# Adventures in Time and Space

Norman Danner
Dept. of Mathematics and Computer Science,
Wesleyan University, Middletown, CN, USA
ndanner@wesleyan.edu

James S. Royer
EECS Dept., Syracuse University, Syracuse, NY, USA
royer@ecs.syr.edu

## Abstract

This paper investigates what is essentially a call-by-value version of PCF under a complexity-theoretically motivated type system. The programming formalism, $\mathsf{ATR}_1$, has its first-order programs characterize the poly-time computable functions, and its second-order programs characterize the type-2 basic feasible functionals of Mehlhorn and of Cook and Urquhart. (The $\mathsf{ATR}_1$-types are confined to levels 0, 1, and 2.) The type system comes in two parts, one that primarily restricts the sizes of values of expressions and a second that primarily restricts the time required to evaluate expressions. The size-restricted part is motivated by Bellantoni and Cook's and Leivant's implicit characterizations of poly-time. The time-restricting part is an affine version of Barber and Plotkin's DILL. Two semantics are constructed for $\mathsf{ATR}_1$. The first is a pruning of the naïve denotational semantics for $\mathsf{ATR}_1$. This pruning removes certain functions that cause otherwise feasible forms of recursion to go wrong. The second semantics is a model for $\mathsf{ATR}_1$'s time complexity relative to a certain abstract machine. This model provides a setting for complexity recurrences arising from $\mathsf{ATR}_1$ recursions, the solutions of which yield second-order polynomial time bounds. The time-complexity semantics is also shown to be sound relative to the costs of interpretation on the abstract machine.

***Categories and Subject Descriptors*** F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs — *program and recursion schemes, type structure*; F.1.3 [*Computation by Abstract Devices*]: Complexity Measures and Classes; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

***General Terms*** Languages, Performance, Theory

## 1. Introduction

> *A Lisp programmer knows the value of everything, but the cost of nothing.* — Alan Perlis

Perlis' quip is an overstatement—but not by much. Programmers in functional (and object-oriented) languages have few tools for reasoning about the efficiency of their programs. Almost all tools from traditional analysis of algorithms are targeted toward roughly the first-order fragment of C. What tools there are from formal methods are interesting, but piecemeal and preliminary.



*POPL'06* January 11–13, 2006, Charleston, South Carolina, USA.


This paper is an effort to fill in part of the puzzle of how to reason about the efficiency of programs that involve higher types. Our approach is, roughly, to take PCF and its conventional denotational semantics [24, 30] and, using types, restrict the language and its semantics to obtain a higher-type "feasible fragment" of PCF and the PCF computable functions. Our notion of higher-type feasibility is based on the *basic feasible functionals* (BFFs) [5, 21], a higher-type analogue of poly-time computability, and Kapron and Cook's [15] machine-based characterization of this class at type-level 2.[1] Basing our work on a higher-type notion of computational complexity provides a connection to the basic notions and tools of traditional analysis of algorithms (and their lifts to higher types). Enforcing feasibility constraints on PCF through types provides a connection to much of the central work in formal methods.

Our approach is in contrast to the work of [3, 11, 18] which also involves higher-type languages and types that guarantee feasibility. These programming formalisms are feasible in the sense that: (i) they have poly-time normalization properties and (ii) the type-level 1 functions expressible by these systems are guaranteed to be (ordinary) poly-time computable. The higher-type constructions of these formalisms are essentially aides for poly-time programming; as of this writing, there is no analysis of what higher-type functions these systems compute.[2]

