On Characterizations of the Basic Feasible Functionals, Part I

— Draft —

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

21 July 1999

Abstract

We introduce a typed programming formalism, type-2 *inflationary tiered loop programs* or ITLP_2, that characterizes the type-2 basic feasible functionals. ITLP_2 is based on Bellantoni and Cook's [BC92] and Leivant's [Lei95] type-theoretic characterization of polynomial-time and turns out to be closely related to Kapron and Cook's [KC91, KC96] machine-based characterization of the type-2 basic feasible functionals.

[*]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 grant CCR-9522987.

1. Introduction

Higher-type functionals and operators have proven to be valuable tools in both theoretical and practical work on programming languages. Indeed, classes, modules, and their kith are—at least in part—higher-type notions, so features of higher-types pervade contemporary computing. There is a great deal of useful theoretical work in support of reasoning about the correctness of programs that make use of higher-order features. In contrast, there has been very little theoretical work in support of reasoning about the performance (e.g., time and space complexity) of such programs. It is clearly a great folly to ignore correctness in program development, but it is nearly as great a folly to ignore performance. Thus there is a serious gap in the scientific underpinnings of programs that use higher-type features—even benchmarking a higher-type procedure is problematic in the absence of a theory to help interpret what the numbers mean.

There is a small body of complexity-theoretic work on higher-type functionals. Most of this work has concentrated on the study of the *basic feasible functionals*. These are a class of functionals, at all simple types, that were intended to be a higher-type analogue of the class of polynomial-time computable functions. The extent to which the basic feasible functionals (BFFs) realize this intent is not yet clear and a focus of current research. But independent of such questions, the BFFs have proven to be a robust class whose various characterizations contain many ideas of independent interest. This paper and its sequel consider several of these characterizations—some old, some new—with the goal of clarifying their underlying ideas and placing these ideas into something like a uniform framework. The end result of this is, we hope, a much clearer picture of the BFFs that is accessible to the programming languages *and* complexity theoretic communities.

Very briefly, this paper concerns characterizations of the type-2 basic feasible functionals, and its sequel concerns the extensions of these characterizations to all simple types. The next section provides some background on some of the prior work on the computational complexity of higher-type functionals. The section following gives an overview of this particular paper and its objectives.

2. A Bit of Background

Initial Notation: N denotes the set of natural numbers. We identify each $x \in \mathrm{N}$ with its dyadic representation over $\{\mathbf{0},\mathbf{1}\}$. So, there is a one-to-one correspondence between N and $\{\mathbf{0},\mathbf{1}\}^*$. Unless we state otherwise, $A \rightarrow B$

denotes the collection of *all* set-theoretic functions from A to B.

Computability at higher-types

Higher-type recursion and computability has a long, rich, and confusing history, some of which is recounted by Gandy and Hyland [GH77] and Cook [Coo91]. The most dramatic difference between ordinary and higher-type computability is that there is no higher-type analogue of Church's Thesis. The difficulty is this: Any notion of higher-type computability starts with the finitary notion of computation and adds to it infinitary objects, the functional parameters. To make such an addition, one must choose what a computation can know about and do with these infinite objects. Different choices can lead to conflicting notions of higher-type computability. We shall see examples of such conflicting notions below; the aforementioned surveys discuss more drastic cases.

Constable's problem and its rationale

Subrecursive higher-type recursion and computation also has a long history—Hilbert [Hil25, Hil67] made use of subrecursive higher-type schemes in his proof theoretic work. In complexity theory, higher-type computability seems to have cropped up first as an object of study (as opposed to a tool) in the early 1970s. Cook reductions [Coo71] are polynomial-time functionals of type $2^{\mathrm{N}} \times \mathrm{N} \to \mathrm{N}$. Constable, in a 1973 paper [Con73], posed the problem of finding a natural analogue of polynomial-time computability for functionals of type $(\mathrm{N} \to \mathrm{N})^k \times \mathrm{N}^\ell \to \mathrm{N}$. This problem has been a dominant issue in the sporadic work on the complexity of higher-type computation since 1973. One might well ask:

1. Why has *this* problem been so prominent?

2. Why, after all this time, is the thing still open?

The first question is easy to answer. A good deal of computational complexity theory concerns trying (and occasionally succeeding) to draw a sharp boundary between the computationally feasible and the computationally infeasible. "Computational feasibility" is a pre-formal notion with a meaning that shifts with the context. However, polynomial-time (in the guises of P, the class of polynomial-time decidable sets, and PF, the class of polynomial-time computable functions) has proven itself to be a good, pliable, first approximation to feasibility. This is because

(i) most tasks that one could consider feasible have polynomial-time algorithms,

(ii) most tasks that sit outside of polynomial-time seem quite infeasible,

(iii) almost all reasonable models of deterministic computation are polynomially related,[1] and

(iv) both P and PF have good closure properties.

So if we want to understand type-2 (and higher) computational complexity at a useful level of detail, then trying to identify what corresponds to polynomial-time is a sensible first step. For the second question we simply note that as there is no higher type Church's thesis, even formally posing the problem may itself be difficult.

The emergence of the basic feasible functionals

Mehlhorn took up Constable's problem in [Meh74, Meh76]. He defined a class of type-2 functionals, denoted $\mathcal{L}()$, through a careful relativization of Cobham's [Cob65] syntactic characterization of polynomial-time. (Cobham's characterization is formally stated as Theorem 2(a) below.) Mehlhorn developed some evidence that this class was a type-2 analogue to PF, but his main motivation was to show that $\mathcal{L}()$ is a sensible extension of Cook reducibilities to function classes. In the ten years following Mehlhorn's papers, there was relatively little work in this area. Then, in 1986 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. While these functionals served his needs, their definition was rather involved. Cook and Urquhart decided to try to develop a simpler class of realizers for IS^1_2 that could be presented as a feasible variant of Gödel's *Dialectica* interpretation of Heyting Arithmetic [Göd58, Göd90]. Their approach to this problem was, like Mehlhorn's work, based on Cobham's [Cob65] syntactic definition of polynomial-time. (Their initial work was independent of Constable and Mehlhorn.) They introduced a

[1] Suppose M_0 and M_1 are two models of computation with associated cost models. M_0 and M_1 are called *polynomially related* if and only if there exist a polynomial q and t_0, $t_1 \in$ PF such that, for $i = 0, 1$:

(a) for each M_i-program p, $t_i(p)$ is a semantically equivalent M_{1-i}-program, and

(b) for all p and x, the M_{1-i}-cost of running $\mathcal{M}_{1-i}$-program $t_i(p)$ on input x is no greater than $q(c_{i,p,x})$, where $c_{i,p,x}$ is the M_i-cost of running M_i-program p on input x.

See Jones's text [Jon97] for a careful, complexity-theoretic treatment of the equivalence of machine models.

formal system PV^ω in which the terms consist of simply typed λ-expressions built from numeric constants, function constants for each element of PF, variables of all finite types, and a type-2 recursor R that corresponds to Cobham's limited recursion on notation (see Definition 1 below). They showed that the resulting class of functionals nicely satisfied their goals [CU89, CU93]. This class of functionals (at all simple types) was named the *basic feasible functionals* (abbreviated, BFF) in Cook and Kapron's work [CK89, CK90], where it was shown that Mehlhorn's $\mathcal{L}()$ exactly corresponds to the type-2 BFFs. The "basic" in "basic feasible functionals" is meant to suggest that any natural higher-type analogue of PF should include the BFFs, but it leaves open the possibility that the BFFs may be too small to be this analogue.

The robustness of the basic feasible functionals

The issue of the naturalness of BFFs as higher-type analogues of PF was taken up by Cook and Kapron in [CK89, CK90]. They establish several function-algebra characterizations of the BFFs and showed that the BFFs satisfy a Ritchie-Cobham property[2] at all simple types. The naturalness question was also a central issue in Cook's [Coo91], where he stated two serious reservations about the type-2 BFFs. *Notation:* Let BFF_2 denote the class of type-2 BFFs (defined formally in Definition 4 below).

Cook's first reservation was based on an example he gave of a functional, L, that was outside of BFF_2, but nonetheless appeared feasible. The criteria for feasibility consisted of three properties that, he proposed, should necessarily be satisfied by any class that is a natural type-2 analogue of PF. The appropriate closure of BFF_2 and L turned out to satisfy these three conditions. Seth pursued this issue in [Set92] and [Set93] (also see [Set94]). He presented a complexity-theoretic based class of functionals that included L and satisfied the three conditions [Set92]. However, he also showed [Set93] that both his class and Cook's $\mathrm{BFF}_2 + L$ class have some undesirable complexity-theoretic properties. Seth proposed a reasonable fourth naturalness property, but in recent work Pezzoli [Pez98] shows that these four properties fail to characterize BFF_2. Pezzoli proposes her own fourth condition that, together with Cook's conditions, does characterize BFF_2, but whether her condition begs the question in some sense is not presently clear.

Cook's second reservation was based on the lack of a pure machine characterization of BFF_2; all of the then-known characterizations of BFF_2 involved

[2]This is an important complexity-theoretic closure property. Very roughly, a collection of functions $\mathcal{C}$ has the Ritchie-Cobham property if and only if, for each $f \in \mathcal{C}$, there is a program p_f for computing f such that $\lambda x.$[the run-time of p_f on x] is also in $\mathcal{C}$.

function algebras in an essential way. In theoretical computer science, machine models and characterizations have been *the* basis for quantitative reasoning about the use of computational resources. Thus, if BFF_2 is a true "type-2 complexity class," it should have a purely machine-based characterization. Such a machine characterization was provided by Kapron and Cook [KC91, KC96]. They introduced the notion of the *length* of a type-1 object and a type-2 notion of polynomial, and proved that the oracle Turing machines that run in time (second-order) polynomial in the lengths of their (type-1 and type-0) inputs compute exactly the type-2 BFFs. (These notions are treated in Section 6.) Seth [Set95] extended Kapron and Cook's argument to give a machine characterization of the BFFs at all simple types. (Seth's characterization is treated at some length in this paper's sequel.) In related work, Royer [Roy97] showed that BFF_2 satisfies a weak analogue of the Kreisel-Lacombe-Shoenfield Theorem [KLS57].

While the evidence is still incomplete, BFF_2 has proven to be a strong candidate for the natural type-2 analogue of PF. For the BFFs at type-3 and above the situation is much less clear, as discussed in this paper's sequel.

3. The Plan of This Paper

In the early 1970s there was an interest among complexity theorists in axiomatically characterizing the *natural* complexity measures—a problem that is still largely open and "not quite abandoned." In commenting upon this work Young [You90, p. 32, fn. 6] notes that the contemporary understanding of what constitutes a natural complexity measure or class is based on two things: (i) well-understood, concrete, computational models and (ii) low complexity translations between these models which establish model equivalences, e.g., polynomial relatedness of footnote 1. Thus when a complexity theorist proves something about a specific model of computation, it is understood (but seldom made explicit) that the result in fact holds for a certain equivalence class of models. This state of affairs is perhaps not mathematically dignified, but it has been remarkably productive.

This paper and its sequel take the complexity-theoretic route of studying specific formalisms and computational models for the BFFs and examining how closely these are related. Our particular interest is in machine/resource-bound characterizations and closely related programming formalisms. The motivation for this focus in only partly complexity theoretic—as we will see, some algorithms have very straightforward expressions in certain formalisms, but seemingly only very clumsy expressions in others. We are thus broadening the BFF naturalness question to include the naturalness and character of the programming formalisms for the BFFs and related classes. We are far from a final resolution of these questions which, because of their intensional character, may not have tidy answers. Be that as it may, this study is a means of addressing some fundamental issues and developing some ideas and observations that may lead to a deeper understanding of the underlying problems.

This paper considers three characterizations of BFF_2.

The *type-2 bounded typed loop programs* (abbreviated, BTLP_2) of Cook and Kapron [CK89, CK90] are introduced in Section 5. BTLP_2 is a simply-typed, imperative programming formalism with a loop construct directly inspired by Cobham's limited recursion on notation. Our official definition of BFF_2 will be as the BTLP_2-computable functionals.

Kapron and Cook's [KC91, KC96] type-2 *basic polynomial-time functionals* are developed in Section 6. This is a class of type-2 functionals computed by "polynomial-time bounded" oracle Turing machines. To formalize what "polynomial-time bounded" should mean in this context, we also introduce the notions of the length of a type-1 object and of second-order polynomials. The Kapron-Cook Theorem (Theorem 9 below) states the equivalence of the type-2 *basic polynomial-time functionals* and BFF_2. To clarify a uniformity issue with this theorem, we discuss Seth's notion of *polynomially-clocked* oracle Turing machines. In many ways this machine-based characterization gives a much clearer picture of BFF_2 than does the BTLP_2 characterization, but in other ways it may appear rather *ad hoc* by the standards of the programming language community.

To better explain this machine-based model, we introduce our *type-2 inflationary tiered loop programs* (abbreviated ITLP_2) formalism in Section 8, with Section 7 laying the groundwork for its definition. ITLP_2 is a typed programming formalism inspired by Bellantoni and Cook's [BC92] and Leivant's [Lei95] type-theoretic characterizations of PF. ITLP_2 is nonetheless very close to the polynomially-clocked oracle Turing machine scheme. The price for this closeness is that certain types and iteration bounds are inflationary in the sense that in the course of a computation they can grow (inflate) with the increase of information about the type-1 arguments. ITLP_2 is original to this paper.

The sequel to this paper uses extensions of these three formalisms to higher-types as the basis for investigation of the BFFs of type-3 and beyond. The ITLP_2 formalism is, however, interesting in its own right as an example of how to use types to capture complexity-theoretic notions.

4. Notation, Conventions, and Such

Numbers and strings

As stated before, 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 sometimes 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 and we treat the elements of ω as numbers, but we identify each $n \in \omega$ with $\underline{n}$. 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 use the elements of ω as tallies to represents lengths, run times, and, generally, the results of size measurements. This is a common type distinction that is implicit in much complexity theory. We make this distinction explicit here—probably to the confusion of all concerned.

Functions and operations

Unless otherwise stated, $A \to B$ denotes the collection of *all* total functions from A to B, and $A \rightharpoonup B$ denotes the collection of *all* (possibly) partial functions from A to B.

For $a_0, \ldots, a_n \in \mathrm{N}$, $\max(\{a_0, \ldots, a_n\}) = \max(a_0, \ldots, a_n) =$ the largest element of the set $\{a_0, \ldots, a_n\}$. By convention, $\max(\emptyset) = 0$. We also have occasion to use the notation: $a \oplus b = \max(a, b)$ and $\bigoplus_{i=m}^{n} a_i = \max(\{a_i \mid m \leq i \leq n\})$. For each x and $y \in \mathrm{N}$, define

$$x \# y \quad = \quad 2^{\ell en(x) \cdot \ell en(y)}.$$

Note that $|x| \cdot |y| = |x \# y|$. So $\#$ (called *smash*) can be thought of 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, p. 64]. By convention, $\langle x \rangle = x$ and $\langle x_0, \ldots, x_n \rangle = \langle x_0, \langle x_1, \ldots, x_n \rangle\rangle$.

Suppose $\alpha\colon X \rightharpoonup Y$ and $A \subseteq X$. Then

$$\alpha|_A \quad = \quad \begin{cases} \alpha(x), & \text{if } x \in A; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

For any given set X, $\mathbf{id}_X$ denotes the identity function on X. Suppose $f\colon X \to X$. $f^{(0)} = \mathbf{id}_X$, and, for each n, $f^{(n+1)} = f \circ f^{(n)}$.

Suppose A is a set and B is a set with a total order $\leq$. For $f, g\colon A \to B$ we write $f \leq g$ if and only if, for all $a \in A$, we have $f(a) \leq g(a)$. Suppose $\mathcal{C}$ is a collection of functions of type $A \to B$. We say that $\mathcal{C}_0$ is *cofinal* in $\mathcal{C}$ if and only if $\mathcal{C}_0 \subseteq \mathcal{C}$ and, for each $f \in \mathcal{C}$, there is a $f_0 \in \mathcal{C}_0$ such that $f \leq f_0$.

