On Characterizations of the Basic Feasible Functionals Part II

Robert J. Irwin[*] Bruce M. Kapron[†] James S. Royer[‡]

17 September 2002

[*]Dept. of Elec. Eng. and Computer Science, Syracuse University, Syracuse, NY 13244, USA. Email: rjirwin@top.cis.syr.edu. Research supported in part by NSF grant CCR-9522987.

[†]Dept. of Computer Science, University of Victoria, Victoria, BC V8W 3P6, Canada. Email: bmkapron@maclure.csc.uvic.ca.

[‡]Dept. of Elec. Eng. and Computer Science, Syracuse University, Syracuse, NY 13244, USA. Email: royer@top.cis.syr.edu. Research supported in part by NSF grants CCR-9522987 and CCR-0098198.

1. Introduction

The *basic feasible functionals* are a class of functionals at all simple types that has been proposed as a higher-type analogue of the class of polynomial-time computable functions. The extent to which this analogy holds is not yet completely clear, but the class of basic feasible functionals (BFF) has proven to be quite robust and its various characterizations contain many ideas of independent interest. This paper and its predecessor [IKR01] consider several of these characterizations—some old, some new—with the goals of clarifying their underlying ideas and of placing these ideas into something like a uniform framework. We hope to provide a much sharper account of BFF than heretofore and to do this in a way that is accessible to both the complexity theory and the programming languages communities.

Higher-types and related constructs (e.g., classes, modules, etc.) pervade contemporary computing. There are a great many tools to help in reasoning about the correctness of programs using these features. In contrast, there is very little to help in reasoning about the efficiency of such programs. For the most part we lack even the means to precisely state questions regarding the efficiency of these programs, much less answer such questions. Studying the basic feasible functionals is one way of developing such a means.

This paper's predecessor [IKR01] (henceforth referred to as *Part I*) concerned the type-2 basic feasible functionals. This sequel concerns the full class at all simple types. The move beyond type-level two entails broadening our investigation in two ways:

- Along with BFF we are lead to consider D, a closely related class of functionals introduced by Seth [Set95]. Seth conjectured that above type-level two, D is a strictly larger than BFF. We confirm this in Proposition 59 below.
- We are also led to study how different semantic domains can support distinct notions of feasible computation. Most prior work on higher-type feasibility concerned classes of functionals defined on the full type hierarchy over N (denoted **Full**).[1] In many respects **Full** is a highly unnatural setting in which to study computation. Difficulties with **Full**

[1] N denotes the set of natural numbers. The full type hierarchy over N consists of the straightforward set-theoretic interpretation of the simple types over N. For more details, see Section 5.

and with the class of continuous functionals lead us to introduce the domain of *bounded continuous functionals* (denoted **BCont**) at all simple types and to study BCBFF and BCD, the **BCont** versions of BFF and D, respectively. The **BCont** domain seems a more natural setting for the ideas behind D than does **Full**. Moreover, relative to **BCont**, there turns out to be a simple, compositional sense in which the elements of BCBFF and BCD are polynomial-time computable. We show that there is also a corresponding sense in which, relative to **Full**, the elements of BFF and D are polynomial-time computable, but this sense seems to be inherently more convoluted.

A basic aim of this paper is to explain, as plainly as we can, the senses in which the elements of D, BFF, BCD, and BCBFF are polynomial-time computable relative to their respective semantic settings.

How does one justify calling something a higher-type version of the class of polynomial-time computable functions? At present all roads to such a justification seem to start with Cobham's 1965 characterization of polynomial-time. Let us officially define the polynomial-time computable functions (PF) as the class of type-level 1 functions computable on deterministic Turing *machines* within time polynomial in the length of the inputs. Cobham [Cob65] introduced a simple function algebra (let us call it L) and showed that the L-computable functions are exactly PF. This characterization provides two analogies to work from in constructing higher-type versions of PF: the first being the function-algebra/programming-language/synthetic side of the characterization, and the second being the machine-based/analytic side. Under the programming language analogy we look at conservative, higher-type extensions of L (and related type-1 systems) and the functions they compute. Under the machine analogy we must specify and justify: (i) a machine/cost model for higher-type computations, (ii) a notion of the length of higher-type arguments, and (iii) a notion of polynomial over these sizes, and then study the functionals computable in time "polynomial" in the "lengths" of their inputs. Clearly, the programming language analogy involves a lot less fuss. However, the machine side of the story would seem to be necessary to develop an understanding of the costs of dynamic resources in a higher-type context. Moreover, such an understanding of costs would seem to be necessary part of any convincing argument that one really does have a sensible analogue of PF. The interplay between the synthetic (programming-languages-based) and the analytic (machine-based) analogies is one of the themes of the work below.

The next section gives a brief synopsis of Part I. Section 3 then provides some background on some of the prior work on computational complexity at type-level three and beyond. Section 4 gives an overview of this paper.

2. A Summary of Part I

Part I centered around three characterizations of BFF_2, the type-2 basic feasible functionals.

The first of these three characterizations is through Cook and Kapron's *type-2 bounded typed loop programs* (abbreviated, BTLP_2) programming formalism [CK89, CK90]. This is a simply typed, imperative programming formalism with a loop construct based on Cobham's limited recursion on notation [Cob65]. BTLP_2 is representative of several restricted programming languages that formalize a type-2 analogue of PF through a careful lift of a programming formalism characterization of PF. Our official definition of BFF_2 is as the BTLP_2-computable functionals.

The second characterization considered is based on a machine model and resource bounds. Kapron and Cook [KC91, KC96] introduced:

(i) the answer-length cost model for oracle Turing machines (OTMs);

(ii) a notion of the length of a type-1 function; and

(iii) second-order polynomials, a type-2 analogue of ordinary polynomials.

By a rather difficult proof they showed what we shall call the *Kapron-Cook Theorem*: BFF_2 is precisely the class of functionals computed by oracle Turing machines that run (under the answer-length cost model) within time bounded by a (second-order) polynomial in the lengths of their (type-1 and type-0) inputs. This characterization provides a crisp, complexity-theoretic account of BFF_2.

The above two characterizations have complementary strengths and weaknesses. BTLP_2 is a simple, straightforward programming language, but it seems much simpler to express certain elements of BFF_2 with a (second-order) polynomial-time bounded OTM than with a BTLP_2 procedure.[2] On the other hand, the polynomial-time bounded OTMs are highly *ad hoc* from a programming languages perspective. *Clocked second-order polynomial-time OTMs* [KC91, KC96, Set92] are only a small step toward programming language respectability.

[2]As evidence for this, the hard direction of the proof of the Kapron-Cook Theorem is the translation of polynomial-time bounded OTMs into equivalent BTLP_2 procedures. In contrast, it is relatively simple to translate a BTLP_2 procedure into an equivalent polynomial-time bounded OTM.

The third characterization was introduced as a compromise between the prior two in order to better explain the machine-based model. Our *type-2 inflationary tiered loop programs* (abbreviated, ITLP_2) system is a typed programming formalism inspired by type-theoretic characterizations of PF due to Bellantoni and Cook [BC92] and Leivant and Marion [Lei95, LM93]. ITLP_2 is nonetheless very close to the polynomially-clocked OTMs. ITLP_2 differs from BTLP_2 in that ITLP_2 types can incorporate certain complexity-theoretic information. It also differs in that certain types and iteration bounds are inflationary in the sense that in the course of a computation their denotations can grow (inflate) with the increase of information about the type-1 arguments. This inflation can be seen as the price for ITLP_2's closeness to the machine model.

The present paper is a direct continuation of Part I. In some sense the only new element is that we now allow λ-expressions as arguments—but of course that addition makes a world of difference. This paper can be read independently of Part I with two major provisos. First, Part I contains a reasonably detailed discussion of prior work on higher-type computational complexity. Here we only briefly survey the prior work that addressed computational complexity at type-level three and above. Second, Part I includes a long and detailed proof that the ITLP_2-computable functionals are (second-order) polynomially bounded. Section 10 below describes how to modify that argument to work for the ITLP formalism at all simple types, but we do not present the full argument here.

Convention: When we refer to a numbered section or paragraph of Part I, we shall prefix the number with "I-", e.g., Section I-7 and Proposition I-18, respectively, refer to Section 7 and Proposition 18 of Part I.

3. Prior Work on Complexity above Type-Level 2

Work related to the basic feasible functionals. Buss [Bus86] introduced a class of "polynomial-time" functionals at all simple types as realizers for IS^1_2, his intuitionistic theory of bounded arithmetic. Cook and Urquhart [CU89, CU93] introduced the formal system PV^ω as an alternative way to provide realizers for IS^1_2. PV^ω terms consist of simply typed λ-expressions built from numeric constants, function constants for elements of PF, variables of simple types, and a type-2 recursor R that corresponds to Cobham's limited recursion on notation. The underlying semantic domain of PV^ω is **Full**. The class of PV^ω-computable functionals (at all simple types) was later named the *basic feasible functionals* by Cook and Kapron [CK89, CK90]. In that work they established several programming-formalism characterizations of the BFFs, including the result that the (general) BTLP-computable functionals are exactly the BFFs. They also showed that the BFFs satisfy a Ritchie-Cobham property at all simple types (see footnote 1 in Part I).

Seth [Set95] investigated machine-based characterizations of the BFFs based on Kleene's approach to defining higher-type functionals over **Full** via machines. Seth defined two related machine models that we shall call D-*machines* and E-*machines*. These respectively define two classes of higher-type functionals, D and E. The E-machines are a restricted class of the D-machines, so E $\subseteq$ D. While the classes of D and E functions coincide at type-levels one and two, Seth conjectured that D is strictly larger than E at type-levels above two. By a lift of the proof of the Cook-Kapron Theorem he showed E = BFF. This clever, insightful result provides a useful machine-based characterization of the BFFs. *However*, it is not a full analogue of the Kapron-Cook Theorem. The problem is that Kapron and Cook provided analyses of the *length* of a type-1 object and of *polynomials* over such lengths, and these analyses are part and parcel of why the Kapron-Cook Theorem is such a strong, convincing result. While measuring sizes and second-order polynomials are part of the considerations in Seth's D- and E-machine models, there is no corresponding analysis of what should be the higher-type analogues of length and polynomial. Thus, Seth's E = BFF Theorem does not make clear in what sense the BFFs correspond to higher-type polynomial-time. A large part of the motivation of this paper is to provide appropriate settings for the E = BFF Theorem in order to extract full analogues of the Kapron-Cook Theorem.

Implicit Computational Complexity. Bellantoni and Cook's [BC92] and Leivant and Marion's [Lei95, LM93] type-theoretic characterizations of PF inspired Bellantoni, Girard, Hofmann, Leivant, Marion, Niggl, Schwichtenberg [BNS00, Gir98, Lei94, LM02, Hof97, Hof99b] and others to investigate the use of type systems in restricted programming languages to characterize various complexity classes. (Hofmann has a recent survey of this work [Hof00].) A key motivation for these investigations is to understand how type-theoretic constraints can temper very strong iteration and recursion constructs so that programs built with these tempered constructs have reasonable complexity. Of particular interest here is the work of Bellantoni, Niggl, and Schwichtenberg [BNS00] and Hofmann [Hof97, Hof99b, Hof99a], who constructed restricted, typed programming formalisms that allow certain forms of higher-type recursion, but nonetheless have precisely PF as their type-1 computable functions. There are implied notions of higher-type feasibility in this work, but as of this writing it is not known how these implied notions are related to BFF.

This paper does not treat higher-type feasible recursions—we have *plenty* of other things to worry about. Complexity-theoretic motivated type systems do play an important role in our ITLP, $\mathsf{ITLP}^\flat$, and $\mathsf{ITLP}^\sharp$ programming formalisms discussed below.

4. An Overview

Preliminaries. In the next section we establish some basic notation and state some standard definitions as background. Section 6 introduces our version of the *Bounded Typed Loop Program* (BTLP) formalism. Our official definition of the basic feasible functionals is as the BTLP-computable functionals over **Full**.

Higher-Type Lengths and Polynomials. The serious work begins in Section 7 where we consider how to lift the Kapron-Cook notion of the length of a type-1 function to type-level 2 and beyond. The direct lift of this notion quickly runs into grave problems. To a great extent, the rest of the paper is devoted to examining two approaches for dealing with these problems. The first approach is to restrict our higher-type objects to roughly the largest subclass of the total continuous functions such that this direct lift makes sense. We call this subclass the *bounded continuous functionals*, and the notion of length developed for this class is called *bounded length*. Under the second approach, we analyze higher-type BTLP procedure calls and find that such calls have a type-1 character that we can exploit. We thus introduce a notion of length, called *relative length*, that is assigned to particular *calls* of higher-type functions rather than to the functions themselves. This notion of length is sensible under the **Full**-based semantics and is roughly the notion of length implicit in the proof of Seth's E = BFF Theorem. Relative length is harder to work with as it involves a tangling of bounds and things bounded. Section 8 introduces *higher type polynomials*, a higher-type analogue of polynomial compatible with bounded lengths. Not surprisingly, the higher-type polynomials are a version of the simply typed λ-calculus. *Seth bounds*, rough analogues of polynomials that work with relative lengths, are introduced in Section 9. The notion of the *depth* of a second-order polynomial played a important role in Part I. This notion is generalized to both higher-type polynomials and Seth bounds and is key to the definition of the series of programming formalisms considered next.

Bridges and Jetties. We use the above notions of length and polynomial to construct certain programming formalisms that serve as bridges between the BTLP and E-machine characterizations of the BFFs. Section 10 introduces the first of these: ITLP, for *inflationary typed loop programs.* ITLP is based on ITLP_2 from Part I, but ITLP has a simpler and more uniform type system. ITLP-types have two components: a *shape* (a standard simple type over N) and a *depth* (a natural number). If an object X is assigned type (σ, d), then the interpretation we want is that X corresponds to an ordinary object of simple-type σ *and* the length of X is bounded by a depth-d polynomial (of a type corresponding to σ). We prove that this interpretation holds for both the **Full**-based semantics (with relative lengths and Seth-bounds) and the **BCont**-based semantics (with bounded lengths and higher-type polynomials). In Section 11 we also show how to translate BTLP procedures to equivalent ITLP procedures. Hence, under both semantics the ITLP-functions and BTLP-functions are appropriately polynomially bounded. ITLP is a simple language, but the cost of ITLP type-inference is wildly expensive as it is tied to normalization of simply typed λ-calculus expressions. Thus in Section 12 we introduce $\mathsf{ITLP}^\flat$, a restricted version of ITLP that turns out to compute the same class of functions as ITLP, but permits polynomial-time type inference. $\mathsf{ITLP}^\flat$ turns out to be related to Seth's original E-machine model in the same tight way that ITLP_2 is related to the Kapron-Cook machine model.[3] *Terminology:* Let E and BCE denote the class of ITLP-computable functionals under the **Full**-semantics and **BCont**-semantics, respectively.

Section 13 considers $\mathsf{ITLP}^\sharp$, an extension of $\mathsf{ITLP}^\flat$ inspired by Seth's D-machine model, obtained by slightly enlarging the allowable expressions and adding one new typing rule. The result is dramatic. Under the **Full**-based semantics, we now have nonterminating loops in certain $\mathsf{ITLP}^\sharp$ programs, *but* under the **BCont**-based semantics, all $\mathsf{ITLP}^\sharp$ programs are nicely total and obey the same sort of growth (and run time) bounds as $\mathsf{ITLP}^\flat$. So, in the **Full** setting there is a huge mismatch between $\mathsf{ITLP}^\flat$ and $\mathsf{ITLP}^\sharp$ computability, but a seeming match in the **BCont** setting. We see in Section 17 that there is a huge mismatch in the **BCont** setting, too. *Terminology:* Let D^- denote the class of *total* $\mathsf{ITLP}^\sharp$-computable functionals under the **Full**-semantics and let

[3] *Confession:* Our version of the E-machine model is more liberal than Seth's in one small respect, but as a consequence, translating from *our* E-machines to $\mathsf{ITLP}^\flat$ may require an exponential increase in program size.

BCD^{-} denote the class of $\mathsf{ITLP}^{\sharp}$-computable functionals under the **BCont**-semantics. (It will turn out that $\mathrm{D}^{-} \subseteq \mathrm{D}$ and $\mathrm{BCD}^{-} \subseteq \mathrm{BCD}$. We conjecture that both containments are strict.)

Machine Models. Our study of machine models for feasible higher-type functions begins in Section 14 with a general discussion of the restrictions one is forced to place on such models. The details of our versions of the D- and E-machine models are given in Section 15. We show that $\mathsf{ITLP}^{\flat}$ procedures and E-machines are effectively intertranslatable in Section 16 and thus conclude that the classes of ITLP-computable and E-machine computable functionals coincide in both semantic settings. Also in that section we show how to translate $\mathsf{ITLP}^{\sharp}$ procedures to equivalent D-machines. The bulk of Section 17 is taken up with the proof that $\mathrm{BCE} \subsetneq \mathrm{BCD}^{-}$. From that separation we can immediately conclude that $\mathrm{BCE} \subsetneq \mathrm{BCD}$ and by a modification of the $\mathrm{BCE} \subsetneq \mathrm{BCD}^{-}$ proof we also show that $\mathrm{E} \subsetneq \mathrm{D}$.

Finally, in Section 18 we give a proof of Seth's $\mathrm{E} = \mathrm{BFF}$ Theorem that also yields a simpler proof of the Kapron-Cook Theorem.

Applications and Conclusions. In Section 19 we apply our results to characterize the sections of BFF and to analyze the computational power of higher-type polynomial-time bounded machines under the unit cost-model. Section 20 states a few conclusions of our work and discusses some directions for further exploration.

What is important in this paper? The technical core of the paper is the *grand chain of translations*, a series of results that show how to translate from one programming formalism to another. These results and their translations are:

Proposition 39	$\mathsf{BTLP} \to \mathsf{ITLP}$
Proposition 44	$\mathsf{ITLP} \to \mathsf{ITLP}^{\flat}$
Proposition 54	$\mathsf{ITLP}^{\flat} \to$ E-machines
Proposition 61	E-machines $\to \mathsf{BTLP}$

However, these results may not be as interesting or important as the analyses that support them (e.g., our discussions of higher-type length, polynomials, and machines and our type system for ITLP) and some of the side issues explored (e.g., our definition of $\mathsf{ITLP}^{\sharp}$ and the $\mathrm{BFF} \neq \mathrm{D}$ result). So, we have built a cathedral, but the best parts may be the buttresses and gargoyles.

5. Notation, Conventions, and Such

Numbers and strings. N denotes the set of natural numbers, and each $x \in \mathrm{N}$ is identified with its dyadic representation over $\{\mathbf{0}, \mathbf{1}\}$. Thus, $0 \equiv \epsilon$, $1 \equiv \mathbf{0}$, $2 \equiv \mathbf{1}$, $3 \equiv \mathbf{00}$, etc. We will freely pun between $x \in \mathrm{N}$ as a number and a **0**-**1**-string. The function $\ell en\colon \mathrm{N} \to \mathrm{N}$ is such that, for each $x \in \mathrm{N}$, $\ell en(x) =$ the length of the dyadic representation of x.

For each natural number n, let $\underline{n}$ denote $\mathbf{0}^n$, i.e., n's *unary* representation over $\{\mathbf{0}\}$. We refer to the elements of $\mathbf{0}^*$ as *tally strings*. The set of finite ordinals is denoted by ω. N and ω are, of course, isomorphic. We treat the elements of ω as numbers, but we identify each $n \in \omega$ with the tally string $\underline{n}$. The function $|\cdot|\colon \mathrm{N} \to \omega$ is such that, for each $x \in \mathrm{N}$, $|x| = \underline{\ell en(x)}$, i.e., the unary representation of $\ell en(x)$. The function $|\cdot|\colon \omega \to \omega$ is simply the identity on ω.

We use the elements of N as integer (and string) values to be computed over, and we use the elements of ω as tallies to represents lengths, run times, and, generally, the results of size measurements.

Functions and operations. Suppose A and B are sets. Then unless otherwise stated, $A \to B$ denotes the collection of all total set-theoretic functions from A to B, and $A \rightharpoonup B$ denotes the collection of all (possibly) partial set-theoretic functions from A to B.

For $x_0, \ldots, x_n \in \mathrm{N}$, $\max(\{x_0, \ldots, x_n\}) = \max(x_0, \ldots, x_n) =$ the largest element of the set $\{x_0, \ldots, x_n\}$. By convention, $\max(\emptyset) = 0$. We use the notation: $x \oplus y = \max(x, y)$ and $\bigoplus_{i=m}^{n} x_i = \max(\{x_i \mid m \leq i \leq n\})$. For each x and $y \in \mathrm{N}$, define $x \# y = 2^{\ell en(x) \cdot \ell en(y)}$. Note that $|x| \cdot |y| = |x \# y|$. So # (called *smash*) can be thought of as a kind of a unary multiplication. By convention, for all x, $(x \bmod 0) = (x \bmod 1) = 0$. Let P be a predicate on integers. Then $(\mu y)[P(\vec{x}, y)]$ denotes the least y such that $P(\vec{x}, y)$ holds, if such a y exists, and is undefined otherwise. (We often abbreviate a list $x_0, \ldots, x_k$ by $\vec{x}$.) Let $\langle \cdot, \cdot \rangle$ be a standard, polynomial-time computable pairing function, e.g., the one from Rogers [Rog67]. We also code finite sequences of natural numbers as follows. For each $x \in \mathrm{N}$, let $E(x) = \mathbf{1}^{|x|}\mathbf{0}x$. (Note that the image of E in $\{\mathbf{0}, \mathbf{1}\}^*$ is a collection of prefix codes [LV97, Section 1.4].) Define $[\,] = 0$ and, for each $n > 0$ and each $x_1, \ldots, x_n \in \mathrm{N}$, define

$$[x_1, \ldots, x_n] \quad = \quad \text{the concatenation of } E(x_1), \ldots, E(x_n).$$

It is clear from our definition that each element of N codes is at most one finite sequence and that there is a polynomial-time procedure that, given $x \in \mathrm{N}$, decides if x codes a sequence, and if so, recovers this sequence. It is clear that, with respect to these coded sequences, concatenations, projections, and so on are all polynomial time computable.

For any given set X, id_X denotes the identity function on X. Suppose $f\colon X \to X$. $f^{(0)} = \mathsf{id}_X$, and, for each n, $f^{(n+1)} = f \circ f^{(n)}$. Suppose that A is some arbitrary set, B is a set with with a total order $\leq$ and $\mathcal{F} \subseteq A \to B$. Then, by default, $\mathcal{F}$ is understood as being partially ordered by the pointwise ordering on the function space, e.g., for $f, g \in \mathcal{F}$, $f \leq g$ if and only if, for all $a \in A$, $f(a) \leq g(a)$.

The simple types. *Syntax.* The *simple types* over the base types b_0, $b_1, \ldots$ consist of the base types themselves together with the inductively constructed terms of the form $\sigma \to \tau$ where σ and τ are simple types over b_0, $b_1, \ldots$. Terms of the form $\sigma \to \tau$ are called *arrow* or *higher* types. The $\to$ operator associates to the right, hence $b_0 \to b_1 \to b_2$ parses as $b_0 \to (b_1 \to b_2)$. By convention, we shall write terms of the form $\tau_0 \to \tau_1 \to \cdots \to \tau_{k-1} \to \tau_k$ as $\tau_0 \times \tau_1 \times \cdots \times \tau_{k-1} \to \tau_k$. (Since we shall always be interpreting types over cartesian closed categories, this notation is justified.) An easy argument shows that any arrow type is equivalent to a unique type of the form $\tau_0 \times \tau_1 \times \cdots \times \tau_{k-1} \to \tau_k$, where τ_k is a base type. We shall almost always put our types in this form. The *level* of a type τ (written: *level*(τ)) is defined by:

$$\begin{aligned} \mathit{level}(b_i) &= 0. \\ \mathit{level}(\tau_0 \times \cdots \times \tau_k \to b_i) &= 1 + \max_{i \leq k} \mathit{level}(\tau_i). \end{aligned}$$

Interpretations. Each base type is usually identified with a particular set, and $\sigma \to \tau$ is usually interpreted as being some class of functions from the set σ names to the set τ names.

Assigning types to terms. We will consider a number of formal systems for which we want to assign types (simple or otherwise) to terms. To help in this we introduce some notation and terminology for type assignments.

A *type assignment* is an expression of the form $M\colon \tau$, where M is a term and τ is a type; M is the *subject* and τ is the *predicate* of this type assignment.

A *type context* Γ is a finite (possibly empty) set of type assignments to variables that is consistent in the sense that no variable occurs in more than one type assignment of Γ. If $\Gamma = \{\, x_1\colon \tau_1, \ldots, x_n\colon \tau_n \,\}$, then *Subjects*$(\Gamma) = \{\, x_1, \ldots, x_n \,\}$.

Suppose R is a set of typing rules. A *type judgement* $\Gamma \vdash_R M{:}\,\tau$ asserts that from the type context Γ one can infer the type assignment $M{:}\,\tau$ via the rules R. We often write $\vdash$ instead of $\vdash_R$ as R is usually understood.

For a sample formal system consider the λ-*calculus*:

$$M \ ::= \ x \mid \lambda x\,\textbf{.}\,M \mid (MM) \qquad\qquad x \ ::= \ \text{Variable}$$

For sample typing rules consider the following two:

$$(\rightarrow E) \quad \frac{\Gamma_1 \vdash M{:}\,\sigma \rightarrow \tau \qquad \Gamma_2 \vdash N{:}\,\sigma}{\Gamma_1 \cup \Gamma_1 \vdash (MN){:}\,\tau} \qquad \text{(if } \Gamma_1 \cup \Gamma_2 \text{ is consistent)}$$

$$(\rightarrow I) \quad \frac{\Gamma \vdash M{:}\,\tau}{(\Gamma - \{\, x{:}\,\sigma \,\}) \vdash \lambda x\,\textbf{.}\,M{:}\,\sigma \rightarrow \tau} \qquad \text{(if } \Gamma \cup \{\, x{:}\,\sigma \,\} \text{ is consistent)}$$

The *simply typable terms* are the λ-calculus terms that have type assignments derivable under the above two rules (using the axioms: $\Gamma \vdash x{:}\,\tau$, where $\{\, x{:}\,\tau \,\} \subseteq \Gamma$). These type assignments are not unique. For example, $\lambda x\,\textbf{.}\,x$ can be assigned type $\sigma \rightarrow \sigma$ for any simple type σ.

The above discussion of type assignment is in the style of Curry [CF58]. An alternative approach due to Church [Chu40] has each variable labeled with a simple type. (We assume an infinite supply of variables for each type.) Then each term built up by the typing rules is assigned a unique type. If we start with the rules $(\rightarrow E)$ and $(\rightarrow I)$, then the resulting set of terms is called the *simply typed* λ*-calculus*. Thus in place of $\lambda x\,\textbf{.}\,x$ that can be assigned any type $\sigma \rightarrow \sigma$, we have, for each σ, the term $\lambda x^\sigma\,\textbf{.}\,x^\sigma$ that has the unique type $\sigma \rightarrow \sigma$. We will have occasion to use both styles of type assignment below. It will be clear from the context when a particular style is in force. Each of the higher type formalisms considered below will be, in one way or another, an extension of the simply typed λ-calculus.

Other syntactic matters. $M[x_1 := N_1, \ldots, x_n := N_n]$ will denote the simultaneous substitution of terms $N_1, \ldots, N_n$ for the free occurrences of variables $x_1, \ldots, x_n$, respectively, in the term M. In such substitutions we assume all needed changes in bound variables are made to avoid any capture of a free variable. We shall assume the reader has at least a passing acquaintance with the λ-calculus, e.g., α-, β-, and η-reductions, $\beta\eta$-normal forms, etc. The first chapter of Hindley's book [Hin97] has a brief summary of this material. We highly recommend this book for a clear and concise introduction to the syntactic side of the simple types and the simply typed λ-calculus.

Cartesian closed categories. There is little explicit category theory in this paper, but we do need to work with cartesian closed categories because of their connection to the simple types. (Chapter 3 of Gunter's text [Gun92] has a nice introduction to cartesian closed categories and how they model simple types.) A category $\mathbf{C}$ is *cartesian closed* provided it has finite products and, for any objects B and C, there is an object $B \Rightarrow C$ and a morphism $\mathsf{apply}\colon (B \Rightarrow C) \times B \to C$ satisfying the property: For each $f\colon A \times B \to C$, there is a unique morphism $\mathsf{curry}(f)\colon A \to (B \Rightarrow C)$ such that $f = \mathsf{apply} \circ (\mathsf{curry}(f) \times \mathsf{id}_B)$. The canonical example of a cartesian closed category is the category of sets where, for each B and C, $B \Rightarrow C$ is the set of all functions from B to C and apply and curry are the obvious things. Cartesian closed categories are essentially the models of the simply typed λ-calculus [Lam80]. Variations on the following three basic cartesian closed categories will concern us.

The full type hierarchy. For each simple type σ over base type N, we define the set Full_σ inductively as follows. Let Full_N denote the set N. Suppose $\sigma = \sigma_0 \times \cdots \times \sigma_n \to \mathrm{N}$. Then Full_σ denotes the collection of functions $\mathrm{Full}_{\sigma_0} \times \cdots \times \mathrm{Full}_{\sigma_n} \to \mathrm{N}$. Let **Full** be the category whose objects consist of the closure of the Full_σ's under finite products (including the empty product) and the morphisms consist of the set-theoretic functions between the objects. It is evident that **Full** is cartesian closed.

The total continuous functionals. It is beyond the scope of this paper to include an introduction to Kleene/Kreisel continuous functions. We refer the reader to Normann's technical survey [Nor99] and to Longley's historical survey [Lon01] for this background. For each simple type σ over N, we define the set TCont_σ inductively as follows. Let $\mathrm{TCont}_\mathrm{N}$ denote the set N. Suppose $\sigma = \sigma_0 \times \cdots \times \sigma_n \to \mathrm{N}$. Then TCont_σ denotes the collection of total continuous functions $f\colon \mathrm{TCont}_{\sigma_0} \times \cdots \times \mathrm{TCont}_{\sigma_n} \to \mathrm{N}$. Let **TCont** be the category formed from the closure of the TCont_σ's under finite products. It is standard that **TCont** is cartesian closed and a subcategory of **Full**.

The monotone functionals. Suppose that for $i = 0, \ldots, n$, the set A_i is partially ordered by $\leq_i$. We say that an $f\colon A_0 \times \ldots \times A_n \to \omega$ is *monotone* (with respect to $(A_0, \leq_0), \ldots, (A_n, \leq_n)$) if and only if, for all $\vec{a}, \vec{b} \in A_0 \times \ldots \times A_n$ with $a_0 \leq_0 b_0, \ldots, a_n \leq_n b_n$, we have that $f(\vec{a}) \leq f(\vec{b})$. For each simple type σ over base type ω, we define the set Mon_σ and ordering $\leq_\sigma$ inductively as follows. Let Mon_ω denote the set ω with $\leq_\omega$ the standare ordering on ω. Suppose $\sigma = \sigma_0 \times \cdots \times \sigma_n \to \omega$. Then Mon_σ denotes the set of all monotone $f\colon \mathrm{Mon}_{\sigma_0} \times \cdots \times \mathrm{Mon}_{\sigma_n} \to \omega$ with respect to $(\mathrm{Mon}_{\sigma_0}, \leq_{\sigma_0}), \ldots, (\mathrm{Mon}_{\sigma_n}, \leq_{\sigma_n})$ and $\leq_\sigma$ is the pointwise ordering on the function space. Let **Mon** be obtained by closing up the Mon_σ's under finite products. **Mon** is also cartesian closed. (Note: These are not the same as Platek's [Pla66] hereditarily monotone functionals.)

Other semantic matters. For a particular semantics **S** for a formalism, $[\![\,\cdot\,]\!]_{\mathbf{S}}$ is the *semantic map* that takes syntactic objects of the formal system to their meaning under the **S**-semantics. $[\![\,\cdot\,]\!]_{\mathbf{S}}$ will be a highly overloaded bit of notation. Given a type τ, $[\![\tau]\!]_{\mathbf{S}}$ is the set of objects named by τ under the **S**-semantics. For example, $[\![\tau]\!]_{\mathbf{Full}} = \mathrm{Full}_\tau$, where in this context **Full** stands for the conventional semantics of the simply typed λ-calculus over the full type hierarchy over N. Given a type context $\Gamma = \{\, x_1{:}\,\tau_1, \dots, x_n{:}\,\tau_n \,\}$, $[\![\Gamma]\!]_{\mathbf{S}}$ is the set of all finite maps of the form $\{\, x_1 \mapsto a_1, \dots, x_n \mapsto a_n \,\}$, where $a_1 \in [\![\tau_1]\!]_{\mathbf{S}}, \dots, a_n \in [\![\tau_n]\!]_{\mathbf{S}}$; such maps are called *environments*. Given a term M, a type context Γ such that $\Gamma \vdash M{:}\,\tau$ for some type τ, and a $\rho \in [\![\Gamma]\!]_{\mathbf{S}}$, $[\![M]\!]_{\mathbf{S}}\,\rho$ denotes the element of $[\![\tau]\!]_{\mathbf{S}}$ that is the meaning of term M under semantics **S** when the free-variables of M have the meanings indicated by ρ. For example, if $M =$ "$(\lambda x\,\mathbf{.}\,y)z$", $\Gamma = \{\, y{:}\,\mathrm{N},\ z{:}\,\mathrm{N} \,\}$, and $\rho = \{\, y \mapsto 2,\ z \mapsto 3 \,\}$, then $[\![(\lambda x\,\mathbf{.}\,y)z]\!]_{\mathbf{Full}}\,\rho = 2$. The meaning of a compound term such as "$(\lambda x\,\mathbf{.}\,y)z$" is formally specified by a set of inductive definitions. It will usually suffice to deal with such matters informally as we typically follow standard conventions.

6. Bounded Typed Loop Programs

Our official definition of the basic feasible functionals is through a version of Cook and Kapron's BTLP (Bounded Typed Loop Programs) [CK90]. In BTLP all variables come equipped with a type. To make the type of a variable explicit, we may decorate the variable with the type as a superscript or, in declarations, add ": *the name of the type*" after the variable. The allowable types in BTLP are the simple types over N. The grammar of our version of BTLP is given in Figure 1. In a λ-term, $\lambda v_1, \dots, v_k\,\mathbf{.}\,v(v'_1, \dots, v'_\ell)$, we insist that v is not among $v_1, \dots, v_k$. In the procedure declaration production, the declared variable v_0 has type $\tau_1 \times \cdots \times \tau_\ell \to \mathrm{N}$, where for $i = 1, \dots, \ell$, v_i has type τ_i; λ-terms are typed similarly. In an application, $v(A_1, \dots, A_n)$, the type of v must be $\tau_1 \times \cdots \times \tau_n \to \mathrm{N}$ where, for $i = 1, \dots, n$, τ_i is the type of the argument A_i. Each variable in a BTLP procedure must be declared either in a procedure declaration, parameter list, local variable declaration, or

P	$::=$	**Procedure** $v_0\,(v_1,\ldots,v_\ell)\ P^*\ V\ I^*$ **Return** v_r^{N} **End**	*[procedures]*
V	$::=$	**var** $v_1^{\mathrm{N}},\ldots,v_m^{\mathrm{N}}$;	*[local var. decls.]*
I	$::=$	L ; | $v^{\mathrm{N}} := E$;	*[instructions]*
L	$::=$	**Loop** v_0^{N} **with** v_1^{N} **do** I^* **Endloop**	*[loops]*
E	$::=$	$1 \mid v^{\mathrm{N}} \mid v_0^{\mathrm{N}} + v_1^{\mathrm{N}} \mid v_0^{\mathrm{N}} \dot{-} v_1^{\mathrm{N}} \mid v_0^{\mathrm{N}} \# v_1^{\mathrm{N}} \mid v_0\,(A_1,\ldots,A_n)$	*[expressions]*
A	$::=$	$v \mid \lambda v_1,\ldots,v_k \,\textbf{.}\, v(v_1',\ldots,v_\ell')$	*[arguments]*

Figure 1: The grammar for BTLP

λ-term. Recursive procedure calls are forbidden.[4] Variable scoping is static and follows standard conventions except that in procedure bodies we forbid nonlocal references to variables of type N. **N.B.** In the body of a λ-term we *do* allow references to type N variables that are declared local in the procedure whose body contains that λ-term.

The operational semantics of BTLP is boringly conventional. We shall discuss only its key points. Parameter passing is call-by-value. From this and our conventions of global variables it follows that procedure calls, and expressions in general, have no side-effects. In expressions, "+" denotes addition, "$\dot{-}$" denotes proper subtraction, and "#" denotes the smash function $((x,y) \mapsto 2^{|x|\cdot|y|})$. The effect of a loop statement of the form

Loop `w` **with** `x` **do** $I_1 \cdots I_n$ **Endloop**

is to iterate on $I_1 \cdots I_n$ $|\mathtt{w}|$-many[5] times *with the following restrictions:* neither `w` nor `x` may be assigned to within $I_1 \cdots I_n$ and, for each assignment "$v := E$" within the body of the loop, whenever this assignment is executed, if the value of E is of length greater than $|\mathtt{x}|$, then the result of the statement's execution is to assign 0 to v.

By a set of standard tricks one can achieve the effect of **If**-statements, a richer set of expressions, etc. By another set of standard tricks one may assume

[4] Because of λ-terms and higher-types, a bit of care is needed in stating this prohibition. Assume, in a given context, that each procedure has a distinct name. Let $G = (N, E)$ be a directed graph in which N consists of procedure names and $(p_1, p_2) \in E$ if and only if in the body of the procedure named by p_1 there is an occurrence of p_2. The prohibition on recursion is then simply the requirement that this graph is acyclic.

[5] Convention: $|\mathtt{w}|$ denotes the length of the value of `w`.

```
Procedure SumUp′(F: (N → N) → N, x: N)
  var bnd, maxi, sum, i;
  (* Find the i for which F(λz: N.i) is maximal. *)
  maxi := 0;   i := 0;
  Loop x with x do
    If F(λz: N.maxi) < F(λz: N.i) then maxi := i; Endif;
    i := i+1;
  Endloop;
  (* Compute an upper bound on the sum. *)
  bnd := (x + 1) · (F(λz: N.maxi) + 1);
  (* Now compute the sum itself. *)
  sum := 0;   i := 0;
  Loop x with bnd do
    sum := sum + F(λz: N.i);
    i := i + 1;
  Endloop;
  Return sum
End
```

Figure 2: A BTLP procedure for T'

without loss of generality that the production $A ::= \lambda v_1, \ldots, v_k \,.\, v(v'_1, \ldots, v'_\ell)$ is liberalized to $A ::= \lambda v_1, \ldots, v_k \,.\, E$, where E is some arbitrary expression. In writing BTLP procedures we shall freely make use of these obvious extensions.

For an example of a higher-type BTLP procedure, let us consider a type-3 version of the T functional of Example I-3. Let $T' \colon ((\mathrm{N} \to \mathrm{N}) \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ be the functional given by:

$$T'(F, x) \quad = \quad \sum_{i<|x|} F(\lambda z \,.\, i). \tag{1}$$

T' is computed by the BTLP procedure given in Figure 2.

We formally define the basic feasible functionals as follows.

Definition 1.

(a) BFF is the class of BTLP-computable functionals over **Full**.

(b) For each $i \geq 1$, BFF_i is the class of type-level i functionals computed by BTLP procedures in which the allowable types are restricted to those of type-levels no greater than i.

(c) For each $i \geq 1$, the *i-section* of BFF (written: $\mathbf{Sec}_i(\text{BFF})$) is the class of type-level i functionals computed by BTLP procedures (with no restriction on allowable types). $\diamond$

It is an easy exercise to show that $\text{BFF}_1 = \text{PF}$ and that the above definition of BFF_2 agrees with its prior definition in Definition I-4. It turns out that, for each $i > 0$, $\text{BFF}_i = \mathbf{Sec}_i(\text{BFF})$ (Proposition 66), but we have some work to do before we can prove this.

7. Notions of Higher Type Length

One of our stated goals is to explain the senses in which the BFFs are polynomial-time computable relative to their semantic setting. The definition of the BFFs as the BTLP-computable functionals and the facts that $\text{BFF}_1 = \text{PF}$ and that BFF_2 agrees with its prior definitions make a start at this, but are hardly convincing evidence. Our route to such evidence is through "analytic" characterizations of the BFFs (and related classes) in which we specify (i) a higher-type notion of length, (ii) a higher-type notion of polynomial, and (iii) a higher-type machine/cost model and then examine the functionals computable in polynomial-time in the lengths of their arguments—relative to our choices of higher-type lengths, polynomials, and machines. We start by considering Kapron and Cook's [KC96] notion of the length of a type-1 function and roughly analogous notions at higher type levels.