For a simple example of a feasible higher-type function, consider $C\colon (\mathsf{N}\to\mathsf{N})\to(\mathsf{N}\to\mathsf{N})\to(\mathsf{N}\to\mathsf{N})$ with $C\,f\,g = f\circ g$. (*Convention:* N is always interpreted as $\{\,\mathbf{0},\mathbf{1}\,\}^*$, i.e., $\mathbf{0}$-$\mathbf{1}$-strings.) In our setting, a reasonable implementation of $C$ has a run-time bound that is a second-order polynomial (see §2) in the complexities of arbitrary $f$ and $g$; in particular, if $f$ and $g$ are poly-time computable, so is $C\ f\ g$. Such a combinator $C$ can be considered as part of the "feasible glue" of a programming environment—when used with other components, its complexity contribution is (higher-type) polynomially bounded in terms of the complexity of the other components *and* the combined complexity can be expressed in a natural, compositional way. More elaborate examples of feasible functionals include many of the deterministic "black box" constructions from cryptography that, for example, map one pseudo-random generator $g$ to another $g'$ with better cryptographic properties in such a way that the complexity properties of the program for $g'$ are not

[1] Mehlhorn [21] originally discovered the class of type-2 basic feasible functionals the mid-1970s. Later Cook and Urquhart [5] independently discovered this class and extended it to all finite types over the full set-theoretic hierarchy. **N.B.** If one restricts attention to continuous models, then starting at type-level 3 there are alternative notions of "higher-type poly-time" [13]. Dealing with type-level 3 and above involves some knotty semantic and complexity-theoretic issues beyond the scope of this paper, hence our restriction of $\mathsf{ATR}_1$ types to orders 2 and below.

[2] In fact, the work of [3, 11] and of this paper sit on different sides of an important divide in higher-type computability between notions of *computation over computable data* (e.g., [3, 11, 18]) and notions of *computation over continuous data* (e.g., this paper) [19, 20].

much worse that of the program of $g$, *but* where this program for $g$ many not run in *deterministic* poly-time.[3] (See Chapter 3 in Goldreich's [8] for examples.)

While our notion of feasibility is based on the BFFs, the full class of functions our formalism computes is more subtle than the BFFs. For example, consider $\mathsf{prn}\colon (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}$ with:

$$\left.\begin{array}{rcl} \mathsf{prn}\ f\ \epsilon & \longrightarrow & f\ \epsilon\ \epsilon. \\ \mathsf{prn}\ f\ (\mathbf{a} \oplus y) & \longrightarrow & f\ (\mathbf{a} \oplus y)\ (\mathsf{prn}\ f\ y). \end{array}\right\} \quad (1)$$

(*Conventions:* $\oplus$ denotes string concatenation and $\mathbf{a} \in \{\mathbf{0}, \mathbf{1}\}$.) So, prn is a string-variant of foldr. It is well-known that starting with poly-time primitives, prn can be used to define any primitive recursive function. *(Hence,* prn *is* ***not*** *a* BFF.*)* But, as Cobham [4] noted, if one adds to (1) the side-condition to that, for some polynomial $p$, we have $|\mathsf{prn}\ f\ x\ y| \leq p(|x|, |y|)$, and if one starts with poly-time primitives, then this modified prn produces definitions of only poly-time computable functions. Bellantoni and Cook [2] showed how get rid of *explicit* use of such a side condition through what amounts to a typing discipline. However, their approach (which has been in large part adopted by the implicit complexity computational community, see Hofmann's survey [10]), requires that prn be a "special form" and that the $f$ in (1) be always given by a syntactic definition. We, on the other hand, want to be able to define prn (see Figure 8) and have the definition's meaning given by a conventional denotational semantics. We thus use Bellantoni and Cook's [2] (and Leivant's [16]) ideas in both syntactic *and semantic* contexts. That is, we extract the growth-rate bounds implicit in the aforementioned systems, extend these bounds to high types, and create a type system, programming language, and semantic models that work to enforce these bounds. As a consequence, we can define prn, with a particular typing, and be assured that, whether the $f$ corresponds to a purely syntactic term or to the interpretation a free variable, prn will not go wrong and produce something of undesirable complexity. The language and its model thus implicitly incorporate side-conditions on growth via types.[4] Handling constructs like prn as first class functions is important because programmers care more about such combinators than about most any BFF.