PF and Cobham's characterization of it

We formally define PF as the class of functions $\subset \bigcup_{k>0}(\mathrm{N}^k \to \mathrm{N})$ such that for each $f\colon \mathrm{N}^k \to \mathrm{N}$ in PF, there are M_f and q_f such that M_f is a deterministic Turing Machine, q_f is a polynomial, and for each input $(x_1, \ldots, x_k)$,

1. M_f outputs $f(\vec{x})$, and
2. M_f runs within $q_f(|x_1|, \ldots, |x_k|)$ time steps.

Definition 1. Suppose $f, h\colon \mathrm{N}^{n+1} \to \mathrm{N}$, $g_0\colon \mathrm{N}^n \to \mathrm{N}$, and $g_1\colon \mathrm{N}^{n+2} \to \mathrm{N}$.

(a) We say that f is defined by *recursion on notation* via g_0 and g_1 if and only if f is given by the following equation:

$$f(x, \vec{y}) \quad = \quad \begin{cases} g_0(\vec{y}), & \text{if } x = 0; \\ g_1(f(\lfloor x/2 \rfloor, \vec{y}), x, \vec{y}), & \text{if } x > 0. \end{cases} \tag{1}$$

(b) We say that f is defined by *limited recursion on notation* via g_0, g_1, and h, if and only if, for all x and $\vec{y}$, both (1) and

$$|f(x, \vec{y})| \quad \leq \quad |h(x, \vec{y})|$$

are satisfied. $\diamond$

Theorem 2 (Cobham [Cob65]).

(a) Let $\mathcal{P}$ be the smallest collection of functions that contains the initial functions $\lambda x.0$, $\lambda x.2x$, $\lambda x.2x+1$, $\lambda x, y.x \# y$, and, for all n and all $j \leq n$, $\lambda x_0, \ldots, x_n.x_j$ and that is closed under composition and limited recursion on notation. Then, $\mathcal{P} = \mathrm{PF}$.

(b) Let $\mathcal{P}'$ be the smallest collection of functions that contains the same initial functions as $\mathcal{P}$ and is closed under composition and recursion on notation. Then $\mathcal{P}' =$ the class of primitive recursive functions.

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 closed cartesian 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 as *level*(τ), is defined by:

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

Semantics. Base types are usually identified with particular sets in mind and $\sigma \to \tau$ is interpreted as being some class of functions from the set named by σ to the set named by τ. In this paper (but not its sequel) we always take $\sigma \to \tau$ as denoting the class of *all* functions from the set named by σ to the set named by τ. We usually use our standard notation for a set serving as a base type as notation for that type, e.g., N in $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$.

5. Type-2 Bounded Typed Loop Programs

Our official definition of the type-2 basic feasible functionals will be through Cook and Kapron's BTLP (Bounded Typed Loop Programs), a straightforward, simply-typed, imperative, programming formalism. The version of BTLP presented here and this paper's sequel differs in several ways from the original version in [CK90], but the two versions are essentially equivalent. This section describes BTLP_2, a type-2 fragment of BTLP. A description of the full version of BTLP can be found in Part II.

In BTLP_2 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_2 are the simple types over N of type-levels 0, 1, and 2. The grammar of our version of BTLP_2 is given below. The intuition behind each of the syntactic categories is: $P \equiv$ procedure declarations, $V \equiv$ local variable declarations, $I \equiv$ instructions, $L \equiv$ loop statements, $E \equiv$ natural number valued expressions, and $v \equiv$ variables.

$$\begin{array}{rcl}
P & ::= & \textbf{Procedure}\ v_0\,(v_1,\ldots,v_\ell)\ P^*\ V\ I^*\ \textbf{Return}\ v_r^{\mathrm{N}}\ \ \textbf{End} \\
V & ::= & \textbf{var}\ v_1^{\mathrm{N}},\ldots,v_m^{\mathrm{N}}\ ; \\
I & ::= & L\ ;\ \mid v^{\mathrm{N}} := E\ ; \\
L & ::= & \textbf{Loop}\ v_0^{\mathrm{N}}\ \textbf{with}\ v_1^{\mathrm{N}}\ \textbf{do}\ I^*\ \textbf{Endloop} \\
E & ::= & 1 \mid v^{\mathrm{N}} \mid v_0^{\mathrm{N}} + v_1^{\mathrm{N}} \mid v_0^{\mathrm{N}} \dot{-} v_1^{\mathrm{N}} \mid v_0^{\mathrm{N}} \# v_1^{\mathrm{N}} \mid v_0\,(v_1,\ldots,v_n)
\end{array}$$

In the procedure declaration production, the declared variable v_0 has type $\tau_1 \times \cdots \times \tau_\ell \to \mathrm{N}$, where for $i = 1,\ldots,\ell$, v_i has type τ_i, and similarly with the procedure application clause of the last production. All variables must be declared either in procedure declarations, parameter lists, or in local variable declarations. Variable scoping is static and follows standard conventions except that we forbid recursive procedure calls and nonlocal references to variables of type N.

Convention: In writing BTLP_2 programs, we shall set key words in bold roman font and variables in typewriter font. Variables in italic font are meta-variables, e.g., they stand for a syntactic category, a semantic object, etc. The range of these meta-variables should be clear from context.

The semantics of BTLP_2 is quite conventional and we shall discuss only its key points. Parameter passing is call-by-value. From this and our conventions of global variables, it thus follows that procedure calls, and expressions in general, have no side-effects. Type N variables local to a block are initialized to 0 every time the block is entered. In expressions, "+" denotes addition, "$\dot{-}$" denotes proper subtraction, and "#" denotes the smash function. The effect of a loop statement of the form

$$\textbf{Loop}\ \mathtt{w}\ \textbf{with}\ \mathtt{x}\ \textbf{do}\ I_1\ \cdots\ I_n\ \textbf{Endloop}$$

is to iterate on $I_1 \cdots I_n$ $|\mathtt{w}|$-many[3] times *with the following restrictions:* neither $\mathtt{w}$ nor $\mathtt{x}$ 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.

Since the types are simple and there are no recursive calls in procedures, one can give a straightforward, inductive semantics to the language. Also, since the only global references are to variables with immutable bindings and since parameter passing is call-by-value, it follows that the semantics

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

of $\mathtt{x}_0(\mathtt{x}_1, \dots, \mathtt{x}_n)$ is simply the application of the function named by $\mathtt{x}_0$ to the values named by $\mathtt{x}_1, \dots, \mathtt{x}_n$.

In BTLP_2 procedures one can—by standard, hoary tricks—achieve the effect of **If**-statements, a richer set of expressions, and so on. Cook and Kapron [CK89, CK90] provide details on this. In writing BTLP_2 procedures we shall blithely make use of these obvious extensions.

Example 3. $T = \lambda f, x . \sum_{i<|x|} f(i)$ is computed by the following BTLP_2 procedure.

```
Procedure SumUp(f: N → N, x: N)
  var bnd, maxarg, sum, i;
  maxarg := 0;    i := 0;
  Loop x with x do
    If f(maxarg) < f(i) then maxarg := i; Endif;
    i := i + 1;
  Endloop;
  bnd := (x + 1) · (f(maxarg) + 1);
  sum := 0;    i := 0;
  Loop x with bnd do
    sum := sum + f(i);
    i := i + 1;
  Endloop;
  Return sum
End
```

The first loop finds a value in $\{0, \dots, |\mathtt{x}| - 1\}$ on which `f` (or more properly, the denotation of `f`) is maximal. Once this value is obtained, an upper bound on the sum is computed and used as the size bound in the second loop that actually computes the sum. ◇

The basic feasible functionals were first defined as the PV^ω-computable functionals and later characterized via BTLP [CK89, CK90]. Here we formally define the type-2 BFF's through BTLP_2.

Definition 4. The *type-2 basic feasible functionals* (abbreviated, BFF_2) are the BTLP_2-computable functionals. ◇

6. Type-2 Basic Polynomial-Time Functionals

Recall that our formal definition of PF was that $f\colon \mathrm{N}^k \to \mathrm{N}$ (for $k > 0$) is in PF if only if there are M_f, a deterministic Turing Machine, and q_f, a polynomial, such that, for each input $(x_1, \ldots, x_k)$,

1. M_f outputs $f(x_1, \ldots, x_k)$, and
2. M_f runs within $q_f(|x_1|, \ldots, |x_k|)$ time steps.

An analogous notion for functionals, say of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$, would run something like: the collection of all functionals F for which there is a deterministic machine $\mathbf{M}_F$ such that for each f and x, the machine $\mathbf{M}_F$ outputs $F(f, x)$ and $\mathbf{M}_F$ runs in time "polynomial" in the "lengths" of f and x. So, to formalize this notion, we must specify (i) what the machine-based model of computation is, (ii) what the "length" an object of type $\mathrm{N} \to \mathrm{N}$ should be, and (iii) what "polynomial" should mean in this context.

Machines

Turing Machines, while anachronistic in many respects, have a conservative and fairly unproblematic cost model (each operation is small and takes unit time), and admit some wonderfully delicate complexity analyses. We shall thus follow complexity-theoretic tradition and make deterministic, multi-tape Turing machines (TMs) our default model of computation. To deal with arguments of type $\mathrm{N} \to \mathrm{N}$, we need to consider *oracle Turing machines* (OTMs). Under our setup OTMs have two special tapes: a *query tape* and a *reply tape.* To make a query of an oracle f, an OTM writes a **0**-**1** string (interpreted as the dyadic representation of an $x \in \mathrm{N}$) on the query tape and goes into its query state, whereupon the contents of the query tape are erased and the contents of the reply tape become the dyadic representation of $f(x)$. The cost model for OTMs is identical to that for TMs except for the issue of the cost of a query. There are two standard models for query costs: (i) the *length-cost model* under which the cost of a query is $\max(|f(x)|, 1)$, where $|f(x)|$ is the length of the string on the reply tape, and (ii) the *unit-cost model* under which queries cost unit time. As the length-cost model is easier to work with, we will use it as our default cost model for OTMs. We will come back to the unit-cost model in Part II. **N.B.** The great bother with Turing machines is that no sensible person wants to read or write a nontrivial Turing machine program. Consequently, the tradition (starting with Turing [Tur36]) is to play loose with the model and speak of Turing machines as having subroutines, counters, arrays, etc. All of these can be realized in the standard model in straightforward ways with polynomial overhead, and this suffices for our purposes.

Lengths

For the length of an $f\colon \mathrm{N} \to \mathrm{N}$, we have the choice of making it either (i) a number or (ii) some sort of function. Option (i) is quite limiting for our purposes. Here is the difficulty. Suppose $\ell(f)$ denotes the number giving the length of f. Then "$\mathbf{M}$, on (f,x), runs in time polynomial in the lengths of f and x" should mean that for some fixed (ordinary) polynomial q, $\mathbf{M}$ on (f,x) should run within $q(\ell(f),|x|)$ steps. But it is easy to show that no such $\mathbf{M}$ can compute the type-2 application functional $(f,x) \mapsto f(x)$. So we choose option (ii). We require the length of f, denoted $|f|$, to be a monotone, nondecreasing function from type-0 lengths to type-0 lengths.[4] We also require that the cost of the query "$f(x) =?$" (under the length-cost model) be bounded above by $|f|$ on $|x|$, that is, $|f(x)| \leq |f|(|x|)$. This will help make application polynomial-time computable. With these requirements in mind, we introduce

Definition 5 (Kapron and Cook [KC91, KC96]). The *length* of $f\colon \mathrm{N} \to \mathrm{N}$ is the function $|f|\colon \omega \to \omega$ such that $|f| = \lambda n.\, \max(\{\, |f(x)| \mid |x| \leq n \,\})$. $\diamond$

One bothersome thing about this definition is that the functional $(f,x) \mapsto |f|(|x|)$ fails to be basic feasible, the problem being that the value of $|f|(|x|)$ depends on $2^{|x|+1} - 1$ many values. It will turn out that, for our purposes, this is usually a surmountable problem.

Polynomials

It is clear from the above discussion that our notion of a polynomial over $|f|$ and $|x|$ must include terms such as $|f|(|x|)$. So, we need to extend the notion of polynomial to incorporate type-1 (function) variables. We are thus led to the following.

Definition 6 (Kapron and Cook [KC91, KC96]). A *second-order polynomial* over type-1 variables $g_0, \ldots, g_m$ and type-0 variables $y_0, \ldots, y_n$ is an expression of one of the following five forms:

1. a 2. y_i 3. $\mathbf{q}_1 + \mathbf{q}_2$ 4. $\mathbf{q}_1 \cdot \mathbf{q}_2$ 5. $g_j(\mathbf{q}_1)$

where $a \in \omega$, $i \leq n$, $j \leq m$, and $\mathbf{q}_1$ and $\mathbf{q}_2$ are second-order polynomials over $\vec{g}$ and $\vec{y}$. A second-order polynomial over m type-1 variables and n type-0 variables is said to be of *rank* (m,n). The *depth* of a second-order polynomial $\mathbf{q}$ is the maximum depth of nesting of applications of the g_j's in $\mathbf{q}$. The value of

[4] This requirement is a bit arbitrary, but fairly standard for complexity-theoretic bounds—we typically want the resources allotted to solve a problem of size $n+1$ to be at least as large as those allotted for a problem of size n.

a second-order polynomial as above on $f_0, \ldots, f_m : \omega \to \omega$ and $x_0, \ldots, x_n \in \omega$ is the obvious thing. $\diamond$

For example, $g(y+23)$ is a rank-$(1,1)$, depth-1, second-order polynomial over type-1 variable g and type-0 variable y, and $g_0((g_0(2 \cdot y \cdot g_1(y^2)) + 6)^3)$ is a rank-$(2,1)$, depth-3, second-order polynomial over type-1 variables g_0 and g_1 and type-0 variable y.

Polynomial-time bounded OTMs

Having specified our machine model and our notions of length and polynomial, we are now in a position to state a machine-based notion of type-2 polynomial-time.

Definition 7 (Kapron and Cook [KC91, KC96]). Suppose $k \geq 1$ and $\ell \geq 0$. Then, $F : (\mathrm{N} \to \mathrm{N})^k \times \mathrm{N}^\ell \to \mathrm{N}$ is a *basic polynomial-time functional* if and only if there is an OTM $\mathbf{M}$ and a second-order polynomial $\mathbf{q}$ such that, for each input $(f_1, \ldots, f_k, x_1, \ldots, x_\ell)$,

1. $\mathbf{M}$ outputs $F(f_1, \ldots, f_k, x_1, \ldots, x_\ell)$, and
2. $\mathbf{M}$ runs within $\mathbf{q}(|f_1|, \ldots, |f_k|, |x_1|, \ldots, |x_\ell|)$ time steps. $\diamond$

Example 8. Let $T = \lambda f, x . \sum_{i<|x|} f(i)$ be as in Example 3. T is computed by the OTM program sketched below.

Program sketch for $\mathbf{M}_T$.

Input (f, x).
$\mathit{sum} := 0$.
For $i = 0, \ldots, |x| - 1$ **do** $\mathit{sum} := \mathit{sum} + f(i)$.
Output sum.

End

If one fills in the details of the sketch in a straightforward way, an easy analysis shows that there is an ordinary, one-variable polynomial q such that $\mathbf{M}_T$ as sketched above runs within $q(|x|+1) \cdot (|f|(|x|)+1))$ time. $\diamond$

The central result of [KC91, KC96] is the following characterization which establishes that, under the above definitions, the type-2 BFFs are the exact type-2 analogue of PF.

Theorem 9 (Kapron and Cook [KC91, KC96]). *The basic polynomial-time functionals and* BFF_2 *are one and the same class. This equivalence is constructive in the sense that:*

(a) For each BTLP P*, one can construct an OTM* $\mathbf{M}$ *and a second-order polynomial* $\mathbf{q}$ *such that* $\mathbf{M}$ *computes the same functional as* P *and* $\mathbf{M}$ *runs within the time bound given by* $\mathbf{q}$.

(b) Given a second-order polynomial $\mathbf{q}$ *and an OTM* $\mathbf{M}$ *that runs within the time bound specified by* $\mathbf{q}$*, one can construct a* BTLP P *that computes the same functional as* $\mathbf{M}$.