Kapron and Cook defined the *length* of an $f\colon \text{N} \to \text{N}$ as the $|f| \in \text{Mon}_{\omega\to\omega}$ such that, for all $n \in \omega$,

$$|f|(n) \quad = \quad \max\{\, |f(x)| : |x| \leq n \,\}. \tag{2}$$

That is, $|f|(n)$ is the length of the largest value f can return on arguments of length no greater than n. (See Definition I-5 and the discussion around it.) We would like to somehow lift this definition to obtain a sensible notion of length for functionals of each simple type. Let us first consider how to do this for functions of type $(\text{N} \to \text{N}) \to \text{N}$. For such functions, the direct lift of (2) is:

$$|F| \quad = \quad \lambda g \,.\, \max\{\, |F(f)| \;\vdots\; |f| \leq g \,\}, \tag{3}$$

where g ranges over $\mathrm{Mon}_{\omega\to\omega}$. This definition has its difficulties. Consider

$$F_0 \quad = \quad \lambda f \mathbin{.} \begin{cases} 0, & \text{if } f = \lambda y \mathbin{.} 0; \\ (\mu y)[\, f(y) \neq 0\,], & \text{otherwise.} \end{cases}$$

Clearly, F_0 is total but unbounded on $\mathrm{N} \to \{\,0,1\,\}$. However, for each $g \in \mathrm{N} \to \{\,0,1\,\}$ we have that $|g| \leq \lambda n \mathbin{.} \underline{1}$, and hence

$$\begin{aligned} |F_0|(\lambda n \mathbin{.} \underline{1}) \quad &= \quad \max\{\, |F_0(f)| \mid |f| \leq \lambda n \mathbin{.} \underline{1} \,\} \\ &= \quad \max\{\, |F_0(f)| \mid \text{ for all } x,\ |f(x)| \leq \underline{1} \,\} \\ &\geq \quad \max\{\, |F_0(f)| \mid f \text{ is 0–1 valued} \,\} \\ &= \quad \infty. \end{aligned}$$

Thus the definition of length given by (3) is useless — we take as a basic principle that total functionals should have total lengths.

We consider two approaches to repairing (3). The first attack is based on the observation that F_0's discontinuity seems to play a crucial role in making $|F_0|(\lambda n \mathbin{.} \underline{1})$ undefined. Restricting F in (3) to continuous functionals may thus solve the problem. The second approach is based on the observation that BTLP procedures happily compute over things of the ilk of F_0 and remain total. Hence, the actual use of an F in a BTLP computation is much weaker than is reflected in the aggressive maximization on the right-hand side of (3). Taming this maximization is thus another possibility for solving the problem.

7.1. The continuous case: Bounded length

For $\mathrm{TCont}_{(\mathrm{N}\to\mathrm{N})\to\mathrm{N}}$ things seem to work out well.

Lemma 2. *Suppose* $F\colon \mathrm{TCont}_{(\mathrm{N}\to\mathrm{N})\to\mathrm{N}}$*. Then* $|F|\colon \mathrm{Mon}_{\omega\to\omega} \rightharpoonup \omega$ *as defined in (3) is total, continuous, and monotone.*

Proof. *Notation:* For each $f\colon \mathrm{N} \to \mathrm{N}$ and $x \in \mathrm{N}$, let $f|_x$ be the finite function with the graph $\{\,(w, f(w)) \mid w < x\,\}$. We write $F(\sigma)\!\downarrow$ if and only if, for some y it is the case that for all $f \supseteq \sigma$, $F(f) = y$.

The argument is a simple application of König's Lemma. Fix $g\colon \mathrm{Mon}_{\omega\to\omega}$ and let $B = \{\, f \mid |f| \leq g \,\}$. We view the elements of B as corresponding to the branches of a finitely branching tree T in the standard fashion: The nodes of T consist of the elements of $\{\, f|_x \mid f \in B \ \ \& \ \ x \in \mathrm{N} \,\}$. T's root is the everywhere undefined function, and the children of a node $f|_x$ are the nodes of the form $f|_x \cup \{\,(x,y)\,\}$ where $|y| \leq g(|x|)$. (Thus T is finitely branching as claimed.) For $f \in B$, the sequence of nodes $f|_0,\ f|_1,\ f|_2, \ldots$ is obviously a branch of T. Also, for a given branch of T the union of the nodes along the branch is obviously an element of B.

For each $f \in B$, let $\delta_F(f) = (\mu x)[\,F(f|_x)\!\downarrow\,]$ and call $\delta_F(f)$ the *modulus of continuity of F on f.* Since F is total and continuous, δ_F is total. Consider the subtree of T in which, for each $f \in B$, the subtree below $f|_{\delta_F(f)}$ is deleted. The result is a finitely branching tree with no infinite paths which, by König's Lemma, is finite. Hence it follows that the value of F on B is bounded and thus that the value of $|F|$ on g is defined and depends on only finitely many values of g. Therefore we have that $|F|$ is total and continuous as required. The monotonicity follows directly from the definition of $|F|$. $\square$

A new trouble emerges when we try to extend this approach to type-level three. For $\Psi\colon \mathrm{TCont}_{((\mathrm{N}\to\mathrm{N})\to\mathrm{N})\to\mathrm{N}}$ the analogous definition of $|\cdot|$ is

$$|\Psi| \quad = \quad \lambda G\,\boldsymbol{.}\, \max(\{\, |\Psi(F)| \;\vdots\; |F| \leq G \,\}),$$

where G ranges over $\mathrm{Mon}_{(\omega\to\omega)\to\omega}$ and F ranges over $\mathrm{TCont}_{(\mathrm{N}\to\mathrm{N})\to\mathrm{N}}$. The problem is that the bounded set $B_G = \{\, F \;\vdots\; |F| \leq G \,\}$ generally fails to be compact in the appropriate topology and so there is no guarantee that a continuous Ψ is bounded on B_G. In particular, consider Ψ_0 given by

$$\Psi_0(F) \quad = \quad (\mu y)[\,F(\chi_\emptyset) = F(\chi_{\{\,y\,\}})\,]$$

where, for each $A \subseteq \mathrm{N}$, χ_A is the characteristic function of A. It is straightforward to show that Ψ_0 is in $\mathrm{TCont}_{((\mathrm{N}\to\mathrm{N})\to\mathrm{N})\to\mathrm{N}}$. However, Ψ_0 is unbounded on $B = \{\, F \in \mathrm{TCont}_{(\mathrm{N}\to\mathrm{N})\to\mathrm{N}} \;\vdots\; F \text{ is 0–1-valued} \,\}$ and each element of B has its length bounded above by $\lambda g\,\boldsymbol{.}\,\underline{1}$ (where g ranges over $\mathrm{Mon}_{\omega\to\omega}$). Hence $|\Psi_0|(\lambda g\,\boldsymbol{.}\,\underline{1})$ is undefined. The root of the difficulty here is that continuous F's can be quite small as judged by (3), but have arbitrarily large moduli of continuity and these can be made use of by a continuous type-3 functional, such as Ψ_0 above, to grow arbitrarily large. If we replace $|\cdot|$ for type-2 functionals with a notion that takes into account the moduli of continuity of such functionals, then total, continuous type-3 functionals are indeed bounded on the length-bounded subsets of $\mathrm{TCont}_{(\mathrm{N}\to\mathrm{N})\to\mathrm{N}}$. We will see other reasons for such a redefinition of $|\cdot|$ in Appendix A. However, such a redefinition would bring up a host of issues we wish to avoid in the present paper. Thus we will do without a notion of length that applies to *all* the continuous functions of type-level 3 and above. Instead we introduce the following subclasses of the continuous functionals which are, by definition, bounded on bounded sets.

Convention: For the rest of this subsection, let σ, σ_0, $\sigma_1, \ldots$ range over simple types over N. Also, for each σ, let $|\sigma|_b$ denote the corresponding simple type over ω, i.e., $\sigma[\mathrm{N} := \omega]$.

Definition 3. For each σ, the classes BCont_σ, $\mathrm{Len}_{|\sigma|_b}$, and $\mathrm{TLen}_{|\sigma|_b}$ and the function $|\cdot|_b : \mathrm{TCont}_\sigma \to \mathrm{Len}_{|\sigma|_b}$ are defined inductively as follows.

(a) $\mathrm{BCont}_\mathrm{N} = \mathrm{N}$. $\mathrm{Len}_\omega = \mathrm{TLen}_\omega = \omega$. For each $x \in \mathrm{N}$, $|x|_b = |x|$.

(b) Suppose $\sigma = \sigma_0 \times \cdots \times \sigma_n \to \mathrm{N}$ and that, for each $i \le n$, $\mathrm{BCont}_{\sigma_i}$, $\mathrm{Len}_{|\sigma_i|_b}$, $\mathrm{TLen}_{|\sigma_i|_b}$, and $|\cdot|_b : \mathrm{TCont}_{\sigma_i} \to \mathrm{Len}_{|\sigma_i|_b}$ are given. For each $f \in \mathrm{TCont}_\sigma$, $\ell_0 \in \mathrm{TLen}_{|\sigma_0|_b}, \ldots, \ell_n \in \mathrm{TLen}_{|\sigma_n|_b}$, define

$$|f|_b(\vec{\ell}) \quad = \quad \sup\left\{ |f(x_0, \ldots, x_n)| \;\vdots\; \begin{array}{l} \text{for each } i \le n,\ x_i \in \\ \mathrm{BCont}_{\sigma_i} \text{ and } |x_i|_b \le \ell_i \end{array} \right\}.$$

We call $|f|_b$ the *bounded length* of f and define $\mathrm{Len}_{|\sigma|_b} = \{\, |f|_b \;\vdots\; f \in \mathrm{TCont}_\sigma \,\}$. We say that $f : \mathrm{TCont}_\sigma$ is *bounded continuous* if and only if $|f|_b$ is total. Let BCont_σ be the collection of all the bounded continuous members of TCont_σ and $\mathrm{TLen}_{|\sigma|_b} = \{\, |x|_b \;\vdots\; x \in \mathrm{BCont}_\sigma \,\}$. ◇

Suppose $\sigma = \sigma_0 \times \cdots \times \sigma_n \to \mathrm{N}$. Then:

$$\begin{aligned} \mathrm{Len}_{|\sigma|_b} &\subseteq \mathrm{TLen}_{|\sigma_0|_b} \times \cdots \times \mathrm{TLen}_{|\sigma_n|_b} \rightharpoonup \omega. \\ \mathrm{TLen}_{|\sigma|_b} &\subseteq \mathrm{TLen}_{|\sigma_0|_b} \times \cdots \times \mathrm{TLen}_{|\sigma_n|_b} \to \omega. \end{aligned}$$

Also, $f \in \mathrm{BCont}_\sigma$ if and only if $f \in \mathrm{TCont}_\sigma$ and is bounded on all the sets of the form

$$\{\, (x_0, \ldots, x_n) \;\vdots\; x_i \in \mathrm{BCont}_{\sigma_i} \text{ and } |x_i|_b \le \ell_i \text{ for } i = 0, \ldots, n \,\},$$

where $\ell_0 \in \mathrm{TLen}_{|\sigma_0|_b}, \ldots, \ell_n \in \mathrm{TLen}_{|\sigma_n|_b}$. For each σ at type-level 1, it is clear that $\mathrm{BCont}_\sigma = \mathrm{TCont}_\sigma$ and $\mathrm{Len}_{|\sigma|_b} = \mathrm{TLen}_{|\sigma|_b} = \mathrm{Mon}_{|\sigma|_b}$, and an easy extension of Lemma 2 shows that the same equalities hold for each σ at type-level 2. Now suppose σ is at type-level 3 or above. Here matters are more complicated. First, BCont_σ is a strict subset of TCont_σ, e.g., for $\sigma = ((\mathrm{N} \to \mathrm{N}) \to \mathrm{N}) \to \mathrm{N}$, $\Psi_0 \in (\mathrm{TCont}_\sigma - \mathrm{BCont}_\sigma)$. However, it turns out that the BTLP-computable functions over BCont_σ are contained within BCont_σ (Corollary 42). So, the BCont_σ classes are quite compatible with BTLP. A more serious complication is that $\mathrm{TLen}_{|\sigma|_b}$ contains members that fail to be continuous. Each element of $\mathrm{TLen}_{|\sigma|_b}$ can, however, be realized as the supremum of finite, lower approximations and this will turn out to be enough of a handle on these functionals for us to make use of them.

Terminology: Define **BCont** to be the category whose objects consist of the closure of the $\mathbf{BCont}_\sigma$'s under finite products (including the empty product) and whose morphisms are the obvious things. Define **TLen** analogously. We observe that:

Lemma 4. **BCont** *and* **TLen** *are both cartesian closed.*

Note that $|\cdot|_b$ is not quite a functor from **BCont** to **TLen** since, in general, $|f \circ g|_b \neq |f|_b \circ |g|_b$. However, by a straightforward argument we do have:

Lemma 5 ($|\cdot|_b$-Subcompositionality). *Suppose that* $\sigma_0 = \sigma_1 \times \cdots \times \sigma_k \to \mathrm{N}$ *and* $x_0 \in \mathrm{BCont}_{\sigma_0}, \dots, x_k \in \mathrm{BCont}_{\sigma_k}$. *Then:*

$$|x_0(x_1,\dots,x_k)|_b \quad \leq \quad |x_0|_b\,(|x_1|_b\,,\dots,|x_k|_b).$$

Bounded length is unsatisfactory in that the BCont_σ classes seem rather *ad hoc*, and worse, bounded lengths are in general discontinuous. However, its definition is in the spirit of the definition of type-1 length in that the length of a higher-type functional is another functional on the lengths of lower-type arguments. Our second notion of length violates this spirit in the interest of generality.

7.2. The general case: Relative length

Consider a particular procedure call, `F(f,z)`, in some BTLP program where `F`, `f`, and `z` are variables of types $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$, $\mathrm{N} \to \mathrm{N}$, and N, respectively. Recall that in any BTLP program, global references to variables of type N are forbidden and each higher-type variable refers to an immutable object. Hence in any execution of a procedure in which the call "`F(f,z)`" occurs, the only way to vary what is returned from this call is to vary the value of `z`. So, this call really amounts to a type-1 procedure call. Now consider the procedure call `F(`λ`x.g(x,y),z)` where `F` and `z` are as before and `g` and `y` are variables of types $\mathrm{N}^2 \to \mathrm{N}$ and N, respectively. In this case we can vary `F`'s type-1 argument, but only through varying the value of the integer variable `y`. So in this case, too, the call really amounts to a type-1 call. It is evident that BTLP is set up so that *all* procedure calls have this type-1 character. We shall exploit this fact to formalize an alternative notion of length, *relative length*, that does not try to assign lengths to functionals, but instead to particular "calls" of functionals.

Since relative length (a would-be semantic notion) depends on the *syntax* of particular calls, we need to proceed a bit circuitously so as to avoid undue tangling of syntax with semantics. We thus define two related "relative lengths." The first, $|\cdot|_r$ (Definition 9), takes each $x \in \mathrm{Full}_\sigma$ to another appropriate semantic object that will be the relative length of x. The second, $\|\cdot\|_r$ (Definition 10(c)), takes an expression E to another syntactic object $\|E\|_r$ such that the relative length of the denotation of E is no greater than the denotation of $\|E\|_r$ (Lemma 11(b)). Before stating these definitions we introduce a few auxiliary notions.

Conventions: Suppose $\mathtt{x}$ is a variable of type N. We shall often treat the expression $|\mathtt{x}|$ as a variable of type ω. This is barbarous, but it helps us avoid worse. For the rest of this subsection, let σ, σ_0, $\sigma_1, \ldots$ range over simple types over N and τ, τ_0, $\tau_1, \ldots$ range over simple *arrow* types over N. Also let j and k range over natural numbers such that $k > 0$ and $0 \leq j \leq k$.

General Convention: When we say that $\|\cdot\|_r$ maps an expression E to another syntactic object, we shall a bit informal in drawing the boundary around what class of expressions the E's are drawn from. The class includes BTLP expressions arguments (as given in the grammar of Figure 1) as well as arbitrary BTLP applications. We will also apply $\|\cdot\|_r$ to expressions (in the broad sense) from other programming formalisms introduced below.

Definition 6. We define $|\cdot|_r$ as a map over type notation as follows. Let $|\mathrm{N}|_r = \omega$ and $\left|\tau_1 \times \cdots \times \tau_j \times \mathrm{N}^{k-j} \to \mathrm{N}\right|_r = \tau_1 \times \cdots \times \tau_j \times \omega^{k-j} \to \omega$. ◇

Definition 7. Let $\mathrm{FL}_{|\mathrm{N}|_r} = \omega$ and, for $\tau = \tau_1 \times \cdots \times \tau_j \times \mathrm{N}^{k-j} \to \mathrm{N}$, let $\mathrm{FL}_{|\tau|_r}$ denote the collection of all elements of $\mathrm{Full}_{\tau_1} \times \cdots \times \mathrm{Full}_{\tau_j} \times \omega^{k-j} \to \omega$ that are monotone nondecreasing in their type-ω arguments. ◇

Definition 8. Given $f\colon \mathrm{N}^k \to \omega$ and $n_1, \ldots, n_k \in \omega$, define

$$(\bigoplus x_1{\restriction}n_1, \ldots, x_k{\restriction}n_k)\, f(\vec{x}) \quad = \quad \max\{\, f(\vec{x}) \mid |x_1| \leq n_1, \ldots, |x_k| \leq n_k \,\}.$$

◇

Note that in the expression "$(\bigoplus x_1{\restriction}n_1, \ldots, x_k{\restriction}n_k)\, f(\vec{x})$", $n_1, \ldots, n_k$ are free, but $x_1, \ldots, x_k$ are not. Clearly, if f is total, then so is $\lambda\vec{n}.\, (\bigoplus x_1{\restriction}n_1, \ldots, x_k{\restriction}n_k)\, f(\vec{x})$.

Definition 9. For each σ we define $|\cdot|_r : \mathrm{Full}_\sigma \to \mathrm{FL}_{|\sigma|_r}$ as follows. For $x \in \mathrm{N}$, let $|x|_r = |x|$; for $x \in \mathrm{Full}_\tau$, where $\tau = \tau_1 \times \cdots \times \tau_j \times \mathrm{N}^{k-j} \to \mathrm{N}$, let

$$|x|_r \quad = \quad \lambda x_1, \ldots, x_j, n_{j+1}, \ldots, n_k \,.\, (\bigoplus x_{j+1}{\restriction} n_{j+1}, \ldots, x_k{\restriction} n_k)\, |x(x_1, \ldots, x_k)|.$$

$\diamond$

Definition 10. Suppose $\Gamma = \{\, \mathtt{x}_1{:}\,\tau_1, \ldots, \mathtt{x}_j{:}\,\tau_j, \mathtt{x}_{j+1}{:}\,\mathrm{N}, \ldots, \mathtt{x}_k{:}\,\mathrm{N} \,\}$.

(a) Define $|\Gamma|_r = \{\, \mathtt{x}_1{:}\,\tau_1, \ldots, \mathtt{x}_j{:}\,\tau_j, |\mathtt{x}_{j+1}|{:}\,\omega, \ldots, |\mathtt{x}_k|{:}\,\omega \,\}$ and $[\![|\Gamma|_r]\!]_{\mathbf{FL}}$ as the collection of all environments $\{\, \mathtt{x}_1 \mapsto x_1, \ldots, \mathtt{x}_j \mapsto x_j, |\mathtt{x}_{j+1}| \mapsto n_{j+1}, \ldots, |\mathtt{x}_k| \mapsto n_k \,\}$ such that $x_i \in \mathrm{Full}_{\tau_i}$ $(i \le j)$ and $n_i \in \omega$ $(i > j)$.

(b) Suppose $\rho = \{\, \mathtt{x}_1 \mapsto x_1, \ldots, \mathtt{x}_k \mapsto x_k \,\} \in [\![\Gamma]\!]_{\mathbf{Full}}$. Define $|\rho|_r = \{\, \mathtt{x}_1 \mapsto x_1, \ldots, \mathtt{x}_j \mapsto x_j, |\mathtt{x}_{j+1}| \mapsto |x_{j+1}|, \ldots, |\mathtt{x}_k| \mapsto |x_k| \,\} \in [\![|\Gamma|_r]\!]_{\mathbf{FL}}$.

(c) Suppose $\Gamma \vdash E{:}\,\sigma$ and $\rho \in [\![\Gamma]\!]_{\mathbf{Full}}$. Define $\|E\|_r$ and $[\![\, \|E\|_r \,]\!]_{\mathbf{FL}}\ |\rho|_r$ as follows. **N.B.** Recall that $\|\cdot\|_r$ is a mapping over syntax.

Case 1: E is a variable or a constant. Then $\|E\|_r = |E|_r$. (Note that $|\cdot|_r$ is used syntactically here and that when $\sigma = \mathrm{N}$ we usually write $|E|$ in place of $|E|_r$.) We also define $[\![|E|_r]\!]_{\mathbf{FL}}\ |\rho|_r = |[\![E]\!]_{\mathbf{Full}}\ \rho|_r$.

Case 2: $\sigma = \mathrm{N}$ and E is neither a variable nor a constant. Then

$$\|E\|_r \quad = \quad (\bigoplus \hat{\mathtt{y}}_1{\restriction}|\mathtt{y}_1|, \ldots, \hat{\mathtt{y}}_\ell{\restriction}|\mathtt{y}_\ell|)\, \big|E[\mathtt{y}_1, \ldots, \mathtt{y}_\ell \mapsto \hat{\mathtt{y}}_1, \ldots, \hat{\mathtt{y}}_\ell]\big|,$$

where $\mathtt{y}_1, \ldots, \mathtt{y}_\ell$ are precisely the type-N free variables of E, and $\hat{\mathtt{y}}_1, \ldots, \hat{\mathtt{y}}_\ell$ are fresh variables. We also define $[\![(\bigoplus \hat{\mathtt{y}}_1{\restriction}|\mathtt{y}_1|, \ldots, \hat{\mathtt{y}}_\ell{\restriction}|\mathtt{y}_\ell|)\, |E|\,]\!]_{\mathbf{FL}}\ |\rho|_r$ to be the obvious thing.

Case 3: $\sigma = \tau'_1 \times \cdots \times \tau'_s \times \mathrm{N}^{t-s} \to \mathrm{N}$ and $E = \lambda \mathtt{y}_1, \ldots, \mathtt{y}_t \,.\, E'$. Then

$$\big\|\lambda \mathtt{y}_1, \ldots, \mathtt{y}_t \,.\, E'\big\|_r \quad = \quad \lambda \mathtt{y}_1, \ldots, \mathtt{y}_s, |\mathtt{y}_{s+1}|, \ldots, |\mathtt{y}_t| \,.\, \big\|E'\big\|_r\,.$$

We also define $[\![\lambda \mathtt{y}_1, \ldots, \mathtt{y}_s, |\mathtt{y}_{s+1}|, \ldots, |\mathtt{y}_t| \,.\, \|E'\|_r]\!]_{\mathbf{FL}}\ |\rho|_r$ to be the obvious thing. $\diamond$

Lemma 11. *Suppose* $\Gamma \vdash E{:}\,\sigma$ *and* $\rho \in [\![\Gamma]\!]_{\mathbf{Full}}$.

(a) $[\![\|E\|_r]\!]_{\mathbf{FL}}\ |\rho|_r$ *is a well-defined element of* $\mathrm{FL}_{|\sigma|_r}$.

(b) $|[\![E]\!]_{\mathbf{Full}}\ \rho|_r \le [\![\|E\|_r]\!]_{\mathbf{FL}}\ |\rho|_r$.

Lemma 12 ($\|\cdot\|_r$-Subcompositionality). *Suppose that* $\tau_0 = \tau_1 \times \cdots \times \tau_j \times \mathrm{N}^{k-j} \to \mathrm{N}$, $\Gamma \vdash E_0{:}\,\tau_0, \ldots, \Gamma \vdash E_j{:}\,\tau_j$, $\Gamma \vdash E_{j+1}{:}\,\mathrm{N}, \ldots, \Gamma \vdash E_k{:}\,\mathrm{N}$, *and* $E =_{\beta\eta} (E_0 \cdots E_k)$. *Then, for each* $\rho \in [\![\Gamma]\!]_{\mathbf{Full}}$,

$$[\![\|E\|_r]\!]_{\mathbf{FL}}\ |\rho|_r \quad \le \quad f_0(x_1, \ldots, x_j, n_{j+1}, \ldots, n_k),$$

where $f_0 = [\![|E_0|_r]\!]_{\mathbf{FL}}\,|\rho|_r$, $x_i = [\![E_i]\!]_{\mathbf{Full}}\,\rho$ $(0 < i \le j)$, *and* $n_i = [\![\|E_i\|_r]\!]_{\mathbf{FL}}\,|\rho|_r$ $(i > j)$.

The proofs of the above two lemmas are straightforward and thus omitted.

Relative length roughly corresponds to the notion of length implicit in Seth's E = BFF Theorem.

7.3. Applying these notions

Example 13. To illustrate the use of these two notions of length, we show how to bound $|T'(F,x)|$ in terms of F and x, where $T' = \lambda F, x \,\mathbf{.}\, \sum_{i<|x|} F(\lambda z \,\mathbf{.}\, i)$ was introduced in Section 6. For the bounded-continuous case, it is easy to check that for all $F \in \mathrm{BCont}_{(\mathrm{N}\to\mathrm{N})\to\mathrm{N}}$ and $x \in \mathrm{N}$,

$$|T'(F,x)| \quad \le \quad |F|_b\,(\lambda n \,\mathbf{.}\, |x|) \cdot |x|. \tag{4}$$

For the general case we have that

$$|T'(F,x)| \quad \le \quad [\![\, \|\mathtt{F}(\lambda\mathtt{y.len(x)})\|_r \cdot |\mathtt{x}|\,]\!]_{\mathbf{FL}}\,|\rho|_r\,, \tag{5}$$

where $\rho = \{\, \mathtt{F} \mapsto F, \mathtt{len} \mapsto \ell en, \mathtt{x} \mapsto x \,\}$ and $\ell en \colon \mathrm{N} \to \mathrm{N}$ is such that $\ell en(w) =$ the length of w. $\diamond$

Compared to (4), the bound of (5) is a bit ugly—but it is the tighter of the two bounds. This is because the maximization implicit in the right-hand side of (5) is usually over a *much* smaller set than the maximization implicit in the right-hand side of (4).

Example 14. Suppose we want to argue that the right-hand sides of (4) and (5) are each, in appropriate senses, feasible/polynomial bounds in terms of F and x. For (5) we can sketch an argument as follows. Let $\rho = \{\, \mathtt{F} \mapsto F, \mathtt{len} \mapsto \ell en, \mathtt{x} \mapsto x \,\}$.

1. Observe that the right-hand side of (5) is equal to $\mathbf{q}(|f|, |x|)$ where $\mathbf{q}$ is the second-order polynomial $g(n) \cdot n$ and where $f = \lambda x \,\mathbf{.}\, F(\lambda y \,\mathbf{.}\, \ell en(x))$, and moreover,

$$|f| \quad = \quad \lambda n \,\mathbf{.}\, \left([\![\, \|\mathtt{F}(\lambda\mathtt{y.len(x)})\|_r]\!]_{\mathbf{FL}} \,\{\, \mathtt{F} \mapsto F, \mathtt{len} \mapsto \ell en, |\mathtt{x}| \mapsto n \,\} \right).$$

2. For each $y \in \mathrm{N}$, let $\rho_y = \rho \cup \{\, \mathtt{y} \mapsto y \,\}$. Then for each y, we have that

$$[\![\|\mathtt{len}(\mathtt{x})\|_r]\!]_{\mathbf{FL}}\, |\rho_y|_r \;\;\leq\;\; |\ell en|(|x|) \;\;\leq\;\; |x| \;\;=\;\; [\![\|\mathtt{x}\|_r]\!]_{\mathbf{FL}}\, |\rho_y|_r\,.$$

So since the relative length of the body of $\lambda\mathtt{y}.\mathtt{len}(\mathtt{x})$ is uniformly polynomially-bounded in terms of the sizes of both its local and global variables, we will accept $\lambda\mathtt{y}.\mathtt{len}(\mathtt{x})$ as feasibly bounded.

3. Since $\lambda\mathtt{y}.\mathtt{len}(\mathtt{x})$ is feasibly bounded, by the principle that "feasible applied to feasible yields feasible," we should accept the right-hand side of (5) as a feasible bound in terms of F and $|x|$.

This argument obtained the type-3 bound by decomposing the problem into proving a type-2 polynomial-bound in step 1, proving a polynomial-bound in step 2, and knitting together these two bounds in step 3. The definition of $\|\cdot\|_r$ was crafted to permit this sort of decomposition. In contrast, the definition of $|\cdot|_b$ brooks no such decomposition and $|F|_b$ demands to be treated as a type-2 object. To justify calling the right-hand side of (4) a "polynomial bound" in $|F|_b$ and $|x|$, we need to extend our notion of polynomial to type-3 and beyond. We attend to this in Section 8. In Section 9 we then apply some of the ideas from Section 8 and develop a proper formal framework for arguments such as the one just sketched for (5). $\diamond$

8. Higher Type Polynomials

Our general stance is that before talking about higher-type polynomial-time, we should have in hand a notion of higher-type polynomial. In particular, we want to justify calling an expression such as $|F|_b\,(\lambda n.|x|) \cdot |x|$, the right-hand side of (4), a "polynomial" in $|F|_b$ and $|x|$. The notion of higher-type polynomial developed below is an extension of Kapron and Cook's [KC96] notion of *second-order polynomials* that was discussed in section I-6. Briefly, second-order polynomials have the syntax given by

$$\mathrm{P} \;\;::=\;\; C \mid V_0 \mid \mathrm{P} + \mathrm{P} \mid \mathrm{P} \cdot \mathrm{P} \mid V_1(\mathrm{P})$$

(where C, V_0, and V_1 are respectively the syntactic categories of constants of ω, type-ω variables, type-$(\omega \to \omega)$ variables) and their semantics is the obvious, straightforward thing. Generalizing second-order polynomials to type-level three and above requires: (i) variables of each simple type and (ii) abstractions over (higher type) polynomials. The following then is the simplest formalism that might meet our needs.

Definition 15 (Syntax). The formalism HTP (for *higher-type polynomial*) consists of the following sorts of expressions with the indicated type assignments.

(a) For each $n \in \omega$, HTP contains the numeral $\overline{n}$ of type ω.

(b) HTP contains the constant symbols $+$ and $\cdot$ of type $\omega^2 \to \omega$.

(c) For each simple type over ω, HTP has a countably infinite collection of variables of that type.

(d) If s is a term of type $\tau_1 \times \cdots \times \tau_n \to \omega$ and $t_1, \dots, t_n$ are terms of type $\tau_1, \dots, \tau_n$, respectively, then $(s\, t_1 \, \dots \, t_n)$ is a term of type ω.

(e) If t is a term of type ω and $y_1, \dots, y_n$ are variables of type $\tau_1, \dots, \tau_n$ respectively, then $(\lambda y_1, \dots, y_n \,\textbf{.}\, t)$ is a term of type $\tau_1 \times \ldots \times \tau_n \to \omega$. $\Diamond$

HTP is just the simply-typed lambda calculus over base type ω with constant type-$(\omega^2 \to \omega)$ symbols $+$ and $\cdot$ thrown in.[6] In writing HTP expressions, we typically take the usual liberties of dropping unnecessary parentheses, using infix forms of $+$ and $\cdot$, writing $x(t_1, \dots, t_n)$ for the application $(x\, t_1 \dots t_n)$ when x is a variable, writing n in place of $\overline{n}$, and writing t^k for the k-fold multiplication of type-ω term t. When we refer to the normal form of an HTP term, we shall mean the $\beta\eta$-normal form.

Definition 16 (Semantics). The intended semantics of HTP is given by:

(a) For each $n \in \omega$, $\overline{n}$ corresponds to n.

(b) '$+$' corresponds to addition over ω and likewise '$\cdot$' to multiplication.

(c) Each variable of type τ ranges over TLen_τ.

(d) Application and abstraction correspond to the obvious operations in the category **TLen**. $\Diamond$

When there is little chance of confusion, we usually write $[\![\cdot]\!]$ in place of $[\![\cdot]\!]_{\mathbf{TLen}}$ in this section. Since **TLen** is a cartesian closed category (Lemma 4) and since addition and multiplication are elements of $\mathrm{TLen}_{\omega^2\to\omega}$ we have:

Lemma 17 (Soundness). *Under the intended semantics, each type τ term of HTP denotes an element of TLen_τ.*

The HTPs turn out to be a conservative extension of ordinary polynomials.

Lemma 18. *Suppose t is a type ω term of HTP such that its only free variables are of type ω. Then t is $\beta\eta$-equivalent to an ordinary polynomial, i.e., a term built up from numerals and type ω variables by applications of $+$ and $\cdot$.*

[6] For a much deeper connection between polynomials and the simply-typed lambda calculus, see Schwichtenberg's [Sch76].

Proof. Without loss of generality, take t to be in normal form. The argument is by induction on the height of the (parse tree of) t. Suppose t's height is 0. Then t is either a numeral or a type-ω variable. So in this case the lemma clearly holds. Now suppose t's height is $n > 0$. Then it follows that the subterm in the head position must be either $+$ or $\cdot$, and the subterms in the argument positions are normal form terms of height less than n. By the induction hypothesis, both subterms are ordinary polynomials. So the lemma holds in this case, too. □

A slight modification of the above argument yields that the HTPs are also a conservative extension of second-order polynomials. That is:

Lemma 19. *Suppose t is a type ω term of* HTP *such that its only free variables are of types ω and $\omega \to \omega$. Then t is $\beta\eta$-equivalent to a second-order polynomial, i.e., a term built up from numerals and type ω variables by applications of $+$, $\cdot$, and the type $\omega \to \omega$ variables.*

Kapron and Cook [KC96] define the *depth* of a second-order polynomial to be the maximum depth of nesting of applications of the type-1 variables in the polynomial (Definition I-6.) We generalize this notion to HTP's as follows.

Definition 20. For each HTP term t in normal form, define

$$depth(t) \quad = \quad \begin{cases} 0, & \text{if } t \text{ is a constant or a variable;} \\ depth(t_1), & \text{if } t = \lambda\vec{x}\,\textbf{.}\,t_1; \\ depth(t_1) \oplus depth(t_2), & \text{if } t = +(t_1, t_2) \text{ or } t = \cdot(t_1, t_2); \\ 1 + \bigoplus_{i=1}^{n} depth(t_i), & \text{if } t = v(t_1, \ldots, t_n) \text{ for some} \\ & \quad \text{variable } v. \end{cases}$$

For each HTP term t that is not in normal form, we define $depth(t) = depth(t')$ where t' is a normal form of t. ◇

Clearly, α-equivalent normal forms have the same depth. Hence $depth(\cdot)$ is well-defined on non-normal forms. We note that if t is a second-order polynomial, then $depth(t)$ agrees with the notion of depth from Definition I-6.

In Section 10 we shall need an estimate of how the depth of an HTP can change under substitutions. More precisely, given HTP terms t, $t_1, \ldots, t_n$ with $a = depth(t)$, $b = \bigoplus_{i=1}^{n} depth(t_i)$, and $\ell =$ the maximum type-level of the types assigned to the t_i's, we want to give an upper bound on the depth of

$t[x_1,\dots,x_n \mapsto t_1,\dots,t_n]$ in terms of a, b, and ℓ. To do this we make use of the following family of functions. For each a, b, and ℓ, define:

$$\begin{aligned} e_0(a,b) &= a+b. \\ e_{\ell+1}(0,b) &= b. \\ e_{\ell+1}(a+1,b) &= (\,1+e_{\ell+1}(a,b)\,) \oplus e_\ell(b, e_{\ell+1}(a,b)). \end{aligned}$$

We note that $e_\ell(a,b)$ is monotone nondecreasing in a, b, and ℓ and that, for all a and ℓ, $e_\ell(a,0)=a$. Also, for all a and all $b>0$, $e_1(a,b)=(a+1)\cdot b$, $e_2(a,b)=(b+1)^a\cdot b$, and $b^{b^a} \le e_3(a,b) \le (b+1)^{(b+1)^a}$. Moreover, e_4 fails to be elementary recursive and, while each e_ℓ is primitive recursive, $\lambda a,b,\ell\,\boldsymbol{.}\,e_\ell(a,b)$ is not.

Lemma 21. *Let t, $t_1,\dots,t_n$ be* HTP*-terms and $s=t[x_1,\dots,x_n \mapsto t_1,\dots,t_n]$. Suppose that $a = depth(t)$, $b=\bigoplus_{i=1}^n depth(t_i)$, and $\ell =$ the maximum type-level of the types assigned to the t_i's. Then, $depth(s) \le e_\ell(a,b)$.*

Proof. Without loss of generality we assume that each of t, $t_1,\dots,t_n$ is in normal form. We proceed by induction on ℓ and the structure of t.

Suppose $\ell=0$. By our assumption that t is in normal form, it follows that Cases 1 through 5 below exhaust the possibilities for the structure of t.

Case 1: t is a constant. Then $depth(t)=depth(s)=0\le a+b$.

Case 2: t is a variable. Then $depth(t)=0$ and the substitution either leaves t unchanged or replaces t with a term of depth at most b. Thus, $depth(s)\le b=a+b$.

Case 3: $t=\lambda\vec{y}\,\boldsymbol{.}\,t'$. Then by the definition of $depth(\cdot)$, $depth(t)=depth(t')$. By the definition of substitution, we have that $depth(s)\le depth(t'[x_1,\dots, x_n \mapsto t_1,\dots,t_n]$. By the induction hypothesis on t, we know $depth(t'[x_1,\dots, x_n\mapsto t_1,\dots,t_n]\le a+b$. Thus, it follows that $depth(s)\le a+b$.

Case 4: $t=+(t_1,t_2)$ or $t=\cdot(t_1,t_1)$. Then, by the definition of $depth(\cdot)$, we have that $depth(t)=depth(t_1)\oplus depth(t_2)$ and $depth(s)=depth(t_1[x_1,\dots,x_n\mapsto t_1,\dots,t_n])\oplus depth(t_2[x_1,\dots,x_n\mapsto t_1,\dots,t_n])$. So it follows from the induction hypothesis on t that $depth(s)\le(a+b)\oplus(a+b)=a+b$.

Case 5: $t=v(t'_1,\dots,t'_m)$ where v is a variable. For $i=1,\dots,m$, let $s_i=t'_i[x_1,\dots,x_n\mapsto t_1,\dots,t_n]$. By the definition of $depth(\cdot)$ we know that $a>0$ and that, for $i=1,\dots,m$, $depth(t_i)\le a-1$. By the induction hypotheses on t we have that $\bigoplus_{i=1}^m depth(s_i)\le(a-1)+b$. Now, since the level of the type of v is at least 1 and since the level of the types of $t_1,\dots,t_n$

are at most $\ell = 0$, it follows that $s = v(s_1, \ldots, s_m)$. Thus, by the definition of $\mathit{depth}(\cdot)$, $\mathit{depth}(s) \leq 1 + (a-1) + b = a + b$.

Therefore, the lemma holds for all $t,\ t_1, \ldots, t_n$ when $\ell = 0$.

Suppose the lemma holds for all $t,\ t_1, \ldots, t_n$ when the maximum type-level of the types of the t_i's is $\leq \ell$. Take $t,\ t_1, \ldots, t_n$ such that $\ell+1$ is the maximum type-level of the types of the t_i's. Then we have the following cases based on the structure of t.

Cases 1 through 4 as above. Simple modifications of the above arguments show that in these cases we have that $\mathit{depth}(s) \leq e_{\ell+1}(a, b)$.

Case 5: $t = v(t'_1, \ldots, t'_m)$ where v is a variable. For $i = 1, \ldots, m$, let $s_i = t'_i[x_1, \ldots, x_n \mapsto t_1, \ldots, t_n]$ as before. By the definition of $\mathit{depth}(\cdot)$ we know that $a = \mathit{depth}(t) > 0$ and that, for $i = 1, \ldots, m$, $\mathit{depth}(t_i) \leq a - 1$. By the induction hypothesis on t, we have that, for $i = 1, \ldots, m$, $\mathit{depth}(s_i) \leq e_{\ell+1}(a-1, b)$. We have two subcases based on how the substitution affects v.