***Outline*** §3 presents the syntactic side of our programming formalism $\mathsf{ATR}_1$ and its type system. §4 explains what can go wrong in a naïve semantics for $\mathsf{ATR}_1$ and its types. §4 also shows how to prune the naïve semantics obtain a semantics for which we can prove *polynomial-size boundedness* for $\mathsf{ATR}_1$, that is, that the size of of the value of each $\mathsf{ATR}_1$ expression $E$ has a second-order polynomial bound over the sizes of the values of $E$'s free variables. §5 gives a time-complexity semantics for $\mathsf{ATR}_1$ expressions that can be shown: (i) sound for the cost model of a particular abstract machine for evaluating $\mathsf{ATR}_1$ and (ii) *polynomial-time bounded*, that is that the time-complexity each $\mathsf{ATR}_1$ expression $E$ has a second-order polynomial bound over the time-complexities of of $E$'s free variables. §5 also states that $\mathsf{ATR}_1$ can compute each basic feasible functional. §6 briefly discusses work related to ours and §7 considers possible extensions of our work. We begin in §2 by stating some basic background definitions. **N.B.** This is an abstract of a much longer paper. The present paper omits proofs, secondary examples, and discussions of finer points of this work.

[3] E.g., $g$ and $g'$ may be computed by a probabilistic poly-time programs or even by families of circuits.

[4] Incorporating side-conditions in models is nothing new. A fixed point combinator has the implicit side-condition that its argument is continuous or at least monotone so that, by Tarski's fixed point theorem [30], we know the result is meaningful. Models of languages with fixed point combinators typically have continuity built-in so the side-condition is always implicit.

$$\begin{array}{l} E ::= K \mid (\mathsf{c}_{\mathbf{a}}\ E) \mid (\mathsf{d}\ E) \mid (\mathsf{t}_{\mathbf{a}}\ E) \mid (\mathsf{down}\ E\ E) \mid V \\ \qquad \mid (E\ E) \mid (\lambda V\,.\,E) \mid (\mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E) \mid (\mathsf{fix}\ E) \\ K ::= \{\mathbf{0}, \mathbf{1}\}^* \qquad \mathbf{a} \in \{\mathbf{0}, \mathbf{1}\} \qquad T ::= \text{the simple types over } \mathsf{N} \end{array}$$

**Figure 1.** PCF syntax

$$\textit{Const-I:}\ \frac{}{\Gamma \vdash K\colon \mathsf{N}} \qquad \textit{Id-I:}\ \frac{}{\Gamma,\, v\colon\sigma \vdash v\colon\sigma} \qquad \mathsf{op}\textit{-I:}\ \frac{\Gamma \vdash E\colon \mathsf{N}}{\Gamma \vdash (\mathsf{op}\ E)\colon \mathsf{N}}$$

$$\mathsf{down}\textit{-I:}\ \frac{\Gamma_0 \vdash E_0\colon \mathsf{N} \quad \Gamma_1 \vdash E_1\colon \mathsf{N}}{\Gamma_0 \cup \Gamma_1 \vdash (\mathsf{down}\ E_0\ E_1)\colon \mathsf{N}} \qquad \to\textit{-I:}\ \frac{\Gamma,\, v\colon\sigma \vdash E\colon\tau}{\Gamma \vdash (\lambda v\,.\,E)\colon \sigma \to \tau}$$

$$\to\textit{-E:}\ \frac{\Gamma_0 \vdash E_0\colon \sigma \to \tau \quad \Gamma_1 \vdash E_1\colon \sigma}{\Gamma_0 \cup \Gamma_1 \vdash (E_0\ E_0)\colon \tau} \qquad \mathsf{fix}\textit{-I:}\ \frac{\Gamma \vdash (\lambda f\,.\,E)\colon \sigma \to \sigma}{\Gamma \vdash (\mathsf{fix}\ (\lambda f\,.\,E))\colon \sigma}$$

$$\textit{If-I:}\ \frac{\Gamma_0 \vdash E_0\colon \mathsf{N} \quad \Gamma_1 \vdash E_1\colon \mathsf{N} \quad \Gamma_2 \vdash E_2\colon \mathsf{N}}{\Gamma_0 \cup \Gamma_1 \cup \Gamma_2 \vdash (\mathsf{if}\ E_0\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2)\colon \mathsf{N}}$$