Note that there is an asymmetry between (a) and (b)—the translation of (a) works for any BTLP procedure P, whereas the translation of (b) works only for $\mathbf{M}$ and $\mathbf{q}$ such that $\mathbf{q}$ happens to bound $\mathbf{M}$'s run time. (The problem of whether $\mathbf{q}$ bounds $\mathbf{M}$'s run time is clearly undecidable in general.) The root cause of this asymmetry is a looseness in Definition 7, which permits any $\mathbf{M}$ that obeys the run-time bound through any method whatsoever. The fact that the run-time bound is itself not generally basic feasibly computable adds to the confusion. We can resolve this mismatch by means of polynomially-clocked machines.

Clocking

Among its many properties, PF admits of a machine/resource-based indexing of the following form. Fix an indexing of TMs $\langle M_i \rangle_{i\in\mathrm{N}}$. For each $i, k \in \mathrm{N}$, let $M_{i,k}$ be a TM that, on input x, simulates M_i on input x for up to $k(|x|+1)^k$ steps and:

- If M_i on x halts within this time with output y, then $M_{i,k}$ outputs y.
- If M_i on x fails to halt within this time, then $M_{i,k}$ outputs 0.

With a bit of care in the construction of the $M_{i,k}$'s, one can arrange that there is a polynomial q_0 such that, for all i, k, and x, $M_{i,k}$ on input x runs within $k(|x|+1)^k + q_0(k, |x|)$ time. Clearly, $M_{i,k}$ computes an element of PF and if, for each input x, a given M_i runs within $k(|x|+1)^k$ time, then M_i and $M_{i,k}$ compute the same element of PF. Hence, the $M_{i,k}$'s determine a subclass of PF. Note that for each one-variable polynomial q, there is a k such that, for all n, $q(n) \leq k(n+1)^k$. It thus follows that the $M_{i,k}$'s determine exactly PF. For obvious reasons $\langle M_{i,k} \rangle_{i,k\in\mathrm{N}}$ is called a *clocked* indexing of PF. (For more than you ever wanted to know about clocked indexings of type-1 subrecursive classes, see Chapter 4 of Royer and Case [RC94].)

The type-2 BFFs have analogous machine/resource-based indexings. This is not an obvious result because of the previously noted failure of $(\vec{f}, \vec{x}) \mapsto \mathbf{q}(|f_1|, \ldots, |f_k|, |x_1|, \ldots, |x_\ell|)$ to be basic feasible. The existence of such an indexing was implicit (but buried) in Kapron and Cook's proof of Theorem

9(b) and was first made explicit by Seth [Set92]. We develop one such indexing below. To keep the notation down, we will restrict our attention to functionals of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$. We first introduce a collection of second-order polynomials analogous to the $k(n+1)^k$'s above.

Definition 10. For each $k \in \mathrm{N}$, let p_k be the two-variable polynomial $k(m+n+1)^k$. For each $d, k \in \mathrm{N}$, define $q^{d,k}$ to be the second-order polynomial over g and y as follows:

$$\begin{aligned} q^{0,k}(g,y) &= p_k(0,y). \\ q^{d+1,k}(g,y) &= p_k(g(q^{d,k}(g,y)),y). \end{aligned}$$

◇

A straightforward argument shows that, for each second-order polynomial $\mathbf{q}$ over f and x, if d is the depth of $\mathbf{q}$, then there is a $k \in \mathrm{N}$ such that, for all f and x, $\mathbf{q}(|f|,|x|) \leq q^{d,k}(|f|,|x|)$.

Now let us consider the problem of how a clocked OTM $\mathbf{M}$ is to deal with a bound of the form $q^{d,k}(|f|,|x|)$ that $\mathbf{M}$ cannot, in general, feasibly compute. The solution is that it suffices for $\mathbf{M}$, on input (f,x), to keep a running, lower approximation to $q^{d,k}(|f|,|x|)$ based on $\mathbf{M}$'s current knowledge of f. At any point in a computation, "$\mathbf{M}$'s knowledge of f" consists simply of $f|_Q$ = the finite function that results from restricting f to Q, the finite set of queries that $\mathbf{M}$ has made of f up to this point. To formalize this approximation process we first extend the notion of length to finite functions. *Notation:* In the following ζ will range over finite functions $\mathrm{N} \rightharpoonup \mathrm{N}$, and for each ζ, let $\widehat{\zeta}$ denote the total function extending ζ that is 0 every place ζ is undefined.

Definition 11. The length of ζ is the function $|\zeta|\colon \omega \to \omega$ such that, for each n, $|\zeta|(n) = \max(\{\, |\zeta(x)| \mid |x| \leq n \;\&\; x \in \mathrm{dom}(\zeta)\,\})$. ◇

Some observations: First note that $|\zeta| = |\widehat{\zeta}|$. Fix, for this paragraph, $f\colon \mathrm{N} \to \mathrm{N}$ and $n \in \omega$. Let $\mathcal{D} = \{\, \zeta \mid \zeta \subset f\,\}$, which is a directed set partially ordered by $\subseteq$. For ζ over $\mathcal{D}$ we have that $|\zeta|(n)$ is monotone nondecreasing in ζ, and $\lim_{\zeta \to f} |\zeta|(n) =$ (least upper bound of $\{\, |\zeta|(n) \mid \zeta \in \mathcal{D}\,\}$) $= |f|(n)$. More generally, a simple argument shows that, for any second order polynomial $\mathbf{q}$, for ζ over $\mathcal{D}$ we also have that $\mathbf{q}(|\zeta|,|x|)$ is monotone, nondecreasing in ζ and $\lim_{\zeta \to f} \mathbf{q}(|\zeta|,|x|) = \mathbf{q}(|f|,|x|)$. Finally, we note a key observation of Kapron and Cook [KC91, KC96], although the statement here is a bit different from theirs.

Lemma 12 (The Continuity of Bounds). *Suppose* $\mathbf{M}$ *is a OTM and* $\mathbf{q}$ *is a second-order polynomial. For each* $f\colon \mathrm{N} \to \mathrm{N}$, $x \in \mathrm{N}$, *and* $s \in \omega$, *let* $\zeta_{f,x,s}$ *denote the finite part of* f *that* $\mathbf{M}$ *discovers through queries when* $\mathbf{M}$, *on input* (f, x), *has run* s *steps. Then, the following are equivalent.*

(a) For each input (f, x), $\mathbf{M}$ *runs within* $\mathbf{q}(|f|, |x|)$ *time.*

(b) For each f, x, *and* s, *on input* (f, x) *the total cost of* $\mathbf{M}$*'s computation up through step* s *is no greater than* $\mathbf{q}(|\zeta_{f,x,s}|, |x|)$.

Proof. *(b)* $\Longrightarrow$ *(a).* This follows from the observations just prior to the statement of the lemma.

(a) $\Longrightarrow$ *(b).* Suppose (a). Fix an input (f, x) and, for each s, let $\zeta_s = \zeta_{f,x,s}$. Suppose by way of contradiction that at step s, the total cost of $\mathbf{M}$'s computation up through step s is greater than $\mathbf{q}(|\zeta_s|, |x|)$. Consider the computation of $\mathbf{M}$ on input $(\widehat{\zeta_s}, x)$. This computation and the computations of $\mathbf{M}$ on (f, x) must be identical up through step s. But by (a), $\mathbf{M}$, on input $(\widehat{\zeta}, x)$, runs within time $\mathbf{q}(|\widehat{\zeta_s}|, |x|) = \mathbf{q}(|\zeta_s|, |x|)$. Hence, the computations of $\mathbf{M}$ on both $(\widehat{\zeta_s}, x)$ and (f, x) halt in fewer than s steps, a contradiction. □

Lemma 12 more or less tells us how to carry out the clocking. Suppose $\langle \mathbf{M}_i \rangle_{i \in \mathrm{N}}$ is an indexing of OTMs. For each d and $k \in \mathrm{N}$, let $\mathbf{M}_{i,d,k}$ be an OTM that, on input (f, x), does something equivalent to the following. $\mathbf{M}_{i,d,k}$ carries out a step-by-step simulation of $\mathbf{M}_i$ and, in the process, keeps track of two auxiliary pieces of data: cost, an integer counter that records the total cost of the steps of $\mathbf{M}_i$ executed thus far, and ζ, a finite function (stored in some appropriate data structure) that records what is known about the type-1 argument thus far. On startup, $\mathbf{M}_{i,d,k}$ initializes $\mathsf{cost} := 0$ and $\zeta := \emptyset$. Then $\mathbf{M}_{i,d,k}$ commences a step-by-step simulation of $\mathbf{M}_i$ on input (f, x). After each simulation of an $\mathbf{M}_i$-step, $\mathbf{M}_{i,d,k}$ goes through:

If $\mathbf{M}_i$ halted on this step with output y,
 then $\mathbf{M}_{i,d,k}$ outputs y and halts,
 else:
  If this step was a query "$f(y) =?$",
   then set $\mathsf{cost} := \mathsf{cost} + \max(1, |f(y)|)$ and $\zeta := \zeta \cup \{\, (y, f(y)) \,\}$,
   else set $\mathsf{cost} := \mathsf{cost} + 1$.
  If $\mathsf{cost} \geq q^{d,k}(|\zeta|, |x|)$,
   then $\mathbf{M}_{i,d,k}$ outputs 0 and halts,
   else the simulation continues.

With a bit of care in the construction of the $\mathbf{M}_{i,d,k}$'s, one can arrange that there is a k_0 such that, for all i, d, k, and x, at any time in the computation, the run time of $\mathbf{M}_{i,d,k}$ on input (f, x) up to that point is bounded by

$q^{d,k}(|\zeta|, |x|) + q^{d+1,k_0}(|\zeta|, k + |x|)$. (Seth [Set92] and Royer [Roy97] provide for details on similar clocking schemes.) Thus by Lemma 12, $\mathbf{M}_{i,d,k}$ computes a basic polynomial-time functional. If $\mathbf{M}_i$ is an OTM that just happens to run within a $q^{d,k}$ time bound, then it follows from the construction and Lemma 12 that $\mathbf{M}_i$ and $\mathbf{M}_{i,d,k}$ compute the same functional. Since the $q^{d,k}$'s are cofinal in the second-order polynomials, we have that the $\mathbf{M}_{i,d,k}$'s determine *exactly* the BFFs of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$.

Lemma 12 plays a critical role in the proof of Theorem 9(b) that can be amended to read:

> *For each i, d, and k, one can construct a* BTLP *P that computes the same functional as* $\mathbf{M}_{i,d,k}$.

(We will see something analogous in our versions of Seth's machine characterization for types 3 and above in Part II.)

The $\mathbf{M}_{i,d,k}$'s provide a crisp, complexity-theoretic way of looking at the type-2 BFFs. Furthermore, in contrast to the program of Example 3, the program of Example 8 corresponds to a more natural way to express programs for type-2 functionals. On the other hand, the $\mathbf{M}_{i,d,k}$'s are less than satisfactory in several regards. First, the program of Example 8 is *not* an OTM program, it is just a sketch of one. An actual OTM program for T would be some rat's nest of tuples. Second, while perhaps palatable to complexity theorists, the explicit clocking in the definition of the $\mathbf{M}_{i,d,k}$'s looks quite *ad hoc*. What would be nice is a fairly simple programming formalism for BFF_2 that is closer in spirit to the $\mathbf{M}_{i,d,k}$'s than BTLP, but that would avoid the grubbiness of OTM programs and the explicit clocking. The next two sections address this problem.

7. Inflationary Tiers and Iterations

This section lays the type-theoretic and recursion-theoretic groundwork for type-2 *inflationary tiered loop programs* (abbreviated as ITLP_2), a second-order programming formalism developed in the next section. ITLP_2 was inspired by Leivant's tiered-recursion formalism [Lei95]. The roots of Leivant's work trace through Nelson's work on predicative arithmetic [Nel86], Leivant's earlier work on predicative recursion [Lei91, Lei94a], and Bellantoni and Cook's safe-recursion formalism [BC92]. Bellantoni and Cook's, Leivant's, and our programming formalisms are all partly based on reforming Cobham's scheme of limited recursion on notation. We thus start with a critique of that scheme.

Recall that f is defined by *limited recursion on notation* via g_0, g_1 and h, if and only if the following two conditions are satisfied for all x and $\vec{y}$:

$$f(x,\vec{y}) = \begin{cases} g_0(\vec{y}), & \text{if } x = 0; \\ g_1(f(\lfloor x/2 \rfloor, \vec{y}), x, \vec{y}), & \text{if } x > 0. \end{cases} \tag{2}$$

$$|f(x,\vec{y})| \leq |h(x,\vec{y})|. \tag{3}$$

The side condition, (3), seems a great necessity and a great nuisance. The necessity is clear from Theorem 2(b)—without something like the tempering influence of (3) one can take feasible initial functions and through (2) produce highly nonfeasible results. The nuisance is also clear if only from aesthetic considerations. More serious for us is that it seems quite awkward to program in formalisms such as PV^ω and BTLP that have a type-2 analogue of limited recursion on notation as their sole iteration construct—compare the programs of Examples 3 and 8. We suspect that this awkwardness is quantifiable.[5] It would be worthwhile, then, to find a less troublesome constraint that would play the same role as (3) in moderating (2). Bellantoni and Cook [BC92] found such a constraint in a simple type system, together with *safe recursion* and *safe composition*, which are, respectively, versions of (2) and composition satisfying some type constraints. Their system provides a quite natural characterization of PF. Following up on their work, Leivant [Lei95] gave an even more elegant characterization of PF and analogous characterizations of several other complexity classes. Below we sketch a function algebra TR that is a variant of Leivant's system for PF.

Predicative tiered recurrence

To motivate the core idea, let us examine a specific problematic use of (2). Consider the function b defined by the recursion

$$b(x) = \begin{cases} 3, & \text{if } x = 0; \\ b(\lfloor x/2 \rfloor) \# b(\lfloor x/2 \rfloor), & \text{if } x > 0. \end{cases}$$

Recall that $x \# y = 2^{|x|\cdot|y|}$, so $|x \# y| = |x| \cdot |y|$. Thus it follows that, for all x, $|b(x)| = 2^{2^{|x|}}$. Now, we certainly want to be able to compute $\lambda y \centerdot y \# y$ and we certainly want recursions of the form of (2), but their combination results in a "vicious spiral," which we most certainly do not want.

To break the vicious growth rates, we apply an idea of Russell's [Rus03, Rus08, Rus67] for breaking vicious circles, the ramified theory of types. Russell's *vicious-circle principle* [Rus08] is:

[5] We conjecture that something like the following is true: There is a real constant $\alpha > 0$ such that, for infinitely many d, i, and k, the shortest BTLP procedure equivalent to $\mathbf{M}_{i,d,k}$ is of length greater than $2^{\alpha \cdot d} \cdot (|i| + |k|)$.

No totality can contain members defined in terms of itself.

In Russell's setting this meant that if Y is defined in terms of a quantification over objects of a particular type, then Y must be of a higher type. In the present setting, this principle should mean something like: if y is a number that results from a recursion on x, then y should be of a type higher than x, where the meaning of "type" needs to be clarified. Towards this end we introduce ω-many copies of N, denoted N_0, N_1, $\mathrm{N}_2, \ldots$, which we will call *tiers*. These tiers are ordered $\mathrm{N}_0 \prec \mathrm{N}_1 \prec \cdots \prec \mathrm{N}_i \prec \cdots$. We usually view input values as residing in tier N_0 and values in tier N_{i+1} as the result of TR-recursions on values from $\mathrm{N}_0, \ldots, \mathrm{N}_i$. (**N.B.** Our conventions on tier-levels are the reverse Leivant's—in his system inputs are from some tier N_k and the values in tier N_i $(i < k)$ arise from recursions on values from $\mathrm{N}_{i+1}, \ldots, \mathrm{N}_k$.)