Subcase 5a: $\mathit{depth}(v[x_1, \ldots, x_n \mapsto t_1, \ldots, t_n]) = 0$. Then it is straightforward that $\mathit{depth}(s) \leq 1 + \bigoplus_{i=1}^m \mathit{depth}(s_i) \leq 1 + e_{\ell+1}(a-1, b) \leq e_{\ell+1}(a, b)$.

Subcase 5b: $\mathit{depth}(v[x_1, \ldots, x_n \mapsto t_1, \ldots, t_n]) > 0$. Since $t,\ t_1, \ldots, t_n$ are all in normal form, it follows that we must have that $v[x_1, \ldots, x_n \mapsto t_1, \ldots, t_n] = \lambda y_1, \ldots, y_m \,.\, t'$ and $s = t'[y_1, \ldots, y_m \mapsto s_1, \ldots, s_m]$, where $\mathit{depth}(t') \leq b$ and the type-level of the types of the s_i's is no greater than ℓ. Thus by the induction hypothesis on ℓ we have that $\mathit{depth}(s) \leq \bigoplus_{\ell'=0}^{\ell} e_{\ell'}(b, e_{\ell+1}(a-1, b)) = e_\ell(b, e_{\ell+1}(a-1, b)) \leq e_{\ell+1}(a, b)$.

Therefore, the $\ell + 1$ case follows. □

9. Polynomial and Seth Bounds

We have unfinished business from Example 14: explaining how the right-hand side of

$$|T'(F, x)| \quad \leq \quad |F|_b\,(\lambda n \,.\, |x|) \cdot |x| \tag{6}$$

expresses a feasible/polynomial bound in terms of $|F|_b$ and $|x|$. The framework of the previous section makes this easy: $|F|_b\,(\lambda n \,.\, |x|) \cdot |x|$ is an HTP over $|F|_b$ and $|x|$, so we are done. In this section we construct an analogous framework for the sets of bounds such as the following from Example 14:

$$|T'(F, x)| \quad \leq \quad [\![\|\mathtt{F}(\lambda\mathtt{y}.\mathtt{len}(\mathtt{x}))\|_r \cdot |\mathtt{x}|_r]\!]_{\mathbf{FL}}\, |\rho|_r \tag{7}$$

$$[\![|\mathtt{len}(\mathtt{x})|_r]\!]_{\mathbf{FL}}\, |\rho|_r \quad \leq \quad [\![|\mathtt{x}|_r]\!]_{\mathbf{FL}}\, |\rho|_r \tag{8}$$

where $\rho = \{\, \mathtt{F} \mapsto F, \mathtt{len} \mapsto \ell en, \mathtt{x} \mapsto x \,\}$, F and x are arbitrary members of $\text{Full}_{(\text{N}\to\text{N})\to\text{N}}$ and N, respectively, and ℓen is as in Example 13. We call these sorts of bounds *Seth bounds.* (Note: We will end up expressing these bounds somewhat differently.) Working with Seth bounds is more awkward than with the polynomial bounds for two reasons. First, they typically consist of a system of related bounds rather than a single, tidy bound. Second, and more seriously, they tangle bounds and things bounded, e.g., the right-hand side of (7) contains a mixture of syntactic and semantic terms (even when the $\|\cdot\|_r$ is fully expanded) in contrast to the right-hand side of (6) which can be reasonably thought of as containing only semantic terms. Before dealing with this tangling, we first introduce a formalization of HTP bounds over certain lengths that involves a tangling of a milder, but related, sort.

Terminology: In inequalities of either the forms '$|E|_b \leq \mathbf{b}$' or '$|E|_r \leq \mathbf{b}$' we shall call E the *subject* expression (since it is the subject of the bound) and $\mathbf{b}$ the *bounding* expression.

9.1. Higher-Type Polynomial Bounds and Their Inference

Our interest in HTPs stems from their use in expressing and deriving bounds on the values of subject expressions involving higher-type terms. Here we sketch the barest beginnings of a formal system for deriving such bounds. While our actual use of HTP bounds will be informal in subsequent sections, the formal system will help justify these informal uses and motivate the formal system for Seth bounds, which plays a more important role in its setting.

Below we shall build upon the definitions and conventions of Section 7.1. Given $\mathtt{x}$, a variable of type σ, we will often treat $|\mathtt{x}|_b$ as a variable of type $|\sigma|_b$. We first formalize the sorts of statements our formal system is supposed to establish and relate such statements to the underlying semantics.

Definition 22. Suppose $\Gamma = \{\, \mathtt{x}_1{:}\,\sigma_1, \ldots, \mathtt{x}_n{:}\,\sigma_n \,\}$, $\rho = \{\, \mathtt{x}_1 \mapsto x_1, \ldots, \mathtt{x}_n \mapsto x_n \,\} \in [\![\Gamma]\!]_{\mathbf{BCont}}$, and E is a subject expression.

(a) Define $|\Gamma|_b = \{\, |\mathtt{x}_1|_b : |\sigma_1|_b \,, \ldots, |\mathtt{x}_n|_b : |\sigma_n|_b \,\}$.

(b) Define $|\rho|_b = \{\, |\mathtt{x}_1|_b \mapsto |x_1|_b \,, \ldots, |\mathtt{x}_n|_b \mapsto |x_n|_b \,\} \in [\![|\Gamma|_b]\!]_{\mathbf{TLen}}$.

(c) We say that an HTP $\mathbf{q}$ is a *bound with respect to* Γ if and only if $FV(\mathbf{q}) \subseteq \text{Subjects}(|\Gamma|_b)$.

(d) We write $\Gamma \overset{\sigma}{\vdash} |E|_b \leq \mathbf{q}$ if and only if $\Gamma \vdash E{:}\,\sigma$ and $|\Gamma|_b \vdash \mathbf{q}{:}\,|\sigma|_b$. We refer to $\Gamma \overset{\sigma}{\vdash} |E|_b \leq \mathbf{q}$ as a $|\cdot|_b$*-bounding judgement* or, usually, simply a *bounding judgement.*

(e) We say that ρ *satisfies* $\Gamma \vdash^{\sigma} |E|_b \leq \mathbf{q}$ if and only if $\big|[\![E]\!]_{\mathbf{BCont}}\,\rho\big|_b \leq [\![\mathbf{q}]\!]_{\mathbf{TLen}}\,|\rho|_b$. ◇

To establish bounding judgements we need axioms, rules of inference, and the soundness of said axioms and rules. We shall ignore the issue of axioms and consider only the following two key rules and their soundness.

Rule 23 (Abstraction). Let $\sigma_0 = \sigma_1 \times \cdots \times \sigma_k \to \mathrm{N}$ and $\Upsilon = \{\,\mathrm{y}_1{:}\,\sigma_1, \ldots, \mathrm{y}_k{:}\,\sigma_k\,\} \subseteq \Gamma$. Then:

$$\frac{\Gamma \vdash^{\mathrm{N}} |E|_b \leq \mathbf{q}}{(\Gamma - \Upsilon) \vdash^{\sigma_0} |\lambda \mathrm{y}_1, \ldots, \mathrm{y}_k \boldsymbol{.} E|_b \leq \lambda\, |\mathrm{y}_1|_b\,, \ldots, |\mathrm{y}_k|_b \boldsymbol{.} \mathbf{q}}.$$

◇

Rule 24 (Application). Let $\sigma_0 = \sigma_1 \times \cdots \times \sigma_k \to \mathrm{N}$, $E =_{\beta\eta} (E_0\, E_1 \ldots E_k)$, and $\mathbf{q} =_{\beta\eta} (\mathbf{q}_0\, \mathbf{q}_1 \ldots \mathbf{q}_k)$. Then:

$$\frac{\Gamma_0 \vdash^{\sigma_0} |E_0|_b \leq \mathbf{q}_0 \quad \ldots \quad \Gamma_k \vdash^{\sigma_k} |E_k|_b \leq \mathbf{q}_k}{(\cup_{i \leq k} \Gamma_i) \vdash^{\mathrm{N}} |E|_b \leq \mathbf{q}}$$

provided $\cup_{i \leq k} \Gamma_i$ is consistent. ◇

Lemma 25 (Soundness). *Rules 23 and 24 are sound with respect to the* **BCont/TLen**-*semantics.*

Proof. This follows from Lemma 5 and the monotonicity of HTPs. □

This is all we really need to formalize about deriving HTP bounds. Rules 23 and 24, in more informal guises, are basic tools in our derivations of HTP bounds in this paper.

9.2. Seth Bounds and Their Inference

We now formalize analogues of HTP bounds and their derivations for the **Full/FL** setting. They are only rough analogues because of some fundamental differences between the two settings that will soon be apparent. Below we shall build upon the definitions and conventions of Section 7.2.

We first formalize the "algebraic closure" of the $\|\cdot\|_r$-expressions. Here is an example that illustrates the need for such a thing. Suppose that the call

$\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\mathtt{y}),\mathtt{z})$ occurs in some BTLP program and that ρ is an environment that includes bindings for F, g, w, y, and z. Then,

$$\begin{aligned} &\big|[\![\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\mathtt{y}),\mathtt{z})]\!]_{\mathbf{Full}}\,\rho\,\big| \\ &\quad\leq\ [\![\, \|\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\mathtt{y}),\mathtt{z})\|_r \,]\!]_{\mathbf{FL}}\ |\rho|_r && \text{(by Lemma 11(b))} \\ &\quad=\ [\![\, (\textstyle\bigoplus \hat{\mathtt{y}}{\upharpoonright}|\mathtt{y}|,\ \hat{\mathtt{z}}{\upharpoonright}|\mathtt{z}|)\,|\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\hat{\mathtt{y}}),\hat{\mathtt{z}})|\,]\!]_{\mathbf{FL}}\ |\rho|_r && \text{(by Definition 10(c)).} \end{aligned}$$

This is all old news. But now suppose that we know that the bounds $|\mathtt{y}| \leq 4|\mathtt{w}|^2$ and $|\mathtt{z}| \leq |\mathtt{w}|^3$ hold whenever the call is made and we want to express a bound on $\big|[\![\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\mathtt{y}),\mathtt{z})]\!]_{\mathbf{Full}}\,\rho\,\big|$ in terms of F, g, and w. It is easily seen that these three bounds can be composed to obtain

$$\big|[\![\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\mathtt{y}),\mathtt{z})]\!]_{\mathbf{Full}}\,\rho\,\big| \quad\leq\quad [\![\, \big(\textstyle\bigoplus \hat{\mathtt{y}}{\upharpoonright}4|\mathtt{w}|^2,\ \mathtt{z}{\upharpoonright}|\mathtt{w}|^3\big)\,|\mathtt{F}(\lambda\mathtt{x}.\mathtt{g}(\mathtt{x},\hat{\mathtt{y}}),\hat{\mathtt{z}})|\,]\!]_{\mathbf{FL}}\,|\rho|_r\,.$$

Formalizing such bounds thus requires formalizing the closure the $\|\cdot\|_r$-expressions under arithmetic and λ-calculus operations. This is done in:

Definition 26 (Basic bounds and their types).

(a) Suppose $\Gamma = \{\,\mathtt{x}_1{:}\,\tau_1,\dots,\mathtt{x}_j{:}\,\tau_j,\ \mathtt{x}_{j+1}{:}\,\mathrm{N},\dots,\mathtt{x}_k{:}\,\mathrm{N}\,\}$. The set of *basic bounds* with respect to Γ, denoted $\mathbf{B}_\Gamma$, is the smallest collection of expressions such that the following hold. For each $i \leq k$, suppose $\mathbf{b}_i \in \mathbf{B}_\Gamma$ and $|\Gamma|_r \vdash \mathbf{b}_i{:}\,|\sigma_i|_r$.

1. If $\Gamma \vdash E{:}\,\sigma$ for some σ, then $\|E\|_r \in \mathbf{B}_\Gamma$ and we write $|\Gamma|_r \vdash \|E\|_r : |\sigma|_r$.
2. If $|\sigma_0|_r = |\sigma_1|_r = \omega$, then $\mathbf{b}_0 + \mathbf{b}_1$ and $\mathbf{b}_0 \cdot \mathbf{b}_1 \in \mathbf{B}_\Gamma$ and we write $|\Gamma|_r \vdash \mathbf{b}_0 + \mathbf{b}_1{:}\,\omega$ and $|\Gamma|_r \vdash \mathbf{b}_0 \cdot \mathbf{b}_1{:}\,\omega$.
3. If $|\sigma_0|_r = \omega$ and $\Upsilon = \{\,\mathtt{y}_1{:}\,\tau_1,\dots,\mathtt{y}_j{:}\,\tau_j,\ \mathtt{z}_{j+1}{:}\,\mathrm{N},\dots,\mathtt{z}_k{:}\,\mathrm{N}\,\} \subseteq \Gamma$, then $\mathbf{b} = \lambda\mathtt{y}_1,\dots,\mathtt{y}_j,|\mathtt{z}_{j+1}|,\dots,|\mathtt{z}_k|.\mathbf{b}_0 \in \mathbf{B}_{\Gamma-\Upsilon}$ and we write $|\Gamma-\Upsilon|_r \vdash \mathbf{b}{:}\,\tau_1 \times\cdots\times \tau_j \times \omega^{k-j} \to \omega$.
4. If $|\sigma_0|_r = \tau_1 \times\cdots\times \tau_j \times \omega^{k-j} \to \omega$, $\ \Gamma \vdash E_i{:}\,\tau_i$ (for $i = 1,\dots,j$), and $|\sigma_{j+1}|_r = \cdots |\sigma_k|_r = \omega$, then for $\mathbf{b} =_{\beta\eta} (\mathbf{b}_0\, E_1\, \cdots\, E_j\, \mathbf{b}_{j+1} \cdots \mathbf{b}_k)$ (i.e., $\mathbf{b}$ is $\beta\eta$-equivalent to the application of $\mathbf{b}_0$ to $E_1,\dots,E_j,\mathbf{b}_{j+1},\dots,\mathbf{b}_k$) we have $\mathbf{b} \in \mathbf{B}_\Gamma$ and we write $|\Gamma|_r \vdash \mathbf{b}{:}\,\omega$.

(b) For each basic bound $\mathbf{b}$, the set of *free base variables* of $\mathbf{b}$, written $FV_*(\mathbf{b})$, is $\{\,\mathtt{x} \mid \mathtt{x} \in FV(\mathbf{b}) \text{ or } |\mathtt{x}| \in FV(\mathbf{b})\,\}$.

(c) We write $\Gamma \overset{\sigma}{\vdash} |E|_r \leq \mathbf{b}$ if and only if $\Gamma \vdash E{:}\,\sigma$ and $|\Gamma|_r \vdash \mathbf{b}{:}\,|\sigma|_r$. We refer to $\Gamma \overset{\sigma}{\vdash} |E|_r \leq \mathbf{b}$ as a *basic* $|\cdot|_b$*-bounding judgement* or, usually, simply as a *basic bounding judgement*.

(d) We say that $\rho \in [\![|\Gamma|_r]\!]_{\mathrm{FL}}$ *satisfies* $\Gamma \vdash^{\sigma} |E|_r \leq \mathbf{b}$ if and only if $\big|[\![E]\!]_{\mathbf{Full}}\,\rho\big|_r \leq [\![\mathbf{b}]\!]_{\mathbf{FL}}\ |\rho|_r$.

(e) Suppose $\Gamma \vdash^{\sigma} |\mathtt{x}|_r \leq \mathbf{b}$ for some variable $\mathtt{x}$. We say that $\mathbf{b}$ is a *closed bound* on $|\mathtt{x}|_r$ if and only if $|\mathtt{x}|_r \notin FV(\mathbf{b})$. ◇

Basic bounds are clearly another variant of the simply-typed lambda calculus. Their normal forms are standard except that they can include type-ω subterms of the form $(\bigoplus \mathtt{y}_1{\restriction}\mathbf{b}_1, \ldots, \mathtt{y}_\ell{\restriction}\mathbf{b}_\ell)\,|E|$, where $\mathbf{b}_1, \ldots, \mathbf{b}_\ell$ are type-ω basic bounds in normal form and E is a normal-form subject expression.

As with HTP-bounds, to formally establish basic bounding judgements we need axioms, rules of inference, and their soundness. Again we shall focus simply on the rules for abstraction and application and their soundness.

Rule 27 (Abstraction). Let $\tau_0 = \tau_1 \times \cdots \times \tau_j \times \mathrm{N}^{k-j} \to \mathrm{N}$, and let $\Upsilon = \{\, \mathtt{y}_1{:}\,\tau_1, \ldots, \mathtt{y}_j{:}\,\tau_j,\ \mathtt{z}_{j+1}{:}\,\mathrm{N}, \ldots, \mathtt{z}_k{:}\,\mathrm{N}\,\} \subseteq \Gamma$. Then:

$$\frac{\Gamma \vdash^{\mathrm{N}} |E|_r \leq \mathbf{b}}{(\Gamma - \Upsilon) \vdash^{\tau_0} |\lambda\vec{\mathtt{y}},\vec{\mathtt{z}}\,\boldsymbol{.}\,E|_r \leq \lambda\vec{\mathtt{y}}, |\mathtt{z}_{j+1}|, \ldots, |\mathtt{z}_k|\,\boldsymbol{.}\,\mathbf{b}}$$

◇

Rule 28 (Application). Let $\tau_0 = \tau_1 \times \cdots \times \tau_j \times \mathrm{N}^{k-j} \to \mathrm{N}$. For $i = 1, \ldots, j$, let $\mathcal{T}_i$ denote the typing judgement $\Gamma_i \vdash E_i{:}\,\tau_i$. For $i = j{+}1, \ldots, k$, let $\mathcal{B}_i$ denote the bounding judgement $\Gamma_i \vdash^{\mathrm{N}} |E_i|_r \leq \mathbf{b}_i$. Also, let $E \ =_{\beta\eta}\ (E_0\, E_1\, \cdots\, E_k)$ and $\mathbf{b} \ =_{\beta\eta}\ (\mathbf{b}_0\, E_1\, \cdots\, E_j\, \mathbf{b}_{j+1}\, \cdots\, \mathbf{b}_k)$. Then:

$$\frac{\Gamma_0 \vdash^{\tau_0} |E_0|_r \leq \mathbf{b}_0 \quad \mathcal{T}_1\ \ldots\ \mathcal{T}_j \quad \mathcal{B}_{j+1}\ \ldots \mathcal{B}_k}{(\cup_{i\leq k}\Gamma_i) \vdash^{\mathrm{N}} |E|_r \leq \mathbf{b}}$$

provided that $\cup_{i\leq k}\Gamma_i$ is consistent. ◇

Lemma 29 (Soundness). *Rules 27 and 28 are sound with respect to the* **Full**/**FL**-*semantics.*

Proof. This follows from Lemma 12 and the fact that basic bounds are monotone with respect to their type ω-arguments. □

Example 14 illustrated that to fully express bounds in the **Full**/**FL** setting we have to employ systems of related bounds. We can re-express (7) and (8), the bounds from Example 14, as the basic bounding judgements:

$$\Gamma \ \vdash^{(\mathrm{N}\to\mathrm{N})\times\mathrm{N}\to\mathrm{N}}\ |\mathtt{T}'|_r \ \leq\ \lambda\mathtt{F}, |\mathtt{x}|\,\boldsymbol{.}\,\big(\,(\bigoplus \hat{\mathtt{x}}{\restriction}|\mathtt{x}|)\,\big|\mathtt{F}(\lambda\mathtt{y}.\mathtt{len}(\hat{\mathtt{x}}))\big| \cdot |\mathtt{x}|_r\,\big) \qquad (9)$$

$$\Gamma \ \vdash^{\mathrm{N}\to\mathrm{N}}\ |\mathtt{len}|_r \ \leq\ \lambda\,|\mathtt{x}|_r\,\boldsymbol{.}\,|\mathtt{x}|_r \qquad (10)$$

where $\Gamma = \{\,\mathtt{T}'\colon (\mathrm{N}\to\mathrm{N})\times\mathrm{N}\to\mathrm{N}, \mathtt{len}\colon \mathrm{N}\to\mathrm{N}\,\}$. Part of the interpretation we wish to place on the system (9)+(10) is that `len` is a "second class" free variable of the right-hand side of (9). The intuition is this: The semantics of that expression obviously depends on the value of `len`. However, we can discount `len` because a primary parameter in the "polynomial" bound on $|\mathtt{T}'|_r$ as $|\mathtt{len}|_r$ has an independent, closed basic bound given by (10). Let us give this notion a name. Suppose that a variable $\mathtt{x}$ occurs free someplace in a system of basic bounds. We say that x is a *confined* variable provided there is a closed basic bound on $|x|_r$ in the system, and otherwise we call $\mathtt{x}$ a *primary* variable. The intuition is that confined variables are "polynomially" bounded in terms of the primary variables. In order for this to make sense, we need to impose some conditions on the allowable systems of basic bounds. To see why, consider the system

$$\Gamma \vdash^{\mathrm{N}\to\mathrm{N}} |\mathtt{g}|_r \le |\mathtt{h}|_r \qquad \Gamma \vdash^{\mathrm{N}\to\mathrm{N}} |\mathtt{h}|_r \le |\mathtt{g}|_r$$

where $\Gamma = \{\,\mathtt{g}\colon \mathrm{N}\to\mathrm{N}, \mathtt{h}\colon \mathrm{N}\to\mathrm{N}\,\}$. Both $|\mathtt{g}|_r$ and $|\mathtt{h}|_r$ have closed basic bounds, so both $|\mathtt{g}|_r$ and $|\mathtt{h}|_r$ are officially confined. Moreover, since there are no primary variables here, one would like to have that $|\mathtt{g}|_r$ and $|\mathtt{h}|_r$ are bounded in some strong sense. However, that certainly is *not* the case here. We shall thus impose a well-foundedness condition on systems of basic bounds to avoid such circularities.

Definition 30 (Seth bounds). Suppose $\mathcal{S} = \{\,\Gamma \vdash^{\sigma_0} |E_0|_r \le \mathbf{b}_0, , \dots, \Gamma \vdash^{\sigma_k} |E_k|_r \le \mathbf{b}_k\,\}$ is a collection of basic bounding judgements and let $FV_*(\mathcal{S}) = \bigcup_{i\le k} FV_*(\mathbf{b}_i)$.

(a) An $\mathtt{x} \in FV_*(\mathcal{S})$ is *confined* if and only if $|\mathtt{x}|_r$ has a closed basic bound in $\mathcal{S}$. An $\mathtt{x} \in FV_*(\mathcal{S})$ that is not confined is said to be *primary.*

(b) We say that $\Gamma \vdash^{\sigma_i} |E_i|_r \le \mathbf{b}_i$ *depends* on $\Gamma \vdash^{\sigma_j} |E_j|_r \le \mathbf{b}_j$ if and only if (i) E_j is a variable $\mathtt{x}$, (ii) $\mathbf{b}_j$ is a closed bound on $|\mathtt{x}|_r$, and (iii) $\mathtt{x}$ or $|\mathtt{x}|_r$ occurs free in $\mathbf{b}_i$.

(c) We say $\mathcal{S}$ is a *Seth bound* (with respect to Γ) if and only if (i) no $\mathtt{x} \in FV_*(\mathcal{S})$ has more than one closed bound in $\mathcal{S}$, and (ii) the transitive closure of the dependency relation on $\mathcal{S}$ has no cycles.

(d) We say that $\rho \in [\![|\Gamma|_r]\!]_{\mathrm{FL}}$ *satisfies* $\mathcal{S}$ if and only if ρ satisfies each element of $\mathcal{S}$. ◇

Inferring a Seth bound $\mathcal{S}$ amounts to inferring each element of $\mathcal{S}$.

9.3. Seth Bounds and Depth

As with higher-type polynomials, we wish to generalize the notion of the depth of a second-order polynomial to Seth bounds. Seth bounds are another variant of the simply-typed lambda calculus (although a somewhat contorted one). Thus, the same principles used in the previous section to define the depth of HTPs apply here also: we want to determine the maximum depth of nesting of applications in a "normal form" of the bound. Two things complicate matters here. First, through $\bigoplus$-expressions, terms from the language of subject expressions can end up inside bounding expressions. Hence, our notion of depth has to apply to both bounding and subject expressions. Second, a Seth bound is made up of a system of related basic bounds; consequently, we end up assigning a depth to each variable and compound expression occurring in the Seth bound. Moreover, this assigned depth is only an upper bound on how deep the nesting can be. (See Example 32 below.) Here are the details of how we assign depth in Seth bounds.

Definition 31. Let $\mathcal{S} = \{\, \Gamma \vdash^{\sigma_0} |E_0|_r \le \mathbf{b}_0, \dots, \Gamma \vdash^{\sigma_k} |E_k|_r \le \mathbf{b}_k \,\}$ be a Seth bound. Suppose for each i and j with $0 \le i < j \le k$ we have that $\Gamma \vdash^{\sigma_i} |E_i|_r \le \mathbf{b}_i$ does *not* depend on $\Gamma \vdash^{\sigma_j} |E_j|_r \le \mathbf{b}_j$. (There must be such an ordering by the well-foundedness condition on Seth bounds.) We assume that each subject and bounding expression in $\mathcal{S}$ is in $\beta\eta$-normal form. We will assign a depth to each variable in $FV_*(\Gamma)$ and each basic bound in $\mathcal{S}$. Each primary variable is assigned depth 0 and each confined variable $\mathtt{x}$ will be assigned the depth of the closed bound on $|\mathtt{x}|_r$. Suppose we have assigned depths to the basic bounds of each of $\Gamma \vdash^{\sigma_0} |E_0|_r \le \mathbf{b}_0, \dots, \Gamma \vdash^{\sigma_{i-1}} |E_{i-1}|_r \le \mathbf{b}_{i-1}$. Let $\Gamma_i = \{\, x_1 \colon \tau_1, \dots, x_m \colon \tau_m \,\}$ be the set of free base variables occurring in $\Gamma \vdash^{\sigma_i} |E_i|_r \le \mathbf{b}_i$. By our hypotheses, each $\mathtt{x}_i$ has already been assigned a depth, say d_i. Relative to this assignment of types and depths, a (normal form) subject expression, E, is inductively assigned a depth as follows.

Case: E is a constant. Then the depth of E is 0.

Case: E is a variable $\mathtt{x}$. Then the depth of E is the same depth as $\mathtt{x}$.

Case: $E = E_0 \odot E_1$ (where $\odot$ is one of $+$, $\dot{-}$, or $\#$). Then the depth of E is the maximum of the depths of E_0 and E_1.

Case: $E = \lambda \mathtt{v}_0, \dots, \mathtt{v}_n \boldsymbol{.} E_0$ and $\Gamma_i \vdash E \colon \tau_1 \times \cdots \times \tau_n \to \mathrm{N}$. Then the depth of E is the same as the depth of E_0 relative to the assignment of types $\Gamma_i \cup \{\, \mathtt{v}_1 \colon \tau_1, \dots, \mathtt{v}_n \colon \tau_n \,\}$, and the assignment of depths in which each $\mathtt{x}_j$ is as before and each $\mathtt{v}_j$ is assigned depth 0.

CASE: $E = \mathtt{x}_0(E_1, \ldots, E_n)$, d_0 is the depth assigned to $\mathtt{x}_0$, d is the maximum of the depths assigned to the E_i's, and ℓ is the maximum level of the types of the E_i's. Then we have the following two subcases.

SUBCASE: $d_0 = 0$. Then the depth of E is $d + 1$.

SUBCASE: $d_0 > 0$. Then the depth of E is $e_\ell(d_0, d)$.

Relative to the same assignments of types and depths, a (normal form) basic bound, $\mathbf{b}$, is inductively assigned a depth along the lines of our assignment of depth to subject expressions. If $\mathtt{x} \in FV_*(\mathcal{S})$, then the variable $|\mathtt{x}|_r$ is assigned the same depth as $\mathtt{x}$. We know that the possible forms of $\mathbf{b}$ are: (i) a constant, (ii) a variable, (iii) the sum or product of two other basic bounds, (iv) a λ-expression, (v) an application, and (vi) an expression of the form $(\oplus \hat{\mathtt{v}}_1 \restriction \mathbf{b}_1, \ldots, \hat{\mathtt{v}}_n \restriction \mathbf{b}_n)\, |E_0|_r$. The depth assignment for forms (i) through (v) is the same as for the analogous cases for subject expressions. For form (vi), the depth of $\mathbf{b}$ is the same as the depth of E_0 relative to the assignment of types $\Gamma_i \cup \{\, \mathtt{v}_1 \colon \mathrm{N}, \ldots, \mathtt{v}_n \colon \mathrm{N} \,\}$, and the assignment of depths in which each $\mathtt{x}_j$ is as before and each $\mathtt{v}_j$ is assigned the same depth as $\mathbf{b}_j$. ◇

Example 32. If we take (9)+(10) as constituting a Seth bound, then under the above rules `len` is assigned depth 0 and $\mathtt{T}'$ is assigned depth 2. By way of contrast, consider the following HTP based on the right-hand side of (7):

$$\lambda\, |F|_b\,, |x|\,\textbf{.}\; |F|_b\, (\lambda n\textbf{.}|x|) \cdot |x|.$$

By Definition 20, this has depth 1. The difference in depths is due to the fact that in the HTP case we can substitute the HTP-bound on $|\mathtt{len}|_b$ for $|\mathtt{len}|_b$ itself and normalize to get rid of the additional application. ◇

At this point we believe we have substantiated our claim that Seth bounds are clumsy to work with. The prospect of using them directly to reason about the complexity of programs looks grim. However, there is an out. In the next section we introduce a typed programming formalism that has a fairly simple type system that is partially based on the assignment of depths to Seth bounds. From the type of an object in this formalism, we will be able to deduce the existence of a Seth bound on the object (under the **Full**-based semantics) and an HTP bound on the object (under the **BCont**-based semantics). Thus through this programming formalism, we can indirectly (or implicitly) work with Seth bounds in a way that suffices for our purposes.

10. Inflationary Tiered Loop Programs

Here we introduce ITLP, a higher-type version of the ITLP_2 programming formalism developed in Part I. The change from ITLP_2 to ITLP is analogous to the change from BTLP_2 to BTLP except that we also revise ITLP_2's type system. In ITLP_2 we made a sharp distinction between the "tier-polymorphic" types of "oracles" and the fixed types of defined procedures. With ITLP we drop this distinction and make *all* arrow types "tier-polymorphic." This results in a simpler and more uniform type system than that of ITLP_2. But all this comes at a price—see Remark 33.

Types

An ITLP type is a pair (σ, d) where σ is a simple type over N and $d \in \omega$; σ and d are called, respectively, the *shape* and the *depth* of (σ, d). For convenience we typically write N_d for (N, d) and $\sigma_0 \times \cdots \times \sigma_k \to_d \mathrm{N}$ for $(\sigma_0 \times \cdots \times \sigma_k \to \mathrm{N}, d)$.

The intended interpretation. We will state things here in terms of the **BCont**-based semantics. Each (σ, d) is intuitively viewed as naming a distinct copy of BCont_σ. The d in (σ, d) figures in as follows. Fix a particular ITLP procedure P and a variable v of P that is assigned type (σ, d). The intended interpretation of this type assignment is that throughout any computation involving P, the length of (the denotation of) v will be $\leq \mathbf{q}$, where $\mathbf{q}$ is some particular depth-d, type-$|\sigma|_b$ HTP bound over the lengths of (the denotations of) the parameters of P. (In general we have to consider not only P's parameters, but the parameters of the procedures in which P's declaration occurs.) Proposition 36 below confirms this interpretation and a similar one for the **Full**-based semantics.

The relation to ITLP_2 *types.* The N_d's here correspond directly to the N_d's of Section I-8 and we shall freely refer to them as *tiers*. The ITLP_2 type $\mathrm{N}^k \to_+ \mathrm{N}$ corresponds precisely to the type $\mathrm{N}^k \to_0 \mathrm{N}$ in the new system. The non-polymorphic arrow types in ITLP_2 do not have direct analogues in ITLP, but each ITLP_2 procedure can be translated to a roughly equivalent tier-polymorphic ITLP procedure.

Notation: For each σ, a simple type over N, and each $d \in \omega$, define $(\sigma)_d = (\sigma, d)$ and $((\sigma, d))_\omega = \sigma$. We also define $((\sigma_0, d_0) \times \cdots \times (\sigma_k, d_k))_\omega$ to be $\sigma_0 \times \cdots \times \sigma_k$.

P	$::=$	**Procedure** $v_0\,(v_1,\ldots,v_\ell)\colon \mathrm{N}_i\ P^*\ V\ I^*$ **Return** $v_r^{\mathrm{N}_i}$ **End**	*[procedures]*
V	$::=$	**var** $v_1^{\mathrm{N}_{i_1}},\ldots,v_m^{\mathrm{N}_{i_m}}$;	*[local var. decls.]*
I	$::=$	C ; $\mid L$; $\mid v^{\mathrm{N}_i} := E$;	*[instructions]*
C	$::=$	**Case** $v_0^{\mathrm{N}_i}$ **of** $\epsilon : I^*$ **or** $\mathbf{0}v_1^{\mathrm{N}_i} : I^*$ **or** $\mathbf{1}v_1^{\mathrm{N}_i} : I^*$ **Endcase**	*[case stms.]*
L	$::=$	**For** $v_0^{\mathrm{N}_i} := \underline{1}$ **to** $v_1^{\mathrm{N}_i}$ **do** I^* **Endfor**	*[loops]*
E	$::=$	$\epsilon \mid v^{\mathrm{N}_i} \mid \mathsf{c_0}(v^{\mathrm{N}_i}) \mid \mathsf{c_1}(v^{\mathrm{N}_i}) \mid \mathsf{down}(v_0^{\mathrm{N}_i}, v_1^{\mathrm{N}_j}) \mid v_0(A_1,\ldots,A_n)$	*[expressions]*
A	$::=$	$v \mid \lambda v_0,\ldots,v_k \,\mathbf{.}\, v(v_0',\ldots,v_\ell')$	*[arguments]*

Figure 3: The grammar for ITLP

Syntax

The grammar of ITLP is given in Figure 3. (Recall that $\underline{n}$ denotes the tally string $\mathbf{0}^n$.) For each λ-term, $\lambda v_0,\ldots,v_k \,\mathbf{.}\, v(v_0',\ldots,v_\ell')$, we insist that v is not among $v_0,\ldots,v_k$. **For**-loops also suffer some syntactic restrictions that shall be discussed shortly.

All the variables in an ITLP procedure must be declared either in procedure declarations, parameter lists, local variable declarations, or in λ-expressions. As in BTLP, recursive procedure calls are forbidden and variable scoping is static and follows standard conventions except that in procedure bodies we forbid nonlocal references to integer variables.[7] **N.B.** In the body of a λ-term we do allow references to integer variables that are declared local in the block containing that λ-term.

Typing

The typing rules and restrictions for an ITLP procedure declaration

$$\textsf{\textbf{Procedure}}\ v\,(v_0,\ldots,v_\ell)\colon \mathrm{N}_i\ P^*\ V\ I^*\ \textsf{\textbf{Return}}\ v_r^{\mathrm{N}_j}\ \ \textsf{\textbf{End}}$$

are (i) that, for $j = 0,\ldots,\ell$, the type of v_j must be of the form $(\sigma_j, 0)$ (i.e., a depth 0 type), and (ii) that the type of v must be of the form $\sigma_0\times\cdots\times\sigma_\ell \to_i \mathrm{N}$.

[7]As in Section I-8, we shall refer to the elements of $\mathrm{N}\cup\mathrm{N}_0\cup\mathrm{N}_1\cup\cdots$ collectively as *integers*, and variables over any of N_0, $\mathrm{N}_1,\ldots$ as *integer variables*.

We type λ-terms analogously:

$$\frac{v_0\colon(\sigma_0,0) \quad \cdots \quad v_n\colon(\sigma_n,0) \quad v(v_0',\dots,v_m')\colon \mathrm{N}_i}{\lambda v_0,\dots,v_n \,\mathbf{.}\, v(v_0',\dots,v_m')\colon \sigma_0\times\cdots\times\sigma_n \to_i \mathrm{N}}$$

(The rules for applications will guarantee that $i > 0$ in all the actual uses of the above rule.) Primitive expressions are typed thus:

$$\frac{}{\epsilon\colon \mathrm{N}_0} \qquad \frac{v\colon \mathrm{N}_i}{\mathsf{c_0}(v)\colon \mathrm{N}_i} \qquad \frac{v\colon \mathrm{N}_i}{\mathsf{c_1}(v)\colon \mathrm{N}_i} \qquad \frac{v_0\colon \mathrm{N}_i \quad v_1\colon \mathrm{N}_j}{\mathsf{down}(v_0,v_1)\colon \mathrm{N}_i} \quad \begin{pmatrix}\text{where}\\ i<j\end{pmatrix}$$

For applications we have the following two rules:

$$\frac{v\colon \sigma_0\times\cdots\times\sigma_n \to_{i+1} \mathrm{N} \quad a_0\colon(\sigma_0,d_0) \quad \cdots \quad a_n\colon(\sigma_n,d_n)}{v(a_0,\dots,a_n)\colon \mathrm{N}_{e_\ell(i+1,d)}} \tag{11}$$

$$\frac{v\colon \sigma_0\times\cdots\times\sigma_n \to_0 \mathrm{N} \quad a_0\colon(\sigma_0,d_0) \quad \cdots \quad a_n\colon(\sigma_n)}{v(a_0,\dots,a_n)\colon \mathrm{N}_{d+1}} \tag{12}$$

where $d = \max(d_0,\dots,d_n)$ and $\ell = \max(\mathit{level}(\sigma_0),\dots,\mathit{level}(\sigma_n))$. The first rule covers applications of defined procedures and the second rule covers "oracle applications." Finally, in an assignment statement, $v^{\mathrm{N}_i} := E$, we require that the type assigned E is N_j for some $j \le i$.

Remark 33. Recall from Section 8 that $\lambda d,i,\ell \,\mathbf{.}\, e_\ell(i,d)$ is a close cousin to Ackerman's function, thus, the presence of "$e_\ell(i+1,d)$" in (11) indicates that ITLP's type system is both too simplistic and too extravagant for practice. The over-simplicity stems from the rule (11). It is easy to come up with examples of terms to which this rule assigns types with depths that are clearly much too large for the terms in question. This problem could be fixed by replacing (11) with a collection of rules that would support finer reasoning about type depths.

The extravagance is an inherent feature of the set of ITLP-typable terms and programs. One can write down reasonably small ITLP procedures (where the smallness is measured by the size of a procedure's parse tree) that *necessarily* involve types with *basso profundo* depths. No refinement of the typing rules can escape this difficulty. To avoid these explosions in depths, our only option is to cull the set of ITLP-typable terms. The $\mathsf{ITLP}^\flat$ formalism in Section 13 explores this option. $\diamond$

Remark 34. Instead of having types consist of a shape and a depth, we could have replaced the depth component with an actual HTP bound or basic bound (depending on the semantics we had in mind) with the intended interpretation that the bound component of a type acts as a bound on the size of the things of that type. This would have enormously increased precision of the things expressible by our types, at the cost of enormously increasing the complexity of the type system. $\diamond$

Semantics

We consider two different interpretations of the ITLP-types, one with respect to **BCont** and another with respect to **Full**. We start with the **BCont** version. *Conventions:* With each of N, N_0, $\mathrm{N}_1, \ldots$ we shall pun between it being a formal type and it naming a distinct copy of the natural numbers. For each i, $x \in \mathrm{N}_i$, and $y \in \mathrm{N}$, $(x)_\omega$ denotes the N version of x and $(y)_i$ denotes the N_i version of y.