**Figure 2.** PCF typing rules

$$\begin{array}{c} (\mathsf{c}_{\mathbf{a}}\ x) \longrightarrow \mathbf{a} \oplus x. \qquad (\mathsf{d}\ (\mathbf{a} \oplus x)) \longrightarrow x. \qquad (\mathsf{d}\ \epsilon) \longrightarrow \epsilon. \\ (\mathsf{t}_{\mathbf{a}}\ x) \longrightarrow [\mathbf{1},\ \text{if } x \in \mathbf{a}\{\mathbf{0}, \mathbf{1}\}^*; \quad \epsilon,\ \text{otherwise}] \\ (\mathsf{down}\ x\ y) \longrightarrow [x,\ \text{if } |x| \leq |y|; \quad \epsilon,\ \text{otherwise}] \\ (\mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z) \longrightarrow [y,\ \text{if } x \neq \epsilon; \quad z,\ \text{otherwise}] \\ \mathsf{fix}\ (\lambda f\,.\,E) \longrightarrow E[f \leftarrow (\mathsf{fix}\ (\lambda f\,.\,E))]. \end{array}$$

**Figure 3.** Some PCF reduction rules

## 2. Background definitions

***Strings and tallies*** Each element of $\mathbb{N}$ is identified with its **0**-**1**-dyadic representation, i.e., $0 \equiv \epsilon,\ 1 \equiv \mathbf{0},\ 2 \equiv \mathbf{1},\ 3 \equiv \mathbf{00}$, etc. Each element of $\omega$ is identified with its **0**-unary representation, i.e., $0 \equiv \epsilon,\ 1 \equiv \mathbf{0},\ 2 \equiv \mathbf{00},\ 3 \equiv \mathbf{000}$, etc. $\mathbb{N}$-values are numeric/string values to be computed over; $\omega$-values are tallies of lengths and run times. *Notation:* For each natural number $k$, $\underline{k} = \mathbf{0}^k$.

***Types*** Below, $\mathbf{b}$ ranges over base types and $B$ ranges over nonempty sets of base types. We use fairly standard notation and terminology for the *simple types* ($T ::= B \mid T \to T$) and the *simple product types* ($T ::= B \mid T \to T \mid () \mid T \times T$) over a set of base types $B$. In particular, we abbreviate $(\sigma_1 \to (\sigma_2 \to (\cdots \to (\sigma_n \to \tau)\ldots))) = \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n \to \tau$ by $(\sigma_1, \ldots, \sigma_n) \to \tau$ and define the *level* of a type by: $\mathit{level}(\mathbf{b}) = \mathit{level}(())\ = 0$; $\mathit{level}(\sigma \times \tau) = \max(\mathit{level}(\sigma), \mathit{level}(\tau))$; and $\mathit{level}(\sigma \to \tau) = \max(1 + \mathit{level}(\sigma), \mathit{level}(\tau))$.

***Type contexts*** A *type context* is a finite mapping of variables to types; these are usually written as a list: $v_1\colon\sigma_1, \ldots, v_k\colon\sigma_k$. $\Gamma, \Gamma'$ denotes the union of two type contexts with disjoint preimages and $\Gamma \cup \Gamma'$ denotes the union of two consistent type contexts.