Before introducing TR, we establish a bit of notation. For each $x \in \mathrm{N} \cup \mathrm{N}_0 \cup \mathrm{N}_1 \cup \cdots$, let $(x)_\omega$ be the N version of x and $(x)_i$ be the N_i version of x. Let $\mathcal{F} = \{\, g{:}\, \mathrm{N}_{i_1} \times \cdots \times \mathrm{N}_{i_r} \to \mathrm{N}_{i_0} \mid r, i_0, \ldots, i_r \in \mathrm{N} \,\}$, the collection of all functions over the tiers. TR will determine a particular subset of $\mathcal{F}$.

Here then is the definition of TR. For each i we define the constructor functions $c^i_\epsilon{:}\, () \to \mathrm{N}_i$, $c^i_{\mathbf{0}}{:}\, \mathrm{N}_i \to \mathrm{N}_i$, and $c^i_{\mathbf{1}}{:}\, \mathrm{N}_i \to \mathrm{N}_i$ by the equations:

$$c^i_\epsilon() \;=\; \epsilon. \qquad c^i_{\mathbf{0}}(w) \;=\; \mathbf{0}w. \qquad c^i_{\mathbf{1}}(w) \;=\; \mathbf{1}w.$$

(Recall that we are identifying the elements of N and $\{\,\mathbf{0}, \mathbf{1}\,\}^*$.) The c^i_ϵ's, $c^i_{\mathbf{0}}$'s, and $c^i_{\mathbf{1}}$'s are TR's initial functions. We usually write ϵ_i for $c^i_\epsilon()$. TR permits the following three ways of defining new functions from old. Let $\mathcal{N}$ (and decorated versions of it) range over finite products of the N_i's.

Explicit Definition. We say $f{:}\, \mathcal{N} \to \mathrm{N}_n$ is defined through explicit definition from a collection of previously introduced functions $\mathcal{S} \subseteq \mathcal{F}$ if and only if the definition consists of a correctly typed equation of the form $f(\vec{x}) = t$, where t is a term over the vocabulary made up of $\vec{x}$ and names for the elements of $\mathcal{S}$.

Definition by Cases. Suppose $m \leq n$. We say that $f{:}\, \mathrm{N}_m \times \mathcal{N} \to \mathrm{N}_n$ is defined by cases from $g_\epsilon{:}\, \mathcal{N} \to \mathrm{N}_n$ and $g_{\mathbf{0}}, g_{\mathbf{1}}{:}\, \mathrm{N}_m \times \mathcal{N} \to \mathrm{N}_n$ if and only if the following equations hold:

$$\begin{aligned} f(\epsilon_m, \vec{x}) &= g_\epsilon(\vec{x}). \\ f(\mathbf{d}w, \vec{x}) &= g_{\mathbf{d}}(w, \vec{x}), \text{ for } \mathbf{d} = \mathbf{0}, \mathbf{1}. \end{aligned}$$

Ramified Recurrence. Suppose $m < n$. We say that $f{:}\, \mathrm{N}_m \times \mathcal{N} \to \mathrm{N}_n$ is defined by ramified recurrence from $g_\epsilon{:}\, \mathcal{N} \to \mathrm{N}_n$ and $g_{\mathbf{0}}, g_{\mathbf{1}}{:}\, \mathrm{N}_n \times \mathrm{N}_m \times$

$\mathcal{N} \to \mathrm{N}_n$ if and only if the following equations hold:

$$\begin{aligned} f(\epsilon_m, \vec{x}) &= g_\epsilon(\vec{x}). \\ f(\mathbf{d}w, \vec{x}) &= g_{\mathbf{d}}(f(w, \vec{x}), w, \vec{x}), \text{ for } \mathbf{d} = \mathbf{0},\ \mathbf{1}. \end{aligned}$$

This completes the definition of TR.

Example 13 (Some sample TR definitions).

(a) The destructor function $destr_i\colon \mathrm{N}_i \to \mathrm{N}_i$ given by the definition by cases:

$$\begin{aligned} destr_i(\epsilon_i) &= \epsilon_i. \\ destr_i(\mathbf{d}w) &= \mathbf{id}_{\mathrm{N}_i}(w), \text{ for } \mathbf{d} = \mathbf{0}, \mathbf{1}. \end{aligned}$$

(Note: $\mathbf{id}_{\mathrm{N}_i}$ can be given by explicit definition.)

(b) Suppose $i < j$. The upward coercion function $up_{ij}\colon \mathrm{N}_i \to \mathrm{N}_j$ is given by the recurrence:

$$\begin{aligned} up_{ij}(\epsilon_i) &= \epsilon_j. \\ up_{ij}(\mathbf{d}w) &= c^j_{\mathbf{d}}(up_{ij}(w)), \text{ for } \mathbf{d} = \mathbf{0}, \mathbf{1}. \end{aligned}$$

(c) The concatenation function $\odot\colon \mathrm{N}_0 \times \mathrm{N}_0 \to \mathrm{N}_1$ is given by:

$$\begin{aligned} \odot(\epsilon_0, x) &= up_{01}(x). \\ \odot(\mathbf{d}w, x) &= c^1_{\mathbf{d}}(\odot(w, x)), \text{ for } \mathbf{d} = \mathbf{0}, \mathbf{1}. \end{aligned}$$

(d) Suppose $s\colon \mathrm{N}_1 \to \mathrm{N}_1$ is given. For each $k \geq 0$, let $iter^k_s\colon (\mathrm{N}_0)^k \times \mathrm{N}_1 \to \mathrm{N}_1$ be the function such that, for all $x_1, \ldots, x_k \in \mathrm{N}_0$ and $y \in \mathrm{N}_1$, $iter^k_s(x_1, \ldots, x_k, y) = s^{(n)}(y)$, where $n = \Pi^k_{i=1}(|x_i| + 1)$. (So, in particular, $iter^k_s(x, \ldots, x, y) = s^{(\,(|x|+1)^k\,)}(y)$.) We can define $iter^0_s$ by: $iter^0_s(y) = s(y)$. Given $iter^k_s$, we can define $iter^{k+1}_s$ by:

$$\begin{aligned} iter^{k+1}_s(\epsilon_0, x_2, \ldots, x_{k+1}, y) &= iter^k_s(x_2, \ldots, x_{k+1}, y). \\ iter^{k+1}_s(\mathbf{d}x_1, x_2, \ldots, x_{k+1}, y) &= iter^k_s(x_2, \ldots, x_{k+1}, y'), \\ \text{where } y' &= iter^{k+1}_s(x_1, x_2, \ldots, x_{k+1}, y) \\ \text{and } \mathbf{d} &= \mathbf{0},\ \mathbf{1}. \end{aligned}$$

◇

To evaluate f on $\vec{x}$, where $f\colon \mathcal{N} \to \mathrm{N}_i$ is given by a set of TR definitions and $\vec{x} \in \mathcal{N}$, we use a standard, call-by-value evaluation scheme. The semantics of TR are thus straightforward. Let TRF $\subset \mathcal{F}$ be the class of TR-definable functions and let $(\mathrm{TRF})_\omega \subset \{\, g\colon \mathrm{N}^k \to \mathrm{N} \mid k \in \omega \,\}$ denote the collection of functions obtained from TRF by identifying all the N_i's in the domains and ranges with N.

We also want to consider the variant of TR that results from replacing ramified recurrence with

Simultaneous Ramified Recurrence. Suppose $m < n$ and $r \in \mathrm{N}$. We say that $f^0, \dots, f^r\colon \mathrm{N}_m \times \mathcal{N} \to \mathrm{N}_n$ are defined by simultaneous ramified recurrence from $g^0_\epsilon, \dots, g^r_\epsilon\colon \mathcal{N} \to \mathrm{N}_n$ and $g^0_{\mathbf{0}}, \dots, g^r_{\mathbf{0}}, g^0_{\mathbf{1}}, \dots, g^r_{\mathbf{1}}\colon (\mathrm{N}_n)^r \times \mathrm{N}_m \times \mathcal{N} \to \mathrm{N}_n$ if and only if the following equations hold, for $j = 0, \dots, r$ and $\mathbf{d} = \mathbf{0}$, $\mathbf{1}$:

$$\begin{aligned} f^j(\epsilon_m, \vec{x}) &= g^j_\epsilon(\vec{x}). \\ f^j(\mathbf{d}w, \vec{x}) &= g^j_{\mathbf{d}}(f^0(w, \vec{x}), \dots, f^r(w, \vec{x}), w, \vec{x}). \end{aligned}$$

We call the resulting function algebra TR^* and define TRF^* and $(\mathrm{TRF}^*)_\omega$ analogously to TRF and $(\mathrm{TRF})_\omega$.

The next proposition compares $(\mathrm{TRF})_\omega$, $(\mathrm{TRF}^*)_\omega$, and PF.

Proposition 14 (After Leivant [Lei95]).

(a) $\mathrm{PF} \subseteq (\mathrm{TRF}^*)_\omega$. *In fact, each* $f \in \mathrm{PF}$ *is given by a* TR^* *definition that uses only tiers* N_0 *and* N_1.

(b) $\mathrm{TRF}^* = \mathrm{TRF}$. *In fact, each* TR^* *definition that uses only tiers* N_0 *and* N_1 *can be uniformly effectively translated to an equivalent* TR *definition that also uses only tiers* N_0 *and* N_1.

(c) $(\mathrm{TRF})_\omega \subseteq \mathrm{PF}$.

Proof Sketch. Proof of the analogous results for Leivant's system can be found in [Lei95]. Here we simply state the core ideas for each part.

Part (a) follows from an elaboration on the definition of $iter^k_s$. Suppose $f\colon \mathrm{N} \to \mathrm{N}$ is an element of PF. (The $\mathrm{N}^k \to \mathrm{N}$ case is analogous.) Then there is a 1-tape deterministic TM M with tape alphabet $\{\, \mathbf{0}, \mathbf{1}, B, \# \,\}$ (where B stands for blank and $\#$ is an end of tape marker) that, for each x, computes $f(x)$ within time $k(|x|+1)^k$ for some fixed $k > 0$. We can simulate M in TR as follows. Suppose the states of M are numbered 0 through j. An *instantaneous description* (abbreviated as ID) of M consists of the current state number plus the current state of the tape, $a_{-m} \dots a_{-2} a_{-1} a_0 a_1 a_2 \dots a_n$, where $a_{-m} = a_n = \#$, $a_i \in \{\, \mathbf{0}, \mathbf{1}, B \,\}$ for $-m < i < n$, and a_0 is the symbol currently scanned by the tape-head. By coding the tape symbols $\mathbf{0}$, $\mathbf{1}$, B, and $\#$ by the strings $b_{\mathbf{0}} = \mathbf{00}$, $b_{\mathbf{1}} = \mathbf{01}$, $b_B = \mathbf{10}$, and $b_\# = \mathbf{11}$, respectively, we can represent such an ID by $(s, \ell, r) \in (\mathrm{N}_1)^3$, where s is the current state number, $\ell =$

$b_{a_{-1}} \dots b_{a_{-m}}$, and $r = b_{a_0} \dots b_{a_n}$. It is straightforward to give TR^* definitions for $next_{state}$, $next_{left}$, $next_{right}$: $(\mathrm{N}_1)^3 \to \mathrm{N}_1$ such that, if (s, ℓ, r) represents an ID, then $(next_{state}(s,\ell,r), next_{left}(s,\ell,r), next_{right}(s,\ell,r))$ is the succeeding ID. The core of the simulation is a simple variation on the definition of $iter_s^k$ to define in TR^* a recursion that simultaneously iterates $next_{state}$, $next_{left}$, and $next_{right}$ an appropriate number of times, starting from the representation of M's initial ID on input x.

Part (b) follows from a careful use of pairing functions.

Part (c) can be argued as follows. For this argument suppose that $f: \mathcal{N} \to \mathrm{N}_i$ is given by $\mathcal{D}_f$, some set of TR definitions of f and all of f's auxiliary functions. Our goal is to show that f is polynomial-time computable. *Conventions:* For each j and each $x \in \mathrm{N}_j$, let $|x| = |(x)_\omega|$. For each j, $\mathcal{N}_0$, and $\vec{x} \in \mathcal{N}_0$, let $|\vec{x}|_{<j}$ denote the maximum length of the elements of $\vec{x}$ of tier less than j and $|\vec{x}|_{=j}$ denote the maximum length of the elements of $\vec{x}$ of tier j. (Recall that $\max(\emptyset) = 0$.)

Claim 1. *For each $g: \mathcal{N}_0 \to \mathrm{N}_j$ defined in $\mathcal{D}_f$, there is a polynomial p_g such that, for each $\vec{x} \in \mathcal{N}_0$, $|g(\vec{x})| \le p_g(|\vec{x}|_{<j}) + |\vec{x}|_{=j}$.*

The proof is a structural induction on the functions defined in $\mathcal{D}_f$. The key case is where $g: \mathrm{N}_k \times \mathcal{N}_1 \to \mathrm{N}_j$ (with $k < j$) is defined by ramified recurrence on h_ϵ, $h_{\mathbf{0}}$, and $h_{\mathbf{1}}$, where p_{h_ϵ}, $p_{h_{\mathbf{0}}}$, and $p_{h_{\mathbf{1}}}$ are polynomials such that, for all $v \in \mathrm{N}_j$, $w \in \mathrm{N}_k$, and $\vec{x} \in \mathcal{N}_1$, we have:

$$\begin{aligned} |h_\epsilon(\vec{x})| &\le p_{h_\epsilon}(|\vec{x}|_{<j}) + |\vec{x}|_{=j}. \\ |h_{\mathbf{d}}(v, w, \vec{x})| &\le p_{h_{\mathbf{d}}}(|w, \vec{x}|_{<j}) + |v, \vec{x}|_{=j}, \text{ for } \mathbf{d} = \mathbf{0}, \mathbf{1}. \end{aligned}$$

(Note that $|w, \vec{x}|_{<j} = \max(|w|, |\vec{x}|_{<j})$ and $|v, \vec{x}|_{=j} = \max(|v|, |\vec{x}|_{=j})$.) Let $\lambda n \centerdot c \cdot (n+1)^m$ (with c, $m \ge 1$) be a polynomial that everywhere dominates p_{h_ϵ}, $p_{h_{\mathbf{0}}}$, and $p_{h_{\mathbf{1}}}$. A simple induction shows that, for all $w \in \mathrm{N}_k$ and $\vec{x} \in \mathcal{N}_1$,

$$|g(w, \vec{x})| \le (|w| + 1) \cdot c \cdot (|w, \vec{x}|_{<j} + 1)^m + |\vec{x}|_{=j}.$$

So, letting $p_g = \lambda n \centerdot c \cdot (n+1)^{m+1}$ suffices.

Now consider a straightforward, RAM-based interpreter for TR definitions. For each $g: \mathcal{N}_0 \to \mathrm{N}_j$ defined in $\mathcal{D}_f$ and each $\vec{x} \in \mathcal{N}_0$, let $t_g(\vec{x})$ = the number steps the interpreter requires to compute $g(\vec{x})$ (via $\mathcal{D}_f$). Also, for each i, define $t_{c^i_\epsilon}() = 1$ and $t_{c^i_{\mathbf{0}}} = t_{c^i_{\mathbf{1}}} = \lambda x{:}\,\mathrm{N}_i \centerdot (|x|+1)$. By a bit of analysis and an argument similar to that for Claim 1, we obtain

Claim 2. *For each $g: \mathcal{N}_0 \to \mathrm{N}_j$ defined in $\mathcal{D}_f$, there is a polynomial q_g such that for each $\vec{x} \in \mathcal{N}_0$, $t_g(\vec{x}) \le q_g(|\vec{x}|_{<j}) + |\vec{x}|_{=j}$.*

It thus follows that f is polynomial-time computable. □

As an immediate corollary to the above, we obtain a characterization of PF which is roughly analogous to that of Bellantoni and Cook.

Corollary 15 (After Leivant [Lei95]). *Suppose* $f\colon \mathrm{N} \to \mathrm{N}$. *Then,* $f \in \mathrm{PF}$ *if and only if there is a* TR *definition using only tiers* N_0 *and* N_1 *that determines* f *as a function from* N_0 *to* N_1.

Thus in TR, tiers 2 and above really are not needed to characterize PF. We put these higher tiers to use just below.