Let σ range over the simple types over N and $d \in \omega$. For each σ and d, define:

$$\mathbb{D}^{\mathrm{BCont}}_{\sigma,d} \quad = \quad \{\, (g,d) \mid g \in \mathrm{BCont}_\sigma \,\}.$$

$$\mathbb{D}^{\mathrm{BCont}}_{\sigma} \quad = \quad \cup_d \mathbb{D}^{\mathrm{BCont}}_{\sigma,d}. \qquad\qquad \mathbb{D}^{\mathrm{BCont}} \quad = \quad \cup_\sigma \mathbb{D}^{\mathrm{BCont}}_{\sigma}.$$

Suppose $\sigma = \sigma_1 \times \cdots \times \sigma_k \to \mathrm{N}$ and $(g,d) \in \mathbb{D}_\sigma$. We identify (g,d) with the function of type $\mathbb{D}^{\mathrm{BCont}}_{\sigma_1} \times \cdots \times \mathbb{D}^{\mathrm{BCont}}_{\sigma_k} \to \mathbb{D}^{\mathrm{BCont}}_{\mathrm{N}}$ such that

$$(g,d)\Big((x_1,d_1),\ldots,(x_k,d_k)\Big) \quad = \quad (g(x_1,\ldots,x_k),d'),$$

where, if $d = 0$, then $d' = 1 + \max(d_1,\ldots,d_k)$, and if $d > 0$ and ℓ is the maximum type-level of the σ_i's, then $d' = e_\ell(d, \max(d_0,\ldots,d_k))$. (That is, we determine d' as per the type assignment rules above.) Each ITLP type (σ,d) is considered as naming $\mathbb{D}^{\mathrm{BCont}}_{\sigma,d}$. *Notation:* For each $x \in \mathrm{BCont}_\sigma$ and $d \in \omega$, define $(x)_d = (x,d) \in \mathbb{D}^{\mathrm{BCont}}_{\sigma,d}$, $\big((x,d)\big)_\omega = x$, and $|(x,d)|_b = |x|_b$. Also define $|(\sigma,d)|_b = |\sigma|_b$.

The **Full** version of the interpretation of types consists merely of replacing BCont with Full and $|\cdot|_b$ with $|\cdot|_r$ in the above.

Under either interpretation of types, the operational semantics of ITLP are quite conventional and we shall discuss only its key points. In expressions, c_0 and c_1 denote the functions that return the result of prefacing their argument with **0** and **1**, respectively. Given $\mathsf{x}\colon \mathrm{N}_i$ and $\mathsf{y}\colon \mathrm{N}_j$ (with $i < j$), the intended interpretation of $\mathsf{down}(\mathsf{x},\mathsf{y})$ is that it returns the N_i version of y's value, *provided* the length of this value is no greater than the length of x's value; otherwise $\mathsf{down}(\mathsf{x},\mathsf{y})$ returns the N_i version of 0. (The rationale for down is discussed in section I-7.) The interpretations of **Case**-statements and assignment statements are standard. A **For**-loop of the form

$$\textbf{For } v_0^{\mathrm{N}_j} := \underline{1} \textbf{ to } v_1^{\mathrm{N}_j} \textbf{ do } \vec{I} \textbf{ Endfor} \tag{13}$$

must satisfy the following restrictions.

1. No assignments to $v_0^{\mathrm{N}_j}$ occur within $\vec{I}$.
2. Each "$v^{\mathrm{N}_i} := E$" with $i \leq j$ occurring within $\vec{I}$ must be such that either:
   (a) Each integer variable in E is a type N_j with $j < i$, or
   (b) E is of the form "$\mathsf{down}(v_2^{\mathrm{N}_i}, v_3^{\mathrm{N}_k})$."

Note that assignments to $v_1^{\mathrm{N}_j}$ *are* permitted in $\vec{I}$, and the initialization and increments to control variables of any inner **For**-loops are not considered violations to the restrictions of the containing **For**-loops. Given these restrictions, the **For**-loop of (13) is equivalent to:

$$v_0^{\mathrm{N}_j} := \mathsf{c_0}(\epsilon); \quad \textbf{While } |v_0^{\mathrm{N}_j}| \leq |v_1^{\mathrm{N}_j}| \textbf{ do } \vec{I} \;\; v_0^{\mathrm{N}_j} := \mathsf{c_0}(v_0^{\mathrm{N}_j}); \textbf{ Endwhile}$$

where **While** ... **do** ... **Endwhile** has the usual meaning. (The rationale for this **For** construct is discussed in section I-7.) So as with BTLP, one can give a straightforward inductive semantics to ITLP and the procedure application turns out to be purely applicative. Note however that, as was the case with ITLP_2, we need to show that **For**-loops always terminate.

Definition 35. Suppose that P is an ITLP-procedure of type $\sigma_0 \times \cdots \times \sigma_n \to_i \mathrm{N}$ with free variables $\mathsf{y}_1, \ldots, \mathsf{y}_m$ of respective types $\tau_1, \ldots, \tau_m$.[8]

(a) For each $\alpha_1 \in \mathbb{D}^{\mathrm{Full}}_{\tau_1}, \ldots, \alpha_m \in \mathbb{D}^{\mathrm{Full}}_{\tau_m}$, define $[\![\mathsf{P}]\!]_{\mathrm{Full}}[\vec{\mathsf{y}} \mapsto \vec{\alpha}]$ to be the functional $\mathbb{D}^{\mathrm{Full}}_{\sigma_0} \times \cdots \times \mathbb{D}^{\mathrm{Full}}_{\sigma_n} \rightharpoonup \mathbb{D}_{\mathrm{N}}$ determined by P under the **Full**-based semantics of ITLP when the variables $\mathsf{y}_1, \ldots, \mathsf{y}_m$ respectively denote $\alpha_1, \ldots, \alpha_m$.

(b) We say that a function $f \in \mathrm{Full}_\sigma$ is ITLP-computable with respect to the **Full**-bases semantics iff and only if there is a $d \in \omega$ and a type (σ, d) ITLP-procedure P with no free variables such that $[\![\mathsf{P}]\!]_{\mathrm{Full}} = (f, d) \in \mathbb{D}^{\mathrm{Full}}_\sigma$.

(c) Define $[\![\,\cdot\,]\!]_{\mathrm{BCont}}[\,\cdot\,]$ and ITLP-computability analogously for the **BCont**-based semantics. ◇

[8] Since the only nonlocal variables reference allowed in ITLP are to variables of noninteger types, each of $\tau_1, \ldots, \tau_m$ is an arrow type. Also, if P has no free variables, then $m = 0$.

Note that in the above definition $[\![\mathsf{P}]\!]_{\mathrm{BCont}}[\vec{\mathsf{y}} \mapsto \vec{\alpha}]$ and $[\![\mathsf{P}]\!]_{\mathrm{Full}}[\vec{\mathsf{y}} \mapsto \vec{\alpha}]$ are presented as possibly partial functions so as not to beg the question of the totality of **For**-loops. That question is part of our next order of business.

Polynomial boundedness

The key fact we need to establish about the ITLP-computable functionals is that they are polynomially bounded, where the notion of polynomial bound is the appropriate one for each of the two possible semantics. This is stated formally in the next proposition.

Proposition 36. *Suppose* P *is an* ITLP *procedure of type* $\tau = \sigma_0 \times \cdots \times \sigma_k \to_i \mathrm{N}$ *with* $FV(\mathsf{P}) \subseteq \Gamma = \{\, \mathsf{y}_1 \colon (\sigma_0', 0) \ldots, \mathsf{y}_m \colon (\sigma_m', m) \,\}$.

(a) There is an HTP $\mathbf{p}_{\mathsf{P}}$ *with respect to* Γ *of depth no greater than* i *such that, for all* $\rho \in [\![\Gamma]\!]_{\mathbf{BCont}}$,

$$|[\![\mathsf{P}]\!]_{\mathbf{BCont}}\,\rho|_b \quad \leq \quad [\![\mathbf{p}_{\mathsf{P}}]\!]_{\mathrm{TLen}}\ |\rho|_b\,.$$

(b) There is Seth bound $\mathcal{S}$ *with respect to* Γ *that includes a basic bound* $\Gamma \vdash^{\tau} |\mathsf{P}|_r \leq \mathbf{b}_{\mathsf{P}}$ *such that the depths assigned to* P *and* $\mathbf{b}_{\mathsf{P}}$ *are no greater than* i *and such that, for all* $\rho \in [\![\Gamma]\!]_{\mathbf{Full}}$,

$$|[\![\mathsf{P}]\!]_{\mathbf{Full}}\,\rho|_r \quad \leq \quad [\![\mathbf{b}_{\mathsf{P}}]\!]_{\mathbf{FL}}\ |\rho|_r\,.$$

Proof Sketch. We argue only part (a). The argument is a modification of the proof of Lemma I-22 to the present setting. The differences in the type systems make for some changes in how the bounds are handled. However, the core of the argument remains essentially the same except for the analyses of (i) assignments of the form $\mathsf{w} := \mathsf{v}(a_0, \ldots, a_k)$, and (ii) λ-terms which can occur as some of the a_i's in such assignments. We sketch these analyses below after setting up some necessary formal machinery.

Without loss of generality we may assume that throughout P the only global references are to variables with types of depth 0. (Recall that in ITLP the only global references allowed are to variables with arrow types, and recursive calls are forbidden. So, if we start with a P that fails to satisfy this assumption, then we can transform P into an equivalent procedure that meets our requirements via the standard trick of making local copies of globally referenced procedures.)

We also assume without loss of generality that throughout P each variable occurs in only one declaration. (Hence, there is no shadowing of declarations.)

For a collection of integer variables $\vec{\mathsf{x}}$, let $|\vec{\mathsf{x}}|_i$ denote the maximum length of the values assigned to the current instantiation of the variables in $\vec{\mathsf{x}}$ of types $\mathrm{N}_0, \ldots, \mathrm{N}_i$ up through this point in the computation. Suppose v is a variable of an arrow type that is declared in a parameter list of a procedure Q. At each point in the computation for a particular call to Q, let $\langle\!\langle \mathsf{v} \rangle\!\rangle$ denote the finite portion of $[\![\mathsf{v}]\!]_{\mathrm{BCont}}$ discovered (via calls involving v) up through this point. Thus a call involving v can enlarge $\langle\!\langle \mathsf{v} \rangle\!\rangle$. As in the proof of Proposition I-19, we shall freely use $\oplus$ as another additive operation in HTPs. We can always get rid of the $\oplus$'s in an HTP $\mathbf{q}$ by replacing them with $+$'s, thus obtaining a standard-issue HTP $\mathbf{q}'$ such that $\mathbf{q} \leq \mathbf{q}'$.

Now suppose that procedure Q occurs within P. (It may be that $\mathsf{P} = \mathsf{Q}$.) Suppose inductively that part (a) holds for each subprocedure of Q. Let $\vec{\mathsf{x}}$ be the list of all the integer variables declared in Q's parameter list and let $i_{\max}$ be the highest tier number of any integer variable declared in Q. Let Υ be a type context that contains exactly the higher-type variables visible from within the body of Q that are declared in the parameter list of some procedure. Suppose inductively that each higher-type variable $\mathsf{w}\colon(\tau', n)$ that occurs in Q has an associated type $|(\tau', n)|_b$, depth n HTP-bound $\mathbf{r}_{\mathsf{w}}$ over $|\Upsilon|_b$ and where

i. if w occurs in Υ, then $\mathbf{r}_{\mathsf{w}} = |\mathsf{w}|_b$, and

ii. if w does not occur in Υ and w names a procedure P' with free variables from $\Gamma' = \{\, \mathsf{y}'_1\colon\tau'_1, \ldots, \mathsf{y}'_m\colon\tau'_m \,\}$, then $\mathbf{r}_{\mathsf{w}} =$ the normal form of $\mathbf{p}_{\mathsf{P}'}[|\mathsf{y}'_1|_b, \ldots, |\mathsf{y}'_m|_b \mapsto \mathbf{r}_{\mathsf{y}'_1}, \ldots, \mathbf{r}_{\mathsf{y}'_k}]$ where $\mathbf{p}_{\mathsf{P}'}$ is as in part (a) of the lemma.

Since recursive calls are forbidden in ITLP, it follows that the $\mathbf{r}_{\mathsf{w}}$'s are well-defined.

Definition 37. Let m be a fresh variable.

(a) We say that $\vec{\mathbf{q}} = \mathbf{q}_0, \ldots, \mathbf{q}_{i_{\max}}$ is a *sequence of tier-bounds for* Q if and only if, for each $i \leq i_{\max}$, $\mathbf{q}_i$ is a type-ω, depth-i HTP with $FV(\mathbf{q}_i) \subseteq |\Upsilon|_b \cup \{\, \mathsf{m}\colon\omega \,\}$ for which we have $\mathbf{q}_0 \leq \mathbf{q}_1 \leq \cdots \leq \mathbf{q}_{i_{\max}}$.

(b) We say that $\vec{\mathbf{q}} = \mathbf{q}_0, \ldots, \mathbf{q}_{i_{\max}}$ *holds* at a particular point in a particular execution of Q if and only if at this point we have, for each $i \leq i_{\max}$, that

$$|\vec{\mathsf{x}}|_i \quad \leq \quad [\![\mathbf{q}_i]\!]_{\mathrm{TLen}}\,\{\, \mathsf{m}, |\mathsf{u}_1|_b, \ldots, |\mathsf{u}_k|_b \mapsto m, |\langle\!\langle \mathsf{u}_1 \rangle\!\rangle|_b, \ldots, |\langle\!\langle \mathsf{u}_k \rangle\!\rangle|_b \,\},$$

where $FV(\mathbf{q}_i) \subseteq \{\, \mathsf{m}, |\mathsf{u}_1|_b, \ldots, |\mathsf{u}_k|_b \,\}$ and where m is the maximum of the lengths of the initial values of the integer arguments of Q in this call. $\diamond$

Definition 38. Suppose that $\vec{I}$ is a sequence of instructions from Q and that $\vec{\mathbf{p}}$, $\vec{\mathbf{p}}'$, $\vec{\mathbf{q}}$, and $\vec{\mathbf{q}}'$ range over sequences of tier-bounds for Q.

(a) We say that $\vec{\mathbf{p}}$, $\vec{I}$, and $\vec{\mathbf{q}}$ form a *weak bounding triple* if and only if, in any execution of Q, *if* $\vec{\mathbf{p}}$ holds just before the execution of $\vec{I}$, *then* $\vec{\mathbf{q}}$ holds just after the execution of $\vec{I}$.

(b) Suppose that $\vec{\mathbf{p}}$, $\vec{I}$, and $\vec{\mathbf{q}}$ form a weak bounding triple. We say that $\mathbf{q}_i$ (i.e., the i-th element of $\vec{\mathbf{q}}$) is *j-dependent* (where $-1 \leq j \leq i$) if and only if, for any $\vec{\mathbf{p}}'$ with $[\![\mathbf{p}'_k]\!] \leq [\![\mathbf{p}_k]\!]$ for each $k \leq j$, there is a $\vec{\mathbf{q}}'$ such that $\mathbf{q}'_k = \mathbf{q}_k$ for each $k \leq i$ and $\vec{\mathbf{p}}'$, $\vec{I}$, and $\vec{\mathbf{q}}'$ form a weak bounds triple.[9]

(c) We write $\{\leq \vec{\mathbf{p}}\}\,\vec{I}\,\{\leq \vec{\mathbf{q}}\}$ and say that $\vec{\mathbf{p}}$, $\vec{I}$, and $\vec{\mathbf{q}}$ form a *bounding triple* if and only if $\vec{\mathbf{p}}$, $\vec{I}$, and $\vec{\mathbf{q}}$ form a weak bounding triple and, for each $i < i_{\max}$, $\mathbf{q}_i$ is i-dependent.

(d) We say that a bounding triple $\{\leq \vec{\mathbf{p}}\}\,\vec{I}\,\{\leq \vec{\mathbf{q}}\}$ is *copacetic through tier i*, if and only if, $\mathbf{q}_0$ is a constant and, for each $j = 1, \ldots, i$, $\mathbf{q}_j$ is is $(j-1)$-dependent and is of the form $\mathbf{s}_j \oplus \mathbf{p}_j$ for some HTP $\mathbf{s}_j$. ◇

Having introduced the requisite technical machinery, we can now focus on the case of interest. Suppose that

$$\mathsf{w} := \mathsf{v}(a_0, \ldots, a_\ell)$$

is an assignment that occurs within the body of Q where w is of type N_i. Note that by the ITLP-typing rules, all the integer variables appearing in "$\mathsf{v}(a_0, \ldots, a_\ell)$" are of tier i or less. Let $\vec{\mathbf{p}}$ be a sequence of tier-bounds for Q. We want to construct a $\vec{\mathbf{q}}$ such that

$$\{\leq \vec{\mathbf{p}}\} \quad \mathsf{w} := \mathsf{v}(a_0, \ldots, a_\ell) \quad \{\leq \vec{\mathbf{q}}\}.$$

(We deal with copaceticity later.) Since w is the only variable that changes as a result of the assignment (excluding the meta-variables of the form: $\langle\!\langle \mathsf{v} \rangle\!\rangle$), it suffices to take

- $\mathbf{q}_j = \mathbf{p}_j$ for each $j < i$, and
- $\mathbf{q}_j = \mathbf{q}_i \oplus \mathbf{p}_j$ for each $j > i$.

So, constructing an appropriate $\vec{\mathbf{q}}$ comes down to constructing an appropriate $\mathbf{q}_i$. To do this, there are four subcases to consider.

[9] In other words, the "post-bounds" $\mathbf{q}_0, \ldots, \mathbf{q}_i$ really depend only on the values of the 0th through the jth "pre-bounds."

Subcase 1: $\mathsf{v}\colon \sigma'_0 \times \cdots \times \sigma'_\ell \to_0 \mathrm{N}$ *and each argument, a_j, is an occurrence of a variable.* It follows that v occurs in Υ and that each a_j must be of a type of the form (σ'_j, d) with $d < i$ (since otherwise, by the ITLP typing rules, the assignment would not be well-typed). Let $\mathbf{q}_i$ be the normal form of the HTP term $(|\mathsf{v}|_b\ t_0\ \ldots\ t_\ell)$ where, for $j = 1, \ldots, \ell,\ t_j$ is

(a) $\mathbf{p}_k$, if a_j is an integer variable of type N_k,
(b) $|a_j|_b$, if a_j is a variable occurring in Υ, and
(c) $\mathbf{r}_{a_j}$, if a_j is a higher-type variable not occurring in Υ.

(For clause c, recall that each $\mathbf{r}_{\mathsf{u}}$ has no type-ω free variables.) It follows that $\mathbf{q}_i$ is as required.

Subcase 2: $\mathsf{v}\colon \sigma'_0 \times \cdots \times \sigma'_\ell \to_{m+1} \mathrm{N}$ *and each argument, a_j, is an occurrence of a variable.* It follows that v does not occur in Υ. Let $\mathbf{q}_i$ be the normal form of $(\mathbf{r}_{\mathsf{v}}\, t_0\ \ldots\ t_\ell)$ where $t_0, \ldots, t_\ell$ are as in the previous subcase. By our induction hypotheses, Lemma 21, and (11), it follows that $\mathbf{q}_i$ is as required.

Subcase 3: $\mathsf{v}\colon \sigma'_0 \times \cdots \times \sigma'_\ell \to_0 \mathrm{N}$ *and there are λ-terms among the a_i's.* Suppose $\lambda \mathsf{z}_0, \ldots, \mathsf{z}_c \centerdot \mathsf{u}(\mathsf{z}'_0, \ldots, \mathsf{z}'_{c'})$ is a λ-term of type $\delta_0 \times \cdots \times \delta_c \to_d \mathrm{N}$ that appears as one of the a_j's. Let $\Upsilon' = \{\, \mathsf{z}_0\colon (\delta_0, 0), \ldots, \mathsf{z}_c\colon (\delta_c, 0) \,\}$ and $\mathbf{s}$ be the normal form of $(\mathbf{r}_{\mathsf{u}}\, t'_0\ \ldots\ t'_{c'})$ where for each $j \le c',\ t'_j$ is

(a′) $\mathbf{p}_k$, if z_j is of type N_k and does not occur in Υ',
(b′) $\left|\mathsf{z}'_j\right|_b$, if z'_j occurs in $\Upsilon \cup \Upsilon'$, and
(c′) $\mathbf{r}_{\mathsf{z}'_j}$, if z'_j is a higher-type variable that does not occur in Υ.

Suppose that $\Upsilon = \{\, \mathsf{y}_0\colon (\gamma_0, 0), \ldots, \mathsf{y}_e\colon (\gamma_e, 0) \,\}$ and that $\rho_0 \in [\![\Upsilon \cup \Upsilon']\!]_{\mathbf{BCont}}$ and $\rho_1 \in [\![\Upsilon]\!]_{\mathbf{BCont}}$ are such that $\rho_0 \supseteq \rho_1$, i.e., $\rho_0(\mathsf{y}_i) = \rho_1(\mathsf{y}_i)$ for each $i \le e$. By our induction hypotheses, we have that

$$\left|[\![\mathsf{u}(\mathsf{z}'_0, \ldots, \mathsf{z}'_{c'})]\!]_{\mathbf{BCont}}\, \rho_0\right| \quad\le\quad [\![\mathbf{s}]\!]_{\mathbf{TLen}}\ |\rho_0|_b$$

and hence

$$\left|[\![\lambda \mathsf{z}_0, \ldots, \mathsf{z}_c \centerdot \mathsf{u}(\mathsf{z}'_0, \ldots, \mathsf{z}'_{c'})]\!]_{\mathbf{BCont}}\, \rho_1\right|_b \quad\le\quad [\![\lambda\, |\mathsf{z}_0|_b, \ldots, |\mathsf{z}_c|_b \centerdot \mathbf{s}]\!]_{\mathbf{TLen}}\ |\rho_1|_b\,.$$

Let $\mathbf{q} = \lambda g_{\mathsf{z}_0}, \ldots, g_{\mathsf{z}_b} \centerdot \mathbf{s}$. Thus, $\mathbf{q}$ serves as an HTP bound on $\lambda \mathsf{z}_0, \ldots, \mathsf{z}_b \centerdot \mathsf{u}(\mathsf{z}'_0, \ldots, \mathsf{z}'_c)$ in the context of the particular call $\mathsf{v}(a_0, \ldots, a_k)$. Note that $\mathbf{q}$ may well have m (recall Definition 37(a)) as a free variable. Now, let $\mathbf{q}_i$ be the normal form of $(|\mathsf{v}|_b\ t_0\ \ldots\ t_\ell)$ where, for $j = 1, \ldots, \ell,\ t_j$ is

(a″) as in Subcase 1, if a_j is not a λ-term and
(b″) an HTP bound as constructed above, if a_j is a lambda-term.

By a bit of higher-type algebra it follows that $\mathbf{q}_i$ is as required.

Subcase 4: $\mathsf{v}\colon \sigma_0' \times \cdots \times \sigma_\ell' \to_{m+1} \mathrm{N}$ *and there are* λ*-terms among the* a_i*'s.* This is the straightforward generalization of the arguments for Subcases 2 and 3.

The copacetic versions of Subcases 1 through 4 are straightforward variations of the above arguments. □

11. Bounded Continuous Basic Feasible Functionals

In this section we show how to translate any given BTLP program into an equivalent ITLP program. This is the first link in our grand chain of translations that will eventually bring us around to BTLP again. As an immediate consequence of this translation result we have, for both **Full**-based and **BCont**-based settings, that the BTLP-computable functionals are contained in the ITLP-computable functionals. By Proposition 36 we then have polynomial boundedness for the BTLP-computable functionals under both semantics. In the **BCont**-case this means that the BTLP-computable functionals over **BCont** are themselves elements of **BCont**. Once this last fact is established we will have the moral authority to define the *bounded continuous basic feasible functionals* as the class of BTLP-computable functionals over **BCont**.

Convention: In this section and in this section *only*, ITLP variables will be set in *slanted sans serif font*. This is to help clearly distinguish them from BTLP variables which, as usual, are set in `typewriter font`.

Proposition 39 (The BTLP to ITLP Translation). *Suppose that* $\mathtt{P}$ *is a* BTLP *procedure of type* σ_0 *with its free variables within* $\{\, \mathtt{z}_1\colon \sigma_1, \ldots, \mathtt{z}_n\colon \sigma_n \,\}$. *Also suppose that* $d_1, \ldots, d_n \in \omega$. *Then one can uniformly, effectively construct an* ITLP *procedure* P *with type* (σ_0, d_0) *(for some* $d_0 > 0$*) with free variables within* $\{\, z_1\colon (\sigma_1, d_1), \ldots, z_n\colon (\sigma_n, d_n) \,\}$ *such that:*

(a) For all $\alpha_1 \in \mathrm{BCont}_{\sigma_1}, \ldots, \alpha_n \in \mathrm{BCont}_{\sigma_n}$,

$$\llbracket \mathtt{P} \rrbracket_{\mathbf{BCont}} \{\, \mathtt{z}_1, \ldots, \mathtt{z}_n \mapsto \alpha_1, \ldots, \alpha_n \,\} = \Big(\llbracket P \rrbracket_{\mathbf{BCont}} \{\, z_1, \ldots, z_n \mapsto (\alpha_1)_{d_1}, \ldots, (\alpha_m)_{d_n} \,\} \Big)_\omega .$$

(b) Similarly for the **Full** *semantics.*

Proof. Since **BCont** is a subcategory of **Full**, part (a) is an immediate consequence of part (b). So we argue only (b).

We show how to translate $\mathtt{P}$ into an appropriate P. This translation is fairly direct and the only difficult parts are in accommodating the differences in the type systems and the looping constructs. Each higher-type variable of $\mathtt{P}$ will have a corresponding higher-type variable of the same name (but in sans serif) in P. For an integer variable of $\mathtt{P}$, say $\mathtt{x}$, the translation may introduce many different versions of $\mathtt{x}$ in P, but there will be at most one such version of $\mathtt{x}$ per tier; we use $\mathsf{x}[i]$ as our name for the tier N_i version of $\mathtt{x}$ in P. In describing the translation we shall ignore the issue of name conflicts, as we may assume that such problems are handled in some standard way in the translation process.

We shall use the ITLP-variables *plus*, *monus*, and *smash* (each of type $\mathrm{N} \times \mathrm{N} \to_1 \mathrm{N}$) as names for ITLP-procedures computing $(\lambda a, b \centerdot a + b)_1$, $(\lambda a, b \centerdot a \mathbin{\dot{-}} b)_1$, and $(\lambda a, b \centerdot a \# b)_1$, respectively. The very last step of the translation will be to add the appropriate procedure declarations for *plus*, *monus*, and *smash* in the subprocedure declaration section of P.

Let $\mathtt{Q}$ be a procedure that occurs within $\mathtt{P}$. (It may be that $\mathtt{P} = \mathtt{Q}$.) Let $\hat{\mathtt{z}}_1, \ldots, \hat{\mathtt{z}}_{\hat{n}}$ be $\mathtt{Q}$'s global variables and let $\hat{\sigma}_1, \ldots, \hat{\sigma}_{\hat{n}}$ be their respective types. Fix $\hat{d}_1, \ldots, \hat{d}_{\hat{n}} \in \omega$.

Suppose $\mathtt{Q}$ is of the form:

Procedure $\mathtt{z}\,(\mathtt{u}_1\colon \tau_1, \ldots, \mathtt{u}_{k_0}\colon \tau_{k_0})$
  $\mathtt{R}_1; \ldots ; \mathtt{R}_{k_1};$
  var $\mathtt{v}_1, \ldots, \mathtt{v}_{k_2};$
  $I_1 \cdots I_\ell$
  Return $\mathtt{v}_r;$
End

where $\mathtt{R}_1, \ldots, \mathtt{R}_k$ are the procedures declared in $\mathtt{Q}$'s subprocedure declaration section. (If $\mathtt{Q}$ has no subprocedures, then $k_1 = 0$.) Let $\mathtt{y}_1, \ldots, \mathtt{y}_m$ be the sublist of $\mathtt{u}_1, \ldots, \mathtt{u}_{k_0}$ consisting of exactly the higher-type variables declared in $\mathtt{Q}$'s parameter list. Suppose $\hat{\tau}_1, \ldots, \hat{\tau}_m$ are the respective types of $\mathtt{y}_1, \ldots, \mathtt{y}_m$. In our translation of $\mathtt{Q}$, each $\mathtt{y}_i$ will have the type $(\hat{\tau}_i, 0)$ and each $\hat{\mathtt{z}}_i$ will have type $(\hat{\sigma}_i, \hat{d}_i)$.

For each $i = 1, \ldots, k_1$, let $\hat{\mathtt{z}}_{\hat{n}+i}$ be the declared name of $\mathtt{R}_i$ in $\mathtt{Q}$ and let $\hat{\sigma}_{\hat{n}+i}$ be $\hat{\mathtt{z}}_{\hat{n}+i}$'s assigned type. Since recursion is forbidden in BTLP, we may assume that the $\mathtt{R}_i$'s are ordered so that, for each i, $\mathtt{R}_i$ contains no global reference to any of $\hat{\mathtt{z}}_{\hat{n}+i}, \ldots, \hat{\mathtt{z}}_{\hat{n}+k_1}$. It follows that for each $i = 1, \ldots, k_1$, the global variables of $\mathtt{R}_i$ are contained in $\{\, \mathtt{y}_1, \ldots, \mathtt{y}_m, \hat{\mathtt{z}}_1, \ldots, \hat{\mathtt{z}}_{\hat{n}+i-1} \,\}$.

Suppose inductively that the lemma holds for each $\mathtt{R}_i$. Apply the lemma to $\mathtt{R}_1$ with $\hat{d}_1, \ldots, \hat{d}_{\hat{n}}$ as the numbers associated with the global variables

$\hat{\mathtt{z}}_1, \ldots, \hat{\mathtt{z}}_{\hat{n}}$. Let R_1 be the translation of $\mathtt{R}_1$ and let $\hat{d}_{\hat{n}+1}$ be the depth of the ITLP-type assigned R_1. For $i = 2, \ldots, k$ in turn, apply the lemma to $\mathtt{R}_i$ with $\hat{d}_1, \ldots, \hat{d}_{\hat{n}+i-1}$ as the numbers associated with the global variables $\hat{\mathtt{z}}_1, \ldots, \hat{\mathtt{z}}_{\hat{n}+i-1}$. Let R_i be the translation of $\mathtt{R}_i$ and let $\hat{d}_{\hat{n}+i}$ be the depth of the ITLP-type assigned R_i.

Now let us consider how to translate $I_1 \cdots I_\ell$, the body of $\mathtt{Q}$.

Let $\mathtt{x}$, $\mathtt{x}_0$, $\mathtt{x}_1, \ldots$ range over the variables of type N declared in $\mathtt{Q}$, either in $\mathtt{Q}$'s parameter list or in $\mathtt{Q}$'s local variable declaration section. For each such variable $\mathtt{x}$, we introduce the ITLP-variable $x[0]\colon \mathrm{N}_0$ and the integer meta-variable $i_{\mathtt{x}}$, which we initialize to 0. During the course of the translation we may introduce variables of the form $x[i]\colon \mathrm{N}_i$, where $i > 0$; and the value of $i_{\mathtt{x}}$ will record the value of the largest i such that $x[i]$ has been introduced as of the current point of the translation.

Suppose inductively that $I_1 \cdots I_j$ have already been translated, where $0 \leq j < \ell$. To translate I_{j+1}, we have the following two case.

Case 1: I_{j+1} is of the form $\mathtt{x}_0 := \mathtt{E}$. First we translate $\mathtt{E}$ to an ITLP expression E as follows.

Subcase 1.1: $\mathtt{E}$ *is of the form* $\mathtt{x}$. Then let E be $x[i_{\mathtt{x}}]$.

Subcase 1.2: $\mathtt{E}$ *is of one of the forms* $\mathtt{x}_1 + \mathtt{x}_2$, $\mathtt{x}_1 \dot{-} \mathtt{x}_2$, or $\mathtt{x}_1 \# \mathtt{x}_2$. Then let E be $\mathit{plus}(x_1[i_{\mathtt{x}_1}], x_2[i_{\mathtt{x}_2}])$, $\mathit{monus}(x_1[i_{\mathtt{x}_1}], x_2[i_{\mathtt{x}_2}])$, or $\mathit{smash}(x_1[i_{\mathtt{x}_1}], x_2[i_{\mathtt{x}_2}])$, respectively.

Subcase 1.3: $\mathtt{E}$ *is of the form* $\mathtt{v}_0(a_1, \ldots, a_u)$. Note that each free higher-type variable in $\mathtt{E}$ must be in $\{\, \mathtt{y}_1, \ldots, \mathtt{y}_m \,\} \cup \{\, \hat{\mathtt{z}}_1, \ldots, \hat{\mathtt{z}}_{\hat{n}+k_1} \,\}$, and hence this variable has a corresponding ITLP-variable of the same name. Let E be the result of: (i) replacing each type N free variable $\mathtt{x}$ with $x[i_{\mathtt{x}}]$, and (ii) replacing each variable $\mathtt{u}\colon \tau$ bound by a λ-expressing in $\mathtt{E}$ by an ITLP-variable $u\colon (\tau, 0)$. (Except for a change in font, the higher-type free variables are left alone in this translation.) For example, if $\mathtt{E} = \hat{\mathtt{z}}_1(\lambda \mathtt{u}\colon \mathrm{N} \,.\, \mathtt{y}_3(\mathtt{u}, \mathtt{x}), \mathtt{z}_8)$, then $E = \hat{z}_1(\lambda u\colon \mathrm{N}_0 \,.\, y_3(u, x[i_{\mathtt{x}}]), z_8)$.

Given E, let N_e be the type assigned E by ITLP's typing rules, and let $i_* = \max(e, i_{\mathtt{x}_0})$. We then translate $\mathtt{x}_0 := \mathtt{E}$ to

$$x_0[i_*] := E$$

where, if $i_* > i_{\mathtt{x}_0}$, we also introduce the new variable $x_0[i_*]$ and reset $i_{\mathtt{x}_0}$ to i_*.

Case 2: I_{j+1} is of the form **Loop** $\mathtt{x}_0$ **with** $\mathtt{x}_1$ **do** $\hat{I}_1 \cdots \hat{I}_{\hat{\ell}}$ **Endloop**. Let $\tilde{\mathtt{x}}_0, \ldots, \tilde{\mathtt{x}}_{\tilde{k}}$ be a list of all the type N variables that occur in I_{j+1} and let $i_* = \max(\{\, i_{\tilde{\mathtt{x}}_j} \mid j \leq \tilde{k} \,\})$. We introduce new variables $b\colon \mathrm{N}_{i_*}$ and $c\colon \mathrm{N}_{i_*}$ and translate I_{j+1} to

$\tilde{x}_0[i_*] := \tilde{x}_0[i_{\tilde{x}_0}];\ \ \dots\ \ \tilde{x}_{\tilde{k}}[i_*] := \tilde{x}_{\tilde{k}}[i_{\tilde{x}_{\tilde{k}}}];$
(* *In the translation, now reset each* $i_{\tilde{x}_j}$ *to* i_*. *)
$b := x_1[i_*];$
For $c := \underline{1}$ **to** $x_0[i_*]$ **do** $\hat{I}'_1 \cdots \hat{I}'_{\hat{\ell}}$ **Endfor**;

where $\hat{I}'_1 \cdots \hat{I}'_{\hat{\ell}}$ are the translations of $\hat{I}_1 \cdots \hat{I}_{\hat{\ell}}$ determined by the following two subcases.

Subcase 2.1: $\hat{I}_{j'}$ *is of the form* $\mathtt{x}'_0 := \mathtt{E}$. Then translate $\mathtt{E}$ to E as in Case 1 above and translate $\hat{I}_{j'}$ to

$x'_0[i_*] := \mathit{down}(b, E);$

where here we are using liberalized ITLP.

Subcase 2.2: $\hat{I}_{j'}$ *is of the form* **Loop** $\mathtt{x}'_0$ **with** $\mathtt{x}'_1$ **do** $\tilde{I}_1 \cdots \tilde{I}_{\tilde{\ell}}$ **Endloop**. We introduce new variables $b'\colon \mathrm{N}_{i_*}$ and $c'\colon \mathrm{N}_{i_*}$ and translate $\hat{I}_{j'}$ to the following liberalized ITLP

$b' := b;$ (* *We push the old value of* b. *)
$b := \min(b, x_1[i_*]);$
For $c' := \underline{1}$ **to** $x'_0[i_*]$ **do** $\tilde{I}'_1 \cdots \tilde{I}'_{\tilde{\ell}}$ **Endfor**;
$b := b';$ (* *We pop the old value of* b. *)

where $\tilde{I}'_1 \cdots \tilde{I}'_{\tilde{\ell}}$ are the translations of $\tilde{I}_1 \cdots \tilde{I}_{\tilde{\ell}}$ obtained by recursively applying Subcases 2.1 and 2.2 to them as appropriate.

Finally, putting the above together, we translate $\mathtt{Q}$ to

Procedure $z(u'_1\colon (\tau_1)_0, \dots, u'_{k_0}\colon (\tau_{k_0})_0)\colon \mathrm{N}_{\max(1,j)}$
  $R_1;\ \dots\ ;\ R_{k_1};$
  var $x_1[i_1]\colon \mathrm{N}_{i_1}, \dots, x_{k_3}[i_{k_3}]\colon \mathrm{N}_{i_{k_3}};$
  — *the translation of* $I_1 \cdots I_\ell$ *as described above* —
  Return $v_r[j];$
End

where:

- for $i = 1, \dots, k_0$, u'_i is $u_i[0]$ if $\tau_i = \mathrm{N}$, and is u_i otherwise;
- $R_1;\ \dots\ ;\ R_{k_1}$ are as above;
- $x_1[i_1], \dots, x_{k_3}[i_{k_3}]$ are the integer variables introduced in the translation of $I_1 \cdots I_\ell$ that are not in the list $u'_1, \dots, u'_{k_0}$; and

- j is the final value of i_{v_r} in the translation of $I_1 \cdots I_\ell$.

The argument for the correctness of this translation is straightforward and omitted. □

Corollary 40. *With respect to either the* **Full**- *or* **BCont**-*based semantics, the class of* BTLP-*computable functionals is a subset of the* ITLP-*computable functionals.*

Corollary 41. *Suppose* P *is a* BTLP *procedure of type* τ *with no free variables.*

(a) There is a closed, type-$|\tau|_b$ HTP $\mathbf{p}$ *such that* $|[\![\mathrm{P}]\!]_{\mathbf{BCont}}|_b \leq [\![\mathbf{p}]\!]_{\mathrm{TLen}}$.

(b) There is a Seth bound Δ *that derives a bounding judgement* $\emptyset; \emptyset; \mathrm{K} \vdash^{\tau} |\mathrm{P}|_r \leq \mathbf{b}$ *such that, for all* $\rho \in [\![\mathrm{K}/\Delta]\!]_{\mathbf{Full}}$, $|[\![\mathrm{P}]\!]_{\mathbf{Full}}\, \rho|_r \leq [\![\mathbf{b}]\!]_{\mathbf{FL}}\ |\rho|_r$.

Given an arbitrary BTLP-procedure with no free variables, it follows from Corollary 41(a) that, over BCont arguments, this procedure determines a continuous functional F with $|F|_b$ total. Hence we have:

Corollary 42. *Suppose* P *is a* BTLP *procedure of type* τ *with no free variables. Then* $[\![\mathrm{P}]\!]_{\mathbf{BCont}} \in \mathrm{BCont}_\tau$.

With this corollary in hand, we can now formally introduce the bounded continuous basic feasible functionals.

Definition 43.

(a) For each τ, a simple type over N, define BCBFF_τ to be the subset of BCont_τ computed by type τ BTLP-procedures over **BCont** arguments. We refer to the elements of the BCBFF_τ's as the *bounded continuous basic feasible functionals.*

(b) Let **BCBFF** be the category formed from the closure of the BCBFF_σ's under finite products. ◇

It is standard that **BCBFF** is cartesian closed and a subcategory of **TCont**.

12. Flattening ITLP

Section I-8 established a close relationship between ITLP_2 and our standard machine model for the class BFF_2. We would like to use ITLP to help formulate reasonable machine models for the class BFF, but ITLP itself is not really suitable for this task. The difficulty is that in a machine model one wants each basic action (e.g., a computation step, a procedure/oracle call, ...) to have fairly low computational complexity. But, as noted in Remark 33, there are certain "small" ITLP-typable terms that necessarily involve enormous tier levels, and hence enormous complexity. Because of this, ITLP cannot be close to any reasonable machine model.

In place of ITLP we consider $\mathsf{ITLP}^\flat$, a restriction of ITLP which eliminates the problematic, explosive ITLP-terms—along with a host of other relatively harmless ones. (Intuitively, $\mathsf{ITLP}^\flat$ is what is left of ITLP after the *IMF* reforms to cut the rate of inflation.) Even though $\mathsf{ITLP}^\flat$ is a much more restrictive language than ITLP, we show that each ITLP program can be translated into an equivalent $\mathsf{ITLP}^\flat$ program. As desired, $\mathsf{ITLP}^\flat$ turns out to be closely related to a machine model—*E-machines* (Section 15.1), our version for one of Seth's machine models for higher-type feasible functionals. Thus this translation result is the second link in our grand chain of translations.