***Semantic conventions*** For a semantics $\mathcal{S}$ for a formalism $\mathcal{F}$, $\mathcal{S}[\![\cdot]\!]$ is the *semantic map* that takes an $\mathcal{F}$-syntactic object to its $\mathcal{S}$-meaning. $\mathcal{S}[\![\tau]\!]$ is the collection of things named by type $\tau$ under $\mathcal{S}$. For a type context $\Gamma = x_1\colon\tau_1, \ldots, x_n\colon\tau_n$, $\mathcal{S}[\![\Gamma]\!]$ is the set of all finite maps $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$, where $a_1 \in \mathcal{S}[\![\tau_1]\!], \ldots, a_n \in \mathcal{S}[\![\tau_n]\!]$; i.e., *environments*. (*Convention:* $\rho$, with or without decorations, ranges over environments.) $\mathcal{S}[\![\Gamma \vdash E\colon\tau]\!]$ is the map from $\mathcal{S}[\![\Gamma]\!]$ to $\mathcal{S}[\![\tau]\!]$ such that $\mathcal{S}[\![\Gamma \vdash E\colon\tau]\!]\,\rho$ is the element of $\mathcal{S}[\![\tau]\!]$ that is the $\mathcal{S}$-meaning of $E$ when $E$'s free-variables have the meanings given by $\rho$. $\mathcal{S}[\![E]\!]$ is typically written for $\mathcal{S}[\![\Gamma \vdash E\colon\tau]\!]$ as the type judgment is usually understood from context.

***Call-by-value PCF*** Figure 1 gives the syntax of our version of PCF. Figure 2 provides the typing rules in which $\mathsf{op} \in \{\mathsf{c_0}, \mathsf{c_1}, \mathsf{d},$

$P ::= K \mid (\odot\ P\ P) \mid V \mid (P\ P) \mid (\lambda V\,.\,P)$

$K ::= \mathbf{0}^* \qquad T ::=$ the level 0, 1, and 2 simple types over $\mathsf{T}$

**Figure 4.** The syntax for second-order polys, where $\odot = \vee, +, *$

$$\textit{Const-I:}\ \frac{}{\Sigma \vdash K\colon \mathsf{T}} \qquad \odot\textit{-I:}\ \frac{\Sigma \vdash P_0\colon \mathsf{T} \quad \Sigma \vdash P_1\colon \mathsf{T}}{\Sigma \vdash (\odot\ P_0\ P_1)\colon \mathsf{T}}\ \ (\odot = \vee, +, *)$$

**Figure 5.** The additional typing rules for second-order polys

$\mathtt{t_0}, \mathtt{t_1}\}$. For emphasis we may write $\lambda v\colon\sigma\,.\,E$ instead of $\lambda v\,.\,E$. $\mathsf{N}$ is interpreted as $\{\mathbf{0}, \mathbf{1}\}^*$. The reduction rules are the standard ones for call-by-value PCF [22, 23] plus the rules of Figure 3. In tests, "$x \neq \epsilon$" is syntactic sugar for "$x$" and "$|E_0| \leq |E_1|$" is syntactic sugar for "(down $\mathsf{c_0}(E_0)$ $\mathsf{c_0}(E_1)$)." A CEK-machine [6] (omitted in this version of the paper) provides an operational semantics. We take $\mathcal{V}$ (for *value*) to be a conventional denotational semantics for PCF [30]. It is standard that these two semantics agree.

***The total continuous functionals*** Let $\sigma$ and $\tau$ be simple product types over base type $\mathsf{N}$. The $\mathbf{TC}_\sigma$ sets are inductively defined by: $\mathbf{TC}_\mathsf{N} = \mathbb{N}$, $\mathbf{TC}_() = \star$, $\mathbf{TC}_{\sigma\times\tau} = \mathbf{TC}_\sigma \times \mathbf{TC}_\tau$, and $\mathbf{TC}_{\sigma\to\tau}$ = the *Kleene-Kreisel total continuous functions* [20] from $\mathbf{TC}_\sigma$ to $\mathbf{TC}_\tau$. This paper is concerned with only the type-level 0, 1, and 2 portions of $\mathbf{TC}$ from which we construct our models for $\mathsf{ATR}_1$.