Extending TR to type-2: A first attempt

Leivant [Lei94b] shows one way of extending his systems to higher types, but the result involves complexity classes much larger than of interest here. We offer another approach inspired by the structure of second-order polynomials.

Recall that the *depth* of a second-order polynomial $\mathbf{q}(g, x)$ is the maximum depth of nesting of applications of the g in $\mathbf{q}$. Our initial idea for a second-order version of TR is to extend the use of tiers so that, in a particular definition, the level of a tier is somehow connected with the depth of a second-order polynomial bound on the length of values that can appear in that tier. Here is a sketch of one way of doing this. Let TR_2 be the function algebra TR augmented with a function symbol $\mathbf{f}$ of polymorphic type $(\forall i)[\mathrm{N}_i \to \mathrm{N}_{i+1}]$. Each TR_2 definition of a function $g\colon \mathcal{N} \to \mathrm{N}_k$ determines a functional $F_g\colon (\mathrm{N} \to \mathrm{N}) \times \mathcal{N} \to \mathrm{N}_k$ such that, on input $(f, \vec{y})$, $F_g(f, \vec{y})$ is the value produced by g on input $\vec{y}$ when $\mathbf{f}$ is the function that, for each i, maps $(x)_i$ to $(\,f(x)\,)_{i+1}$. Let us consider an example. Define $t\colon \mathrm{N}_0 \to \mathrm{N}_3$ by:

$$t(x) \quad = \quad \begin{cases} (0)_3, & \text{if (i) } x = \epsilon; \\ \mathbf{f}(\ell en(x')) + t(x'), & \text{if (ii) } x = \mathbf{d}x' \text{ for } \mathbf{d} = \mathbf{0},\, \mathbf{1}; \end{cases}$$

where $+\colon \mathrm{N}_2 \times \mathrm{N}_3 \to \mathrm{N}_3$ is addition for the indicated tiers, $\ell en\colon \mathrm{N}_0 \to \mathrm{N}_1$ is the length function for elements of N_0, and $\mathbf{f}$ is used as a function from N_1 to N_2. Then $F_t(f, (x)_0) = (\,T(f, x)\,)_3$, where T is the functional from Examples 3 and 8. We can adapt the argument for Proposition 14(c) to show that, for each TR_2-defined $g\colon \mathrm{N}_0 \to \mathrm{N}_k$, there is a depth k second-order polynomial $\mathbf{q}$ such that, for all $f\colon \mathrm{N} \to \mathrm{N}$ and $x \in \mathrm{N}_0$, $|F_g(f, x)| \le \mathbf{q}(|f|, |x|)$. The key point is that since there is no downward coercion in tiers, if a value v is somehow obtained through j-many nested applications of the oracle, then v must be in tier j or higher. Thus all the TR_2-definable functionals have a second-order polynomial growth rate as desired. There is trouble, however. Consider the functional given by the equation:

$$G(f,x,y,z) \quad = \quad \begin{cases} y \bmod z, & \text{if } x = \epsilon; \\ f(y') \bmod z, & \text{if } x = \mathbf{d}x' \text{ for } \mathbf{d} = \mathbf{0},\, \mathbf{1} \text{ and} \\ & \quad \text{where } y' = G(f,x',y,z). \end{cases} \tag{4}$$

This is a perfectly respectable basic feasible functional, but it is not TR_2-definable. The problem is that successive applications of f push the result higher and higher in the tiers, so no definition of G can be finitely tiered.

Inflationary tiers

The problem represented by (4) is a difficulty with pure ramified type systems: intuitively mundane things can end up at enormous type levels or at no type level at all. Because of this difficulty, ramified type systems sometimes include downward coercion rules to bring mundane things down to mundane type levels. Russell's *axiom of reducibility* from [Rus08] is precisely a rule of this sort. For our setting we will choose "mundane" to mean "of small length." So following Bertie's lead, we introduce the following length-based, downward coercion functions. For each i and j with $i < j$, $\mathit{down}_{i,j}\colon \mathrm{N}_i \times \mathrm{N}_j \to \mathrm{N}_i$ is defined through the equation:

$$\mathit{down}_{i,j}(x,y) \quad = \quad \begin{cases} (y)_i, & \text{if } |y| \le |x|; \\ \epsilon_i, & \text{otherwise.} \end{cases}$$

Let TR'_2 be the function algebra obtained from TR_2 by adding the $\mathit{down}_{i,j}$'s to the set of initial functions. TR'_2 definitions determine functionals in the same way TR_2 definitions do. Note that, as with the TR_2-defined functionals, each TR'_2-defined functional has a second-order polynomial upper bound on its growth rate. This is because any downward coercion of a value into a tier does not produce a value of length any greater than already found at that tier. The TR'_2-definable functionals (when all the tiers are identified with N) turn out to be precisely the BFFs with types of the form $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N}^k \to \mathrm{N}$. Before considering this characterization we need to discuss some points related to the changed meaning of the tiers in TR'_2.

1. TR'_2 *is impredicative.* Unlike the previous systems of this section, TR'_2 fails to satisfy our interpretation of Russell's vicious-circle principle. For example, we can take the leftmost "bit" of any value in any tier and coerce it down to N_0. Thus in TR'_2 one can no longer think of N_{i+1} as the values arising from recursions and oracle-queries applied to N_i values.

2. *Tiers classify relative lengths.* Since the downward coercions produce values of length no greater than values already in the lower tier, it follows that, like TR_2, the TR'_2-definable functionals have growth rates that are bounded by second-order polynomials. In fact, the argument shows that, given a TR'_2 definition of a functional $F\colon (\mathrm{N} \to \mathrm{N}) \times \mathrm{N}^k \to \mathrm{N}$ and given an i, there is a depth i second-order polynomial $\mathbf{q}_i$ such that, for each input $(f, \vec{x})$, the values that appear in tier i in the "computation" of F are of length $\leq \mathbf{q}_i(|f|, |x_1|, \ldots, |x_k|)$. Moreover, if x is a variable in this TR'_2 definition that is typed at N_i, then this means that we can prove that any value bound to x in the course of the computation must satisfy the $\mathbf{q}_i$ length bound.

3. *Tiers are inflationary.* The phenomenon noted in Lemma 12 occurs in this context as well. The bounds represented by the tiers are based on "current" knowledge of the oracle. As the computation progresses, the information on the oracle increases, the bounds can thus increase, and the tiers can be thought of as inflating with the evolution of the computation. Thus, a value v that cannot be classified at tier i at an early stage of the computation may be so classifiable later after tier i has inflated to the point where it includes values of length $|v|$ or greater.

TR'_2 still is not satisfactory. As mentioned, TR'_2-definable functionals turn out to be precisely the BFFs with types of the form $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N}^k \to \mathrm{N}$, but the proof of this seems to require a fairly involved argument along the lines of the proof of the Kapron-Cook Theorem (Theorem 9). Thus we do not have the evidence to claim that TR'_2 is "close in spirit to the second-order polynomially clocked OTMs." The root of our problem here is that we have not been thorough enough in our reform of Cobham's limited recursion on notation scheme. Our attention has been on the side condition (3), which has been replaced by tiers that control, in an inflationary fashion, growth rates throughout the computation. The present trouble lies with (2). The bound on the number of recursions is set on the initial call and remains static throughout the computation for that call. So, as with the tiers, we will let this bound on the number recursions be inflationary. To formalize this, in the next section we shall introduce a programming formalism based on BTLP.

8. Type-2 Inflationary Tiered Loop Programs

ITLP_2 (for "Type-2 Inflationary Tiered Loop Programs") is a modification of the BTLP_2 formalism. The base types consist of N_0, $\mathrm{N}_1, \ldots$. (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 $\mathrm{N}_0, \mathrm{N}_1, \ldots$ as *integer variables.*) For each $k \geq 1$, we include the polymorphic type $(\forall i)[\mathrm{N}_i^k \to \mathrm{N}_{i+1}]$ which we abbreviate to $\mathrm{N}^k \to_+ \mathrm{N}$. These polymorphic types are considered to be at type-level one. The remaining allowable types in ITLP_2 are those of the form $\tau_0 \times \cdots \times \tau_n \to \mathrm{N}_i$ built up from the N_i's and $\mathrm{N}^k \to_+ \mathrm{N}$'s subject to the restrictions:

1. $\tau_0 \times \cdots \times \tau_n \to \mathrm{N}_i$ is of type-level one or two, and
2. i is greater than the tier number of any base type among $\tau_0, \ldots, \tau_n$.

Note that if there are no base types among $\tau_0, \ldots, \tau_n$, then i may be zero.

To cut down on notation we make upward tier-coercions automatic in ITLP_2. We say that N_i *is coercible to* N_j (written: $\mathrm{N}_i \preceq \mathrm{N}_j$) if and only if $i \leq j$.

All variables in ITLP_2 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 add ": *the name of the type*" after the variable in declarations. The grammar of ITLP_2 is given below. The intuitions behind the syntactic categories are the same as those of BTLP_2 with the one addition: $C \equiv$ case statements. (Recall that $\underline{n}$ denotes the tally string $\mathbf{0}^n$.)

$$
\begin{array}{rcl}
P & ::= & \textsf{\textbf{Procedure}}\ v_0\,(v_1, \ldots, v_\ell){:}\,\mathrm{N}_i\ P^*\ V\ I^*\ \textsf{\textbf{Return}}\ v_r\ \ \textsf{\textbf{End}} \\
V & ::= & \textsf{\textbf{var}}\ v_1^{\mathrm{N}_{i_1}}, \ldots, v_m^{\mathrm{N}_{i_m}}\ ; \\
I & ::= & C\ ;\ |\ L\ ;\ |\ v^{\mathrm{N}_i} := E\ ; \\
C & ::= & \textsf{\textbf{Case}}\ v_0^{\mathrm{N}_i}\ \textsf{\textbf{of}}\ \ \epsilon:\ I^*\ \ \textsf{\textbf{or}}\ \ \mathbf{0}v_1^{\mathrm{N}_i}:\ I^*\ \ \textsf{\textbf{or}}\ \ \mathbf{1}v_1^{\mathrm{N}_i}:\ I^*\ \ \textsf{\textbf{Endcase}} \\
L & ::= & \textsf{\textbf{For}}\ v_0^{\mathrm{N}_i} := \underline{1}\ \textsf{\textbf{to}}\ v_1^{\mathrm{N}_i}\ \textsf{\textbf{do}}\ I^*\ \textsf{\textbf{Endfor}} \\
E & ::= & \epsilon\ |\ v^{\mathrm{N}_i}\ |\ \mathsf{c}_{\mathbf{0}}(v^{\mathrm{N}_i})\ |\ \mathsf{c}_{\mathbf{1}}(v^{\mathrm{N}_i})\ |\ \mathsf{down}(v_0^{\mathrm{N}_i}, v_1^{\mathrm{N}_j})\ |\ v_0(v_1, \ldots, v_n)
\end{array}
$$

Convention: In writing ITLP_2 programs, we shall set key words in bold sans serif font and variables in normal sans serif font. As before, variables in italic font are meta-variables.

We require all ITLP_2 procedures to be well-typed with the types allowable in ITLP_2. In the procedure declaration production the declared variable v_0 has type $\tau_1 \times \cdots \times \tau_\ell \to \mathrm{N}_i$, where for $j = 1, \ldots, \ell$, v_j has type τ_j, and the type of v_r must be coercible to N_i. In an assignment of the form: '$v^{\mathrm{N}_i} := E$', the type of the expression must be coercible to N_i. Expressions are typed according to the following rules.

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

$$\frac{v\colon \tau_0 \times \cdots \times \tau_n \to \mathrm{N}_j \quad v_0\colon \sigma_0 \quad \dots \quad v_n\colon \sigma_n}{v(v_0, \dots, v_n)\colon \mathrm{N}_j} \qquad \begin{pmatrix}\text{where for each } i \le n,\\ \sigma_i = \tau_i \text{ or } \sigma_i \preceq \tau_i\end{pmatrix}$$

$$\frac{v\colon \mathrm{N}^k \to_{+} \mathrm{N} \quad v_0\colon \mathrm{N}_{i_0} \quad \dots \quad v_n\colon \mathrm{N}_{i_n}}{v(v_0, \dots, v_n)\colon \mathrm{N}_j} \qquad \big(\text{where } j = 1 + \max(i_0, \dots, i_n)\big)$$

All variables must be declared either in procedure declarations, parameter lists, or local variable declarations. Variable scoping is static and follows standard conventions, except that we forbid recursive procedure calls and references to nonlocal integer variables.

As with BTLP_2, the semantics of ITLP_2 are quite conventional and we shall discuss only its key points. Parameter passing is call-by-value. Variables local to a block are initialized to ϵ every time the block is entered. In expressions, "ϵ" denotes ϵ_0 and "$\mathsf{c_0}$," "$\mathsf{c_1}$," and "down" each denotes the obvious thing. The interpretation of **Case**-statements is standard. A **For**-loop of the form

$$\textbf{For } v_0^{\mathrm{N}_j} := \underline{1} \textbf{ to } v_1^{\mathrm{N}_j} \textbf{ do } \vec{I} \textbf{ Endfor} \tag{5}$$

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 $i \le j$ occurring within $\vec{I}$ must be such that either
   (a) each integer variable in E is of tier strictly less than 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 (5) is equivalent to:

$v_0^{\mathrm{N}_j} := \mathsf{c_0}(\epsilon_j);$
While $|v_0^{\mathrm{N}_j}| \le |v_1^{\mathrm{N}_j}|$ **do**
  $\vec{I}$
  $v_0^{\mathrm{N}_j} := \mathsf{c_0}(v_0^{\mathrm{N}_j});$
Endwhile

where **While** ... **do** ... **Endwhile** has the usual meaning. The not-so-obvious fact that such **For**-loops always terminate will fall out from the proof of Proposition 19 below.

Since the types are simple and there are no recursive calls in procedures, one can give a straightforward, inductive semantics to the language. Also, since the only global references are to variables with immutable bindings and since parameter passing is call-by-value, it follows that the semantics of $\mathsf{x}(\mathsf{x}_0, \dots, \mathsf{x}_n)$ is simply the application of the function named by x to the values named by $\mathsf{x}_0, \dots, \mathsf{x}_n$.

As with BTLP, one can apply variations on standard tricks to achieve the effect of a richer set of control structures, expressions, and so on, in ITLP_2 procedures. We shall occasionally make use of these extensions.

Definition 16.

(a) For each $f\colon \mathrm{N}^k \to \mathrm{N}$, let $(f)_0$ be the function that, for each $x_1, \dots, x_k \in \mathrm{N}$ and each $i_1, \dots, i_k \in \omega$, satisfies

$$(f)_0(\,(x_1)_{i_1}, \dots, (x_k)_{i_k}\,) \quad = \quad (\,f(x_1, \dots, x_k)\,)_{\max(i_1,\dots,i_k)+1}.$$

Also, let $(\mathrm{N})_0 = \mathrm{N}_0$ and $(\mathrm{N}^k \to \mathrm{N})_0 = \mathrm{N}^k \to_+ \mathrm{N}$.

(b) Suppose $\tau_1 \times \cdots \times \tau_n \to \mathrm{N}$ is a type over N at type-level two. We say that $F\colon \tau_1 \times \cdots \times \tau_n \to \mathrm{N}$ is ITLP_2-*computable* if and only if there is an ITLP_2 procedure P of type $(\tau_1)_0 \times \cdots \times (\tau_n)_0 \to \mathrm{N}_j$ for some j such that, for each $z_1\colon \tau_1, \dots, z_n\colon \tau_n$, we have that P on input $((z_1)_0, \dots, (z_n)_0)$ returns $(\,F(z_1, \dots, z_n)\,)_j$. ◇

For example, the procedure Gdef below witnesses that G defined in (4) is ITLP_2-computable. In the following $\mathsf{mod}\colon \mathrm{N}_1 \times \mathrm{N}_0 \to \mathrm{N}_2$ is the modulus function over the indicated tiers.