The $\mathsf{ITLP}^\flat$ system

To obtain $\mathsf{ITLP}^\flat$ from ITLP, we first replace (11), ITLP's typing rule for procedure applications, with:

$$\frac{\mathsf{v}\colon \sigma_0 \times \cdots \times \sigma_n \to_{i+1} \mathrm{N} \quad a_0\colon (\sigma_0, d_0) \quad \cdots \quad a_n\colon (\sigma_n, d_n)}{\mathsf{v}(a_0, \ldots, a_n)\colon \mathrm{N}_{i+d+1}} \tag{14}$$

where $d = \max(d_0, \ldots, d_n)$ and where, for $j = 0, \ldots, n$, if σ_j is an arrow type, then we *must* have $d_j = 0$. $\mathsf{ITLP}^\flat$ consists of those ITLP procedures that are typable under this revised type-system. We shall sometimes refer to $\mathsf{ITLP}^\flat$ as a subsystem of ITLP. Strictly speaking, this is not the case—there are terms for which $\mathsf{ITLP}^\flat$'s typing rules assign types of lower depth than those assigned by the ITLP rules. This new typing regime is quite confining, but it does not restrict computational power.

Proposition 44 (The ITLP to $\mathsf{ITLP}^\flat$ Translation). *Suppose that* P *is an* ITLP *procedure of type* (σ, i) *with global references. Then from* P*, one can effectively construct an* $i' \in \omega$ *and a* $\mathsf{ITLP}^\flat$ *procedure* P' *of type* (σ, i') *(also with no global references) such that* $[\![\mathsf{P}]\!]_{\mathbf{Full}} = [\![\mathsf{P}']\!]_{\mathbf{Full}}$ *and similarly for the* **BCont**-*based semantics.*

Proof Sketch. An example translation will suffice to indicate what is required in general. Consider the ITLP program given in Figure 4. The only

Procedure Before $(\mathsf{F}\colon (\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N},\ \mathsf{f}\colon \mathrm{N} \to \mathrm{N},\ \mathsf{g}\colon \mathrm{N} \times \mathrm{N} \to \mathrm{N},\ \mathsf{x}\colon \mathrm{N})\colon \mathrm{N}_{12}$

  Procedure h $(\mathsf{u}\colon \mathrm{N})\colon \mathrm{N}_1$
    var $\mathsf{v}\colon \mathrm{N}_1$; $\mathsf{u} := \mathsf{c_0}(\mathsf{u})$; $\mathsf{v} := \mathsf{f}(\mathsf{u})$; **Return** v
  End (* *of* h *)

  Procedure H $(\mathsf{p}\colon \mathrm{N} \to \mathrm{N},\ \mathsf{u}\colon \mathrm{N})\colon \mathrm{N}_2$
    var $\mathsf{v}\colon \mathrm{N}_1,\ \mathsf{w}\colon \mathrm{N}_2$; $\mathsf{v} := \mathsf{p}(\mathsf{u})$; $\mathsf{w} := \mathsf{F}(\mathsf{p}, \mathsf{v})$; **Return** w
  End (* *of* H *)

  var $\mathsf{y}\colon \mathrm{N}_3,\ \mathsf{z}\colon \mathrm{N}_{12}$
  $\mathsf{y} := \mathsf{H}(\mathsf{h}, \mathsf{x})$; (* *The first call to* H. *)
  $\mathsf{z} := \mathsf{H}(\lambda \mathsf{a}\colon \mathrm{N}.\mathsf{g}(\mathsf{a}, \mathsf{y}), \mathsf{x})$; (* *The second call to* H. *)
  Return z
End (* *of* Before *)

Figure 4: A sample ITLP procedure

problematic parts of this procedure are the two calls to H. To translate this procedure to $\mathsf{ITLP}^\flat$, we create a new version of H for each call so that we can replace the old call with an $\mathsf{ITLP}^\flat$-legal one. Our translation is given in Figure 5. Note that the translation eliminates all global references in the subprocedures. The key idea of the translation is very simple: each new call passes the raw material (integer and depth-0, higher-type arguments) to new versions of H to reconstruct the equivalents of the original higher-type arguments of positive depth. Note that these reconstructions may involve new λ-expressions, e.g., the $\lambda \mathsf{a}\colon \mathrm{N}.\mathsf{h}(\mathsf{f}', \mathsf{a})$ in H_1. The general translation algorithm is just a systemization of these simple tricks. □

Corollary 45. *With respect to either the* **Full**- *or* **BCont**-*based semantics, the class of* $\mathsf{ITLP}^\flat$-*computable functionals equals the class of* ITLP-*computable functionals.*

Note that there is a conservation of explosions—even though the translations of the above proposition are simple, their result can be enormously longer than the original.

We will return to $\mathsf{ITLP}^\flat$ in Section 15 in connection with E-machines. Before that, we examine in the next section a second variation on ITLP that

Procedure After (F: (N → N) × N → N, f: N → N, g: N × N → N, x: N): N_4
  Procedure H_1 (F′: (N → N) × N → N, f′: N → N, u: N): N_2
    Procedure h (f″: N → N, u: N): N_1
      var v: N_1;
      u := c_0(u); v := f″(u); **Return** v
    End (* *of* h *)
    var v: N_2, w: N_2;
    v := h(f′, u); w := F′(λa: N. h(f′, a), v); **Return** w
  End (* *of* H_1 *)
  Procedure H_2 (F′: (N → N) × N → N, g′: N × N → N, y′: N, u: N): N_2
    var v: N_1, w: N_2;
    v := g′(u, y′); w := F′(λa: N. g(a, y′), v); **Return** w
  End (* *of* H_2 *)
  var y: N_2, z: N_4
  y := H_1(F, f, x); (* *The equivalent of the first call to* H. *)
  z := H_2(F, g, y, x); (* *The equivalent of the second call to* H. *)
  Return z
End (* *of* After *)

Figure 5: The translation of Before

leads us outside of the class of ITLP-computable functionals.

13. Adding Higher-Type Downward Coercions

Seth [Set94, Set95] introduced two machine models for computing higher-type feasible functionals. We shall introduce our versions of these models, D-*machines* and E-*machines*, in Section 15. $\mathsf{ITLP}^\flat$ was introduced as a counterpart to the E-machine model. Here we introduce $\mathsf{ITLP}^\sharp$, an extension of $\mathsf{ITLP}^\flat$, to capture part of the idea behind D-machines.

Motivations

We obtain $\mathsf{ITLP}^\sharp$ by adding a form of higher-type downward coercion to $\mathsf{ITLP}^\flat$. Here is the idea: Extend the grammar of $\mathsf{ITLP}^\flat$ to permit λ-terms of the sort

$$\lambda \mathsf{y}\,\textbf{.}\, \mathsf{down}\,(\mathsf{bnd}(\mathsf{y}), \mathsf{g}(\mathsf{x},\mathsf{y})) \tag{15}$$

where, say, $\mathsf{g}\colon \mathrm{N}\times\mathrm{N} \to_1 \mathrm{N}$, $\mathsf{bnd}\colon \mathrm{N} \to_1 \mathrm{N}$, and $\mathsf{x}\colon \mathrm{N}_3$. In such an extension what type should this term be assigned? If we treat down as if it were of type, say, $\mathrm{N}\times\mathrm{N} \to_0 \mathrm{N}$, then the $\mathsf{ITLP}^\flat$ typing rules would assign (15) the type $\mathrm{N} \to_6 \mathrm{N}$. However, we know that, independent of the value of x,

$$\Big|\, [\![\lambda \mathsf{y}\,\textbf{.}\, \mathsf{down}\,(\mathsf{bnd}(\mathsf{y}), \mathsf{g}(\mathsf{x},\mathsf{y}))]\!]_{\mathbf{S}} \Big|_s \quad\le\quad \Big|[\![\mathsf{bnd}]\!]_{\mathbf{S}}\Big|_s$$

(where either $\mathbf{S} = \mathbf{BCont}$ and $s = b$ or else $\mathbf{S} = \mathbf{Full}$ and $s = r$). This suggests that a higher-type version of the typing rule for down-terms should assign (15) the same type as bnd. If we adopt this latter type assignment, then, given $\mathsf{F}\colon (\mathrm{N}\to\mathrm{N}) \to_0 \mathrm{N}$ and $\mathsf{z}\colon \mathrm{N}_2$, the assignment statement

$$\mathsf{z} := \mathsf{F}\left(\lambda \mathsf{y}\,\textbf{.}\, \mathsf{down}\left(\mathsf{bnd}(\mathsf{y}), \mathsf{g}(\mathsf{x},\mathsf{y})\right)\right)$$

would be permitted. As we will see, this opens the door to a new sort of higher-type mischief; specifically, with the right choices of g and F, the above assignment can convey a surprising amount of information about the (tier 3) value of x through the (tier 2) value of $\mathsf{F}(\lambda \mathsf{y}\,\textbf{.}\, \mathsf{down}(\mathsf{bnd}(\mathsf{y}), \mathsf{g}(\mathsf{x},\mathsf{y})))$.

The $\mathsf{ITLP}^\sharp$ system

$\mathsf{ITLP}^\sharp$ is obtained by making the following changes to $\mathsf{ITLP}^\flat$. First, we add the following production to $\mathsf{ITLP}^\flat$'s grammar:

$$A \;::=\; \lambda \mathsf{v}_0,\ldots,\mathsf{v}_k\,\textbf{.}\, \mathsf{down}\left(\mathsf{v}'(\mathsf{v}'_0,\ldots,\mathsf{v}'_\ell), \mathsf{v}''(\mathsf{v}''_0,\ldots,\mathsf{v}''_m)\right) \tag{16}$$

where the list $\mathsf{v}_0,\ldots,\mathsf{v}_k$ contains neither v' nor v''. We shall call these new terms *coerced applications* (although *accursed applications* would also be appropriate, given some of their properties). Second, we introduce the following typing rule for coerced applications:

$$\frac{\mathsf{v}_0\colon(\sigma_0,0) \quad \cdots \quad \mathsf{v}_n\colon(\sigma_n,0) \qquad \mathsf{v}'(\mathsf{v}'_0,\ldots,\mathsf{v}'_\ell)\colon \mathrm{N}_i \qquad \mathsf{v}''(\mathsf{v}''_0,\ldots,\mathsf{v}''_m)\colon \mathrm{N}_j}{\lambda \mathsf{v}_0,\ldots,\mathsf{v}_n\,\textbf{.}\, \mathsf{down}\left(\mathsf{v}'(\mathsf{v}'_0,\ldots,\mathsf{v}'_\ell), \mathsf{v}''(\mathsf{v}''_0,\ldots,\mathsf{v}''_m)\right) : \sigma_0\times\cdots\times\sigma_n \to_i \mathrm{N}}$$

```
Procedure H(F: (N → N) → N): N_2
  Procedure bnd(y: N): N_1  Return 1  End
  Procedure g(x: N, y: N): N_1  If x = y then Return 1 else Return 0
    End
  var w: N_2,  x: N_2;
  x := 2;
  For w := 1 to x do  x := F (λy. down (bnd(y), g(x, y)) );  Endfor;
  Return x
End
```

Figure 6: The $\mathsf{ITLP}^\sharp$ procedure H

where $i < j$. Finally, we revise the syntactic/typing restrictions on **For**-loops as follows. *Terminology:* In any expression E that includes a subexpression of the form "$\mathsf{down}(\mathsf{E}_1, \mathsf{E}_2)$," we say that each occurrence of a variable in E_2 is a *covered occurrence*; and if a variable v has an occurrence in E that is not covered, then we say that v is *uncovered in* E. Now, a **For**-loop of the form

$$\textbf{For } \mathsf{v}_0^{\mathrm{N}_j} := \underline{1} \textbf{ to } \mathsf{v}_1^{\mathrm{N}_j} \textbf{ do } \vec{I} \textbf{ Endfor}$$

must satisfy the following restrictions:

1. No assignments to $\mathsf{v}_0^{\mathrm{N}_j}$ occur within $\vec{I}$.
2. Each '$\mathsf{v}^{\mathrm{N}_i} := \mathsf{E}$' with $i \leq j$ occurring within $\vec{I}$ must be such that either
   (a) each *uncovered* integer variable in E is of tier less than i, or
   (b) E is of the form "$\mathsf{down}(\mathsf{x}^{\mathrm{N}_i}, \mathsf{y}^{\mathrm{N}_k})$."

(The only change from before is in 2(a).)

Before proceeding further we consider a disconcerting example.

Example 46. Consider the $\mathsf{ITLP}^\sharp$ procedure given in Figure 6—in which we take our usual liberty of employing a few unofficial constructs. Let $\mathcal{H}\colon ((\mathrm{N} \to \mathrm{N}) \to \mathrm{N}) \rightharpoonup \mathrm{N}$ denote the functional determined by H under the **Full** semantics and let $\mathcal{F}_0\colon (\mathrm{N} \to \mathrm{N}) \to \mathrm{N}$ be the (discontinuous) functional such that, for each $f\colon \mathrm{N} \to \mathrm{N}$,

$$\mathcal{F}_0(f) \quad = \quad \begin{cases} \mathbf{1}x, & \text{if } f = C_{\{x\}}; \\ 0, & \text{if, for all } x,\ \ f \neq C_{\{x\}}. \end{cases}$$

It is easily seen that in the computation of H on input $\mathcal{F}_0$, the value of x increases without bound. Hence, $\mathcal{H}(\mathcal{F}_0)$ is undefined. However, consider an $\mathcal{F} \in \mathrm{BCont}_{(\omega\to\omega)\to\omega}$. By the definition of $\mathrm{BCont}_{(\omega\to\omega)\to\omega}$, we have

$$\max\left(\left\{\, |\mathcal{F}(f)| \,\vdots\, |f| \le \lambda x \,\boldsymbol{.}\, \underline{1} \,\right\}\right) \quad = \quad |\mathcal{F}|_b(\lambda x \,\boldsymbol{.}\, \underline{1}) \quad < \quad \infty.$$

In the computation of the procedure H on input $\mathcal{F}$, the value of x cannot grow beyond $|\mathcal{F}|_b(\lambda x \,\boldsymbol{.}\, \underline{1})$, hence the procedure must halt and $\mathcal{H}(\mathcal{F})\!\downarrow$. In fact, H determines the polynomially-bounded member of $\mathrm{BCont}_{((\omega\to\omega)\to\omega)\to\omega}$, as follows from the next proposition. $\diamond$

Proposition 47. *Suppose* P *is an* $\mathsf{ITLP}^\sharp$ *procedure of type* $\tau = \sigma_0 \times \cdots \times \sigma_k \to_i \mathrm{N}$ *with free variables from* $\Gamma = \{\, \mathsf{y}_1\colon (\sigma'_1, 0), \ldots, \mathsf{y}_m\colon (\sigma'_m, 0) \,\}$. *Then, there is a type-*$|\tau|_b$*, depth-*i HTP *bound* $\mathbf{p}_\mathsf{P}$ *over* $|\Gamma|_b$ *such that for all* $\rho \in [\![\Gamma]\!]_{\mathbf{BCont}}$, $|[\![\mathsf{P}]\!]_{\mathbf{BCont}}\, \rho|_b \le [\![\mathbf{p}_\mathsf{P}]\!]_{\mathrm{TLen}}\ |\rho|_b$.

Proof. The proof is a straightforward modification of the argument for Proposition 36. $\square$

The above example and proposition cast $\mathsf{ITLP}^\sharp$ in a curious light. Under the **Full**-based semantics we have divergent $\mathsf{ITLP}^\sharp$ computations, yet under the **BCont**-based semantics, $\mathsf{ITLP}^\sharp$-computable functionals are polynomially-bounded in the same sense as $\mathsf{ITLP}^\flat$-computable functionals. It turns out that under the **BCont**-based semantics, $\mathsf{ITLP}^\sharp$-computable functionals are a strictly larger class than $\mathsf{ITLP}^\flat$-computable functionals. To state this separation, we introduce some terminology.

Definition 48.

(a) D^- is the class of *hereditarily total* $\mathsf{ITLP}^\sharp$-computable functionals over **Full**.

(b) E is the class of $\mathsf{ITLP}^\flat$-computable functionals over **Full**.

(c) BCD^- is the class of $\mathsf{ITLP}^\sharp$-computable functionals over **BCont**.

(d) BCE is the class of $\mathsf{ITLP}^\flat$-computable functionals over **BCont**. $\diamond$

It is evident that $\mathrm{E} \subseteq \mathrm{D}^-$ and $\mathrm{BCE} \subseteq \mathrm{BCD}^-$. We shall see in Propositions 58 and 59 that both of these containments are strict. In proving these separations it is convenient to work with machines for higher types, which is what we turn to next.

14. Prolegomena Machana

We now begin our study of machine models for higher-type feasible functionals. The present section examines some general problems in the construction of such models. The specifics of our models are covered in the next section. Before considering machine models in the present context, it is worth recalling the point of bothering with such things in any context at all.

Complexity classes and models of computation

The basic objects of study in computational complexity theory are *complexity classes*. Typically, three ingredients go into a definition of a complexity class:

1. a general *model of computation* (e.g., deterministic TMs, nondeterminism RAMs) together with a notion of *cost* for that model's computations,
2. a notion of the *size* of an input (e.g., the length of a string, the number of nodes and edges in a graph), and
3. a family of functions (e.g., polynomials, poly-logarithmic functions) on sizes that express *upper bounds* on computational costs.

The complexity class is then the collection of all things computable by means of the given model of computation with a cost that is no greater than allowed by some particular upper bound on the size of the input. It is almost always understood (but almost always unstated) that the computational/cost model is merely a convenient representative of a class of equivalent models, any of which would suffice as the model of computation for defining the class. (See Section I-3 for a fuller discussion of this.) The complexity class itself thus abstracts away from particular algorithms and petty details about particular models to capture the power inherent in a general class of computational models relative to a given collection of resource bounds. In contrast, a machine model used to define a complexity class is usually quite concrete and specific so as to support close reasoning about costs.

The models used for complexity-theoretic ends are thus generally chosen for the simplicity and clarity of their basic computational operations and associated costs. Ease and elegance of programming on these models is typically not a concern and, yes, it shows.

As we saw in Part I, to characterize BFF_2 it suffices to use the ingredients: (1) standard Oracle Turing machines (OTMs) with the answer-length cost model for queries, (2) the usual notion of length of elements of N and Kapron and Cook's notion of length for elements of $\mathrm{N}^k \to \mathrm{N}$ (Definition I-5), and (3) Kapron and Cook's notion of a second-order polynomial (Definition I-6). Justifying these choices was fairly straightforward, especially in contrast to the work we have had to go through in previous sections in justifying our choices for higher-type analogues of lengths and polynomials. Choosing and justifying appropriate computational models for type-level 3 and above is not so straightforward either.

Higher-type queries

The first and most basic question about higher-type machines is: What should the form of oracle queries be? Concretely, if we have an oracle for an $F\colon(\mathrm{N} \to \mathrm{N})\times\mathrm{N} \to \mathrm{N}$, how should we specify an $\mathrm{N} \to \mathrm{N}$ argument in a query? Machines deal in syntax, so it is natural to specify an $\mathrm{N} \to \mathrm{N}$ argument by writing down a piece of syntax that somehow names the $\mathrm{N} \to \mathrm{N}$ object we have in mind. To do this we shall follow Kleene (see, for example, [Kle60, Kle62]) and construct a machine-based indexing of a class of functionals at all simple types and name arguments to oracles by indices of this indexing. Thus, to query an oracle for F as above, we present the oracle with a $(i, x) \in \mathrm{N}^2$ ($\equiv (\{\,\mathbf{0},\mathbf{1}\,\}^*)^2$) where i is an appropriated code of some machine that computes some $g\colon\mathrm{N} \to \mathrm{N}$. Note that the index i may describe a program that references the oracles present in its defining context, e.g., the above i could be an index of $\lambda y\,\boldsymbol{.}\,F(\lambda z\,\boldsymbol{.}\,(z{+}1)^2, y)$. While the query (i, x) lacks the decorum of, say, an ITLP oracle/procedure call $\mathsf{F(g,x)}$, both query methods offer the oracle bits of syntax in place of an actual element of $\mathrm{N} \to \mathrm{N}$, so one is no more silly than the other. The disadvantage of the procedure-call form of query here is that it implicitly begs several questions that must be addressed to justify our models.

Restrictions on query indices

Having committed to using dodgy indices, the first thing we discover is that one cannot allow arbitrary indices to be used in queries. Here is an example of the problem (based on a similar example in [Kap91]). Suppose $\Phi\colon((\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \{\,0,1\,\}) \times \mathrm{N} \to \{\,0,1\,\}$ is given by:

$$\Phi(F,x) \quad = \quad F(\lambda y\,\boldsymbol{.}\,2^x, x).$$

One can "feasibly" compute Φ as follows. On input F and x, construct j_x, an index for $\lambda y\,\boldsymbol{.}\,2^x$, submit (j_x, x) to the oracle for F, and output the 0–1 result of this query. Given a reasonable indexing, the computation of j_x from x can be done in time linear in $|x|$. Hence, everything in our sketch is "feasible." To see the difficulty with this Φ, first let C be a set decidable in $O(2^n)$ time that is not in P. Let $g_C\colon \mathrm{N}^2 \to \{0,1\}$ be a member of PF such that, for all x,

$$x \in C \quad \Longleftrightarrow \quad g_C(2^x, x) = 1.$$

(Proving such a g_C exists is straightforward.) Now let $G\colon (\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \{0,1\}$ be given by

$$G(f,x) \quad = \quad g_C(f(0),x).$$

Clearly, G is basic feasible. Hence, if Φ is feasible in any reasonable sense, then $\lambda x\,.\,\Phi(G,x)$ (= the characteristic function of C) should be in PF, which it most certainly is not.

This example suggests that if we want a guarantee that a particular machine **M** computes a feasible functional, then one of the things we must do is restrict **M**'s query indices so that they themselves determine feasible functionals. We conclude from this that, in the present context, our "general model of computation" ingredient cannot be so general after all, at least on the matter of the indices used in queries. Put another way, for type-3 and above it seems that our choices of our three ingredients for a complexity class cannot be as independent as are the cases for the type-1 and type-2 settings. In the type-1 and type-2 settings one can specify a machine model *and then* introduce feasibility bounds to restrict machine run times. In this setting we have to have a notion of feasibility in place before we allow any of our supposedly unrestricted machines to make its first query. For these reasons, the indexing we referred to above will be based on "feasibly clocked machines" defined inductively in a reasonably straightforward manner.

Uniform feasibility bounds

Our general strategy thus is that, in making queries, machines for feasible functionals may use only those indices that are themselves descriptions of machines for feasible functionals. Naively applied this strategy does not eliminate Φ as infeasible, since $\lambda y\,.\,2^x$ is a constant function and hence linear (in $|y|$) time computable. As Seth [Set95, Set94] points out, a way around this particular difficulty is to impose a uniformity requirement for the feasible bounds on the things named by the query indices. Our argument that $\lambda y\,.\,2^x$ is feasible was that for each value of x there is a polynomial p_x such that, for all y, 2^x is computable within time $p_x(|y|)$. But, for any reasonable notion of higher-type polynomial, it *should not* be the case that there is a fixed polynomial bound $\mathbf{p}$ such that, for all F, x, and y, 2^x is computable within time $\mathbf{p}$(the size of $F, |x|, |y|$).

Therefore, given a machine $\mathbf{M}$ that has its run time bounded by some "polynomial" $\mathbf{p}$ in the sizes of its arguments $\vec{x}$, we shall require that for each type τ there is a fixed "polynomial" $\mathbf{p}_\tau$ (in the sizes of $\vec{x}$ and perhaps some other arguments) that bounds the run time of all query indices coding machines for computing objects of type τ. Moreover, we require that the machines coded by the query indices satisfy these same requirements (using the same $\mathbf{p}_\tau$'s) on their own query indices.[10] These requirements are quite restrictive, but anything more permissive is open to the difficulty illustrated by the Φ example.

Recursion through the back door

The above uniformity requirements do not banish all difficulties. Here is an example of a different sort of trouble. Suppose $\Psi\colon ((\mathrm{N}^2 \to \mathrm{N})\times\mathrm{N}^2 \to \mathrm{N})\times(\mathrm{N} \to \mathrm{N}) \times \mathrm{N}^2 \to \mathrm{N}$ is given by:

$$\Psi(F,f,x,y) \quad = \quad \begin{cases} y, & \text{if } x = 0; \\ z, & \text{if } x > 0 \text{ and } \max(|f(y)|,|z|) \le |y|, \text{ where} \\ & \qquad z = F(\lambda u,v\,\boldsymbol{.}\,\Psi(F,f,u,v), x-1, f(y)); \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

It follows from this definition that, for all F, f, x, and y, $|\Psi(F,f,x,y)| \le |y|$. We can compute Ψ as follows.

Program sketch for $\mathbf{M}_\Psi$.

Input (F,f,x,y).
If $x = 0$ **then** output y and halt.
$w :=$ the result of querying the oracle for f on y.
If $|w| > |y|$ **then** output 0 and halt.
$e :=$ a particular index for $\lambda u,v\,\boldsymbol{.}\,\mathbf{M}_\Psi(F,f,u,v)$. *(More on this shortly.)*
$z :=$ the result of querying the oracle for F on $(e, x-1, w)$.
If $|z| \le |y|$ **then** output z **else** output 0.

End

[10] We are also assuming that, for a given machine, there is a fixed, *a priori*, finite number of types that can arise in any of the computations of this machine, and the machines specified by its query indices, or the machines specified by the query indices of the machines specified by the original machine's query indices, and so on.

The index e can be constructed through an application of an appropriate version of Kleene's recursion theorem [Rog67, Odi81, RC94]) through which we can arrange that the run time of (the machine named by) e on input (u, v) is no greater than a constant plus the run time of $\mathbf{M}_\Psi$ on (F, f, u, v). Clearly, $\mathbf{M}_\Psi$ computes Ψ. Let us sketch an argument that it does so "feasibly."[11] For simplicity, let us assume for the moment that our machines are restricted to arguments from **BCont**. In the sketch of $\mathbf{M}_\Psi$ everything outside of the query to F is perfectly feasible. We know that e determines an $h\colon \mathrm{N}^2 \to \mathrm{N}$ such $|h(u, v)| \leq |v|$ for all u and v. It thus follows that the size of the result of the query is bounded by an HTP bound in $|F|_b$, $|f|_b$, $|x|$, and $|y|$, and hence that the run time of $\mathbf{M}_F$ on (F, f, x, y) is no larger than the value of some HTP bound in $|F|_b$, $|f|_b$, $|x|$, and $|y|$. By our remark above on the run time of e, it follows that, for all F, f, x, y, u, v, the machine named by e has a run time on (u, v) that is within an HTP bound in $|F|_b$, $|f|_b$, $|u|$ and $|v|$. Hence, it follows that $\mathbf{M}_\Psi$ satisfies a uniformity requirement on query indices as discussed above. Therefore, we must accept $\mathbf{M}_\Psi$ as feasible by our current criteria.

To see why $\mathbf{M}_\Psi$'s feasibility is suspect, first suppose $A_2\colon (\mathrm{N}^2 \to \mathrm{N}) \times \mathrm{N}^2 \to \mathrm{N}$ is given by: $A_2(g, u, v) = g(u, v)$. Clearly, A_2 is basic feasible. Next let $G = \lambda f, x, y \,\boldsymbol{.}\, \Psi(A_2, f, x, y)$. Since Ψ and A_2 are "feasible," so is G. But

$$G(f, x, y) \quad = \quad \begin{cases} y, & \text{if } x = 0; \\ z, & \text{if } x > 0 \text{ and } \max(|f(y)|, |z|) \leq |y|, \\ & \quad \text{where } z = G(f, x-1, f(y)); \\ 0 & \text{otherwise.} \end{cases}$$

Thus G expresses a length-bounded version of primitive recursion on f. This seems worrisome, especially after one notes that with appropriate choices of $f, g, h \in \mathrm{PF}$, one can have $\lambda w \,\boldsymbol{.}\, G(f, g(w), h(w))$ decide any predetermined problem in PSPACE. (Proving this is straightforward.) PSPACE is certainly *not* the brand of feasibility for which we are in the market.

The general opacity of programs

The above tale of trouble features many suspicious characters. Our problem now is to find the guilty parties among them. If we look at the definition of Ψ given in (17), the presence of a primitive recursion is quite clear. Moreover,

[11] **N.B.** To fully justify this sketch takes a bit of work which, for obvious reasons, we omit. In particular, the indexings of the next section are *not* suitable for this argument.

if we alter (17) to change the primitive recursion to a recursion on notation, the revised version of Ψ is easily seen to be basic feasible. From this it may seem that imposing the usual restrictions on recursions is all that is required to fix the problem. However our argument for the "feasibility" of Ψ was not through (17), but through our sketch of $\mathbf{M}_\Psi$, and a careful look at this sketch reveals the true depth of the trouble. Our description of the construction of e clearly stated what e computes, but in general all that we currently require of a query-index i is that i satisfy certain run-time constraints. Rice's Theorem and its subrecursive variants (see for example Kozen's work [Koz80]) strongly suggest that in the present setting if those constraints are all we require of such an i, then in general we cannot expect to know much more about the behavior of the machine that i codes. An innocent looking query, such as $(e, x-1, w)$ in $\mathbf{M}_\Psi$, may thus entail huge recursions and we have no hope of detecting this under our current constraints.

Levels

To remedy this new problem we follow Seth [Set95, Set94] in stratifying our clocked machines. Each clocked machine (and indices for them) will have an associated *level*, an element of ω. A clocked machine of level ℓ is restricted to using only those query indices with levels less than ℓ. (This is roughly analogous to our prohibition of recursive calls in BTLP and ITLP.) As with the requirement of uniform polynomial bounds, this stratification into levels is quite restrictive. Less restrictive requirements are clearly possible here, but these would seem to involve a treatment of feasible higher-type recursion—a topic avoided in this paper.

Bounding the size of queries

The third, and by far most problematic, restriction concerns measuring the "size" of queries in determining clock bounds. To explain the underlying difficulty, we first briefly restate the clocking scheme of Section I-6 for OTMs computing type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ functionals. We then consider the problem that arises when one tries to lift this clocking scheme to higher type-levels and query indices enter the picture.

The type-2 clocking scheme

We start with $\langle \mathbf{M}_i \rangle_{i \in \mathrm{N}}$, an indexing of arbitrary OTMs of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \rightharpoonup \mathrm{N}$. Then, for each i, k, and d, we let the clocked machine $\mathbf{M}_{i,k,d}$ be an OTM that, on input (f, x), runs $\mathbf{M}_i$'s program step-by-step and, over the course of this $\mathbf{M}_i$-computation, keeps updating the following values: *cost* = the current accumulated cost of the $\mathbf{M}$-computation; $b[0] = k \cdot (|x| + 1)^k$, and for each $i = 1, \dots, d$,

$$b[i] \quad = \quad \max \left\{ k \cdot (|a| + 1)^k \;\middle|\; \begin{array}{l} a = x \text{ or } a \text{ is the answer to a query} \\ \text{“}f(z) = ?\text{” with } |z| \leq b[i-1] \text{ that} \\ \text{was made at or before this point} \\ \text{in the } \mathbf{M}\text{-computation} \end{array} \right\}. \tag{18}$$

If *cost* $> b[d]$ after a set of updates, then $\mathbf{M}_{i,k,d}$ halts with output 0. (Thus, $\mathbf{M}_{i.d.k}$ cannot run forever since *cost*'s value increases by at least one at every $\mathbf{M}_i$-step and since $b[d]$'s value can never exceed $q^{d,k}(|f|, |x|)$—see Definition I-10.) If the $\mathbf{M}_i$-computation runs to completion without incurring a cost overrun, then $\mathbf{M}_{i,d,k}$ outputs the result of this $\mathbf{M}_i$-computation and halts.

In the above scheme the value of $b[i+1]$ depends on the answers to queries of size no greater that $b[i]$, where the “size” of a query “$f(z) = ?$” is simply $|z|$. If we extend this scheme to handle machines with oracles of type $\mathrm{N}^c \to \mathrm{N}$, then the size of a query “$f(z_1, \dots, z_c) = ?$” can be simply $\max(|z_1|, \dots, |z_c|)$. This is all very straightforward and meshes well with the notions of the length of a type-1 function and second-order polynomial bounds. But now let us consider what happens above type-level 2.

Clocking and query size at higher type-levels

We want to set up an analogous clocking scheme for machines for higher type-levels. We thus have the question of how to measure the size of queries when query indices are involved. By the uniform-polynomial-bounds restriction, a query index, q, is already bounded—where the bound is on the complexity of the thing q that names, not $|q|$. Moreover, as such a q is merely a name of the thing of interest (the higher-type object we want as a parameter), it seems odd to consider bounding the size of this name. Thus one option here is not to bother with bounds on query indices in the analogues of (18). This road leads us to the D-machine model (Section 15.2), which, in turn, leads us to very strange territory. For example, under the **Full**-semantics, clocked D-machines on total arguments may run forever. Even under the **BCont**-semantics, where this totality problem does not arise, it turns out that the collection of functionals one can now compute are *far* outside the basic feasible functionals (Section 17).

The other obvious option here is to bound query indices just like type-N queries in the analogues of (18). This choice still has an air of the *ad hoc* about it, but it leads us to the E-machine model (Section 15.1) which turns out to characterize the basic feasible functionals (Section 16).

Our machine models

We have identified the three major restrictions we shall place on our machines: uniform polynomial bounds, levels, and query size bounds—each of which derives from Seth's work [Set95, Set94]. We are now in a position to lay out our machine models and their indexings, which we do in the next section. We note that our machine models are similar in many respects to the higher-type machine models of Kleene (see, for example, [Kle63, Kle60, Kle62]), of Savonov [Saz76], and, of course, of Seth [Set95, Set94].

15. D- and E-Machines

Oracles and queries

Our basic model of computation is essentially the multi-tape deterministic oracle Turing machine of Part I with some changes in how oracles and type-N inputs are handled. Each argument of a machine is treated as an oracle, including the type-N arguments, as explained below. A machine may also have a finite number of additional oracles, called *context oracles*, of various types. Suppose $\mathbf{M}$ is a machine with u many context oracles and that $\mathbf{M}$ determines a function of type $\sigma_u \times \sigma_{u+1} \times \cdots \times \sigma_v \rightharpoonup \mathrm{N}$, where $u \leq v$. Each oracle is assigned a number: the context oracles receive numbers 0 through $u-1$, and the arguments of the machine are, in order, assigned numbers u through v. Two special tapes are involved in oracles queries: the *query tape*, and the *answer tape*. To query an oracle, $\mathbf{M}$ writes a well-formed query on the query tape and issues the query—the OTM's query instruction indicates the number of the oracle being queried. In response to the query, the query tape is blanked and the oracle's reply is placed on the answer tape. Whether a query is well-formed depends on the type of the oracle and the constraints on $\mathbf{M}$. For an oracle of type N, the only well-formed query is $[\,]$. (A type-N oracle is thus more properly on oracle of type $() \to \mathrm{N}$.) For an oracle of type $\gamma_1 \times \cdots \times \gamma_c \to \mathrm{N}$, a well-formed query is of the form $[z_1, \ldots, z_c]$ where for each j, if γ_j is an arrow type, then z_j is the index of a machine $\mathbf{M}'$ that: (a) determines a function of type γ_j, (b) has $(v+1)$-many context oracles corresponding directly to the $(v+1)$-many oracles available to $\mathbf{M}$, and (c) is subject to some additional restrictions depending on $\mathbf{M}$ as discussed below. If $\mathbf{M}$ ever makes an ill-formed query or specifies an oracle with a number greater than v, then at that point $\mathbf{M}$ halts with output 0. The cost of a query is, as in Part I, the length of the answer.

15.1. E-Machines

Conventions. In the following, let i range over N; k, d, and ℓ range over ω; $\vec{\sigma}$ range over finite (and possibly empty) sequences of simple types; and τ range over simple arrow types. If $\mathbf{M}$ is an OTM with u-many context oracles of respective types $\sigma_0, \ldots, \sigma_{u-1}$, and $\mathbf{M}$ determines a function of type $\sigma_u \times \cdots \times \sigma_v \rightharpoonup \mathrm{N}$ and $x_0 \in \mathrm{Full}_{\sigma_0}, \ldots, x_{u-1} \in \mathrm{Full}_{\sigma_{u-1}}$, $y_u \in \mathrm{Full}_{\sigma_u}, \ldots, y_v \in \mathrm{Full}_{\sigma_v}$, then $\mathbf{M}(\vec{x}; \vec{y})$ denotes the result of running $\mathbf{M}$ with respective context oracles for $\vec{x}$ and respective input oracles for $\vec{y}$. (Note: Undefined is a possible result of the computation.) Until further notice, we fix i, k, d, ℓ, $\vec{\sigma} = \sigma_0, \ldots, \sigma_{u-1}$, and $\tau = \sigma_u \times \cdots \times \sigma_v \to \mathrm{N}$, where $u \le v$ and $v > 0$. (If $u = 0$, then $\vec{\sigma} = ()$.)

We define two machines $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ and $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ for which the parameter i describes a program, k and d together determine certain clock bounds, ℓ sets the level of the machine, $\sigma_0, \ldots, \sigma_{u-1}$ are the respective types of the context oracles numbered 0 through $u-1$, and τ describes the type of the function determined by the machine. The $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$'s are what we shall call E-*machines*.

A description of $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$

The machine $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is either an *oracular machine* (when i is even) or a *procedural machine* (when i is odd).

The oracular case. Suppose $i = 2 \cdot i'$. We say that $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is a *proper* oracular machine provided $i' < u$ and $\sigma_{i'} = \tau$. (Hence, $u > 0$ and $\sigma_{i'}$ is an arrow type.) The intended semantics is that $\widetilde{\mathbf{E}}^{\vec{\sigma};\tau}_{i,k,d,\ell}(\vec{x}; \vec{y}) = x_{i'}(\vec{y})$ for $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ proper, and $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}(\vec{x}; \vec{y}) = 0$ otherwise. In either case, $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ determines a function of type τ. Oracular machines provide a means of naming oracles with indices, so these oracles may serve as arguments in queries to other oracles.

The procedural case. Now suppose that $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is procedural, in which case $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ corresponds to an OTM that determines a function of type $\sigma_u \times \cdots \times \sigma_v \rightharpoonup \mathrm{N}$. $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ has no constraints on its run time, but it *is* constrained in its use of query indices as follows. If $\ell = 0$, then no use of query indices is legal for $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, so the only well-formed oracle queries allowed are those to oracles of type-levels 0 and 1. If $\ell > 0$, then the legal query indices for $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ are those that describe machines of the form $\mathbf{E}^{\vec{\sigma}',\tau'}_{i',k,d,\ell-1}$ where $i' \in \mathrm{N}$, $\vec{\sigma}' = \sigma_1, \ldots, \sigma_u, \sigma_{u+1}, \ldots, \sigma_v$, and τ' is some arrow type.

A description of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$

As with $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is either oracular (when i is even) or procedural (when i is odd). The oracular case is identical to that for $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$. So suppose $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is procedural, in which case $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ corresponds to an OTM that is a clocked version of $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$. The clocking works as follows. $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ runs the program of $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ step-by-step and, over the course of this $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation, keeps track of the following $(d+2)$-many values:

$$
\begin{aligned}
\mathit{cost} \quad &= \quad \text{the current accumulated cost of the } \widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}\text{-computation.} \\
b[0] \quad &= \quad \max\left\{ k\cdot(|a|+1)^k \;\middle|\; \begin{array}{l} a \text{ is the answer to a query } [\,] \text{ to a} \\ \text{type N oracle made at or before} \\ \text{this point of the computation} \end{array} \right\}. \\
b[i] \quad &= \quad \max\left\{ k\cdot(|a|+1)^k \;\middle|\; \begin{array}{l} a \text{ is the answer to a query } [z_1, \ldots, \\ z_c] \text{ to an oracle made at or be-} \\ \text{fore this point of the computation} \\ \text{where } |z_1|, \ldots, |z_c| \le b[i-1] \end{array} \right\},
\end{aligned}
$$

where $0 < i \le d$. (Note: When $c = 0$, $[z_1, \ldots, z_c] = [\,]$. So the maximization for $b[i]$ above includes queries to type N oracles.) The values of *cost*, $b[0], \ldots, b[d]$ are updated as needed after each $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation step. If it is ever the case that after an update we have $\mathit{cost} > b[d]$, then $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ halts with output 0. If, on the other hand, the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation runs to completion without incurring a cost overrun, then $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ outputs the result of this computation and halts. The third case, that the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation runs forever without incurring a cost overrun, cannot happen, as shown by a simple, finite induction argument that we leave to the reader. (See Section 6 for a similar argument.) Hence, $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is total and determines a function of type τ. The clocking as described above can be done with only polynomial overhead. That is, there is (independent of i, $k, \ldots, \tau$) a polynomial q such that, at any point in running the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation, the cost of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$'s computation is no more than $q\big(k, d, \text{the cost of the } \widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}\text{-computation}\big)$.