Let $\sigma$ and $\tau$ be simple product types over base type $\mathsf{T}$ (for *tally*). The $\mathbf{MC}_\sigma$ sets and partial orders $\leq_\sigma$ are inductively defined by: $\mathbf{MC}_\mathsf{T} = \omega$ and $\leq_\mathsf{T}$ = the usual ordering on $\omega$; $\mathbf{MC}_() = \star$ and $\star \leq_() \star$; $\mathbf{MC}_{\sigma\times\tau} = \mathbf{MC}_\sigma \times \mathbf{MC}_\tau$ and $(a,b) \leq_{\sigma\times\tau} (a',b') \iff a \leq_\sigma a'$ and $b \leq_\tau b'$; and $\mathbf{MC}_{\sigma\to\tau}$ = the Kleene-Kreisel total continuous functions from $\mathbf{MC}_\sigma$ to $\mathbf{MC}_\tau$ that are monotone (w.r.t. $\leq_\sigma$ and $\leq_\tau$), and $\leq_{\sigma\to\tau}$ is the point-wise ordering on $\mathbf{MC}_{\sigma\to\tau}$. (E.g., $\mathbf{MC}_{\mathsf{T}\to\mathsf{T}} = \{\, f\colon\omega\to\omega \mid f(0) \leq f(1) \leq f(2) \leq \cdots \,\}$.) As with $\mathbf{TC}$, our concern is with only the type-level 0, 1, and 2 portions of $\mathbf{MC}$ from which we construct our models of size and time bounds.

***Lengths*** For $x \in \mathbb{N}$, let $|x| = \underline{x}$, where $k$ is the length of $x$'s dyadic representation (e.g., $|\mathbf{011}| = \underline{3} = \mathbf{000}$). Following Kapron and Cook [15], for $f \in \mathbf{TC}_{(\mathsf{N}^k)\to\mathsf{N}}$, define $|f| \in \mathbf{MC}_{(\mathsf{T}^k)\to\mathsf{T}}$ by:

$$|f|(\vec{\ell}) \;=\; \max\{\, |f(\vec{x})| \mid |x_i| \leq \ell_i \text{ for } i = 1,\ldots,k \,\}. \qquad (2)$$

For each $\sigma$, a simple type over $\mathsf{N}$, let $|\sigma| = \sigma[\mathsf{N} \leftarrow \mathsf{T}]$ (e.g., $|\mathsf{N}\to\mathsf{N}| = \mathsf{T}\to\mathsf{T}$). Note that if $x \in \mathbf{TC}_\sigma$ and $\gamma$ is level 0 or 1, then $|x| \in \mathbf{MC}_{|\sigma|}$. For $\gamma = (\sigma_1,\ldots,\sigma_k)\to\mathsf{N}$ of level-2, $F \in \mathbf{TC}_\gamma$, and $\ell_1 \in \mathbf{MC}_{|\sigma_1|}, \ldots, \ell_k \in \mathbf{MC}_{|\sigma_k|}$, define

$$|F|(\vec{\ell}) = \max\{\, |F(\vec{x})| \mid |x_i| \leq_{|\sigma_i|} \ell_i \text{ for } i = 1,\ldots,k \,\}. \qquad (3)$$

$|F|$ as defined above turns out to be an element of $\mathbf{MC}_{|\gamma|}$.

***Maximums and polynomials*** Let $x_1 \vee x_2 = \max(\{x_1, x_2\})$ and $\bigvee_{i=1}^k x_i = \max(\{x_1,\ldots,x_k\})$ for $x_1,\ldots,x_k \in \omega$. By convention, $\max(\emptyset) = 0$. We allow $\vee$ as another arithmetic operation in polynomials; $\vee$ binds closer than either multiplication or addition. Coefficients in polynomials will always be nonnegative; hence polynomials denote monotone nondecreasing functions.