Procedure $\mathsf{Gdef}(\mathsf{f}\colon \mathrm{N} \to_+ \mathrm{N},\ \mathsf{x}\colon \mathrm{N}_0,\ \mathsf{y}\colon \mathrm{N}_0,\ \mathsf{z}\colon \mathrm{N}_0)\colon \mathrm{N}_1$

  var $\mathsf{w}\colon \mathrm{N}_0$, $\mathsf{temp}\colon \mathrm{N}_2$;

  For $\mathsf{w} := \underline{1}$ **to** x **do**

    $\mathsf{temp} := \mathsf{f}(\mathsf{y}) \bmod \mathsf{z}$;

    $\mathsf{y} := \mathsf{down}(\mathsf{z}, \mathsf{temp})$;

  Endfor;

  Return y

End

For a crucial example of the use of the inflationary aspect of **For**-loops, see the procedure Sim in the proof of Proposition 18 below.

The following is our promised characterization of BFF_2. Its proof takes up the remainder of this section.

Proposition 17. *The class of* ITLP_2*-computable functionals is exactly* BFF_2*.*

To keep the notation manageable, we shall argue this proposition only for functionals of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$. We first show

Proposition 18. *Suppose that* $F\colon(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ *is computed by an OTM* $\mathbf{M}$ *and that* $\mathbf{q}$ *is a second-order polynomial that bounds* $\mathbf{M}$*'s run time. Then,* F *is* ITLP_2*-computable.*

Proof. Recall from Definition 10 that the second-order polynomials $q^{d,k}$ ($d, k \in \omega$) are given by the equations:

$$
\begin{aligned}
q^{0,k}(g, y) &= p_k(0, y) \\
q^{d+1,k}(g, y) &= p_k(g(q^{d,k}(g, y)), y)
\end{aligned}
$$

where $p_k = \lambda m, n.(m + n + 1)^k$. Without loss of generality, we take $\mathbf{q}$ to be $q^{d,k}$ for some $d \geq 0$ and $k > 0$. Also without loss of generality we assume that each $\mathbf{M}$ instruction (other than the oracle query instruction) involves a single tape and makes no changes to the contents or head positions on the other tapes. We sketch a ITLP_2 procedure, Sim, that simulates $\mathbf{M}$. We make use of the following observation in the construction.

Claim 1. *For each* $f\colon\mathrm{N} \to \mathrm{N}$*,* $x \in \mathrm{N}$*, and* $s \in \omega$*, let* $\zeta_{f,x,s}$ *denote the finite part of* f *that* $\mathbf{M}$ *discovers through queries when* $\mathbf{M}$*, on input* (f, x)*, has run* s *steps. Then, for each* f*,* x*, and* s*, on input* (f, x)*, all of* $\mathbf{M}$*'s queries up through (and including) step* s *are of length no greater than* $q^{d-1,k}(|\zeta_{f,x,s}|, |x|)$*.*

Proof: Fix f and x and, for each s, let $\zeta_s = \zeta_{f,x,s}$. The value of $q^{d,k}(|\zeta_s|, |x|)$ depends exclusively on the values of ζ_s (and hence f) of length no greater than $q^{d-1,k}(|\zeta_s|, |x|)$. Hence, if at some step s, $\mathbf{M}$ had the temerity to query its oracle on some value of length greater than $q^{d-1,k}(|\zeta_s|, |x|)$, then there are f's consistent with ζ_s such that the length of answer received to the query (and hence the cost of $\mathbf{M}$'s computation) would exceed $q^{d,k}(|f|, |x|)$, a contradiction. Hence, Claim 1 follows.

Sim is of type $(\mathrm{N} \to_+ \mathrm{N}) \times \mathrm{N}_0 \to \mathrm{N}_{2d+2}$ and uses integer variables of tiers 0 through $2d + 2$. The variables concerned with simulating tape contents are of tier N_{2d+2}. Since we have seen the simulation of a non-oracular TM in a similar setting (the proof of Proposition 14(a)), here we will suppress all the details of simulating the steps of $\mathbf{M}$ *except* for the case of oracle queries. Among the variables concerned with the clocking there are a few that merit special mention. (*Note:* For each i, ω_i denotes the subtype of N_i consisting of the elements of N_i whose dyadic representation is in $\mathbf{0}^*$.)

- The variable $\mathsf{Queries}\colon \mathrm{N}_{2d+1}$ will store the list of oracle queries that have been made up to the current point in the computation.[6]

- The variables $\mathsf{m}_i\colon \omega_{2i}$ and $\mathsf{n}_i\colon \omega_{2i+1}$, where $i \le d$, are used in computing the clock bound. In the main loop of the simulation we shall maintain the following invariants on the m_i's and n_i's. For $i \le d$, $\mathsf{n}_i = p_k(\mathsf{m}_i, |x|)$ (where the arithmetic is unary), $\mathsf{m}_0 = \epsilon_0$, and, for $i > 0$,

$$\mathsf{m}_i \quad = \quad \max(\{\, |f(w)| \mid |w| \le |\mathsf{n}_{i-1}| \text{ and } w \text{ is listed in } \mathsf{Queries}\,\}).$$

  It follows from the definition of the $q^{d,k}$'s that, for each $i \le d$, after each step in the simulation, the value of n_i will be $q^{i,k}(|\zeta|, |x|)$, where ζ is the finite part of the oracle currently known. Thus, the value of n_d is the (inflationary) run-time bound. Note that if the simulation of $\mathbf{M}$ halts in a given step, we violate the invariant on n_d by assigning it ϵ_{2d+1}; this provides us a quick exit from the **For**-loop.

Sim also uses the nested procedures $\mathsf{P}_0, \ldots, \mathsf{P}_d$. For each i, P_i is of type $\mathrm{N}_{2i} \times \mathrm{N}_0 \to \mathrm{N}_{2i+1}$ and computes the function $\lambda y\colon \mathrm{N}_{2i}, x\colon \mathrm{N}_0 \,.\, (\, \underline{p_k(|y|, |x|)}\,)_{2i+1}$. By adapting the technique from Example 13(d), we can give ITLP_2 procedure definitions for the P_i's. Here then is a high-level sketch of Sim.

Procedure $\mathsf{Sim}(\mathsf{f}\colon \mathrm{N} \to_+ \mathrm{N},\ \mathsf{x}\colon \mathrm{N}_0)\colon \mathrm{N}_{2d+1}$

(∗ *Notation: f and x respectively denote the values of* f *and* x. ∗)

$\cdots$ *declarations of the* P_i*'s* $\cdots$

var $\mathsf{query}_0\colon \mathrm{N}_1, \ldots, \mathsf{query}_{d-1}\colon \mathrm{N}_{2d-1},\ \ \mathsf{answer}_1\colon \mathrm{N}_2, \ldots, \mathsf{answer}_d\colon \mathrm{N}_{2d},$

$\mathsf{m}_0\colon \mathrm{N}_0, \ldots, \mathsf{m}_d\colon \mathrm{N}_{2d},\ \ \mathsf{n}_0\colon \mathrm{N}_1, \ldots, \mathsf{n}_d\colon \mathrm{N}_{2d+1},$

$\mathsf{Queries}\colon \mathrm{N}_{2d+1},\ \ \mathsf{result}\colon \mathrm{N}_{2d+2},\ \ \cdots;$

(∗ *Initializations* ∗)

$\mathsf{Queries} :=$ the empty list;

$\mathsf{m}_0 := \epsilon_0;\ \ \mathsf{m}_1 := \epsilon_2;\ \ \ldots;\ \ \mathsf{m}_d := \epsilon_{2d};$

[6]To be specific: if $\mathsf{Queries}$ represents the list "$w_1, w_2, \ldots, w_j$," then the actual value of $\mathsf{Queries}$ is $h(w_1)\mathbf{11}h(w_2)\mathbf{11}\cdots\mathbf{11}h(w_j)$, where h is the string homomorphism given by $h(\mathbf{0}) = \mathbf{00}$ and $h(\mathbf{1}) = \mathbf{01}$. The empty list is represented by ϵ.

$\mathsf{n}_0 := \mathsf{P}_0(\mathsf{m}_0, \mathsf{x})$; $\mathsf{n}_1 := \mathsf{P}_1(\mathsf{m}_1, \mathsf{x})$; ... ; $\mathsf{n}_d := \mathsf{P}_d(\mathsf{m}_d, \mathsf{x})$;
$\cdots$ *other initializations* $\cdots$

For i := $\underline{1}$ **to** n_d **do**
(* *The Simulation Section* *)
If the next step of **M** is an oracle query **then**
$\mathsf{query}_{d-1} := \mathsf{down}(\mathsf{n}_{d-1},$ the contents of the query tape);
(* *By Claim 1,* query_{d-1} *always receives the query tape contents.* *)
the contents of the answer tape := $\mathsf{f}(\mathsf{query}_{d-1})$;
the contents of the query tape := ϵ;
Add query_{d-1} to the list kept in Queries;
else (* *the next step is an ordinary instruction* *)
$\cdots$ *the simulation of such instructions* $\cdots$
Endif;

(* *The Bookkeeping Section* *)
If the computation of **M** halts in the step just simulated **then**
$\mathsf{n}_d := \epsilon_{2d+1}$;
ElseIf the step just simulated was an oracle query **then**
(* *Update* m_1 *and* n_1 *)
Examine the w's of length no greater than n_0 in the Queries list and find the smallest such that $|f(w)|$ is maximal among these w's;
$\mathsf{query}_0 := \mathsf{down}(\mathsf{n}_0,$ the w found in the search);
$\mathsf{answer}_1 := \mathsf{f}(\mathsf{query}_0)$;
$\mathsf{m}_1 := |\mathsf{answer}_1|$; $\mathsf{n}_1 := \mathsf{P}_1(\mathsf{m}_1, x)$;

$\cdots$ *Analogous updates of* m_i *and* n_i *for* $i = 2, \ldots, d$. $\cdots$
Endif;
Endfor;
If $\mathsf{n}_d = \epsilon_{2d+1}$
then result := the contents of the **M**'s output tape;
else result := ϵ_{2d+2}; **Endif**;
Return result
End

The correctness of Sim is straightforward. □ **Proposition 18**

Proposition 19. *Every* ITLP_2*-computable functional of type* $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ *is in* BFF_2.

Proof. This argument is modeled after the proof of Proposition 14(c). We first establish second-order polynomial bounds on the growth rates of ITLP_2-computable functionals. Suppose $F\colon (\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ is computed via an ITLP_2-procedure P of type $(\mathrm{N} \to_+ \mathrm{N}) \times \mathrm{N}_0 \to \mathrm{N}_j$ for some j. Let N_k be the highest tier occurring in P.

Conventions: To argue about computations of P, we adapt some of the notions from the proof of Proposition 14(c) to this imperative setting. Suppose we are considering a particular step in a given P-computation and $\vec{\mathsf{v}}$ is a list of integer variables visible (with respect to the block structure) at the current point in the program. Then $|\vec{\mathsf{v}}|_i$ denotes the maximum length of the values assigned to the current instantiation of the variables in $\vec{\mathsf{v}}$ of types $\mathrm{N}_0, \dots, \mathrm{N}_i$ up through this point in the computation. That is, $|\vec{\mathsf{v}}|_i$ is the current high-water mark of the present instantiation of the variables of types $\mathrm{N}_0, \dots, \mathrm{N}_i$ in $\vec{\mathsf{v}}$. Also, at each point in the computation, ζ denotes the finite portion of the oracle f discovered (via queries) up through this point. Thus, an oracle query can enlarge ζ. Recall that for each a and $b \in \omega$, $a \oplus b = \max(a, b)$. In the following we allow ourselves the liberty of using $\oplus$ as another "additive" operator in first- and second-order polynomials. We can always get rid of the $\oplus$'s in a $\mathbf{q}$ by replacing them with $+$'s and thus obtain a pure second-order polynomial $\mathbf{q}'$ such that $\mathbf{q} \le \mathbf{q}'$. Below, let $\vec{p}$, $\vec{q}$, and $\vec{r}$ range over finite sequences of one variable, nonconstant, first-order polynomials.

Definition 20. Given $\vec{p} = p_0, \dots, p_n$, we define $\mathbf{b}_{\vec{p},0}, \dots, \mathbf{b}_{\vec{p},n}$ and $\mathbf{c}_{\vec{p},0}, \dots, \mathbf{c}_{\vec{p},n}$ to be the second-order polynomials such that, for $g\colon \omega \to \omega$ and $m_0, \dots, m_n \in \omega$:

$$\begin{aligned}
\mathbf{b}_{\vec{p},0}(g, m_0) &= p_0(0) + m_0. \\
\mathbf{b}_{\vec{p},i+1}(g, m_{i+1}, \dots, m_0) &= p_{i+1}(b_i) + \Big(g(b_i) \oplus m_{i+1}\Big), \text{ where} \\
&\qquad b_i = \mathbf{b}_{\vec{p},i}(g, m_i, \dots, m_0).
\end{aligned}$$

$$\begin{aligned}
\mathbf{c}_{\vec{p},0}(g, m_0) &= p_0(0) \oplus m_0. \\
\mathbf{c}_{\vec{p},i+1}(g, m_{i+1}, \dots, m_0) &= \Big(2 \cdot p_{i+1}(b_i) + g(b_i)\Big) \oplus m_{i+1}, \text{ where} \\
&\qquad b_i = \mathbf{b}_{\vec{p},i}(g, c_i, \dots, c_0) \text{ and} \\
&\qquad c_j = \mathbf{c}_{\vec{p},j}(g, m_j, \dots, m_0), \text{ for each } j \le i.
\end{aligned}$$

◇

For a given choice of $\vec{p}$, $\mathbf{p}_{\vec{p},i}(g, m_i, \dots, m_0)$ is intended to express a bound on the lengths of possible tier-i values at a particular point in an ITLP_2 code fragment, where g corresponds to the estimated length of the oracular argument and m_j (for $j = 0, \dots, i$) is roughly the maximum length of the values of the tier-j variables at the beginning of the execution of the fragment; $\mathbf{c}_{\vec{p},i}$ serves an analogous purpose for the lengths of possible tier-i values inside of **For**-loops where, in this case i, is no greater than the tier of the loop's control variable. We formalize these intentions in

Definition 21.

(a) Suppose $\vec{I}$ is a sequence of instructions occurring within P and that $\vec{\mathsf{v}}$ is a list of all the integer variables visible in $\vec{I}$. We say that $\vec{p} = p_0, \dots, p_k$ is *a sequence of growth bounds for* $\vec{I}$ if and only if it is the case that whenever *before* executing $\vec{I}$ we have $m_i = |\vec{\mathsf{v}}|_i$ for each $i \le k$, then *after* executing $\vec{I}$ we have $|\vec{\mathsf{v}}|_i \le \mathbf{b}_{\vec{p},i}(|\zeta|, m_i, \dots, m_0)$ for each $i \le k$, where ζ denotes the finite part of f known *after* the execution of $\vec{I}$.

(b) Suppose $\vec{I}$ and $\vec{\mathsf{v}}$ are as in part (b). We say that $\vec{p} = p_0, \dots, p_j$ is *a sequence of copacetic growth bounds for* $\vec{I}$ *through tier* j if and only if it is the case that whenever *before* executing $\vec{I}$ we have $m_i = |\vec{\mathsf{v}}|_i$ for each $i \le j$, then *after* executing $\vec{I}$ we have $|\vec{\mathsf{v}}|_i \le \mathbf{c}_{\vec{p},i}(|\zeta|, m_i, \dots, m_0)$ for each $i \le j$, where ζ denotes the finite part of f known *after* the execution of $\vec{I}$.

(c) Suppose Q is a procedure occurring within P that has return type N_i and that $\vec{\mathsf{v}}$ is a list of all the integer variables that appear as arguments in a particular call to Q. Then $\vec{p} = p_0, \dots, p_i$ gives a *growth bound* for Q if and only if, in any computation of P, if $m_j = |\vec{\mathsf{v}}|_j$ for each $j \le i$ before a particular call to Q, then the length of the value returned by this call is no greater than $\mathbf{b}_{\vec{p},i}(|\zeta|, m_i, \dots, m_0)$, where ζ denotes the finite portion of f known as of the *end* of this particular call.