Remark 49. Some care must be taken in achieving the aforementioned polynomial overhead. The problem is that since $(k, n) \mapsto k \cdot (n+1)^k$ is exponential in k, we have to be a bit indirect in using the clock bounds. The key observation is that there is an ordinary TM M and a constant c_0 such that on input $\langle k, n\rangle$ (where k and n are understood to be tally strings), (i) M outputs the tally string $\underline{k \cdot (n+1)^k}$, and (ii) M runs within $c_0 \cdot (k \cdot (n+1)^k + n + 1)$ steps. So, given $t, k, n \in \omega$, to test whether $t \leq k \cdot (n+1)^k$, we run M on input $\langle k, n\rangle$ for $c_0 \cdot (t+n+1)$ steps. If the computation fails to halt within that number of steps, then $t < k \cdot (n+1)^k$. If the computation did halt, then we compare t to the output of that computation and report the result. Clearly, this comparison can be done within time polynomial in $|t| + |k| + |n|$. So, by a straightforward elaboration of this standard trick, we can achieve the polynomial overhead in the clocking. $\diamond$

Along with the restrictions imposed by the clocking, $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ adds one more restriction to the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation. That is, if at any point in carrying out the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation there is an attempt to issue a query $[z_1, \ldots, z_c]$ with $\max(|z_1|, \ldots, |z_c|) > b[d-1]$, then at that point $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ halts with output 0. As a consequence of this query-size restriction, the answer to every (completed) query is taken into account in the clocking, and this turns out to considerably simplify several arguments below. Moreover, this addition of the query-size restriction is essentially not any more limiting than simply clocking by itself. Here is what we mean by this: Suppose we have a version of our E-machines that ignores this query-size restriction. Then these two forms of E-machines are effectively inter-translatable. (This follows from Lemma 51 below.) We shall consider the enforcement of this query-size restriction as part of the clocking. Its addition does not change the fact that $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ has only polynomial overhead.[12]

OTM programs

Conventions. Past this point, i, d, k, ℓ, $\vec{\sigma}$, and τ are no longer fixed. Recall the convention that we can treat ω as the subset of N consisting of those elements

[12] In Seth's formulation of his version of E-machines, he adds another restriction on queries. He states that an E-machine "can use only a fixed finite set of (parameterizable) functionals for querying. Furthermore, this is also required to hold for all machines which compute functionals used for querying." Roughly, this means Seth's E-machines look rather like $\mathsf{ITLP}^\flat$ programs in that the subprocedures are fixed by the program and cannot be constructed in the course of the computation as is the case with our version of the E-machine model.

of N that have their representation over $\{\,\mathbf{0},\mathbf{1}\,\}^*$ consisting of all $\mathbf{0}$'s.

In order to be precise about what an OTM program is, we now need to develop a few more details about OTMs. First, OTM tapes are numbered 0 to t, where the output, query, and answer tapes receive numbers 0, 1, and 2, respectively, and the work tapes receive numbers 3 through t. Thus, t varies from machine to machine but is always at least 2. Each OTM instruction is either of the form $(\mathit{state}_0, n_0, a_0, n_1, a_1, d_1, \mathit{state}_1)$ or $(\mathit{state}_0, j, \mathit{state}_1)$. The interpretation of an instruction of the first form is:

> If the current state is state_0 and the symbol on tape number n_0 is a_0, then write symbol a_1 on tape number n_1, then move that tape's head one square in direction d_1 (either *left* or *right*), and finally make state_1 the new current state.

The interpretation of an instruction of the second form is:

> If the current state is state_0, then issue a query to oracle number j and make state_1 the new current state.

We assume a straightforward coding of such instructions into **0**-**1**-strings. A *proper OTM program* is an element of N of the form: $2\cdot[m, \mathit{instr}_1, \mathit{instr}_2, \dots, \mathit{instr}_n]+1$, where $m\in\omega$ is the number of work tapes of the OTM (so the OTM has $m+3$ tapes total) and the instr_j's are encodings of OTM instructions. An *improper OTM program* is an odd element of N that is not a proper OTM program.

Universal Machines

The operational semantics of the $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d.\ell}$'s is given by a family of interpreters. Given $\vec{\sigma}=\sigma_0,\dots,\sigma_{u-1}$ and $\tau=\sigma_u\times\cdots\times\sigma_v\to \mathrm{N}$, we define $\mathbf{U}^{\vec{\sigma},\tau}_{\tilde{\mathbf{E}}}$ to be an OTM with u-many context oracles of respective types $\sigma_0,\dots,\sigma_{u-1}$ and that determines a function of type $\sigma_u\times\cdots\times\sigma_v\times \mathrm{N}\rightharpoonup \mathrm{N}$ as follows. Fix $p\in \mathrm{N}$ and $x_0\in \mathrm{Full}_{\sigma_0},\dots,x_v\in\mathrm{Full}_{\sigma_v}$. We have three cases, based on p, for the result of running $\mathbf{U}^{\vec{\sigma},\tau}_{\tilde{\mathbf{E}}}$ on input $\vec{x};\vec{y},p$.

Case 1: $p=[i,k,d,\ell]$, where $i=2\cdot[m,\mathit{instr}_1,\mathit{instr}_2,\dots,\mathit{instr}_n]+1$ is a proper OTM program. Then, the computation of $\mathbf{U}^{\vec{\sigma},\tau}_{\tilde{\mathbf{E}}}$ proceeds as follows. First, $\mathbf{U}^{\vec{\sigma},\tau}_{\tilde{\mathbf{E}}}$ sets up the simulation of the m work tapes on two of its own work tapes, sets the simulated current state to 0, and then goes off and interprets the OTM program for $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ in the obvious fashion with the following provisos.

1. If at some point multiple instructions in i apply, then $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ chooses the first applicable instruction within the $instr_1, instr_2, \ldots, instr_n$ sequence.

2. If at some point no instruction in i applies, then this is the halting condition for $\widetilde{\mathbf{E}}_{i,k,d,\ell}^{\vec{\sigma},\tau}$ and so $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ halts with the same output as $\widetilde{\mathbf{E}}_{i,k,d,\ell}^{\vec{\sigma},\tau}$.

3. If at some point an instruction commits a *faux pas* in issuing an ill-formed oracle query, or referring to a nonexistent work tape or the like, then at that point $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ halts with output 0.

Using standard techniques we can arrange that $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ carries out its work with only polynomial overhead. That is, there is a polynomial q_1 such that, for all p, at any point in simulating the $\widetilde{\mathbf{E}}_{i,k,d,\ell}^{\vec{\sigma},\tau}$-computation (where $[i,k,d,\ell] = p$), the cost of $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$'s computation is no more than $q_1\big(|p|$, the cost of the $\widetilde{\mathbf{E}}_{i,k,d,\ell}^{\vec{\sigma},\tau}$-computation$\big)$.

Case 2: $p = [2{\cdot}i, k, d, \ell]$ where $i < u$ and $\sigma_i = \tau$. Then $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ on input $\vec{x}; \vec{y}, p$ simply returns the answer to the query $[z_u, \ldots, z_v]$ to its i-th context oracle, where for $j = u, \ldots, v$, $z_j = y_j$ if $\sigma_i = \mathrm{N}$ and $z_j = [2 \cdot j, k, d, \ell]$ (an appropriate index for the j-th oracle), otherwise. Hence, $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}(\vec{x}; \vec{y}, p) = x_i(\vec{y})$ and the cost of the computation is no more than $q_2(|p|, |x_i(\vec{y})|)$ for some polynomial q_2.

Case 3: Neither Case 1 nor 2. Then $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ simply outputs 0. The cost of this computation is no more than $q_3(|p|)$ for some polynomial q_3.

For each $\vec{\sigma}$ and τ, we also define $\mathbf{U}_{\mathbf{E}}^{\vec{\sigma},\tau}$ to be a version of $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ that additionally carries out the clocking as in the description of the $\mathbf{E}_{i,k,d,\ell}^{\vec{\sigma},\tau}$ machines. By our discussions of the overheads of clocking and the $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma}',\tau'}$ machines, it follows that we may assume that the $\mathbf{U}_{\mathbf{E}}^{\vec{\sigma},\tau}$ machine carries out its work with only polynomial overhead.

Thus, for each $\vec{\sigma}$ and τ, $\mathbf{U}_{\tilde{\mathbf{E}}}^{\vec{\sigma},\tau}$ and $\mathbf{U}_{\mathbf{E}}^{\vec{\sigma},\tau}$ are universal programs for the $\widetilde{\mathbf{E}}_{i,k,d,\ell}^{\vec{\sigma},\tau}$ and $\mathbf{E}_{i,k,d,\ell}^{\vec{\sigma},\tau}$ machines, respectively, that have only polynomial overhead in run-time complexity.

The above sketches a **Full**-based (operational) semantics for the E-machines. The **BCont**-based operational semantics is identical, except that to show that the semantics is well-defined, we have to prove that an E-machine over **BCont** arguments determines a **BCont** functional. (This will be a consequence of Proposition 55.)

Three basic technical lemmas

The following structural lemma will help with constructing query indices.

Lemma 50 (S-m-n/Partial evaluation). *There is a polynomial-time computable* $s\colon \mathrm{N}^3 \to \mathrm{N}$ *as follows. Given:* i*, a proper OTM program;* m_1, m_2. n_1, $n_2 \in \mathrm{N}$ *with* $m_1 + m_2 > 0$ *and* $n_1 + n_2 > 0$*;* $\vec{\sigma} = \sigma_0, \ldots, \sigma_{u-1}$*;* $\tau = \gamma_1 \times \cdots \times \gamma_{m_1+n_1} \times \mathrm{N}^{m_2+n_2} \to \mathrm{N}$*;* k, d, $\ell \in \omega$*;* $a_1, \ldots, a_{m_1} < u$ *with* $\gamma_j = \sigma_{a_j}$ *for each* $j = 1, \ldots, m_1$*;* $v_0 \in \mathrm{Full}_{\sigma_0}, \ldots, v_{u-1} \in \mathrm{Full}_{\sigma_{u-1}}$*;* $x_{m_1+1} \in \mathrm{Full}\gamma_{m_1+1}, \ldots, x_{m_1+m_2} \in \mathrm{Full}\gamma_{m_1+m_2}$*; and* $y_1, \ldots, y_{m_2}, z_{m_2+1}, \ldots, z_{m_2+n_2} \in \mathrm{N}$*, we have*

$$\widetilde{\mathbf{E}}^{\vec{\sigma};\tau'}_{s(i,[\vec{a}],[\vec{y}]),k,d,l}(\vec{v};\vec{x},\vec{z}) \quad = \quad \widetilde{\mathbf{E}}^{\vec{\sigma};\tau}_{i,k,d,l}(\vec{v};\vec{w},\vec{x},\vec{y},\vec{z}), \tag{19}$$

where $\vec{w} = v_{a_1}, \ldots, v_{a_{m_1}}$ *and* $\tau' = \gamma_{m_1+1} \times \cdots \times \gamma_{m_1+n_1} \times \mathrm{N}^{n_2} \to \mathrm{N}$*. Moreover, the cost of running this* $\widetilde{\mathbf{E}}^{\vec{\sigma};\tau'}_{s(i,[\vec{a}],[\vec{x}]),k,d,l}$ *on* $\vec{v};\vec{x},\vec{z}$ *is no greater than* $c \cdot |i| \cdot \big($*the cost of running* $\widetilde{\mathbf{E}}^{\vec{\sigma};\tau}_{i,k,d,l}$ *on* $\vec{v};\vec{w},\vec{x},\vec{y},\vec{z}\big)$*, where* c *is a constant independent of* i, $m_1, \ldots$ *.*

Proof Sketch. Given i, m_1, $m_2, \ldots$ as above, let $s(i,[\vec{a}],[\vec{y}])$ be a straightforward modification of i that systematically redirects certain oracle queries and has the values $y_1, \ldots, y_{m_2}$ stored in its program text. The behavior of $i_* = s(i,[\vec{a}],[\vec{y}])$ differs from that of i only in making oracle queries and in constructing query indices. In regard to oracle queries:

1. Where i would query oracle number j, where $u \le j \le u + m_1$, i_* queries oracle number a_j.

2. Where i would query oracle number j, where $u + m_1 < j \le u + m_1 + n_1$, i_* queries oracle number $j - m_1$.

3. Where i would query oracle number j, where $u + m_1 + n_1 < j \le u + m_1 + n_1 + m_2$, i_* supplies the appropriate bits of $y_{j-u-m_1-n_1}$ as needed.

4. Where i would query oracle number j, where $u + m_1 + n_1 + m_2 < j \le u + m_1 + n_1 + m_2 + n_2$, i_* queries oracle number $j - m_1 - m_2$.

In regard to query indices, i_* has to revise the oracle references in the indices i would build so as to be consistent with the redirections noted above. With some care in programming, one can easily guarantee the complexity requirements. $\square$

In practice we shall allow ourselves to use a liberalized (and even more dreadful to state) version of Lemma 50 in which the "partially evaluated" arguments (i.e., w's and the y's) and the "unevaluated" arguments (i.e., the x's and the z's) can be mixed in any order in the right-hand side of (19).

Terminology. We say that an $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ is *well-behaved* if and only if in any computation of this clocked machine we have that (i) no cost overrun occurs, and (ii) each query, $[z_1,\ldots,z_c]$, made is well-formed and has $|z_1|,\ldots,|z_c| \leq b[d-1]$ at the time of the query. We say that an $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ satisfies a property *hereditarily* if and only if $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ satisfies the property and any query-indices used by $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ describe machines that satisfies the property hereditarily.

Lemma 51. *There is a polynomial-time procedure that, given i, d, k, ℓ, $\vec{\sigma}$, and τ, finds i' and k' such that $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ and $\mathbf{E}^{\vec{\sigma},\tau}_{i',d+\ell+1,k',\ell}$ determine the same functional and $\mathbf{E}^{\vec{\sigma},\tau}_{i',d+\ell+1,k',\ell}$ is hereditarily well-behaved.*

Proof Sketch. *Some observations.* Recall that our clocking scheme is such that there is (independent of i, $k,\ldots,\tau$) a polynomial q such that, at any point in running the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation, the cost of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$'s computation is no more than $q\big(k, d, \text{the cost of the } \widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}\text{-computation}\big)$. Also recall that checking for the appropriate form of well-formedness has a similar polynomial overhead. Note that each query, $[z_1,\ldots,z_c]$, of $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ trivially has $|z_1|,\ldots,|z_c| \leq b[d]$ at the time it is issued. Hence, if $\mathbf{E}^{\vec{\sigma},\tau}_{i,d,k,\ell}$ never experiences a cost overrun, then for $d' = d+1$, each query, $[z_1,\ldots,z_c]$, issued by $\mathbf{E}^{\vec{\sigma},\tau}_{i,d',k,\ell}$ has $|z_1|,\ldots,|z_c| \leq b[d'-1]$.

Using these observations, it is straightforward to construct a a simple recursive (in ℓ) procedure that suffices for the lemma. □

Lemma 52. *Given an $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ and a $t \in \omega$, if on a particular input the cost of the computation of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is greater than t, then at some point before the cost of this computation exceeds cost t, the machine makes a query that has an answer a with $k \cdot (|a|+1)^k > t$.*

Proof. This is a simple consequence of the way the clocking scheme works. □

15.2. D-Machines

Except for some changes in the way clocking is done and some corresponding changes in the semantics, the machines $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, $\mathbf{U}^{\vec{\sigma},\tau}_{\widetilde{\mathbf{D}}}$ and $\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{D}}$ are, respectively, defined in exactly the same way as $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, $\mathbf{U}^{\vec{\sigma},\tau}_{\widetilde{\mathbf{E}}}$ and $\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}$.

The $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$'s are what we shall call D-*machines*. The details of clocking D-machines are as follows. A procedural $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is a clocked version of $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$. $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ runs the program of $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ step-by-step and, over the course of this $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation, keeps track of:

$$
\begin{aligned}
\mathit{cost} &= \text{the current accumulated cost of the } \widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}\text{-computation.} \\
b[0] &= \max\left\{ k\cdot(|a|+1)^k \;\middle\vert\; \begin{array}{l} a \text{ was the answer to a query } [\,] \text{ to a type} \\ \text{N oracle made at or before this point of} \\ \text{the computation} \end{array} \right\}. \\
b[i] &= \max\left\{ k\cdot(|a|+1)^k \;\middle\vert\; \begin{array}{l} a \text{ was the answer to a query } [z_1,\dots,z_c] \\ \text{to an oracle of type } \gamma_1\times\ldots\times\gamma_c\to \mathrm{N} \\ \text{made at or before this point of the com-} \\ \text{putation where, for } j=1,\dots,c, \text{ if } \gamma_j = \\ \text{N, then } |z_j|\le b[i-1] \end{array} \right\},
\end{aligned}
$$

where $0 < i \le d$. The values of *cost*, $b[0],\dots,b[d]$ are updated as needed after each $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation step. If it is ever the case that after an update we have $\mathit{cost} > b[d]$, then $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ halts with output 0. On the other hand, if the $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation runs to completion without being cut off by a cost overrun, then $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ outputs the result of this computation and halts.

The occurrence of the third case in the clocking, that the $\widetilde{\mathbf{D}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$-computation runs forever without incurring a cost overrun, depends on the underlying semantics. If a type-τ oracle can be an arbitrary element of Full_τ, then this third case is quite possible—Example 46 can be easily adapted to the present setting. Therefore, with arbitrary **Full** oracles we have a well-defined semantics only for hereditarily total D-machines, where this means that the machine is total and all the indices it uses describe machines that are also hereditarily total. (The use of the level parameter in the indices makes this a well-founded notion.) If, on the other hand, we draw our oracles exclusively from **BCont**, then a modification of the proof of Proposition 47 shows that this third case cannot occur. (The run-time bounds on the indices imply size bounds on the function named by these indices.) Therefore, with **BCont** oracles, it follows that $\mathbf{D}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is total and determines an element of BCont_τ.

Under either semantics it follows, as in the E-machine case, that the clocking as described above can be done with only polynomial overhead and, moreover, that the universal machines also have polynomial overhead in run-time complexity.

16. Relating $\mathsf{ITLP}^\flat$ and $\mathsf{ITLP}^\sharp$ to Machines

Here we examine the respective relations of $\mathsf{ITLP}^\flat$ and $\mathsf{ITLP}^\sharp$ to the E- and D-machine models.

E-Machines and $\mathsf{ITLP}^\flat$

Terminology: Let EF denote the class of functionals computed by E-machines of the form $\mathbf{E}^{(),\tau}_{i,k,d,\ell}$ and let BCEF denote the **BCont** version of EF.

Proposition 53. E $=$ EF *and* BCE $=$ BCEF.

This will follow from Propositions 54 and 55 below. Proposition 54 is the third link in our grand chain of translations. *Terminology:* An $\mathsf{ITLP}^\flat$ (or $\mathsf{ITLP}^\sharp$) procedure containing no global references is said to be *closed*.

Proposition 54. *One can effectively translate any given closed* $\mathsf{ITLP}^\flat$ *procedure into an equivalent E-machine of the form* $\mathbf{E}^{(),\tau}_{i,k,d,\ell}$. *Hence,* E $\subseteq$ EF *and* BCE $\subseteq$ BCEF.

Proof Sketch. The argument is a lift of the one given for Proposition I-19. The hard work, establishing the polynomial boundedness (and hence the totality) of the $\mathsf{ITLP}^\flat$-computable functionals, has already been done in proving Proposition 36.

Suppose P is a closed $\mathsf{ITLP}^\flat$ procedure. Without loss of generality we assume that each procedure occurring within P is also closed. (See the proof of Proposition 44 for justification.) To simplify the following, we assume that P has been preprocessed so that nowhere in P is there a λ-expression of the form $\lambda\vec{\mathsf{u}}.\mathsf{v}(\vec{\mathsf{w}})$ where v has a type of depth 0. Eliminating such expressions involves adding a few more simple procedures and slightly increasing the depth of the types of some variables. Let $\mathsf{P}_0,\dots,\mathsf{P}_m = \mathsf{P}$ be a list of all the procedures occurring within P. As $\mathsf{ITLP}^\flat$ forbids recursion, we may assume without loss of generality that for each j, P_j contains no references to any of $\mathsf{P}_j,\dots,\mathsf{P}_m$.

Suppose $0 \le j \le m$ and that for each $r < j$ we have already constructed $\mathbf{E}^{(),\tau_r}_{i_r,k_r,d_r,\ell_r}$, an E-machine equivalent to P_r. Moreover we assume for each r that the clocking parameters, k_r and d_r, have been chosen so that no cost overrun occurs in any computation of $\mathbf{E}^{(),\tau_r}_{i_r,k_r,d_r,\ell_r}$. Our goal is to construct an analogous E-machine corresponding to P_j.

Suppose P_j determines a functional of type $\tau = \sigma_0 \times \cdots \times \sigma_{u-1} \to \mathrm{N}$. We first show how to construct an $\widetilde{\mathbf{E}}^{(),\tau}_{i,k,d,\ell}$ equivalent to P_j. This construction is largely a repeat of the "ITLP_2 to type-2 OTM" translation in the proof of Proposition I-19. Thus in what follows we show just how to translate $\mathsf{ITLP}^\flat$ procedure calls and λ-expressions, which are the only two cases substantially different from the type-2 construction.

Consider a statement of the form $\mathsf{z} := \mathsf{v}(\vec{\mathsf{a}})$ where the type of v is $(\sigma_u \times \cdots \times \sigma_v \to \mathrm{N}, d')$. In translating the call $\mathsf{v}(\vec{\mathsf{a}})$ there are two cases to consider.

Case 1: $d' > 0$. Then v must name a P_r with $r < j$. By the $\mathsf{ITLP}^\flat$ typing restrictions, each argument in $\vec{\mathsf{a}}$ must be either an integer variable or a variable of an arrow type with depth 0. We can thus make use of standard compiling tricks to incorporate $\mathbf{E}^{(),\tau_r}_{i_r,k_r,d_r,\ell_r}$ as an "OTM subroutine" of the OTM we are constructing and translate $\mathsf{v}(\vec{\mathsf{a}})$ as a call to this OTM subroutine.

Case 2: $d' = 0$. We translate $\mathsf{v}(\vec{\mathsf{a}})$ as an oracle query that is preceded by the computations of appropriate query indices for this query. A member of $\vec{\mathsf{a}}$ may be (a) an integer variable, (b) a higher-type variable with type of depth 0, (c) a higher-type variable with type of positive depth, or (d) a λ-expression. We consider each of these possibilities in turn. Let $\vec{\sigma} = \sigma_0, \ldots, \sigma_{u-1}$.

Subcase 2a: An integer variable is no problem.

Subcase 2b: Since P_j is closed, the only variables with depth-0 types are the parameters of P_j. Hence, the argument in this subcase must be a parameter of P_j, say the b-th (counting from 0). Thus, $[2b, 0, 0, 0]$, the index of the oracular E-machine $\mathbf{E}^{\vec{\sigma},\sigma_b}_{2b,0,0,0}$, suffices as the query index for this argument.

Subcase 2c: In this subcase the argument must name a P_r for some $r < j$. Let i'_r be the modification of i_r that shifts each reference to oracle number b to oracle number $b + u$. It follows that, except for the difference in context oracles, $\mathbf{E}^{(),\tau_r}_{i_r,k_r,d_r,\ell_r}$ and $\mathbf{E}^{\vec{\sigma},\tau_r}_{i'_r,k_r,d_r,\ell_r}$ are equivalent machines. Hence $[i'_r, k_r, d_r, \ell_r]$ suffices as the query index for this argument.

Subcase 2d: Let $\lambda\vec{\mathsf{e}}.\mathsf{t}(\vec{\mathsf{b}})$ be the λ-expression in question. By our assumptions on such expressions, t must name a P_r for some $r < j$. So each argument in $\vec{\mathsf{b}}$ must be either an integer variable or a variable of an arrow type with depth 0. For simplicity let us suppose that the form of $\lambda\vec{\mathsf{e}}.\mathsf{t}(\vec{\mathsf{b}})$ is $\lambda\vec{\mathsf{x}},\vec{\mathsf{z}}.\mathsf{t}(\vec{\mathsf{w}},\vec{\mathsf{x}},\vec{\mathsf{y}},\vec{\mathsf{z}})$, where $\vec{\mathsf{w}}$ and $\vec{\mathsf{x}}$ consist of variables of higher types and where $\vec{\mathsf{y}}$ and $\vec{\mathsf{z}}$ consist of integer variables. (This parallels the statement of Lemma 50. If the λ-expression is not of this special form, we can argue this subcase by means of the liberalized version of that lemma.) Let i'_r be as in Subcase 2c. We construct a sequence OTM instructions to compute $i''_r = s(i'_r, [\vec{a}], [\vec{y}])$, where s is as in Lemma 50, $\vec{a}$ is the sequence of the numbers of the oracles that correspond to the free variables $\vec{\mathsf{w}}$, and $\vec{y}$ is the sequence of numbers that are the current (at the time of the call) values of the free variables $\vec{\mathsf{y}}$. (Since these

$\vec{y}$'s can be different each time the call is made, the computation of i''_r must be at run time, not compile time.) Let k''_r be a number $\geq k_r$ such that no cost overrun occurs in any computation of $\mathbf{E}^{\vec{\sigma},\tau_r}_{i''_r,k''_r,d_r,\ell_r}$. By the complexity bound of Lemma 50 it follows that such a k''_r exists and can be found effectively during the translation. Hence $[i''_r, k''_r, d_r, \ell_r]$ suffices as the query index for this argument.

Under Case 2 we thus translate $\mathsf{v}(\vec{\mathsf{a}})$ to the computation of each of the appropriate query indices as sketched above, followed by a query to the appropriate oracle using these indices.

Adding the above treatment of procedure calls to the translation of the proof of Proposition I-19 provides us with a method to generate an OTM program, say i_*, equivalent to P_j. Let k_* be the maximum of all the "k components" of the indices that i_* is set up to construct. (By the above analysis, these values are known at compile-time.) If i_* does not involve the construction of any indices, let k_* be 0. Define d_* and ℓ_* similarly. It then follows that $\widetilde{\mathbf{E}}^{(),\tau_r}_{2\cdot i_*+1,k_*,d_*,\ell_*+1}$ is equivalent to P_j.

Let $i_j = 2 \cdot i_* + 1$ and $\ell_j = \ell_* + 1$. By Lemma 50 and the above analysis, we have that the work $\widetilde{\mathbf{E}}^{(),\tau_r}_{i_j,k_*,d_*,\ell_j}$ does in building the query indices for those procedure calls falling under Case 2 has no more than polynomial cost (where this polynomial depends only on P). Recall that the translation for Proposition I-19 produced a program for a type-2 OTM and a second-order polynomial that bounded the run time of that OTM. By adapting the second-order polynomial bound analysis of Proposition I-19 to the present translation, and taking into account the cost of constructing query indices, it follows that we can effectively find a $k_i \geq k_*$ and a $d_j \geq d_*$ such that $\mathbf{E}^{(),\tau_j}_{i_j,k_j,d_j,\ell_j}$ is equivalent to $\widetilde{\mathbf{E}}^{(),\tau_r}_{2\cdot i_*+1,k_*,d_*,\ell_*+1}$ (and hence is equivalent to P_j). Moreover, k_j and d_j can be effectively chosen so that no cost overrun occurs in any computation of $\mathbf{E}^{(),\tau_r}_{i_r,k_r,d_r,\ell_r}$. Therefore, $\mathbf{E}^{(),\tau_r}_{i_r,k_r,d_r,\ell_r}$ is as required. □

Under the above translation the sizes of the k's can grow rather large. A more careful translation could do much better.

Proposition 55. *There is an effective procedure that, given i, k, d, ℓ, and τ, constructs an* $\mathsf{ITLP}^\flat$ *program equivalent to* $\mathbf{E}^{(),\tau}_{i,k,d,\ell}$. *Hence,* $\mathrm{E} \supseteq \mathrm{EF}$.

Proof Sketch. The argument is a lift of the one given for Proposition I-18. The one new problem in this setting is to devise a way for $\mathsf{ITLP}^\flat$ procedures to simulate the ability of E-machines to generate query indices (procedural objects) in the course of their computations. Since there is no *a priori* bound on the number of query indices generated during a given computation, the present translation obviously cannot construct an $\mathsf{ITLP}^\flat$ procedure for every possible query index. The solution is to interpret rather than compile. Recall that $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ is equivalent to $\lambda\vec{w},\vec{x}\,\boldsymbol{.}\,\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}(\vec{w};\vec{x},[i,k,d,\ell])$. Thus it suffices to construct, for various ℓ's, $\vec{\sigma}$'s, and τ's, an $\mathsf{ITLP}^\flat$ procedure, $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$, equivalent to $\lambda\vec{w},\vec{x},j\,\boldsymbol{.}\,\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}(\vec{w};\vec{x},[j,k,d,\ell])$ and deal with the indices through these $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$'s. (Recall that k and d are fixed by the "topmost" machine, so they become constants of the construction. This is a good thing, since $\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}$ is very far from being polynomially bounded.) The question then is, for which $\vec{\sigma}$'s, τ's, and ℓ's do we need to construct $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$?

Suppose $\mathbf{E}^{(),\tau_0}_{i_0,k,d,\ell_0}$ is the E-machine we want to translate. Consider the query indices that can be generated by this machine, or by the machines specified by these query indices, or by the machines specified by the query indices of the machines specified by $\mathbf{E}^{(),\tau_0}_{i_0,k,d,\ell_0}$'s query indices, or by .... It is clear that all these possible query indices describe machines of the form $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ where $\ell \leq \ell_0$, τ is a subexpression of τ_0 and $\vec{\sigma}$ is, roughly, the concatenation of at most $(\ell_0 - \ell)$-many left-hand sides of arrow types that occur as subexpressions of τ_0. It is thus clear what $\vec{\sigma}$'s, τ's, and ℓ's we need consider in constructing the $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$'s. However, in general we will need to construct $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$'s for all of the possible $\vec{\sigma}$'s, τ's, and ℓ's. Let $|\tau_0|$ denote the length of τ_0 as an expression. Then there is a lower bound of $\Omega(2^{\ell_0})$ and an upper bound of $O(|\tau_0|^{2\cdot\ell_0})$ on the number of possible triples $(\ell,\vec{\sigma},\tau)$. Therefore, the translation of $\mathbf{E}^{(),\tau_0}_{i_0,k,d,\ell_0}$ to an equivalent $\mathsf{ITLP}^\flat$ procedure is quite direct, but yields an exponential increase in program size. $\square$

The above explosion in program size is especially irritating when one notes that all of the $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$'s have essentially the same program. There is no cure for this explosion without radically altering $\mathsf{ITLP}^\flat$ to compensate for the "unfair" advantage that an E-machine has in constructing most of its program (that is, "program" in $\mathsf{ITLP}^\flat$ terms) at run time. Note, however, that if we restrict the translation to work only on E-machines as originally formulated by Seth (see footnote 12 in the previous section), then the explosion in program size is much milder. How mild depends on the details of the indexing of this other version of E-machines.

D-Machines and $\mathsf{ITLP}^\sharp$

Definition 56. D denotes the class of total functionals computed by D-machines of the form $\mathbf{D}^{(),\tau}_{i,k,d,\ell}$ over **Full**. BCD denotes the **BCont** version of D. $\diamond$

Proposition 57. *One can effectively translate any given closed* $\mathsf{ITLP}^\sharp$ *procedure into an equivalent D-machine of the form* $\mathbf{D}^{(),\tau}_{i,k,d,\ell}$*. Hence,* $\mathrm{D}^- \subseteq \mathrm{D}$ *and* $\mathrm{BCD}^- \subseteq \mathrm{BCD}$*.*

Proof Sketch. The argument is simply the $\mathsf{ITLP}^\sharp$/D-machine version of the proof of Proposition 54. The more liberal clocking scheme for the D-machines makes it reasonably straightforward to adapt the translation of Propositions 54 to handle $\mathsf{ITLP}^\sharp$ and the programming construct allowed by (16). $\square$

We do not have a proof of either $\mathrm{D}^- \supseteq \mathrm{D}$ or $\mathrm{BCD}^- \supseteq \mathrm{BCD}$. We conjecture that neither containment holds. This is the problem: Suppose $\mathsf{UD}^{\vec{\sigma},\tau}_{\ell}$ is the analogue of $\mathsf{UE}^{\vec{\sigma},\tau}_{k,d,\ell}$ from the proof of Proposition 55. In a statement such as

$$\mathsf{x} := \mathsf{F}\left(\lambda\vec{\mathsf{y}}\,\mathbf{.}\ \mathsf{down}\left(\mathsf{bnd}(\vec{\mathsf{y}}), \mathsf{UD}^{\vec{\sigma},\tau}_{\ell}(\vec{\mathsf{y}},\mathsf{p})\right)\right),$$

the bounding expression $\mathsf{bnd}(\vec{\mathsf{y}})$ is static and exterior to the call to $\mathsf{UD}^{\vec{\sigma},\tau}_{\ell}$, and this seems much too restrictive to deal with the strong sort of inflationary clocking that can occur in this call to $\mathsf{UD}^{\vec{\sigma},\tau}_{\ell}$.

17. The D Versus E Separations

This section contains the proofs of $\mathrm{BCD}^- \neq \mathrm{BCE}$ and $\mathrm{D}^- \neq \mathrm{E}$. The discussion here is reasonably self-contained. However, as the following are diagonalization results, the exposition is more recursion/complexity theoretic than heretofore.

Proposition 58. $\mathrm{BCD}^- \neq \mathrm{BCE}$.

Proof. The proof is a more sophisticated version of Example 46. In place of the H of the example, we have Diag, a slightly more elaborate $\mathsf{ITLP}^\sharp$ procedure based on the same ideas as H. In place of $\mathcal{F}_0$, we have a more complex functional $\mathcal{G}$ that handles the lion's share of the work of the diagonalization, although the actual diagonalization is done by Diag upon advice from $\mathcal{G}$. Before defining either Diag or $\mathcal{G}$ we consider a few preliminary matters.

For each finite function $\gamma\colon \mathrm{N} \rightharpoonup \mathrm{N}$, let $\widehat{\gamma}$ denote the total function extending γ that is 0 every place that γ is undefined. Given $v \in \mathrm{N}$ $(\equiv \{\, \mathbf{0}, \mathbf{1} \,\}^*)$, let $a_0, \ldots, a_z \in \{\, \mathbf{0}, \mathbf{1} \,\}$ be such that $v = a_0 \cdots a_z$ (if $v = \epsilon$, then $z = -1$) and define

$$\mathit{init}_v \quad = \quad \lambda y\colon \mathrm{N}\,\mathbf{.} \begin{cases} 0, & \text{if } y \leq z \text{, and } a_y = \mathbf{0}; \\ 1, & \text{if } y \leq z \text{, and } a_y = \mathbf{1}; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Let INIT $= \{\, \mathit{init}_v \mid v \in \mathrm{N} \,\}$ = the collection of all $\gamma\colon \mathrm{N} \rightharpoonup \{\, 0, 1 \,\}$ defined on some finite initial segment of N. Let γ, and decorated versions of γ, range over INIT. For each $\alpha\colon \mathrm{N} \rightharpoonup \mathrm{N}$ and $n \in \omega$, we define $\alpha{\restriction}_{=n}$ to be the finite function with the graph $\{\, (x, \alpha(x)) \mid |x| = n \,\}$ and $\alpha{\restriction}_{\leq n}$ to be the finite function with the graph $\{\, (x, \alpha(x)) \mid |x| \leq n \,\}$. For α, $\beta\colon \mathrm{N} \rightharpoonup \mathrm{N}$, we say that α and β are *inconsistent* if, for some x, $\alpha(x){\downarrow} \neq \beta(x){\downarrow}$, otherwise we say that they are *consistent*. We say that $\mathcal{A} \subseteq (\mathrm{N} \rightharpoonup \{\, 0, 1 \,\})$ *covers* $\mathrm{N} \to \{\, 0, 1 \,\}$ if and only if each $f\colon \mathrm{N} \to \{\, 0, 1 \,\}$ is consistent with some element of $\mathcal{A}$.

Recall that $\langle \cdot, \cdot \rangle$ denotes a polynomial-time computable pairing function that is nondecreasing in both arguments. Let π_1 and π_2 be the two (polynomial-time computable) projection functions for $\langle \cdot, \cdot \rangle$, i.e., $\langle \pi_1(x), \pi_2(x) \rangle = x$ for all x. For each $k > 2$, define $\langle x_1, \ldots, x_k \rangle = \langle x_1, \langle x_2, \ldots, x_k \rangle \rangle$. Note that $\lambda x_1, \ldots, x_k \,\mathbf{.}\, \langle x_1, \ldots, x_k \rangle$ is a polynomial-time isomorphism between N^k and N.

In this section "query" will be synonymous with a query to a higher-type oracle. Queries to type-N oracles will not be mentioned explicitly.

The $\mathsf{ITLP}^\sharp$ procedure Diag is sketched in Figure 7. In this sketch $\mathsf{bnd}\colon \mathrm{N} \to_1 \mathrm{N}$, $\mathsf{q}\colon \mathrm{N} \times \mathrm{N} \to_1 \mathrm{N}$, $\mathsf{first}\colon \mathrm{N} \to_1 \mathrm{N}$, and $\mathsf{second}\colon \mathrm{N} \to_1 \mathrm{N}$ respectively compute $\lambda x \,\mathbf{.}\, \underline{1}$, $\lambda v, y \,\mathbf{.}\, \widehat{\mathit{init}_v}(y)$, π_1, and π_2; we omit their actual declarations. Note that $\lambda \mathsf{y} \,\mathbf{.}\, \mathsf{q}(0, \mathsf{y})$ denotes $\lambda y \,\mathbf{.}\, 0$. Let $\tau_{\mathcal{G}} = (\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ and $\tau_{\mathsf{D}} = \tau_{\mathcal{G}} \times \mathrm{N} \to \mathrm{N}$.

The functional $\mathcal{G} \in \mathrm{BCont}_{\tau_{\mathcal{G}}}$ is defined through a collection $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{S}_2, \ldots$ of subsets of INIT $\times$ N constructed below. These $\mathcal{S}_a$'s will be such that:

1. For each distinct (γ_0, y_0) and (γ_1, y_1) in an $\mathcal{S}_a$, γ_0 and γ_1 are inconsistent.
2. Each $\mathcal{S}_a$ is finite.
3. The function $\lambda a \,\mathbf{.}\, \Big([\langle v_0, y_0 \rangle, \langle v_1, y_1 \rangle, \ldots, \langle v_k, y_k \rangle]$, where $v_0 < v_1 < \cdots < v_k$ and $\mathcal{S}_a = \{\, (\mathit{init}_{v_i}, y_i) \mid i \leq k \,\} \Big)$ is computable.

Given the $\mathcal{S}_a$'s, we define $\mathcal{G}$ by:

$$\mathcal{G}(f, a) \quad = \quad \begin{cases} y, & \text{if, for some } (\gamma, y) \in \mathcal{S}_a,\ \gamma \subset f; \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

```
Procedure Diag(G: τ_G, a: N): N_2
    ...  Declarations of bnd, q, first, and second  ...
  var t: N_2, u: N_2, v: N_2, w: N_2;
  v := 0;   w := 1;
  For u := 1 to w do
    t := G (λy. down (bnd(y), q(v, y)) , a);
    v := down(t, first(t));   w := down(t, second(t));
  Endfor;
  Return v
End
```

Figure 7: The Diag procedure

By property 1, $\mathcal{G}$ is well defined. By its definition, $\mathcal{G}$ is total. By property 2, $\mathcal{G}$ is continuous. Hence, since $\mathcal{G}$ is of type-level 2, $\mathcal{G}$ is bounded continuous. By property 3, $\mathcal{G}$ is computable. ($\mathcal{G}$'s computability is used in the next proof, but not in this one.)