***The second-order polynomials*** We define *second-order polynomials* [15] as a type-level 2 fragment of the simply typed $\lambda$-calculus over base type $\mathsf{T}$ with arithmetic operations $\vee$, $+$, and $*$. Figure 4 gives the syntax. (We often write $\vee$-, $+$-, and $*$-expressions in infix form.) The typing rules are *Id-I*, $\to$*-E*, and $\to$*-I* from Figure 2 plus the rules in Figure 5. The $\mathcal{L}$-semantics for second-order polynomials is as follows. For $\sigma$, a simple type over $\mathsf{T}$, take $\mathcal{L}[\![\sigma]\!] = \mathbf{MC}_\sigma$. $\mathcal{L}[\![\Sigma \vdash p\colon\sigma]\!]$ is defined in the standard way. **Important:** The *depth* a second-order polynomial $q$ is the maximal depth of nesting of applications in $q$'s $\beta$-normal form, e.g., $g_0((g_0(2*y*g_1(y^2))\vee 6)^3)$ has depth 3. For second-order polynomials, depth plays something like the role degree does for ordinary polynomials.

***Time complexity*** The CEK machine, mentioned above, provides an operational semantics for both PCF and $\mathsf{ATR}_1$. As our concern is with the costs of evaluating expressions, we use the CEK machine as our standard model of computation and use a simple cost model to the CEK as the standard notion of time complexity.[5] Our CEK machine handles oracles (type-level 1 functions over $\mathbb{N}$) as the values of particular variables in the initial environment for an evaluation. As with Kapron and Cook's answer-length cost model for oracle Turing machines [15], part of the CEK-cost of querying an oracle includes the length of the answer.

***The basic feasible functionals*** Suppose $\tau = (\sigma_1,\ldots,\sigma_k)\to\mathsf{N}$ is a simple type over $\mathsf{N}$ of level 1 or 2. We say that $f \in \mathcal{V}[\![\tau]\!]$ is *basic feasible* when there is a closed type-$\tau$, PCF-expression $E_f$ and a second-order polynomial $q_f$ such that $\mathcal{V}[\![E_f]\!] = f$ and, for all $v_i \in \mathcal{V}[\![\sigma_1]\!], \ldots, v_k \in \mathcal{V}[\![\sigma_k]\!]$, CEK-time$(E_f, v_1, \ldots, v_k) \leq q_f(|v_1|,\ldots,|v_k|)$. (Kapron and Cook's machine-based characterization [15] of the type-2 BFFs is the basis of the above definition. For level-1 $\tau$'s, the above yields ordinary poly-time computability.)

## 3. Affine tail recursion

### 3.1 Ingredients

***Ramified base types*** Bellantoni and Cook's [2] well-known poly-time programming formalism features a recursion construct along the lines of prn of (1) *without a side condition*. They avoided the side condition by separating the roles of $\mathbb{N}$-values into *normal* and *safe* values. Roughly, a normal value can be used to drive a recursion, but cannot be the result of a recursion, whereas a safe value can be the result of a recursion, but cannot be used to drive a recursion.[6] The separation of roles allows enough power to compute poly-time, but forbids "bad feedbacks" through which the unconstrained prn can produce non-poly-time results.

At type-level 2 there are new "bad feedbacks" to be avoided, e.g., $F_0 = \lambda f \in \mathbb{N}\to\mathbb{N},\ x \in \mathbb{N}\,.\,(f^{(|x|)}(x))$ fails to be a BFF as $|F_0(\lambda w \in \mathbb{N}\,.\,(w \oplus w), x)| = |x| \cdot 2^{|x|}$. Note however that both $F_1 = \lambda f \in \mathbb{N}\to\mathbb{N},\ x \in \mathbb{N}\,.\,f(f(x))$ and $F_2 = \lambda f \in \mathbb{N}\to\mathbb{N},\ x \in \mathbb{N}\,.\,[\,g^{(|x|)}(x),\ \text{where } g(w) = f(w) \bmod (x+1)\,]$ are BFFs. So, any restrictions that prohibit $F_0$'s definition must also leave enough room for $F_1$'s and $F_2$'s definitions. To deal with these problems we use the facts that each expression in our language will need to have a second-order polynomial size bound and that each second-order polynomial has a specific depth, e.g., $f(f(x))$ has the size bound $|f|(|f|(|x|))$ which has depth 2. We shall have depth-$d$ versions of normal and safe base types, for $d = 0, 1, \ldots$, with the subtype ordering: depth-0-normal $\leq$: depth-0-safe $\leq$: depth-1-normal $\leq$: depth-1-safe $\leq$: $\cdots$. An expression $E$ being of a depth-$d$ type will turn out to mean that $E$ has a depth-$d$ second-order polynomial size bound. Moreover, if $E$ is of the depth-$d$-normal type, then $E$ will turn out to an input value (or else size bounded by some such value), where a depth-0 input is just an ordinary string input and a depth-$(d+1)$ is the answer returned by a type-1 input $f$ when