(d) Suppose $\vec{I}$ is a sequence of instructions occurring within P. We say that $\vec{I}$ is *copacetic through tier* j if and only if every assignment $v^{\mathrm{N}_i} := E$ occurring within $\vec{I}$ with $i \le j$ is such that either (i) each integer variable in E is of tier less than i, or (ii) E is of the form $\mathsf{down}(x^{\mathrm{N}_i}, y^{\mathrm{N}_j})$. $\diamond$

Lemma 22.

(a) Suppose Q *is a procedure occurring within* P *that has return type* N_i*. Then there is a* $\vec{p}$ *that gives a growth bound for* Q*.*

(b) Suppose $\vec{I}$ *is a sequence of instructions occurring within* P*. Then there is an associated sequence of growth bounds for* $\vec{I}$*.*

(c) Suppose $\vec{I}$ *is a sequence of instructions occurring within* P *that is copacetic through tier* j*. Then there is an associated sequence of copacetic growth bounds for* $\vec{I}$ *through tier* j*.*

Proof. The argument is a structural induction on P. Recall that N_k is the highest tier occurring in P.

Part (a). Suppose that Q is a declared procedure occurring within P with return type N_i and with body $\vec{I}_{\mathsf{Q}}$. Let $\vec{\mathsf{v}}$ be a list of the integer variables occurring as arguments of Q in a particular call and, for $i = 0, \ldots, k$, let m_i be the value of $|\vec{\mathsf{v}}|_i$ just before this call in a particular computation. Let $\vec{p}$ be a sequence of growth bounds for $\vec{I}_{\mathsf{Q}}$. (By the induction hypothesis, such a $\vec{p}$ must exist.) Let $\vec{\mathsf{u}}$ be a list of all the integer variables visible within $\vec{I}_{\mathsf{Q}}$. Recall that there are no nonlocal references to integer variables in ITLP_2 and that the local variables introduced in a procedure's **var** declaration are initialized to ϵ in every execution of the procedure. It follows then that $|\vec{\mathsf{u}}|_0 \leq m_0$, $\cdots$, $|\vec{\mathsf{u}}|_k \leq m_k$ just before the execution of $\vec{I}_{\mathsf{Q}}$ on this call. So since $\vec{p}$ is a sequence of growth bounds on $\vec{I}_{\mathsf{Q}}$ and since all of the polynomials involved are monotone, nondecreasing, we have that bound on the value returned by Q is as required.

Parts (b) and (c): The case of individual assignments. Consider an assignment $v^{\mathrm{N}_i} := E$. Let $\vec{\mathsf{v}}$ be a list of all the integer variables appearing in this assignment. Suppose that $m_{-1} = 0$ and, for each $\ell = 0, \ldots, k$, m_ℓ is the value of $|\vec{\mathsf{v}}|_\ell$ before the assignment in some execution of P.

Subcase (i). Suppose E is of the form: $\mathsf{down}(v_0, v_1)$. Then the length of value of E is no greater than m_i. For each $\ell = 0, \ldots, k$, let $p_\ell = \lambda n \,.\, n$. Then straightforward arguments show that $p_0, \ldots, p_k$ is a sequence of growth bounds for the assignment and that $p_0, \ldots, p_k$ is also a sequence of copacetic growth bounds for the assignment through tier k.

Subcase (ii). Suppose E is of one of the forms ϵ, v_0, $\mathsf{c}_{\mathbf{0}}(v_0)$, or $\mathsf{c}_{\mathbf{1}}(v_0)$. Then the length of the value of E is clearly no greater than $m_i + 1$. Let $p_i = \lambda n \,.\, n{+}1$ and $p_j = \lambda n \,.\, n$ for $j \neq i$. Then a straightforward argument shows that $p_0, \ldots, p_k$ is a sequence of growth bounds for the assignment. Suppose all the variables in E are from tiers less than i. Then the length of the value of E is clearly no greater than $m_{i-1} + 1$. Another straightforward argument shows that the same $p_0, \ldots, p_k$ is a copacetic sequence of growth bounds for the assignment through tier k.

Subcase (iii). Suppose E is of the form $v_0(v_1)$ where v_0 is of type $\mathrm{N} \rightarrow_+ \mathrm{N}$ and v_1 is of tier less than i. (Hence, $i > 0$.) Then the length of the value of E is clearly no greater than $|\zeta|(m_{i-1})$. Let the p_i's be as in subcase (i). Then straightforward arguments show that $p_0, \ldots, p_k$ is both a sequence of growth bounds for the assignment and a sequence of copacetic growth bounds for the assignment through tier k.

Subcase (iv). Suppose E is a call to a procedure Q with return type N_j, where $j \leq i$. By part (a), there is a sequence $p_0, \ldots, p_j$ that gives a growth bound for Q. For each $\ell \in \{\, j+1, \ldots, k\,\}$, let $p_\ell = \lambda n \,.\, n$. Then it follows that $\vec{p} = p_0, \ldots, p_k$ is a sequence of growth bounds for the assignment. Now let us consider the copacetic case. By the restrictions on ITLP_2 types, we know that each of the integer variables appearing in the call has its tier-level less than i. If $i = 0$, then by part (a) the length of the value returned by the call is no greater than $p_0(0)$ and it follows that $\vec{p}$ is a sequence of copacetic growth bounds for the assignment through tier k. Suppose $i > 0$. Then by our hypothesis on $\vec{p}$ we have that the value returned by the call is of length no greater than $\mathbf{b}_{\vec{p},i}(|\zeta|, m_i, \dots, m_0) = p_i(b_{i-1}) + (\,|\zeta|(b_{i-1}) \oplus m_i\,)$, where $b_{i-1} = \mathbf{b}_{\vec{p},i-1}(|\zeta|, m_{i-1}, \dots, m_0)$. Since there are no tier i variables in $\vec{\mathsf{v}}$ we have that $m_i = m_{i-1}$. Hence, $p_i(b_{i-1}) + (\,|\zeta|(b_{i-1}) \oplus m_i\,) = p_i(b_{i-1}) + (\,|\zeta|(b_{i-1}) \oplus m_{i-1}\,)$. Also, by our assumption that all of the p_i's are non-constant, it follows that $p_i(b_{i-1}) \geq m_{i-1}$. Hence $p_i(b_{i-1}) + (\,|\zeta|(b_{i-1}) \oplus m_{i-1}\,) \leq 2 \cdot p_i(b_{i-1}) + |\zeta|(b_{i-1})$. It thus follows that in this case too, $\vec{p}$ is a sequence of copacetic growth bounds for the assignment through tier k.

Part (b): The sequencing case. The following claim shows how to construct growth bounds for a sequence of instructions from growth bounds for the constituent instructions in the sequence.

Claim 1. *Suppose that $I_1 \cdots I_{n+1}$ is a sequence of instructions occurring within* P*, that $\vec{p}$ is a sequence of growth bounds for $I_1 \cdots I_n$, and that $\vec{q}$ is a sequence of growth bounds for I_{n+1}. For each $i \leq k$, let $r_i = p_i + q_i$. Then $\vec{r}$ is a sequence of growth bounds for $I_1 \cdots I_{n+1}$.*

Proof of Claim 1. Suppose that $\vec{\mathsf{v}}$ is the list of all variables visible within $I_1 \cdots I_{n+1}$ and that at some point of an execution of P, the execution passes through $I_1 \cdots I_{n+1}$ and we have that, for each $i \leq k$,

$$\begin{aligned} m_i^a &= |\vec{\mathsf{v}}|_i \text{ just before executing } I_1, \\ m_i^b &= |\vec{\mathsf{v}}|_i \text{ just after executing } I_n \text{ (but before executing } I_{n+1}\text{), and} \\ m_i^c &= |\vec{\mathsf{v}}|_i \text{ just after executing } I_{n+1}. \end{aligned}$$

Also let ζ^b (respectively, ζ^c) be the finite portion of f known just before (respectively, after) I_{n+1} is executed. By hypothesis we have that, for each $i \leq k$, $m_i^b \leq \mathbf{b}_{\vec{p},i}(|\zeta^b|, \vec{m}^a)$ and $m_i^c \leq \mathbf{b}_{\vec{q},i}(|\zeta^c|, \vec{m}^b)$. For $i = 0$ we have, by Definition 20 and our hypotheses:

$$\begin{aligned} m_0^c \;\leq\; \mathbf{b}_{\vec{q},0}(|\zeta^c|, \vec{m}^b) \;&=\; q_0(0) + m_0^b \;\leq\; q_0(0) + \mathbf{b}_{\vec{p},0}(|\zeta^b|, \vec{m}^a) \\ &=\; q_0(0) + p_0(0) + m_0^a \;=\; \mathbf{b}_{\vec{r},0}(|\zeta^c|, \vec{m}^a). \end{aligned}$$

Thus, $m_0^c \leq \mathbf{b}_{\vec{q},0}(|\zeta^c|, \vec{m}^b) \leq \mathbf{b}_{\vec{r},0}(|\zeta^c|, \vec{m}^a)$. Now suppose as an induction hypothesis that $m_i^c \leq \mathbf{b}_{\vec{q},i}(|\zeta^c|, \vec{m}^b) \leq \mathbf{b}_{\vec{r},i}(|\zeta^c|, \vec{m}^a)$. Let $b_i^p = \mathbf{b}_{\vec{p},i}(|\zeta^b|, \vec{m}^a)$,

$b_i^q = \mathbf{b}_{\vec{q},i}(|\zeta^c|, \vec{m}^b)$, and $b_i^r = \mathbf{b}_{\vec{r},i}(|\zeta^c|, \vec{m}^a)$. By the monotonicity of the functions involved we have $b_i^p \leq b_i^r$ and by the induction hypothesis we have $b_i^q \leq b_i^r$. Thus,

$$\begin{aligned}
m_{i+1}^c &\leq \mathbf{b}_{\vec{q},i+1}(|\zeta^c|, \vec{m}^b) \\
&\qquad \text{(by our hypothesis on } \vec{q}\text{)} \\
&= q_{i+1}(b_i^q) + \Big(|\zeta^c|(b_i^q) \oplus m_{i+1}^b\Big) \\
&\qquad \text{(by the definition of } \mathbf{b}_{\vec{q},i+1}\text{)} \\
&\leq q_{i+1}(b_i^r) + \Big(|\zeta^c|(b_i^r) \oplus m_{i+1}^b\Big) \\
&\qquad \text{(by the monotonicity of the functions involved} \\
&\qquad\ \text{and since } b_i^q \leq b_i^r\text{)} \\
&\leq q_{i+1}(b_i^r) + \Big(|\zeta^c|(b_i^r) \oplus \mathbf{b}_{\vec{p},i+1}(|\zeta^b|, \vec{m}^a)\Big) \\
&\qquad \text{(by our hypothesis on } \vec{p}\text{)} \\
&\leq q_{i+1}(b_i^r) + \Big(|\zeta^c|(b_i^r) \oplus \Big(p_{i+1}(b_i^p) + (\,|\zeta^b|(b_i^p) \oplus m_{i+1}^a)\Big)\Big) \\
&\qquad \text{(by the definition of } \mathbf{b}_{\vec{p},i+1}\text{)} \\
&\leq q_{i+1}(b_i^r) + \Big(|\zeta^c|(b_i^r) \oplus \big(p_{i+1}(b_i^r) + (|\zeta^c|(b_i^r) \oplus m_{i+1}^a)\big)\Big) \\
&\qquad \text{(by the monotonicity of the functions involved)} \\
&\leq q_{i+1}(b_i^r) + p_{i+1}(b_i^r) + \Big(|\zeta^c|(b_i^r) \oplus m_{i+1}^a\Big) \\
&\qquad \text{(by algebra)} \\
&= r_{i+1}(b_i^r) + \Big(|\zeta^c|(b_i^r) \oplus m_{i+1}^a\Big) \\
&\qquad \text{(by the definition of } r_{i+1}\text{)} \\
&= \mathbf{b}_{\vec{r},i+1}(|\zeta^c|, \vec{m}^a) \\
&\qquad \text{(by the definition of } \mathbf{b}_{\vec{r},i+1}\text{).}
\end{aligned}$$

Thus, $m_{i+1}^c \leq \mathbf{b}_{\vec{q},i+1}(|\zeta^c|, \vec{m}^b) \leq \mathbf{b}_{\vec{r},i+1}(|\zeta^c|, \vec{m}^a)$. Therefore, $\vec{r}$ is a sequence of growth bounds for $I_1 \cdots I_{n+1}$ and Claim 1 follows.

Part (c): The sequencing case. The following shows how to compose copacetic growth bounds.

Claim 2. *Suppose that $I_1 \cdots I_{n+1}$ is a sequence of instructions occurring within* P, *that $\vec{p}$ is a sequence of copacetic growth bounds for $I_1 \cdots I_n$ through* *tier j, and that $\vec{q}$ is a sequence of copacetic growth bounds for I_{n+1} through tier j. For each $i \leq k$, let $r_i = p_i \oplus q_i$. Then $\vec{r}$ is a sequence of copacetic growth bounds for $I_1 \cdots I_{n+1}$ through tier j.*

Proof of Claim 2. Suppose that $\vec{\mathsf{v}}$, ζ^b, ζ^c, and, for each $i \leq k$, m_i^a, m_i^b, and m_i^c are as in the proof of Claim 1. By hypothesis we have that, for $i = 0, \dots, j$, $m_i^b \leq \mathbf{c}_{\vec{p},i}(|\zeta^b|, \vec{m}^a)$ and $m_i^c \leq \mathbf{c}_{\vec{q},i}(|\zeta^c|, \vec{m}^b)$. For $i = 0$ we have, by Definition 20 and our hypotheses:

$$\begin{aligned} m_0^c \;\leq\; & \mathbf{c}_{\vec{q},0}(|\zeta^c|, \vec{m}^b) &=\;& q_0(0) \oplus m_0^b \\ \leq\; & q_0(0) \oplus \mathbf{c}_{\vec{p},0}(|\zeta^b|, \vec{m}^a) &=\;& q_0(0) \oplus (p_0(0) \oplus m_0^a) \\ =\; & (p_0(0) \oplus q_0(0)) \oplus m_0^a &=\;& \mathbf{c}_{\vec{r},0}(|\zeta^c|, \vec{m}^a). \end{aligned}$$