The $\mathcal{S}_a$'s are constructed in stages $0, 1, 2, \ldots$ in turn. The aim of each stage s is to diagonalize against $\mathbf{E}^{(),\tau_{\mathsf{D}}}_{i,k,d,\ell}$ where $s = \langle i, k, d, \ell \rangle$. That is, the construction will guarantee that Diag and $\mathbf{E}^{(),\tau_{\mathsf{D}}}_{i,k,d,\ell}$ differ on input $(\mathcal{G}, a_s)$ for a particular $a_s \in \mathrm{N}$.

Before the beginning of stage 0, $\mathcal{S}_a = \emptyset$ for each a. During the course of the construction, finitely-many pairs (γ, y) are added to particular $\mathcal{S}_a$'s. For each a, there will come a point in the construction when $\mathcal{S}_a$ is *closed*, which means that past that point no more elements may enter $\mathcal{S}_a$. Initially, each $\mathcal{S}_a$ is *open*, i.e., not closed.

At each point in the construction, $\widehat{\mathcal{G}}$ denotes the functional defined by the current contents of the $\mathcal{S}_a$'s. That is, if we closed every $\mathcal{S}_a$ at that moment, then $\widehat{\mathcal{G}}$ would be the functional defined by the right-hand side of (20).

Let $a_0 = 0$. At the end of each stage s, $a_{s+1} \in \mathrm{N}$ is defined. (These a_s's are the same ones mentioned a few paragraphs ago.) At the beginning of each stage $s+1$, it will be the case that for each $a \geq a_{s+1}$, $\mathcal{S}_a$ will be open and empty, and for each $a < a_{s+1}$, $\mathcal{S}_a$ will be closed and

$$\max(\{\, |x| + |y| \;\vdots\; (\gamma, y) \in \mathcal{S}_a \;\;\&\;\; \gamma(x)\!\downarrow \;\;\&\;\; a < a_{s+1} \,\}) \quad < \quad |a_{s+1}|. \tag{21}$$

The purpose of (21) is to help make sure the work done in stage $s+1$ will not injure the work accomplished in prior stages.

Before giving the construction of the $\mathcal{S}_a$'s, it is helpful to consider what sorts of query indices can arise in an interaction between $\mathcal{G}$ and an $\mathbf{E}^{();\tau_D}_{i,k.d,\ell}$ on input $(\mathcal{G}, a)$.

- If $\ell = 0$, then no query indices can be employed. Thus, since the only function oracle of $\mathbf{E}^{();\tau_D}_{i,k.d,\ell}$ is type-level 2, the machine can have no dependence on its functional oracle.

- If $\ell > 0$, then the machine may employ level-$(\ell-1)$ query indices that name machines that may employ level $(\ell-2)$ query-indices, and so on. Each level-ℓ' query index names a machine of the form $\mathbf{E}^{\vec{\sigma}_{\ell'},\mathrm{N}\to\mathrm{N}}_{i',k.d,\ell'}$ where $\vec{\sigma}_{\ell-1} = \tau_{\mathcal{G}}, \mathrm{N}$; $\vec{\sigma}_{\ell-2} = \tau_{\mathcal{G}}, \mathrm{N}, \mathrm{N}$; $\vec{\sigma}_{\ell-3} = \tau_{\mathcal{G}}, \mathrm{N}, \mathrm{N}, \mathrm{N}$, and so on. Moreover, each such machine is equivalent to

$$\lambda x \,.\, \mathbf{U}^{\vec{\sigma}_{\ell'},\mathrm{N}\to\mathrm{N}}_{\mathbf{E}}(\mathcal{G}, a, b_{\ell-1}, \ldots, b_{\ell'+1}; [i', k, d, \ell'], x),$$

  where $b_{\ell'+1}$ is the argument of the invocation of the level-$(\ell'+1)$ query index that constructed this instance of the level-ℓ' query index, $b_{\ell'+2}$ is the argument of the invocation of the level-$(\ell'+2)$ query index that constructed the aforementioned instance of the level-$(\ell'+1)$ query index, and so on. As in the $\ell = 0$ case, level-0 query indices have no dependence on their functional oracle.

We consider four special cases of stage s that will build in complexity. Recall that we regard s as coding the quadruple (i, k, d, ℓ) where $s = \langle i, k, d, \ell \rangle$.

Stage $s = \langle i, k, d, \ell \rangle$ for $\ell = 0$.

Since $\ell = 0$, by the discussion above, $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$ has no dependence on its functional oracle, and so the construction sketched in Figure 8 suffices.

To see that this works, first note that by construction $\mathsf{Diag}(\mathcal{G}, a_s) = 1 \dot{-} y$. Since $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$ has no dependence on its type-$\tau_{\mathcal{G}}$ argument, $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}(\mathcal{F}, a_s) = y$ for any type $\tau_{\mathcal{G}}$ oracle $\mathcal{F}$. Hence, Diag and $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$ differ on $(\mathcal{G}, a_s)$ as required.

Stage $s = \langle i, k, d, \ell \rangle$ for $\ell = 1$.

By the general discussion of query indices, under this case $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$ can query its functional oracle, but the only legal queries involve level-0 query indices and these indices name machines that have no dependence on their functional

Construction for the $\ell = 0$ case.

Let $y = \mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}(\widehat{\mathcal{G}}, a_s)$ and set $\mathcal{S}_{a_s} := \{ (\emptyset, \langle 1 \dot{-} y, \underline{0} \rangle) \}$.

Let a_{s+1} be the least number $> a_s$ such that (21) holds and close each $\mathcal{S}_a$ with $a < a_{s+1}$.

End construction

Figure 8: The $\ell = 0$ construction

oracle. Let $\vec{\sigma}_0 = \tau_{\mathcal{G}}, \mathrm{N}$. Let $q_k = \lambda n \centerdot k \cdot (n+1)^k$. Let $\mathcal{Z} \in \mathrm{BCont}_{\tau_{\mathcal{G}}}$ be such that $\mathcal{Z} = \lambda f, x \centerdot 0$. Also, for each p and x, define:

$$f^0_p(x) \quad = \quad \begin{cases} \mathbf{U}^{\vec{\sigma}_0, \mathrm{N} \to \mathrm{N}}_{\mathbf{E}}(\mathcal{Z}, a_s; p, x), & \text{if } p \text{ is a legal query index for this case;} \\ 0, & \text{otherwise.} \end{cases}$$

The construction is sketched in Figure 9. The intuition behind the construction is that $\mathcal{G}$ views $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$ as trying to easesdrop on $\mathcal{G}$'s conversation with Diag. To confound $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$, $\mathcal{G}$ equivocates. That is, $\mathcal{G}$ makes sure that its end of the conversation with Diag on input $(\mathcal{G}, a_s)$ is sufficiently evasive and long-winded to exceed the abilities of $\mathbf{E}^{();\tau_{\mathsf{D}}}_{i,k,d,\ell}$ to follow it.

Claim 1.1. *Diag*$(\mathcal{G}, a_s) = 1 \dot{-} y$.

By the constructions of Diag and $\mathcal{G}$, on input $(\mathcal{G}, a_s)$, Diag engages in a $(d+1)$-round conversation with $\mathcal{G}$. In round j $(1 \leq j \leq d)$, Diag asks a question and receives the answer $\langle v_j, \underline{d+1} \rangle$, where v_j encodes the next question Diag should ask $\mathcal{G}$ and $\underline{d+1}$ is large enough so that we are sure that Diag will get to ask this next question. In round $d+1$, Diag asks its question and the reply, $\langle 1 \dot{-} y, \underline{0} \rangle$, signals that $1 \dot{-} y$ is the result and that the conversation is at an end, whereupon Diag outputs $1 \dot{-} y$ as claimed.

Claim 1.2. *The invariant holds throughout stage s.*

At the beginning of stage 0, $\mathcal{A} = \emptyset$, so the invariant clearly holds at that point. If $d = 0$, then $\mathcal{A} = \emptyset$ throughout stage s and so we are done. Suppose $d > 0$ and suppose $j \in \{ 1, \ldots, d \}$ is such that the invariant holds through all iterations of the for-loop up to the j-th. If $j = 1$, then $\mathcal{A} = \emptyset$ before the j-th iteration and if $j > 1$, then by construction we have that: $|\max(\mathrm{dom}(\mathit{init}_{v_0}))| \leq n_0$ and, for $j' = 1, \ldots, j-1$, $n_{j'-1} + 1 < |\max(\mathrm{dom}(\mathit{init}_{v_{j'}}))| \leq n_{j'}$. In either

Construction for the $\ell = 1$ case.

Let $v_0 = 0$, let $n_0 = q_k(|a_s|)$, and set $\mathcal{A} := \emptyset$.

(* ***Invariant:*** *$\mathcal{A}$ fails to cover* $\mathrm{N} \to \{0,1\}$. *)

(* *The equivocation phase* *)

For $j = 1, \dots, d$ in turn, do:

Let w_j be such that $init_{w_j} = (\widehat{init_{v_{j-1}}})\restriction_{\leq 1+n_{j-1}}$.

Set $\mathcal{A} := \mathcal{A} \cup \{ f^0_p\restriction_{=n_{j-1}} \mid |p| \leq n_{j-1} \} \cup \{ init_{w_j}\restriction_{=1+n_{j-1}} \}$.

Let v_j be the least number such that $init_{v_j}$ is inconsistent with each element of $\mathcal{A}$. (* *By the invariant, such a v_j must exist.* *)

Set $\mathcal{S}_{a_s} := \mathcal{S}_{a_s} \cup \{ (init_{w_j}, \langle v_j, \underline{d+1}\rangle) \}$.

$$\text{Let } n_j = 1 + \max\Big(\{ 1 + n_{j-1} \} \cup \{ |x| \mid init_{v_j}(x)\downarrow \} \cup \{ q_k(|\widehat{\mathcal{G}}(f^0_p, y)|) \mid |p|, |y| \leq n_{j-1} \} \Big).$$

(* *The main diagonalization* *)

Let $y = \mathbf{E}^{();\tau_D}_{i,k,d,\ell}(\widehat{\mathcal{G}}, a_s)$ and set $\mathcal{S}_{a_s} := \mathcal{S}_{a_s} \cup \{ (init_{v_d}, \langle 1 \dot{-} y, \underline{0}\rangle) \}$.

(* *Clean up* *)

Let a_{s+1} be the least number $> a_s$ such that (21) holds and close each $\mathcal{S}_a$ with $a < a_{s+1}$.

End construction

Figure 9: The $\ell = 1$ construction

case, it follows by construction that the functions added to $\mathcal{A}$ before the j-th iteration have ranges that are disjoint from the functions added in iteration j. Recall that for each n, $\|\{ p \mid |p| \leq n \}\| = 2^{n+1} - 1$, $\|\{ f\restriction_{=n} \mid f\colon \mathrm{N} \to \{0,1\} \}\| = 2^{2^n}$, and $2^{n+1} - 1 < 2^{2^n}$. So if there is an element of $\mathrm{N} \to \{0,1\}$ that is inconsistent with each element of $\mathcal{A}$ before the j-th iteration, then it follows easily that we can find an element of $\mathrm{N} \to \{0,1\}$ that is inconsistent with each old element of $\mathcal{A}$ and each element of $\mathcal{A}$ added in the j-th iteration. Therefore, Claim 1.2 follows.

Conventions: For each $j < d$, let p_j range over numbers of the form $[i, k, d, 0]$ with $|p_j| \leq n_j$ and let y_j range over numbers with $|y_j| \leq n_j$. Let $\widehat{\mathcal{G}}_0$ be the version of $\widehat{\mathcal{G}}$ current as of the beginning of stage s. For $j = 1, \dots, d$, let $\widehat{\mathcal{G}}_j$ be the version of $\widehat{\mathcal{G}}$ current as of the *end* of the j-th iteration of the for-loop. Let $\widehat{\mathcal{G}}_{d+1}$ be the version of $\widehat{\mathcal{G}}$ current as of the end of stage s.

Claim 1.3. *For each $j < d$, each j' with $j < j' \leq d+1$, and each p_j and y_j, we have that $\widehat{\mathcal{G}}_j(f^0_{p_j}, y_j) = \widehat{\mathcal{G}}_{j'}(f^0_{p_j}, y_j)$.*

If $d = 0$ there is nothing to prove. So suppose $d > 0$ and suppose $j_* < d$ is such that the claim holds for each $j < j_*$. By Claim 1.2 and the invariant, each (γ, z) added to $\mathcal{S}_{a_s}$ in iteration j_* through the end of stage s is inconsistent with each $f^0_{p_{j_*}}$. Hence, the claim follows for j_* and we are done.

Claim 1.4. *For each j and j' with $1 \leq j < j' \leq d+1$, we have that $n_j = 1 + \max\Big(\{\, n_{j-1}+1 \,\} \cup \{\, |x| \mid init_{v_j}(x)\downarrow \,\} \cup \{\, q_k(|\widehat{\mathcal{G}}_{j'}(f^0_p, y)|) \mid |p|, |y| \leq n_{j-1} \,\}\Big)$.*

This follows from the definition of the n_j's and Claim 1.3.

Claim 1.5. $\mathbf{E}^{(),\tau_\mathsf{D}}_{i,k,d,\ell}(\widehat{\mathcal{G}}_{d+1}, a_s) = y$.

Clearly, $\mathbf{E}^{(),\tau_\mathsf{D}}_{i,k,d,\ell}(\widehat{\mathcal{G}}_d, a_s) = y$. So we have to argue that the last addition to $\mathcal{S}_{a_s}$ does not change the computation of $\mathbf{E}^{(),\tau_\mathsf{D}}_{i,k,d,\ell}$ on input $(\widehat{\mathcal{G}}_{d+1}, a_s)$ from the computation on input $(\widehat{\mathcal{G}}_d, a_s)$. If $d = 0$, then the construction simplifies to be the same as that for the $\ell = 0$ case and the argument for that case shows the present claim. So suppose $d > 0$. It follows by Claim 1.4 and by the query size restriction in the clocking scheme for the E-machines that all the possible queries of the machine on these two inputs must be of the form (p_{d-1}, y_{d-1}). By Claim 1.3, $\widehat{\mathcal{G}}_d$ and $\widehat{\mathcal{G}}_{d+1}$ must agree on the answers to all such queries. Therefore, because the answers to all queries are the same, the computations are the same, and the claim follows.

Claim 1.6. $\mathbf{E}^{(),\tau_\mathsf{D}}_{i,k,d,\ell}(\mathcal{G}, a_s) = y$.

As we argued in the proof of Claim 1.5, all of the possible queries of $\mathbf{E}^{(),\tau_\mathsf{D}}_{i,k,d,\ell}$ on input $(\widehat{\mathcal{G}}_{d+1}, a_s)$ are a subset of those of the form (p_{d-1}, y_{d-1}). Recall from the construction that for each y_{d-1} we have $|y_{d-1}| \leq n_{d-1}$ and also that $n_{d-1} < |\max(\mathrm{dom}(init_{v_d}))| < |v_d|$ and $(init_{v_{d-1}}, \langle v_d, |v_d|\rangle) \in \mathcal{S}_{a_s}$. So by the actions of the clean-up phase, all $\mathcal{S}_a$ with $|a| \leq n_{d-1}$ are closed at the end of stage s. Hence, all additions to $\mathcal{S}_a$'s past stage s will be unseen by the computation of $\mathbf{E}^{(),\tau_\mathsf{D}}_{i,k,d,\ell}$ on input $(\widehat{\mathcal{G}}, a_s)$ and so the computation on that input will be identical to the one on input $(\widehat{\mathcal{G}}_{d+1}, a_s)$. The claim thus follows.

Therefore, from Claims 1.1 and 1.6 we can conclude that the construction for the $\ell = 1$ case works as required.

Note that after the d-th iteration of the for-loop we can add (γ, y)'s to $\mathcal{S}_{a_s}$, and the computation of the machine on $(\widehat{\mathcal{G}}_d, a_s)$ will match that on $(\widehat{\mathcal{G}}, a_s)$, *provided* all of these γ's are inconsistent with the final version of $\mathcal{A}$. Most of the work of the above construction went into "preserving" the computation of $\mathbf{E}^{(),\tau_{\mathrm{D}}}_{i,k,d,\ell}$ on $(\mathcal{G}, a_s)$ in the above sense. We use this trick of preserving computations more extensively in the next case.

Stage $s = \langle i, k, d, \ell\rangle$ for $\ell = 2$.

By the general discussion of query indices, $\mathbf{E}^{(),\tau_{\mathrm{D}}}_{i,d,k,\ell}$ can make use of level-1 query indices and these name machines that, in turn, may make use of level-0 query indices. Let $\vec{\sigma}_1 = \tau_{\mathcal{G}}, \mathrm{N}$. Let $\vec{\sigma}_0 = \tau_{\mathcal{G}}, \mathrm{N}, \mathrm{N}$. Let $q_k = \lambda n.\, k \cdot (n+1)^k$. Let $\mathcal{Z}$ be as before. For each p, b, and $x \in \mathrm{N}$ and each $\mathcal{F} \in \mathrm{BCont}_{\tau_{\mathcal{G}}}$, define:

$$f^1_{p,\mathcal{F}}(x) = \begin{cases} \mathbf{U}_{\mathbf{E}}^{\vec{\sigma}_1,\mathrm{N}\to\mathrm{N}}(\mathcal{F}, a_s; p, x), & \text{if } p \text{ is a legal level-1 query} \\ & \quad \text{index for this case;} \\ 0, & \text{otherwise.} \end{cases}$$

$$f^0_{p,b}(x) = \begin{cases} \mathbf{U}_{\mathbf{E}}^{\vec{\sigma}_0,\mathrm{N}\to\mathrm{N}}(\mathcal{Z}, a_s, b; p, x), & \text{if } p \text{ is a legal level-0 query} \\ & \quad \text{index for this case;} \\ 0, & \text{otherwise.} \end{cases}$$

The construction for this case is split into a main construction (Figure 10), which deals with confounding level-1 queries, and a procedure equivocate$_0$ (Figure 11), which deals with confounding level-0 queries. Both the main construction and the procedure are adaptations of the construction for the $\ell = 0$ case. Note that $\mathcal{A}$ and $\mathcal{S}_{a_s}$ are "globals" of the construction and that a call to equivocate$_0$ will change their values as a side-effect.

Claim 2.1. *Diag*$(\mathcal{G}, a_s) = 1 \dot{-} y$.

The proof is a simple lift of the argument for Claim 1.1. Note that each call of equivocate$_0$ adds d many rounds to the conversation. Hence, since there are d calls to equivocate$_0$ in the main construction and $d+1$ many rounds added by the main construction outside of the ones generated equivocate$_0$, there are a total of $d^2 + d + 1$ rounds in the conversation for this case.

Claim 2.2. *The invariant holds throughout stage s.*

This is a lift of the argument for Claim 1.2.

Conventions: For each $j < d$, let p_j range over numbers of the form $[i, k, d, 1]$ with $|p_j| \le n_j$ and let y_j range over numbers with $|y_j| \le n_j$. Let $\widehat{\mathcal{G}}_{j,0}$ = the version of $\widehat{\mathcal{G}}$ current as of the return from the call to equivocate$_0$ in the

Main construction for the $\ell = 2$ case.

Let $v_0 = 0$, let $n_0 = q_k(|a_s|)$, and set $\mathcal{A} := \emptyset$.
(* ***Invariant:*** $\mathcal{A}$ *fails to cover* $\mathrm{N} \to \{0,1\}$. *)

(* *The equivocation phase* *)
For $j = 1, \ldots, d$ in turn, do:
Let $(u_j, m_j) = \mathrm{equivocate}_0(v_{j-1}, n_{j-1})$.
Let w_j be such that $init_{w_j} = (\widehat{init_{u_j}})\restriction_{\leq 1+m_j}$.
Set $\mathcal{A} := \mathcal{A} \cup \{\, f^1_{p,\widehat{\mathcal{G}}}\restriction_{=n_{j-1}} \,¦\, |p| \leq n_{j-1} \,\} \cup \{\, init_{w_j}\restriction_{=1+m_j} \,\}$.
Let v_j be the least number such that $init_{v_j}$ is inconsistent with each element of $\mathcal{A}$. (* *By the invariant, such a* v_j *must exist.* *)
Set $\mathcal{S}_{a_s} := \mathcal{S}_{a_s} \cup \{\, (init_{w_j}, \langle v_j, \underline{d^2+d+1}\rangle) \,\}$.

$$\text{Let } n_j = 1 + \max\Big(\{\, 1 + m_j \,\} \cup \{\, |x| \,¦\, init_{v_j}(x)\!\downarrow \,\} \cup \{\, q_k(|\widehat{\mathcal{G}}(f^1_{p,\widehat{\mathcal{G}}}, y)|) \,¦\, |p|, |y| \leq n_{j-1} \,\} \Big).$$

(* *The main diagonalization* *)
Let $y = \mathbf{E}^{();\tau_D}_{i,k,d,\ell}(\widehat{\mathcal{G}}, a_s)$ and set $\mathcal{S}_{a_s} := \mathcal{S}_{a_s} \cup \{\, (init_{v_d}, \langle 1 \dot{-} y, \underline{0}\rangle) \,\}$.

(* *Clean up* *)
Let a_{s+1} be the least number $> a_s$ such that (21) holds and close each $\mathcal{S}_a$ with $a < a_{s+1}$.

End main construction

Figure 10: The $\ell = 2$ construction

j-th iteration of the main construction's for-loop and let $\widehat{\mathcal{G}}_{j,1}$ = the version of $\widehat{\mathcal{G}}$ current as of the end of the j-th iteration of this for-loop. Let $\widehat{\mathcal{G}}_{d+1}$ = the version of $\widehat{\mathcal{G}}$ current as of the end of stage s.

Claim 2.3. *For each* $j = 1, \ldots, d$, *each* p_{j-1} *and* y_{j-1}, *and each* $\widehat{\mathcal{G}}_*$, *some version of* $\widehat{\mathcal{G}}$ *current between the versions* $\widehat{\mathcal{G}}_{j,0}$ *and* $\widehat{\mathcal{G}}_{d+1}$ *(inclusive), we have that* $f_{p,\widehat{\mathcal{G}}_{j,0}}(y_{j-1}) = f_{p,\widehat{\mathcal{G}}_*}(y_{j-1})$.

Roughly, the call to $\mathrm{equivocate}_0$ in the j-th iteration of the main construction's for-loop has the purpose of running the level-1 construction so as to henceforth fix the value of $f^1_{p,\widehat{\mathcal{G}}}(y_{j-1})$. Thus, the proof this claim is simply a lift of the argument for Claim 1.5, together with supporting arguments from the level-1 case.

Procedure $\mathrm{equivocate}_0(v, n)$

Let $\widetilde{u}_0 = v$ and $\widetilde{m}_0 = q_k(n)$.

(* ***Invariant:*** *$\mathcal{A}$ fails to cover* $\mathrm{N} \to \{0,1\}$. *)

(* *The equivocation phase* *)

For $\widetilde{\jmath} = 1, \ldots, d$ in turn, do:

Let $\widetilde{w}_{\widetilde{\jmath}}$ be such that $init_{\widetilde{w}_{\widetilde{\jmath}}} = (\widehat{init_{\widetilde{u}_{\widetilde{\jmath}-1}}})\restriction_{\leq 2+\widetilde{m}_{\widetilde{\jmath}-1}}$.

Set $\mathcal{A} := \mathcal{A} \cup \{ f^0_{p,b}\restriction_{=1+\widetilde{m}_{\widetilde{\jmath}-1}} \mid |p|, |b| \leq \widetilde{m}_{\widetilde{\jmath}-1} \} \cup \{ init_{\widetilde{w}_{\widetilde{\jmath}}}\restriction_{=2+\widetilde{m}_{\widetilde{\jmath}-1}} \}$.

(* *Note that for each* m, $\|\{ (p,b) \mid |p|, |b| \leq m \}\| = (2^{m+1}-1)^2 < 2^{2^{m+1}} = \big\|\{ f\restriction_{=m+1} \mid f\colon \mathrm{N} \to \{0,1\} \}\big\|$. *)

Let $\widetilde{u}_{\widetilde{\jmath}}$ be the least number such that $init_{\widetilde{u}_{\widetilde{\jmath}}}$ is inconsistent with each element of $\mathcal{A}$. (* *By the invariant, such a* $\widetilde{u}_{\widetilde{\jmath}}$ *must exist.* *)

Set $\mathcal{S}_{a_s} := \mathcal{S}_{a_s} \cup \{ (init_{\widetilde{w}_{\widetilde{\jmath}}}, \langle \widetilde{u}_{\widetilde{\jmath}}, \underline{d^2 + d + 1} \rangle) \}$.

Let $\widetilde{m}_{\widetilde{\jmath}} = 1 + \max\Big(\{ 1 + \widetilde{m}_{\widetilde{\jmath}-1} \} \cup \{ |x| \mid init_{\widetilde{u}_{\widetilde{\jmath}}}(x)\downarrow \}$
$\cup \{ q_k(|\widehat{\mathcal{G}}(f^0_{p,b}, y)|) \mid |p|, |b|, |y| \leq \widetilde{m}_{\widetilde{\jmath}-1} \}\Big)$.

Return $(\widetilde{u}_d, \widetilde{m}_d)$.

End procedure

Figure 11: The $\mathrm{equivocate}_0$ procedure

Claim 2.4. *For each $j = 1, \ldots, d$, each p_{j-1} and y_{j-1}, and each $\widehat{\mathcal{G}}_*$, a version of $\widehat{\mathcal{G}}$ current between $\widehat{\mathcal{G}}_{j,1}$ and $\widehat{\mathcal{G}}_{d+1}$ (inclusive), we have that $\widehat{\mathcal{G}}_{j,1}(f_{p,\widehat{\mathcal{G}}_{j,0}}, y_{j-1}) = \widehat{\mathcal{G}}_*(f_{p,\widehat{\mathcal{G}}_{j,0}}, y_{j-1})$.*

We can use Claim 2.3 to adapt the argument for Claim 1.3 to show the present claim.

Claim 2.5. *For each j with $1 \leq j \leq d$ and each $\widehat{\mathcal{G}}_*$, a version of $\widehat{\mathcal{G}}$ current between $\widehat{\mathcal{G}}_{j,1}$ and $\widehat{\mathcal{G}}_{d+1}$, we have $n_j = 1+\max\Big(\{ 1 + m_j \} \cup \{ |x| \mid init_{v_j}(x)\downarrow \} \cup \{ q_k(|\widehat{\mathcal{G}}_*(f^1_{p,\widehat{G}_*}, y)|) \mid |p|, |y| \leq n_{j-1} \}\Big)$.*

This follows from the definition of the n_j's and Claim 2.4.

Claim 2.6. $\mathbf{E}^{();\tau_D}_{i,k,d,\ell}(\widehat{\mathcal{G}}_{d+1}, a_s) = y$.

The proof is a straightforward adaptation of the proof of Claim 1.5.

Claim 2.7. $\mathbf{E}^{();\tau_D}_{i,k,d,\ell}(\mathcal{G}, a_s) = y$.

The proof is a straightforward adaptation of the proof of Claim 1.6.

STAGE $s = \langle i, k, d, \ell \rangle$ for $\ell > 2$.

We omit the details for this case. There are really no new ideas required. All that is needed is to extend the construction and argument for the $\ell = 2$ case in a reasonably straightforward manner. $\square$ **Proposition 58**

Proposition 59. $\mathrm{D}^- \neq \mathrm{E}$.

Proof Sketch. The proof is a modification of the argument for the previous proposition. The construction of $\mathcal{G}$ is left unchanged, but we modify Diag (let us call the new version Diag') by adding a looking-back construction to its **For**-loop. This looking-back construction: (a) records the conversation between Diag' and its current oracle, (b) (slowly) reruns the construction of $\mathcal{G}$ and $\mathcal{G}$'s conversations with Diag', (c) compares the two conversations, and (d) makes Diag' halt with output 0 if a difference between the two conversations is ever detected. In any given iteration of the loop, only a few steps of the reconstruction of (b) are done, but as w increases, more and more of the reconstruction takes place. Thus, the only way Diag' can run forever is if the two conversations are infinite and identical. But we know from the argument for Proposition 58 that any conversation with $\mathcal{G}$ must terminate. Hence, Diag' computes a total functional. By the argument for Proposition 58 we also know that for each $\mathbf{E}^{(),\tau_{\mathsf{D}}}_{i,k,d,\ell}$, there is an a such that $\mathsf{Diag}'(\mathcal{G}, a) \neq \mathbf{E}^{(),\tau_{\mathsf{D}}}_{i,k,d,\ell}(\mathcal{G}, a)$. Hence Diag' determines an element of D^- that is not in E. $\square$

Remark 60. The above construction makes no real use of the fact that the $\mathbf{E}^{(),\tau}_{i,k,d,\ell}$'s are *polynomially* bounded. For example, in place of the k in $\mathbf{E}^{(),\tau}_{i,k,d,\ell}$ naming $\lambda n \,.\, k \cdot (n+1)^k$, it could just as well name a $f_k \in \mathcal{E}_{k+1}$ that majorizes every element of $\mathcal{E}_k$ and this f_k would replace the use of $\lambda n \,.\, k \cdot (n+1)^k$ in the machine's clocking scheme.[13] With such a change, the construction and proof would go through with Diag unchanged. The same is true if we changed the E-machines (now denoted, say, $\ddot{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$) so that the level of an outermost machine (an $\ddot{\mathbf{E}}^{(),\tau}_{i,k,d,\ell}$) can depend on a type-0 input, e.g, $\ddot{\mathbf{E}}^{(),\tau_{\mathsf{D}}}_{i,k,d,\ell}(\mathcal{F}, a) = \mathbf{E}^{(),\tau_{\mathsf{D}}}_{i,k,d,2^{a+\ell}}(\mathcal{F}, a)$. (Seth [Set95] calls these transfinite levels.) Thus, BCD^- (respectively, D^-) is a profoundly larger class than BCE (respectively, E). Whether BCD^- and D^- are too large to be considered feasible is a question taken up in Section 20. $\diamond$

[13] We suspect that Gödel's primitive recursive functionals [Göd58, Göd90] are exactly the class of functionals such machines compute.

18. From E-Machines to BTLP

The following proposition is the fourth, and final, link in our grand chain of translations.

Proposition 61. *There is an effective procedure that, given i, k, d, ℓ, and τ, constructs a* BTLP *program equivalent to* $\mathbf{E}^{();\tau}_{i,k,d,\ell}$ *under the* **Full***-based semantics.*

Proof Sketch. The general argument follows the pattern of the proof of Proposition 55. That is, given i, k, d, ℓ_0, and τ_0, to translate $\mathbf{E}^{();\tau_0}_{i,k,d,\ell_0}$ to an equivalent BTLP, it suffices to construct, for particular values of ℓ, $\vec{\sigma}$, and τ, a BTLP procedure $\mathtt{U}^{\vec{\sigma},\tau}_{k,d,\ell}$ equivalent to $\lambda\vec{w},\vec{x},j\,.\,\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}(\vec{w};\vec{x},[j,k,d,\ell])$. These "particular values" are the same as those in the proof of Proposition 55. So, as in that proof, there is an exponential (in ℓ_0) number of these procedures to construct.

In the proof of Proposition 55 we did not need to worry about the details of the individual procedures because they are each a straightforward modification of the procedure in the proof of Proposition I-18. We are not so lucky with the $\mathtt{U}^{\vec{\sigma},\tau}_{k,d,\ell}$'s. The restrictiveness of BTLP's sole iteration construct, **Loop**, makes translating the $\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}$'s difficult. The problem posed by these translations is essentially the problem solved in Kapron and Cook's paper [KC96]. By a difficult construction they showed how to translate type-2 polynomial-clocked machines into equivalent BTLP procedures. Below we sketch a simpler solution to the present problem that applies as well to the type-2 case.

Given k, d, ℓ, $\vec{\sigma}$ and τ, our goal is to construct a BTLP procedure $\mathtt{U}^{\vec{\sigma},\tau}_{k,d,\ell}$ equivalent to $\lambda\vec{w},\vec{x},i\,.\,\mathbf{U}^{\vec{\sigma},\tau}_{\mathbf{E}}(\vec{w};\vec{x},[i,k,d,\ell])$ that, in turn, we know is equivalent to $\lambda\vec{w},\vec{x},i\,.\,\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}(\vec{w};\vec{x})$. We shall assume that for each $\ell' < \ell$ all the required $\mathtt{U}^{\vec{\sigma}',\tau'}_{k,d,\ell'}$'s have already been constructed. We shall also assume, without loss of generality, that $k > 0$. The key parameter for the construction is d, the depth parameter. When $d = 0$, queries are forbidden in any computation of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$, so that case is straightforward and omitted. When $d = 1$, matters are more complicated and when $d > 1$ things are more complicated still. Below we consider the $d = 1$ and $d = 2$ cases. First we set up some conventions and note one important lemma.

Preliminaries

In this section we shall use a very liberalized version of BTLP in our constructions. In particular, we will permit **If-then-else**'s, breaks in **Loop**'s, and **Return**'s in the middle of procedures. Since the distinction between context and input oracles is not really used here, we shall replace "$\vec{w};\vec{x}$" with "$\vec{y}$" = "$y_0,\dots,y_v$" to simplify notation. Moreover, in definitions and procedure declarations, we will not bother mentioning the types of the elements of $\vec{y}$ since these will be fixed throughout. To simplify matters further, we shall use a central clearing house for queries. That is, for each $q \in \mathrm{N}$ and $\vec{y}$ we define

$$F(\vec{y},q) \;=\; \begin{cases} a, & \text{if } q=[j,z_1,\dots,z_c]\text{, where } j\le v,\ \ [z_1,\dots,z_c] \text{ is a well-formed} \\ & \quad \text{query to } y_j\text{, and } a \text{ is the result of this query;} \\ 0, & \text{otherwise.} \end{cases}$$

In an expression such as "$F(\vec{y},q)$" we shall refer to q as an *F-query* or simply a *query*. In the constructions below, in place of making direct queries to particular y_j's, we shall typically use F-queries. It follows from the properties of our coding of sequences and the fact that the types of the y_j's are fixed that there is an $m\in\omega$ such that for each $j\le v$ and each $s\in\omega$, if $[z_1,\dots,z_c]$ is a well-formed query to y_j with $|z_1|,\dots,|z_c|\le s$, then $\big|[z_1,\dots,z_c,j]\big|\le m\cdot s$.

Let p_k be the polynomial function $\lambda n\,.\,k\cdot(n+1)^k$. As noted in Section 15, our clocking scheme has polynomial overhead. In fact, it follows our description of this scheme that we can effectively find, given k and d, a polynomial $bnd_{k,d}$ such that, for all $n\in\omega$ and all $\vec{y}$, whatever part of the computation of $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,k}$ on input $\vec{y}$ can be completed within cost $p_k(n)$, the machine $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,k}$ on input $\vec{y}$ can complete the same part of the $\widetilde{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,k}$-computation within cost $bnd_{k,d}(n)$. We assume that $bnd_{k,d}$ is everywhere positive.

In the following we shall use functions and BTLP procedures that involve a mix of variables and values of types N and ω. We will usually write a type-ω value as $\underline{a}$ where $a\in\mathrm{N}$. Also, $\odot$ will denote multiplication in ω. That is, $\underline{a}\odot\underline{b}=\underline{a\cdot b}$.

For each $i\in\mathrm{N}$, $\underline{n}\in\omega$, and $\vec{y}$, we define

$$Run(\vec{y},i,\underline{n}) \;=\; \begin{cases} \mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}(\vec{y}), & \text{if the cost of the computation of } \mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell} \\ & \quad \text{on input } \vec{y} \text{ is no more than } \underline{n}; \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, Run is basic feasible. For each $q\in\mathrm{N}$, $\underline{n}$ and $\underline{s}\in\omega$, and $\vec{y}$, we define

$$MaxQ(\vec{y},q,\underline{n},\underline{s}) \;=$$

$$\begin{cases} q', & \text{if } F(\vec{y},q) < F(\vec{y},q'), \text{ where } q' \text{ is least number} < n \text{ with } |q'| \leq \underline{s} \\ & \quad \text{such that } F(\vec{y},q') \text{ is maximal among such numbers;} \\ q, & \text{otherwise;} \end{cases}$$

Note that since $\mathbf{Card}(\{\, q' \mid q' < n \,\}) = n = |\underline{n}|$, it follows that *MaxQ* is also basic feasible.

In our constructions we shall assume that we have defined BTLP procedures $\mathtt{bnd}_{k,d}$, F, MaxQ, $\mathtt{p}_k$, and Run that compute *bnd*, F, *MaxQ*, p_k, and *Run*, respectively.

Most of the construction below is centered around the following notions.

Definition 62.

(a) We say that q is a *depth-0 dominating query* relative to $(\vec{y}, i)$ if and only if $|F(\vec{y},q)| \geq \max\{\, |y| \mid y \text{ is a type-N value in } \vec{y} \,\}$.

(b) Suppose that $d' \in \{\, 1,\ldots,d \,\}$ and that $q_{d'-1} \in \mathrm{N}$. We say that q is a *depth-d' dominating query* relative to $(\vec{y}, i)$ and $q_{d'-1}$ if and only if $|q| \leq p_k(|F(\vec{y},q_{d'-1})|)$ and

$$|F(\vec{y},q)| \quad \geq \quad \max \left\{ |F(\vec{y},q')| \;\middle|\; \begin{array}{r} |q'| \leq p_k(|F(\vec{y},q_{d'-1})|) \text{ and } q' \text{ codes} \\ \text{a query made in the course of the} \\ \text{computation of } \mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell} \text{ on input } \vec{y} \end{array} \right\}.$$

$\diamond$

To see why dominating queries are important, we note:

Lemma 63. *Given, $\vec{y}$ and i, suppose q_0 is a depth-0 dominating query relative to $(\vec{y}, i)$ and, for each $d' = 1,\ldots,d$, suppose $q_{d'}$ is a depth-d' dominating query relative to $(\vec{y}, i)$ and $q_{d'-1}$. Then $bnd_{k,d}(|F(\vec{y},q_d)|)$ is an upper bound on the cost of running $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ on input $\vec{y}$.*

The proof is an easy consequence of the way the E-machines have been set up. Our constructions will find dominating queries through a series of approximations. It is useful to give a name to an almost final one of these approximations.

Definition 64. Suppose that $d' \in \{\, 1,\ldots,d \,\}$ and that $q_{d'-1} \in \mathrm{N}$. We say that q is a *depth-d' near-dominating query* relative to $(\vec{y}, i)$ and $q_{d'-1}$ if and only if $|q| \leq p_k(|F(\vec{y},q_{d'-1})|)$ and, for some q_d, a depth-d' dominating query relative to $(\vec{y}, i)$ and $q_{d'-1}$ we have that $\underline{2} \odot bnd_{k,d}(|F(\vec{y},q)|) \geq bnd_{k,d}(|F(\vec{y},q_d)|)$. $\diamond$

Procedure $\texttt{RunUntil}_1(\vec{\texttt{y}}, \texttt{i}\colon \mathrm{N}, \texttt{q}_0\colon \mathrm{N}, \texttt{q}_1\colon \mathrm{N})$

  var $\texttt{q}^{\max}\colon \mathrm{N}$, $\texttt{q}\colon \mathrm{N}$, $\texttt{s}_0\colon \omega$, $\texttt{t}_1\colon \omega$;

  . . . Initialize a simulation of $\mathbf{E}^{\vec{\sigma},\tau}_{\texttt{i},k,1,\ell}$ *on input* $\vec{\texttt{y}}$ *. . .*

  $\texttt{q}^{\max} := \texttt{q}_1$; $\texttt{s}_0 := \underline{m} \odot \texttt{p}_k\big(|\texttt{F}(\vec{\texttt{y}}, \texttt{q}_0)|\big)$; $\texttt{t}_1 := \underline{2} \odot \texttt{bnd}_{k,d}(|\texttt{F}(\vec{\texttt{y}}, \texttt{q}_1)|)$;

  Loop $\texttt{t}_1$ **with** $\texttt{s}_0$ **do**

    If the simulation has completed

      then break;

    else if the next step to be simulated is *not* a query

      then simulate the step;

    else (* *the next step is a query* *)

      Suppose the query in question is $(z_1, \ldots, z_c)$ to $\texttt{y}_j$;

      Set $\texttt{q} := [j, z_1, \ldots, z_c]$;

      If $\texttt{bnd}_{k,d}(\vec{\texttt{y}}, |\texttt{F}(\vec{\texttt{y}}, \texttt{q})|) > \texttt{t}_1$

        then Return $\texttt{q}$;

      If $|\texttt{F}(\vec{\texttt{y}}, \texttt{q})| > |\texttt{F}(\vec{\texttt{y}}, \texttt{q}^{\max})|$

        then $\texttt{q}^{\max} := \texttt{q}$;

      Simulate the query;

  Endloop;

  Return $\texttt{q}^{\max}$

End

Figure 12: The $\texttt{RunUntil}_1$ procedure

The Depth-1 Case

Our definition of $\texttt{U}^{\vec{\sigma},\tau}_{k,1,\ell}$ uses the aforementioned auxiliary procedures $\texttt{bnd}_{k,d}$, $\texttt{F}$, $\texttt{MaxQ}$, $\texttt{p}_k$, and $\texttt{Run}$ and two other procedures: $\texttt{FindDom}^1_1$ and $\texttt{RunUntil}_1$. $\texttt{FindDom}^1_1$ finds depth-1 dominating queries using $\texttt{RunUntil}_1$. $\texttt{RunUntil}_1$ takes an approximation to a depth-1 dominating query and returns either a depth-1 dominating query or else an improved approximation. The details of $\texttt{RunUntil}_1$, $\texttt{FindDom}^1_1$, and $\texttt{U}^{\vec{\sigma},\tau}_{k,1,\ell}$ are given in Figures 12 and 14.