Thus, $m_0^c \leq \mathbf{c}_{\vec{q},0}(|\zeta^c|, \vec{m}^b) \leq \mathbf{c}_{\vec{r},0}(|\zeta^c|, \vec{m}^a)$. Now suppose as an induction hypothesis that for each $\ell \leq i$ we have $m_\ell^c \leq \mathbf{c}_{\vec{q},\ell}(|\zeta^c|, \vec{m}^b) \leq \mathbf{c}_{\vec{r},\ell}(|\zeta^c|, \vec{m}^a)$. For each $\ell \leq i$, let $\mathsf{c}_\ell^p = \mathbf{c}_{\vec{p},\ell}(|\zeta^b|, m_\ell^a, \dots, m_0^a)$, $\mathsf{c}_\ell^q = \mathbf{c}_{\vec{q},\ell}(|\zeta^c|, m_\ell^b, \dots, m_0^b)$, and $\mathsf{c}_\ell^r = \mathbf{c}_{\vec{r},\ell}(|\zeta^c|, m_\ell^a, \dots, m_0^a)$. For each $\ell \leq i$, we have by the monotonicity of the functions involved that $\mathsf{c}_\ell^p \leq \mathsf{c}_\ell^r$ and by our induction hypothesis that $\mathsf{c}_\ell^q \leq \mathsf{c}_\ell^r$. Let $b_i^p = \mathbf{b}_{\vec{p},i}(|\zeta^b|, \mathsf{c}_i^p, \dots, \mathsf{c}_0^p)$, $b_i^q = \mathbf{b}_{\vec{q},i}(|\zeta^c|, \mathsf{c}_i^q, \dots, \mathsf{c}_0^q)$, and $b_i^r = \mathbf{b}_{\vec{r},i}(|\zeta^c|, \mathsf{c}_i^r, \dots, \mathsf{c}_0^r)$. Since $c_\ell^p \oplus c_\ell^q \leq c_\ell^r$ for each $\ell \leq i$, it follows by the monotonicity of the functions involved that $b_i^p \oplus b_i^q \leq b_i^r$. Thus,

$$\begin{aligned} m_{i+1}^c \;\leq\; & \mathbf{c}_{\vec{q},i+1}(|\zeta^c|, \vec{m}^b) \\ & \qquad \text{(by our hypothesis on } \vec{q}\text{)} \\ =\; & \Big(2 \cdot q_{i+1}(b_i^q) + |\zeta^c|(b_i^q)\Big) \oplus m_{i+1}^b \\ & \qquad \text{(by the definition of } \mathbf{b}_{\vec{q},i+1}\text{)} \\ \leq\; & \Big(2 \cdot q_{i+1}(b_i^r) + |\zeta^c|(b_i^r)\Big) \oplus m_{i+1}^b \\ & \qquad \text{(by the monotonicity of the functions involved} \\ & \qquad \text{ and since } b_i^q \leq b_i^r\text{)} \\ \leq\; & \Big(2 \cdot q_{i+1}(b_i^r) + |\zeta^c|(b_i^r)\Big) \oplus \Big(2 \cdot p_{i+1}(b_i^p) + |\zeta^b|(b_i^p)\Big) \oplus m_{i+1}^a \\ & \qquad \text{(by our hypothesis on } \vec{p}\text{)} \\ \leq\; & \Big(2 \cdot q_{i+1}(b_i^r) + |\zeta^c|(b_i^r)\Big) \oplus \Big(2 \cdot p_{i+1}(b_i^r) + |\zeta^c|(b_i^r)\Big) \oplus m_{i+1}^a \\ & \qquad \text{(by the monotonicity of the functions involved} \\ & \qquad \text{ and since } b_i^p \leq b_i^r\text{)} \\ \leq\; & \Big(2 \cdot (\, p_{i+1}(b_i^r) \oplus q_{i+1}(b_i^r)\,) + |\zeta^c|(b_i^r)\Big) \oplus m_{i+1}^a \end{aligned}$$

$$\begin{aligned}
&\qquad \text{(by algebra)}\\
= &\quad \Big(2\cdot r_{i+1}(b_i^r) + |\zeta^c|(b_i^r)\Big) \oplus m_{i+1}^a\\
&\qquad \text{(by the definition of } r_{i+1})\\
= &\quad \mathbf{c}_{\vec{r},i+1}(|\zeta^c|, \vec{m}^a)\\
&\qquad \text{(by the definition of } \mathbf{c}_{\vec{r},i+1}).
\end{aligned}$$

Thus, $m_{i+1}^c \leq \mathbf{c}_{\vec{q},i+1}(|\zeta^c|, \vec{m}^b) \leq \mathbf{c}_{\vec{r},i+1}(|\zeta^c|, \vec{m}^a)$. Therefore, $\vec{r}$ is a sequence of growth bounds for $I_1 \cdots I_{n+1}$ and Claim 2 follows.

Parts (b) and (c): The **Case**-*statement case.* This is straightforward and omitted.

Parts (b) and (c): The **For**-*loop case.* Given an $n \in \omega$ and an $\vec{I}$, a sequence of instructions, let $(\vec{I}\,)^n$ denote the sequence of instructions obtained by repeating $\vec{I}$ n-many times. As a corollary to Claims 1 and 2 we have

Claim 3. *Fix n.*

(a) Suppose $\vec{p}$ is a sequence of growth bounds for $\vec{I}$. Let $n' = \max(n,1)$ and, for each $i \leq k$, let $q_i = \lambda m.\, (n' \cdot p_i(m))$. Then $\vec{q}$ is a sequence of growth bounds for $(\vec{I}\,)^n$.

(b) If $\vec{p}$ is a sequence of copacetic growth bounds for $\vec{I}$ through tier j, then $\vec{p}$ is also a sequence of copacetic growth bounds for $(\vec{I}\,)^n$ through tier j.

The proof of part (a) of the claim follows directly from Claim 1. The proof of part (b) of the claim follows from Claim 2 and the fact that, for all $a \in \omega$, $a \oplus a = a$.

The next claim provides us with one more fact we need before proceeding with the analysis of **For**-loops.

Claim 4. *Given $\vec{q} = q_0, \ldots, q_j$, there exists $\vec{r} = r_0, \ldots, r_j$ such that for all monotone nondecreasing $g\colon \omega \to \omega$ and for all $m_j, \ldots, m_0$ we have that for $i = 0, \ldots, j$, $\mathbf{c}_{\vec{q},i}(g, m_i, \ldots, m_0) \leq \mathbf{b}_{\vec{r},i}(g, m_i, \ldots, m_0)$.*

This claim follows by more dreadful algebra—which we omit.

Now, suppose that

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

occurs in P and that $\vec{\mathsf{v}}$ is a list of all the integer variables visible in the **For**-loop.

To argue part (c), suppose the **For**-loop meets the copaceticity conditions through tier j'. Then it follows that $\vec{I}$ also meets copaceticity conditions through tier j'. So, by the induction hypothesis there is a $\vec{q} = q_0, \ldots, q_{j'}$ that is a sequence of copacetic growth bounds for $\vec{I}$. It follows by Claim 3(b) that $\vec{q}$ is also a sequence of copacetic growth bounds for the entire **For**-loop through tier j'. Hence, part (c) follows for this case.

To argue part (b), we first note that, by the induction hypothesis, there is a $\vec{p} = p_0, \ldots, p_k$ that is a sequence of growth bounds for $\vec{I}$. By the conditions on bodies of **For**-loops, it must be the case that $\vec{I}$ is copacetic through tier j. So, by the argument for part (c) we have that there is a $\vec{q} = q_0, \ldots, q_j$ which is a sequence of copacetic growth bounds for the **For**-loop through tier j. Therefore, if some execution of P executes this **For**-loop and before this particular execution we have, for $i = 0, \ldots, j$, $m_i = |\vec{\mathsf{v}}|_i$, then we have that this execution goes through at most $\mathbf{c}_{\vec{q},j}(|f|, m_j, \ldots, m_0)$ iterations. Hence, the execution of this **For**-loop terminates and, in fact, goes through no more than $\mathbf{c}_{\vec{q},j}(|\zeta|, m_j, \ldots, m_0)$ iterations, where ζ is that part of f known as of the end of this execution. Let $\vec{r} = r_0, \ldots, r_j$ be as in Claim 4. Therefore, in this execution, the number of iterations is no more than $\mathbf{b}_{\vec{r},j}(|\zeta|, m_j, \ldots, m_0)$ iterations. For each $\ell = j+1, \ldots, k$, let $r_\ell = \lambda m.(\, m \cdot p_\ell(m)\,)$. Then some algebra shows that $\vec{r}$ is a sequence of growth bounds for the **For**-loop. Hence, part (b) follows in this case. $\square$ **Lemma 22**

Scholium: Note that the argument for the **For**-loop case of Lemma 22 amounts to determining a "fixed-point" for the bounds on tier values.

By Lemma 22(a) it follows that there is a depth j second-order polynomial $\mathbf{q}$ such that the value returned by P on input (f, x) is bounded above by $\mathbf{q}(|f|, |x|)$. Let $\mathbf{q}'$ be the result of replacing each '$\oplus$' in $\mathbf{q}$ by '$+$'. Clearly, $\mathbf{q}'$ is also a depth j second-order polynomial such that $\mathbf{q}'(|f|, |x|)$ bounds the value returned by P on input (f, x). Therefore, we have the desired (conventional) second-order polynomial bounds on the growth rates of ITLP_2-computable functionals.

The rest of the proof of Proposition 19 is blessedly conventional. One defines a simple OTM-based interpreter for ITLP_2 programs and a cost model for ITLP_2 programs based on this interpreter and argues that each ITLP_2-program has a second-order polynomial bound on the cost of running it. The details of this are standard and straightforward and thus omitted. The proposition thus follows. $\square$ **Proposition 19**

9. Conclusions

Have our goals been meet?

Our aim in constructing ITLP_2 was to replace the machines and clocks of the $\mathbf{M}_{i,d,k}$'s with programming constructs and types to achieve a more structured and understandable version of the programming formalism of the $\mathbf{M}_{i,d,k}$'s. In some respects ITLP_2 clearly satisfies our goals, in other respects things are not so clear.

First, let us argue that the $\mathbf{M}_{i,d,k}$'s and ITLP_2 are in a reasonable sense closely related. Note that the only difficult part of showing Proposition 17 comes in arguing Lemma 22—the confirmation of the connection between the tiers and second-order polynomial bounds. Beyond the proof of that lemma, the argument for Proposition 17 consists of two straightforward simulations. So if we measure closeness in terms of the ease of the simulations and translations, ITLP_2 and the clocked OTMs are indeed close.

It is more questionable as to whether ITLP_2 provides more understandable programs than the $\mathbf{M}_{i,d,k}$'s. Clearly, ITLP_2 procedures are more palatable than OTM code. However, many of the dynamic aspects of the $\mathbf{M}_{i,d,k}$'s are inherited by ITLP_2 in its inflationary tiers and **For**-loop and, as the proof of Lemma 22 shows, these inflationary features take some work to analyze. We suspect that the dynamics are inherent to the situation, which in our view is one of its charms.

Types for complexity analyses

Bellantoni and Cook's safe-recursion formalism [BC92], Leivant's tiered-recursion formalisms, and our ITLP_2 all provide examples of how to use types to capture complexity classes. (For other examples, we refer the reader to the work of Asperti [Asp98], Bellantoni, Niggl, and Schwichtenberg [BNS], Hofmann [Hof97, Hof99b, Hof99a], Girard [Gir98], Girard, Scedrov, and Scott [GSS92], and Otto [Ott95].) A natural question is whether these ideas can be developed to the point of providing type systems to aide in practical complexity analyses of programs—particularly, programs that use higher-types.

ITLP_2 certainly is *not* a practical tool for the analysis of type-2 algorithms. In any given ITLP_2 program, the set of bounds corresponding to any particular tier is simply too loose and big to support precise reasoning about sizes and complexity. A tighter type system for reasoning about complexity would link tiers/types with smaller classes of complexity bounds, e.g., $O(\mathbf{q})$ or $\mathbf{q} + O(1)$, where $\mathbf{q}$ is a second-order polynomial. It is clearly possible to push the ITLP_2 system in this direction. We suspect, however, something more radical is required to produce a practical tool. The problem is that the ITLP_2 type system respects abstraction barriers, but tight complexity analyses of algorithms are often forced to ignore these barriers. A possible way around this would be to have a family of related type-systems for a given programming formalism, each of which would give a different view of the program—some views appropriate for reasoning about correctness, others for reasoning about complexity. This, of course, is simply speculation. However, types are such powerful and flexible tools that it would be surprising if one could not craft a type system that would closely support reasoning about algorithmic complexity.

Acknowledgments

Thanks to Stuart Kurtz, Gary Leavens, Jack Lutz, Peter O'Hearn, and Sue Older for their comments on various stages of this work. The anonymous referees also made many very helpful comments. We happily acknowledge the influence of Neil Jones's text [Jon97] that provided us with an extended example of how to carefully mesh programming language and complexity-theoretic concerns. Thanks also to Elaine Weinman for her careful comments on the text. Preliminary reports on this work were presented at the 14th Annual *Mathematical Foundations of Programming Semantics Workshop* (London, May 1998) and the *Implicit Computational Complexity in Programming Language Design and Methodology Workshop* (Baltimore, September 1998).

References

[Asp98] A. Asperti, *Light affine logic*, Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, 1998.

[BC92] S. Bellantoni and S. Cook, *A new recursion-theoretic characterization of the polytime functions*, Computational Complexity **2** (1992), 97–110.

[Bel92] S. Bellantoni, *Predicative recursion and computational complexity*, Ph.D. thesis, University of Toronto, 1992, appears as University of Toronto Computer Science Dept. Technical Report 264/92.

[BNS] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg, *Characterising polytime through higher type recursion*, Annals of Pure and Applied Logic, to appear.

[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.

[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.

[Con73] R. Constable, *Type two computational complexity*, Proceedings of the Fifth Annual ACM Symposium on the Theory of Computing, 1973, pp. 108–121.

[Coo71] S. Cook, *The complexity of theorem proving procedures*, Proceedings of the Third Annual ACM Symposium on the Theory of Computing, 1971, pp. 151–158.

[Coo91] S. Cook, *Computability and complexity of higher type functions*, Logic from Computer Science (Y.N. Moschovakis, ed.), Springer-Verlag, 1991, pp. 51–72.

[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.

[GH77] R. Gandy and J. Hyland, *Computable and recursively countable functions of higher type*, Logic Colloquium 76, North-Holland, 1977, pp. 407–438.

[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.

[GSS92] J.-Y. Girard, A. Scedrov, and P.J. Scott, *Bounded linear logic: A modular approach to polynomial time computability*, Theoretical Computer Science **97** (1992), 1–66.

[Hil25] D. Hilbert, *Über das unendliche*, Mathematische Annalen **95** (1925), 161–190.

[Hil67] D. Hilbert, *On the infinite*, From Frege to Gödel: A source book in mathematical logic, 1879–1931 (J. van Heijenoort, ed.), Harvard University Press, 1967, (an English translation of [Hil25]), pp. 367–392.

[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.

[Jon97] N. Jones, *Computability and complexity from a programming perspective*, MIT Press, 1997.

[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.

[KLS57] G. Kreisel, D. Lacombe, and J. Shoenfield, *Partial recursive functionals and effective operations*, Constructivity in Mathematics: Proceedings of the Colloquium held at Amsterdam (A. Heyting, ed.), North-Holland, 1957, pp. 195–207.

[Lei91] D. Leivant, *A foundational delineation of computational feasibility*, Proceedings of the Sixth IEEE Conference on Logic in Computer Science, 1991, pp. 2–11.

[Lei94a] D. Leivant, *A foundational delineation of poly-time*, Information and Computation **110** (1994), 391–420.

[Lei94b] D. Leivant, *Predicative recurrence in finite types*, Logical Foundations of Computer Science: Third International Symposium, LFCS '94 (A. Nerode and Yu. Matiyasevich, eds.), Lecture Notes in Computer Science, vol. 813, Springer-Verlag, 1994, pp. 227–239.

[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.

[Meh74] K. Mehlhorn, *Polynomial and abstract subrecursive classes*, Proceedings of the Sixth Annual ACM Symposuium on the Theory of Computing, 1974, pp. 96–109.

[Meh76] K. Mehlhorn, *Polynomial and abstract subrecursive classes*, Journal of Computer and System Science **12** (1976), 147–178.

[Nel86] E. Nelson, *Predicative arithmetic*, Princeton University Press, 1986.

[Ott95] J. Otto, *Complexity doctrines*, Ph.D. thesis, McGill University, 1995.

[Pez98] E. Pezzoli, *On the computational complexity of type two functionals*, Proceedings of the Conference for Computer Science Logic '97 (D. van Dalen and M Bezem, eds.), Lecture Notes in Computer Science, vol. 1414, Springer-Verlag, 1998, pp. 373–388.

[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.

[Roy97] J. Royer, *Semantics versus syntax versus computations: Machine models for type-2 polynomial-time bounded functionals*, Journal of Computer and System Science **54** (1997), 424–436.

[Rus03] B. Russell, *The principles of mathematics, Vol. I*, Cambridge University Press, 1903.

[Rus08] B. Russell, *Mathematical logic as based on the theory of types*, American Journal of Mathematics **30** (1908), 222–262.

[Rus67] B. Russell, *Mathematical logic as based on the theory of types*, From Frege to Gödel: A source book in mathematical logic, 1879–1931 (J. van Heijenoort, ed.), Harvard University Press, 1967, pp. 152–182.

[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.

[Set93] A. Seth, *Some desirable conditions for feasible functions of type 2*, Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science, 1993, pp. 320–331.

[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.

[Tow90] M. Townsend, *Complexity for type-2 relations*, Nortre Dame Journal of Formal Logic **31** (1990), 241–262.

[Tur36] A. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society **42** (1936), 230–265.

[You90] P. Young, *Juris Hartmanis: Fundamental contributions to isomorphism problems*, Complexity Theory Retrospective (A. Selman, ed.), Springer-Verlag, 1990, pp. 28–58.