Claim 1.1. *Fix an input* $(\vec{y}, i, q_0, q_1)$ *to* $\texttt{RunUntil}_1$*, where* q_0 *is a depth-0 dominating query. Let* $\underline{s_0}$ *be the value of* $\texttt{s}_0$ *and let* q' *be the value returned by the procedure on this input. Then either*

(i) q_1 *is a depth-1 near-dominating query relative to* $(\vec{y}, i)$ *and* q_0*, in which case* q' *is a depth-1 dominating query relative to* $(\vec{y}, i)$ *and* q_0*, or*

Procedure $\mathtt{FindDom}_1^1(\vec{\mathtt{y}}, \mathtt{i}\colon \mathrm{N}, \mathtt{q}_0\colon \mathrm{N})$
  var $\mathtt{q}_1\colon \mathrm{N}$, $\mathtt{s}_0\colon \omega$;
  $\mathtt{q}_1 := \mathtt{q}_0$; $\mathtt{s}_0 := \underline{m} \odot \mathtt{p}_k\big(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_0)|\big)$;
  Loop $\mathtt{s}_0$**1 with** $\mathtt{s}_0$ **do** $\mathtt{q}_1 := \mathtt{RunUntil}_1(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0, \mathtt{q}_1)$; **Endloop**;
  Return $\mathtt{MaxQ}(\vec{\mathtt{y}}, \mathtt{q}_1, \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|), \mathtt{s}_0)$
End

Figure 13: The $\mathtt{FindDom}_1^1$ procedure

(ii) $|q'| \leq \underline{s_0}$ *and* $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q')|) > \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q_1)|)$.

From the query size bound in the E-machine clocking scheme it follows that in the computation of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,1,\ell}$ on input $\vec{y}$, every query $[z_1, \dots, z_c]$ has $|z_1|, \dots, |z_c| \leq p_k(|F(\vec{y}, q_0)|)$ and hence, for each $j \leq u$, $\|[j, z_1, \dots, z_c]\| \leq m \cdot p_k(|F(\vec{y}, q_0)|) = s_0$. Therefore, the $\mathtt{s}_0$ bound in the **Loop** will be at least as large as an F-query that arises in the simulation of $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,1,\ell}$ on input $\vec{y}$.

If q_1 is a depth-1 near-dominating query relative to $(\vec{y}, i)$ and q_0, then the simulation runs to completion and $\mathtt{q}^{\max}$ ends up with the value of a depth-1 dominating query. So suppose q_1 is not near dominating. Then by Lemma 52 and the construction, $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,1,\ell}$ on input $\vec{y}$ must make a query q' such that $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q')|) > \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q_1)|)$ in which case the first such q' is returned, and hence (ii) and the claim follow.

Claim 1.2. *Fix an input* $(\vec{y}, i, q_0)$ *to* $\mathtt{FindDom}_1^1$. *Then the procedure returns a depth-1 dominating query relative to* $(\vec{y}, i)$ *and* q_0.

It follows as in the argument for Claim 1.1 that the **Loop**'s $\mathtt{s}_0$ bound will be at least as large as the values returned by the calls to $\mathtt{RunUntil}_1$ in the **Loop**. Let q_1^{fin} be the value of $\mathtt{q}_1$ as of the end of the **Loop**. It follows from the definition of $\mathtt{MaxQ}$ that if q_1^{fin} is a depth-1 dominating query relative to $(\vec{y}, i)$ and q_0, then so is the value returned by $\mathtt{FindDom}_1^1$. Thus suppose that q_1^{fin} is not dominating and that $\underline{s_0}$ is the value of $\mathtt{s}_0$. Then it follows from Claim 1.1 that the value of $\mathtt{q}_1$ changes with each iteration of the loop. Hence, by Claim 1.1(ii), the value of $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|)$ must at least double with each iteration. Thus,

$$\mathtt{p}_k\Big(|\mathtt{F}(\vec{\mathtt{y}}, q_1^{\text{fin}})|\Big) \quad \geq \quad \underline{2^{s_0+1}} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_0)|).$$

Procedure $\mathtt{U}^{\vec{\sigma},\tau}_{k,1,\ell}(\vec{\mathtt{y}}, \mathtt{i}\colon \mathrm{N})$

. . . declarations of $\mathtt{p}_k$, $\mathtt{Run}$, $\mathtt{RunUntil}_1$, $\mathtt{FindDom}^1_1$, *etc. . . .*

var $\mathtt{q}_0\colon \mathrm{N}$, $\mathtt{q}_1\colon \mathrm{N}$;

$$\mathtt{q}_0 := \begin{cases} [j], & \text{if there are type-N oracles in } \vec{\mathtt{y}} \text{ and } j \text{ is the least number} \\ & \quad \text{such that } \mathtt{y}_i \text{ is a type-N oracle of maximal length;} \\ [0], & \text{if there are no type-N oracles in } \vec{\mathtt{y}}; \end{cases}$$

$\mathtt{q}_1 := \mathtt{FindDom}^1_1(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0)$;

Return $\mathtt{Run}(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{p}_k(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|))$

End

Figure 14: The $\mathtt{U}^{\vec{\sigma},\tau}_{k,1,\ell}$ procedure

Since the value of $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_0)|)$ is positive and there are only $2^{s_0+1}-1$ many queries of length $\leq \underline{s_0}$, it follows from the definition of $\mathtt{MaxQ}$ that $\mathtt{FindDom}^1_1$ returns a q' of length $\leq \underline{s_0}$ such that

$$|\mathtt{F}(\vec{\mathtt{y}}, q')| \quad = \quad \max\{\, |\mathtt{F}(\vec{\mathtt{y}}, q)| \mid |q| \leq \underline{s_0}\,\}.$$

So, q' is a depth-1 dominating query relative to $(\vec{y}, i)$ and q_0. Thus the claim follows.

Claim 1.3. $\mathtt{U}^{\vec{\sigma},\tau}_{k,1,\ell}$ *is equivalent to* $\lambda\vec{y}, i\,.\,\mathbf{E}^{\vec{\sigma},\tau}_{i,k,1,\ell}(\vec{y})$.

Fix an input $(\vec{y}, i)$ to $\mathtt{U}^{\vec{\sigma},\tau}_{k,1,\ell}$. Clearly, the value of $\mathtt{q}_0$ is a depth-0 dominating query relative to $(\vec{y}, i)$. Hence, by Claim 1.2, the value of $\mathtt{q}_1$ is a depth-1 dominating query relative to $(\vec{y}, i)$ and $\mathtt{q}_0$. Therefore, by Lemma 63 and the definition of $\mathtt{Run}$, it follows that the procedure returns $\mathbf{E}^{\vec{\sigma},\tau}_{i,k,1,\ell}(\vec{y})$. Thus the claim, and the depth-1 case, follows.

The Depth-2 Case

Our definition of $\mathtt{U}^{\vec{\sigma},\tau}_{k,2,\ell}$ uses the auxiliary procedures $\mathtt{bnd}_{k,d}$, $\mathtt{F}$, $\mathtt{MaxQ}$, $\mathtt{p}_k$, and $\mathtt{Run}$ and three other procedure, $\mathtt{FindDom}^1_2$, $\mathtt{FindDom}^2_2$, and $\mathtt{RunUntil}_2$, that are based on the tricks used in the definitions of $\mathtt{RunUntil}_1$ and $\mathtt{FindDom}^1_1$ above. $\mathtt{FindDom}^1_2$ finds depth-1 dominating queries using $\mathtt{FindDom}^2_2$ and $\mathtt{RunUntil}_1$. $\mathtt{FindDom}^2_2$ takes an approximation to a depth-1 dominating query and returns

Procedure $\texttt{RunUntil}_2(\vec{\mathtt{y}}, \mathtt{i}\colon \mathrm{N}, \mathtt{q}_0\colon \mathrm{N}, \mathtt{q}_1\colon \mathrm{N}, \mathtt{q}_2\colon \mathrm{N})$

var $\mathtt{q}^{\max}\colon \mathrm{N}$, $\mathtt{q}\colon \mathrm{N}$, $\mathtt{s}_0\colon \omega$, $\mathtt{s}_1\colon \omega$, $\mathtt{t}_1\colon \omega$, $\mathtt{t}_2\colon \omega$;

... *Initialize a simulation of* $\mathbf{E}^{\vec{\sigma},\tau}_{\mathtt{i},k,2,\ell}$ *on* $(\vec{\mathtt{y}})$ ...

$\mathtt{q}^{\max} := \mathtt{q}_2$;

$\mathtt{s}_0 := \underline{m} \odot \mathtt{p}_k\big(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_0)|\big)$; $\quad \mathtt{s}_1 := \underline{m} \odot \mathtt{p}_k\big(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|\big)$;

$\mathtt{t}_1 := \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|)$; $\quad \mathtt{t}_2 := \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_2)|)$;

Loop $\mathtt{t}_2$ **with** $\mathtt{s}_1$ **do**

If the simulation has completed

then break;

else if the next step to be simulated is *not* a query

then simulate the step;

else (* *the next step is a query* *)

Suppose the query in question is $(z_1, \dots, z_c)$ to $\mathtt{y}_j$;

Set $\mathtt{q} := [j, z_1, \dots, z_c]$;

If $\big(\,\big|\mathtt{q}\big| \le \mathtt{s}_0$ & $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q})|) > \mathtt{t}_1\,\big)$ or $(\,\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q})|) > \mathtt{t}_2\,)$

then Return $\mathtt{q}$;

If $|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q})| > |\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}^{\max})|$

then $\mathtt{q}^{\max} := \mathtt{q}$;

Simulate the query;

Endloop;

Return $\mathtt{q}^{\max}$

End

Figure 15: The $\texttt{RunUntil}_2$ procedure

either a depth-2 dominating query or a better approximation to a depth-1 dominating query, or something that can be used to obtain a depth-1 dominating query (see Claim 2.2b(ii)). $\texttt{RunUntil}_2$ takes an approximation to a depth-1 dominating query and an approximation to a depth-2 dominating query and returns either a depth-2 dominating query, or else an improved approximation of either the depth-1 or the depth-2 dominating queries. $\texttt{RunUntil}_2$, $\texttt{FindDom}^2_2$, $\texttt{FindDom}^1_2$, and $\mathtt{U}^{\vec{\sigma},\tau}_{k,2,\ell}$ are sketched in Figures 15, 16, and 17.

Claim 2.1. *Fix an input* $(\vec{y}, i, q_0, q_1, q_2)$ *to* $\texttt{RunUntil}_2$*. Let* $\underline{s_0}$ *and* $\underline{s_1}$ *be the respective values of* $\mathtt{s}_0$ *and* $\mathtt{s}_1$ *and let* q' *be the value returned by the procedure on this input. Then either:*

(i) q_2 *is a depth-2 near-dominating query relative to* $(\vec{y}, i)$ *and* q_1*, in which*

Procedure $\mathtt{FindDom}_2^2(\vec{\mathtt{y}}, \mathtt{i}\colon \mathrm{N}, \mathtt{q}_0\colon \mathrm{N}, \mathtt{q}_1\colon \mathrm{N})$
  var $\mathtt{q}_2\colon \mathrm{N}$, $\mathtt{s}_0\colon \omega$, $\mathtt{s}_1\colon \omega$, $\mathtt{t}_1\colon \omega$;
  $\mathtt{q}_2 := \mathtt{q}_1$; $\mathtt{s}_0 := \underline{m} \odot \mathtt{p}_k\big(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_0)|\big)$; $\mathtt{s}_1 := \underline{m} \odot \mathtt{p}_k\big(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|\big)$;
  $\mathtt{t}_1 := \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|)$;
  Loop $\mathtt{s}_1$**1 with** $\mathtt{s}_1$ **do**
    $\mathtt{q}_2 := \mathtt{RunUntil}_2(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2)$;
    If $|\mathtt{q}_2| \leq \mathtt{s}_0$ & $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_2))|) > \mathtt{t}_1$ **then Return** $\mathtt{q}_2$;
  Endloop;
  Return $\mathtt{MaxQ}(\vec{\mathtt{y}}, \mathtt{q}_2, \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_2)|), \mathtt{s}_1)$
End

Procedure $\mathtt{FindDom}_2^1(\vec{\mathtt{y}}, \mathtt{i}\colon \mathrm{N}, \mathtt{q}_0\colon \mathrm{N})$
  var $\mathtt{q}_1\colon \mathrm{N}$, $\mathtt{s}_0\colon \omega$;
  $\mathtt{q}_1 := \mathtt{q}_0$; $\mathtt{s}_0 := \underline{m} \odot \mathtt{p}_k\big(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_0)|\big)$;
  Loop $\mathtt{s}_0$**1 with** $\mathtt{s}_0$ **do**
    If $|\mathtt{FindDom}_2^2(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0, \mathtt{q}_1)| > \mathtt{s}_0$ **then** break
    $\mathtt{q}_1 := \mathtt{FindDom}_2^2(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0, \mathtt{q}_1)$;
  Endloop;
  $\mathtt{t} := \max(\underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_1)|), \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{FindDom}_2^2(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0, \mathtt{q}_1))|))$;
  Return $\mathtt{MaxQ}(\vec{\mathtt{y}}, \mathtt{q}_1, \mathtt{t}, \mathtt{s}_0)$
End

Figure 16: The $\mathtt{FindDom}_2^2$ and $\mathtt{FindDom}_2^1$ procedures

case q' is a depth-2 dominating query relative to $(\vec{y}, i)$ and q_1, or
(ii) $|q'| \leq \underline{s_0}$ and $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q')|) > 2 \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q_1)|)$, or
(iii) $\underline{s_0} < |q'| \leq \underline{s_1}$ and $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q')|) > 2 \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q_2)|)$.

The argument for Claim 2.1 is a straightforward modification of the one given for Claim 1.1.

Claim 2.2. *Fix an input $(\vec{y}, i, q_0, q_1)$ to $\mathtt{FindDom}_2^2$. Let $\underline{s_0}$, and $\underline{s_1}$ be the respective values of $\mathtt{s}_0$ and $\mathtt{s}_1$ and let q' be the value returned by the procedure. Then:*

(a) If q_1 is a depth-1 near-dominating query relative to $(\vec{y}, i)$ and q_0, then q' is is a depth-2 dominating query relative to $(\vec{y}, i)$ and q_1.

Procedure $\mathtt{U}^{\vec{\sigma},\tau}_{k,2,\ell}(\vec{\mathtt{y}}, \mathtt{i}\colon \mathrm{N})$

... *declarations of* p_k, $\mathtt{Run}$, $\mathtt{RunUntil}_2$, $\mathtt{FindDom}^2_1$, $\mathtt{FindDom}^2_2$, *etc.* ...

var $\mathtt{j}\colon \mathrm{N}$, $\mathtt{q}_0\colon \mathrm{N}$, $\mathtt{q}_1\colon \mathrm{N}$, $\mathtt{q}_2\colon \mathrm{N}$;

$$\mathtt{q}_0 := \begin{cases} [j], & \text{if there are type-N oracles in } \vec{\mathtt{y}} \text{ and } j \text{ is the least number} \\ & \quad \text{such that } \mathtt{y}_i \text{ is a type-N oracle of maximal length;} \\ [0], & \text{if there are no type-N oracles in } \vec{\mathtt{y}}; \end{cases}$$

$\mathtt{q}_1 := \mathtt{FindDom}^1_2(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0)$;

$\mathtt{q}_2 := \mathtt{FindDom}^2_2(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{q}_0, \mathtt{q}_1)$;

Return $\mathtt{Run}(\vec{\mathtt{y}}, \mathtt{i}, \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, \mathtt{q}_2)|))$;

End

Figure 17: The $\mathtt{U}^{\vec{\sigma},\tau}_{k,2,\ell}$ procedure

(b) If q_1 *is not a depth-1 near-dominating query relative to* $(\vec{y}, i)$ *and* q_0*, then either:*

(i) $|q'| \leq \underline{s_0}$ *and* $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q')|) > \underline{2} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{y}, q_1)|)$ *or*

(ii) $\underline{s_0} < |q'| \leq \underline{s_1}$ *and* $\mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, q')|) > \underline{2^{s_0+1}} \odot \mathtt{bnd}_{k,d}(|\mathtt{F}(\vec{\mathtt{y}}, q_0)|)$.

Part (a) follows from Claim 2.1 and an argument similar to the one given for Claim 1.2. For part (b), suppose (i) does not hold. Then (ii) follows from Claim 2.1 and another argument similar to the one given for Claim 1.2. Thus, we have the claim.

Claim 2.3. *Fix an input* $(\vec{y}, i, q_0)$ *to* $\mathtt{FindDom}^1_2$*. Then the procedure returns a depth-1 dominating query relative to* $(\vec{y}, i)$ *and* q_0*.*

Let $\underline{s_0}$ be the value of $\mathtt{s}_0$. Let q_1^* be the value of $\mathtt{q}_1$ as of the end of the **Loop**. If q_1^* is a depth-1 near-dominating query relative to $(\vec{y}, i)$ and q_0, then the present claim follows as in the argument for Claim 1.2. So, suppose q_1^* is *not* near dominating. If the **Loop** terminated through the break, then by Claim 2.2(b.ii) and the definition of $\mathtt{MaxQ}$, the present claim follows. If the **Loop** did not terminate through the break, then by essentially the argument for Claim 1.2 again, we have that the present claim follows under this case too.

Claim 2.4. $\mathtt{U}^{\vec{\sigma},\tau}_{k,2,\ell}$ *is equivalent to* $\lambda\vec{y}, i \,.\, \mathbf{E}^{\vec{\sigma},\tau}_{i,k,2,\ell}(\vec{y})$.

This follows from Claims 2.2 and 2.3 by a straightforward modification of the argument for Claim 1.3.

The General Case

The general depth-d, $d > 2$, case is a fairly straightforward modification of the depth-2 case and is left to the reader.

19. Some Applications

The Grand Equivalence

Proposition 61 closes a ring of our chain of translations and containments. We thus have the following corollary.

Terminology: Let $\mathrm{ITLPF}_{\mathrm{Full}}$ and $\mathrm{ITLPF}_{\mathrm{BCont}}$, respectively, denote the class of ITLP-computable functions with respect to the **Full**- and **BCont**-based semantics. Similarly, define $\mathrm{ITLPF}^{\flat}_{\mathrm{Full}}$ and $\mathrm{ITLPF}^{\flat}_{\mathrm{BCont}}$ for the $\mathsf{ITLP}^{\flat}$ formalism.

Corollary 65.
(a) $\mathrm{BFF} = \mathrm{ITLPF}_{\mathrm{Full}} = \mathrm{ITLPF}^{\flat}_{\mathrm{Full}} = \mathrm{E}$.
(b) $\mathrm{BCBFF} = \mathrm{ITLPF}_{\mathrm{BCont}} = \mathrm{ITLPF}^{\flat}_{\mathrm{BCont}} = \mathrm{BCE}$.
(c) There are effective translations between each of BTLP, ITLP, $\mathsf{ITLP}^{\flat}$, *and* E*-machines that realize the equivalences of parts (a) and (b).*

Proof. This follows from Propositions 39, 44, 54, and 61. ☐

Sections

Recall from Definition 1 that for each $i \geq 1$, BFF_i is the class of type-level i functionals computed by BTLP procedures in which the allowable types are restricted to those of type-levels no greater than i and $\mathbf{Sec}_i(\mathrm{BFF})$ (the *i-section* of BFF) is the class of type-level i functionals computed by BTLP procedures with no restriction on allowable types. In his thesis Kapron [Kap91] proved $\mathbf{Sec}_1(\mathrm{BFF}) = \mathrm{PF}$. Our various translations now permit us to show

Proposition 66. *For each $i \geq 1$, $\mathbf{Sec}_i(\mathrm{BFF}) = \mathrm{BFF}_i$.*

Proof. Let $\mathtt{P}$ be a BTLP-program that computes a function f, where σ is of type σ of type-level $i \geq 1$. Let $\mathtt{P}'$ be the result of moving $\mathtt{P}$ through the chain of translations:

$$\mathsf{BTLP} \rightarrow \mathsf{ITLP} \rightarrow \mathsf{ITLP}^{\flat} \rightarrow \text{E-machines} \rightarrow \mathsf{BTLP}.$$

It is very likely that $\mathtt{P}'$ is a truly dreadful program, but we argue that it at least witnesses that $f \in \mathrm{BFF}_i$. The key step is the $\mathsf{ITLP} \rightarrow \mathsf{ITLP}^{\flat}$ translation that will, as a consequence of its other work, normalize away any procedures of type-level greater than its input procedure. Next observe that both of the $\mathsf{ITLP}^{\flat} \rightarrow$ E-machines and E-machines $\rightarrow \mathsf{BTLP}$ translations never introduce any machines or procedures that compute functions of type-level higher than the type-level of the input procedure. Hence it follows that $\mathtt{P}'$ is as desired. □

The Unit Cost Model

Throughout this paper and its predecessor we have used the *answer-length cost-model* for our higher-type machines. Under that model the cost of a query '$F(\vec{x}) =?$' is $\max(1, |F(\vec{x})|)$. We now consider the *unit cost-model* for higher-type machines, under which all queries have cost 1. We also consider the class of functionals computable under this cost model in time "polynomial" in the size of the inputs. As usual, "polynomial" here means a Seth bound in the case of the **Full**-based semantics and an HTP bound in the case of the **BCont**-based semantics. Note that there is a seeming miss-match between the cost of a query and the time granted to a computation because of a query. For example, suppose that answers to queries are written so that just after a query the answer-tape head starts on the least significant character of the answer. Then, in querying an $f\colon \mathrm{N} \rightarrow \mathrm{N}$, a machine might read just the least significant character of the answer to '$f(x) =?$' at cost 1, but be granted a large amount of additional time to finish its computation because $|f(x)|$ is huge. Thus, "polynomial" time-bounded machines under this cost model would appear to be quite powerful.

This puzzle is resolved by noting that a machine that reads only one character of an answer knows only that the length of the answer is at least 1. In order to be sure that $|f(x)|$ is huge, the machine must read a large portion of the answer. Thus, in order for a machine to guarantee that it runs under the appropriate polynomial bound, the machine has to base its computation on what it has actually observed of its inputs. We can formalize this observation through a revised clocking scheme for our higher-type machines. Under this revision, a clocked computation will keep a table of *all* the queries that have been made thus far and the observed lower bound of the length of the answer to each of these queries. (A particular query might be made several times with ever deeper readings into the answer.) This continuously updated information on lengths is used as data for essentially the clocking scheme described in Section 15.1, the difference being that in the revised scheme there is potentially an update to the estimate to the polynomial bound after each character of an answer is read. The overhead of this revised scheme is higher than the original, but it is still polynomial. The argument that these clocked machines are equivalent to the machines that respect the "polynomial" run-time bound is similar to the arguments for the analogous results of Sections I-6 and 15. We are now in a position to sketch a proof of the following result that was announced by Aleksandar Ignjatovic in 1994. (See Ignjatovic and Sharma [IS02] for a type-level 2 version of the general result.)

Terminology. Let $\widehat{\mathbf{E}}^{\vec{\sigma},\tau}_{i,k,d,\ell}$ denote the revision of the machines from Section 15.1 to conform to the new clocking scheme and let $\widehat{\mathrm{E}}$ denote the class of functionals computed by this revised class of machines.

Proposition 67. BFF $= \widehat{\mathrm{E}}$.

Proof Sketch. BFF $\subseteq \widehat{\mathrm{E}}$: Consider $\widehat{\mathrm{E}}$-machines that always read the entire answer to each query they make. Modulo some bookkeeping details, these $\widehat{\mathrm{E}}$-machines behave essentially as standard E-machines, and so the containment follows.

$\widehat{\mathrm{E}} \subseteq$ BFF: We note that the E-machine $\rightarrow$ $\mathsf{ITLP}^\flat$ translation adapts readily to provide an $\widehat{\mathrm{E}}$-machine $\rightarrow$ $\mathsf{ITLP}^\flat$ translation. Hence, by Corollary 65 this second containment follows. ☐

The **BCont** version of Proposition 67 follows by the same argument.

20. Conclusions

One of our main goals was to provide appropriate settings for Seth's E = BFF Theorem in order to extract full analogues of the Kapron-Cook Theorem. This we have clearly done, and in the process met another of our goals: explaining the senses in which the elements of D, BFF, BCD, and BCBFF are polynomial-time computable relative to their respective semantic settings. We also clarified the nature of Seth's class D to the point where we could provide proofs of BFF $\neq$ D and BCBFF $\neq$ BCD.

Given our work in this paper, are we any closer to understanding what the correct analogue of polynomial time is for higher-types? To this also we think we have enough data to give an answer: there is no such notion. Rather, different semantic domains can support distinct notions of feasibility. For example, since we now know that BCD $\neq$ BCBFF and D $\neq$ BFF, we have four contenders for the title of "higher-type polynomial time." In the context of **Full**-based semantics, BFF appears more natural than D, if only because it seems unlikely that D has a recursive presentation. In the context of **BCont**-based semantics, it is almost certain that BCD is recursively presentable and in fact has reasonable language characterizations, Thus in regard to a **BCont**-based semantics, BCD seems the more natural analogue of polynomial-time.

However, is either **Full** or **BCont** a reasonable setting for higher-type complexity theory? **Full** seems too large a domain for the study of computation, much less complexity theory. The difficulty is that **Full** contains elements that are extraordinarily far from the computable, and the presence of such monsters, notions of feasible computability (e.g., BFF) are necessarily quite constrained. For **BCont** we have the opposite problem: it may be too small a domain for complexity theory. The evidence for this is that, as noted in Remark 60, BCD contains some extremely powerful functionals. That is, because **BCont** is so small, notions of feasible computability (e.g., BCD) can be relatively unconstrained, and thus they can breed their own monsters.

These remarks on **Full** and **BCont** as suitable domains for complexity theory are quite provisional. Currently, we do not have firm criteria for judging a domain too large or too small, natural or unnatural.

Independent of how reasonable **Full** and **BCont** are, they are not the end of the story. There are other domains of computational interest (we have in mind the sequentially realizable functionals [vO99, Lon98] and the Scott continuous functionals [Sco93]), and they will have their own brands of feasibility. For the sequentially realizable functionals, there is a hint of what an appropriate notion of feasibility might look like in [Roy00]. For the continuous functionals, Appendix A sketches an example of a type-level 3 continuous functional that we argue is feasibly computable, but that is neither basic feasible nor in D. So while there is a fairly clear understanding of the class of basic feasible functions (due to the work of Cook, Urquhart, Seth, the present authors, and many others), in the realm of higher-type complexity there are still many other very basic questions to explore.

Acknowledgments

Thanks to Stuart Kurtz, Gary Leavens, Jack Lutz, Peter O'Hearn, Jean-Yves Marion, and Sue Older for their comments on various stages of this work. Thanks once again to Elaine Weinman for her careful comments on the text.

A. A Feasible Continuous Functional That Is Not Basic Feasible

Preliminaries. For each $x \in \mathrm{N}$, let $c_x = \chi_{\{\,2^x\,\}}$, the characteristic function of the singleton set $\{\,2^x\,\}$, and let $c_\infty = \lambda y\,.\,0$.

Claim 1. *Relative to multi-tape Turing machines:*
  (a) Given x and $y \in \mathrm{N}$, deciding whether $2^x = y$ takes $O(|x| + |y|)$ time.
  (b) Given x and $y \in \mathrm{N}$, computing $c_x(y)$ can be done in time $O(|x| + |y|)$.

Proof sketch. Part (a) is standard and part (b) follows immediately from part (a).

Now we define Φ, Ψ: $\big((\mathrm{N} \to \mathrm{N}) \to \{\,0,1\,\}\big) \to \{\,0,1\,\}$ by:

$$\Phi(F,x) \quad = \quad \begin{cases} 1, & \text{if } F(c_x) \neq F(c_\infty); \\ 0, & \text{otherwise.} \end{cases} \tag{22}$$

$$\Psi(F,x) \quad = \quad \begin{cases} 2^x, & \text{if } F(c_x) \neq F(c_\infty); \\ 0, & \text{otherwise.} \end{cases} \tag{23}$$

Claim 2.
  (a) (22) and (23) define total continuous functionals.
  (b) Φ is basic feasible.
  (c) Ψ is neither basic feasible nor is it in D.

Proof sketch. Part (a) is obvious from the definitions. Part (b) follows from Claim 1(b). Since F, c_∞, and all the c_x's are 0-1 valued, it is clear that there is no Seth bound in F and $|x|$ that provides an upper bound on $|\Psi(F,x)|$. Hence, part (c) follows.

We argue that Φ and Ψ have roughly the same computational complexity. That is,

Claim 3 (Informal). *Given any program p_Φ for Φ, there is a program p_Ψ for Ψ such that, in any reasonable, computational experiment, one has that,* *for all F and x, the run time of p_Ψ on (F, x) is no more than twice the run time of p_Φ on (F, x).*

The informality of Claim 3 rests on the meaning of "a reasonable, computational experiment." To explain what we mean by this, consider using an actual computer to run timing experiments on p_Φ and p_Ψ for various inputs F and x. To do this, F would be represented by some procedure (say p_F), x would be represented in dyadic or binary. Moreover, the run time of p_Φ (respectively, p_Ψ) would be the total observed time between starting the program and the time the program halts. This total run time thus includes the time costs of running the procedure p_F.

Proof sketch of Claim 3. Fix a p_Φ. The corresponding p_Ψ does the following on input (F, x):

Let y be the result of running p_Φ on (F, x).
If $y = 0$ then output 0 else output 2^x.

Now fix an input (F, x). There are clearly two cases to consider.

Case 1: $F(c_x) = F(c_\infty)$. For such inputs, the claim clearly holds.

Case 2: $F(c_x) \neq F(c_\infty)$. For all $y \in \mathrm{N}$, $c_x(y) = c_\infty(y)$ *except* when $y = 2^x$. Thus, since $F(c_x) \neq F(c_\infty)$, any procedure for F (recall that we are assuming F is represented by a procedure) must actually write down 2^x to make the query "$f(2^x) =?$" of both c_x and c_∞. Hence, the run-time of p_Φ on this (F, x) already includes the cost of writing down 2^x twice. Thus, the cost of p_Ψ writing down 2^x one more time for its output no more than doubles its run time over that of p_Φ. Therefore, the claim holds in this case also.

Intuitively, Ψ is feasible (in its arguments) because if Ψ observes that $F(c_x) \neq F(c_\infty)$, then Ψ can infer that the procedure for F, call it p_F, made a particular (big) query of its type-1 argument and so Ψ can output this (big) query in time commensurate with p_F's run time. So even though F is 0-1 valued and a "black box," information about the run time of p_F can leak out.

Suppose that in place of a program for F we had a demon, able to instantaneously do things far outside the computable, but which is cursed to write down queries to its type-level 1 argument one character per time step. Then the argument for Claim 3 applies to this situation as well. So it is more accurate to say that, even though F is a 0–1 valued "black box," information about F's modulus of continuity (see the proof of Proposition 2) can leak out. It is because of this observation and our remarks on modulus of continuity and compactness in Section 7 that we suspect that any notion of feasibility for the continuous functionals must take modulus of continuity into account, and hence must be quite different than the notions of higher-type feasibility studied above.

References

[BC92] S. Bellantoni and S. Cook, *A new recursion-theoretic characterization of the polytime functions*, Computational Complexity **2** (1992), 97–110.

[BNS00] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg, *Characterising polytime through higher type recursion*, Annals of Pure and Applied Logic (2000), 17–30.

[Bus86] S. Buss, *The polynomial hierarchy and intuitionistic bounded arithmetic*, Structure in Complexity Theory (A. Selman, ed.), Lecture Notes in Computer Science, vol. 223, Springer-Verlag, 1986, pp. 77–103.

[CF58] H. Curry and R. Feys, *Combinatory logic, Vol. I*, North-Holland, 1958.

[Chu40] A. Church, *A formulation of the simple theory of types*, Journal of Symbolic Logic **5** (1940), 56–68.

[CK89] S. Cook and B. Kapron, *Characterizations of the basic feasible functions of finite type*, Proceedings of the 30nd Annual IEEE Symposium on the Foundations of Computer Science, 1989, pp. 154–159.

[CK90] S. Cook and B. Kapron, *Characterizations of the basic feasible functions of finite type*, Feasible Mathematics: A Mathematical Sciences Institute Workshop (S. Buss and P. Scott, eds.), Birkhäuser, 1990, pp. 71–95.

[Cob65] A. Cobham, *The intrinsic computational difficulty of functions*, Proceedings of the International Conference on Logic, Methodology and Philosophy (Y. Bar Hillel, ed.), North-Holland, 1965, pp. 24–30.

[CU89] S. Cook and A. Urquhart, *Functional interpretations of feasibly constructive arithmetic*, Proceedings of the 21st Annual ACM Symposium on the Theory of Computing, 1989, pp. 107–112.

[CU93] S. Cook and A. Urquhart, *Functional interpretations of feasibly constructive arithmetic*, Annals of Pure and Applied Logic **63** (1993), 103–200.

[Gir98] J.-Y. Girard, *Light linear logic*, Information and Computation **143** (1998), 175–204.

[Göd58] K. Gödel, *Über eine bisher noch nicht benützte Erweiterung des finiten*, Dialectica **12** (1958), 280–287.

[Göd90] K. Gödel, *On a hitherto unutilized extension of the finitary standpoint*, Kurt Gödel: Collected Works, Volume II (S. Feferman, J. Dawson, S. Kleene, G. Moore, R. Solovay, and J. van Heijenoort, eds.), Oxford University Press, 1990, pp. 241–251.

[Gun92] C. Gunter, *Semantics of programming languages: Structures and techniques*, MIT Press, 1992.

[Hin97] J. R. Hindley, *Basic simple type theory*, Cambridge University Press, 1997.

[Hof97] M. Hofmann, *An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras*, Bulletin of Symbolic Logic **3** (1997), 469–486.

[Hof99a] M. Hofmann, *Linear types and non-size-increasing polynomial time computation*, Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, 1999.

[Hof99b] M. Hofmann, *Type systems for polynomial-time computation*, Habilitation thesis, Darmstadt, 1999, appears as University of Edinburgh LFCS Technical Report ECS-LFCS-99-406.

[Hof00] M. Hofmann, *Programming languages capturing complexity classes*, SIGACT News **31** (2000), 31–42.

[IKR01] R. Irwin, B. Kapron, and J. Royer, *On characterizations of the basic feasible functional, Part I*, Journal of Functional Programming **11** (2001), 117–153.

[IS02] A. Ignjatovic and A. Sharma, *Some applications of logic to feasibility in higher types*, available as http://xxx.lanl.gov/abs/cs.LO/0204045, 2002.

[Kap91] B. Kapron, *Feasible computation in higher types*, Ph.D. thesis, Department of Computer Science, University of Toronto, 1991.

[KC91] B. Kapron and S. Cook, *A new characterization of Mehlhorn's polynomial time functionals*, Proceedings of the 32nd Annual IEEE Symposium Foundations of Computer Science, 1991, pp. 342–347.

[KC96] B. Kapron and S. Cook, *A new characterization of type 2 feasibility*, SIAM Journal on Computing **25** (1996), 117–132.

[Kle60] S. Kleene, *Turing machine computable functionals of finite types I*, Proc. of the 1960 Congess for Logic, Methodology and the Philosophy of Science (P. Suppes, ed.), Stanford University Press, 1960, pp. 38–45.

[Kle62] S. Kleene, *Turing-machine computable functionals of finite types II*, Proceedings of the London Mathematical Society **12** (1962), 245–258.

[Kle63] S. Kleene, *Recursive functionals and quantifiers of finite types, II*, Transactions of the American Mathematical Society **108** (1963), 106–142.

[Koz80] D. Kozen, *Indexings of subrecursive classes*, Theoretical Computer Science **11** (1980), 277–301.

[Lam80] J. Lambek, *From λ-calculus to cartesian closed categories*, To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (J. Seldin and J. Hindley, eds.), Academic Press, 1980, pp. 375–402.

[Lei94] D. Leivant, *A foundational delineation of poly-time*, Information and Computation **110** (1994), 391–420.

[Lei95] D. Leivant, *Ramified recurrence and computational complexity I: Word recurrence and poly-time*, Feasible Mathematics II (P. Clote and J. Remmel, eds.), Birkhäuser, 1995, pp. 320–343.

[LM93] D. Leivant and J.-Y. Marion, *Lambda calculus characterizations of polytime*, FundamentæInformaticæ**19** (1993), 167–184.

[LM02] D. Leivant and J.Y. Marion, *Predicative recurrence and computational complexity IV: Predicative functionals and poly-space*, Information and Computation (2002), To appear.

[Lon98] J. Longley, *The sequentially realizable functionals*, Tech. Report ECS-LFCS-98-402, Dept. of Computer Science, University of Edinburgh, 1998, to appear in *Annals of Pure and Applied Logic*.

[Lon01] J. Longley, *Notions of computability at higher types, I*, draft manuscript available from http://www.dcs.ed.ac.uk/∼jrl/, 2001.

[LV97] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications, second edition*, Springer-Verlag, 1997.

[Nor99] D. Normann, *The continuous functionals*, Handbook of Computability Theory (E. R. Griffor, ed.), North-Holland, 1999, pp. 251–275.

[Odi81] P. Odifreddi, *Strong reducibilities*, Bulletin of the American Mathematical Society **4** (1981), 37–86.

[Pla66] R. Platek, *Foundations of recursion theory*, Ph.D. thesis, Stanford University, 1966.

[RC94] J. Royer and J. Case, *Subrecursive programming systems: Complexity & succinctness*, Birkhäuser, 1994.

[Rog67] H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967, reprinted, MIT Press, 1987.

[Roy00] J. Royer, *On the computational complexity of Longley's H functional*, presented at ICC'00, 2000.

[Saz76] V. Sazonov, *Expressibility of functions in Scott's LCF language*, Algebra and Logic **15** (1976), 308–330.

[Sch76] H. Schwichtenberg, *Definierbare funktionen im λ-kalkül mit typen*, Arch. Math. Logik **17** (1976), 113–114.

[Sco93] D. Scott, *A type-theoretical alternative to ISWIM, CUCH, and OWHY*, Theoretical Computer Science **121** (1993), 411–440, Orginally written in 1969.

[Set92] A. Seth, *There is no recursive axiomatization for feasible functionals of type 2*, Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, 1992, pp. 286–295.

[Set94] A. Seth, *Complexity theory of higher type functionals*, Ph.D. thesis, University of Bombay, 1994.

[Set95] A. Seth, *Turing machine characterizations of feasible functionals of all finite types*, Feasible Mathematics II (P. Clote and J. Remmel, eds.), Birkhauser, 1995, pp. 407–428.

[vO99] J. van Oosten, *A combinatory algebra for sequential functionals of finite type*, Models and Computability (Leeds, 1997) (S. Cooper and J. Truss, eds.), Cambridge University Press, 1999, pp. 389–405.