[5] Under the CEK cost model: (i) operations that involve the entirety of string $x$ (in reading, writing, or testing $x$) have cost $\underline{1} \vee |x|$, and (ii) all other operations have unit cost. The standard model of computation underlying this CEK machine is Schönhage's *storage modification machine* [27], which is known to be polynomially related to standard Turing machines.

[6] Bellantoni and Cook did not use typing per se, but most of the followup work has recast their ideas in terms of type systems.

queried on depth-$d$-safe values. If $E$ is of the depth-$d$-safe type, then its value is the result of (type-2) poly-time computation over depth-$d$-normal terms (or else size bounded by some such value).

***Clocked, linear, tail-recursions*** In place of fix, $\mathsf{ATR}_1$ has the combinator crec (for *clocked recursion*) with the reduction rule:

$$\mathsf{crec}\ c\ (\lambda_r f\,.\,E) \longrightarrow \lambda\vec{v}\,.\,\big(\mathsf{if}\ |c| \leq |v_1|\ \mathsf{then}\ \big(E'\ \vec{v}\big)\ \mathsf{else}\ \epsilon\big)$$
$$\text{with } E' = E[f \leftarrow \big(\mathsf{crec}\ \ (\mathbf{0} \oplus c)\ (\lambda_r f\,.\,E)\big)],$$

where $c$ is a constant and $\vec{v} = v_1, \ldots, v_k$ is a sequence of variables. Roughly, $|c|$ acts as the tally of the number of recursions thus far and $\mathbf{0} \oplus c$ is the result of a tick of the clock. The value of $v_1$ is program's estimate of the total number of recursions it needs to do its job. Typing constraints will make sure that each crec-recursion recursion terminates after polynomially many steps. Without these constraints, crec is essentially equivalent to fix.[7]

Besides being clocked, recursions have two other restrictions.

ONE USE. In any expression $(\mathsf{crec}\ a\ (\lambda_r f\,.\,E))$, we require that $f$ has at most one *use* in $E$. Operationally this means that, in any possible evaluation of $E$, at most one application of $f$ takes place. One consequence of this restriction is that no free occurrence of $f$ is allowed within any inner crec expression. (Even if $f$ *occurs* but once in an inner crec, the presumption is that $f$ may be *used* many times.) Affine typing constraints enforce this one-use restriction.

Under the one-use restriction, bounds on the cost of $m$ steps of a crec recursion are provided by recurrences of the form $T(m, \vec{n}) \leq T(m-1, \vec{n}) + q(\vec{n})$, where $\vec{n}$ represents the other parameters and $q$ is a (second-order) polynomial. Such $T$'s grow polynomially in $m$. Thus, a polynomial bound on the depth of a crec recursion implies a polynomial bound on the recursion's total cost.[8]

TAIL RECURSIONS. We restrict crec terms to expressing just tail recursions. Primarily, this is just a simplifying restriction; but secondarily, we focus on tail recursions because almost all of the implicit complexity literature has focused on primitive recursions.

### 3.2 $\mathsf{ATR}_1$ syntax, types, and typing

***Syntax*** $\mathsf{ATR}_1$ (for *affine tail recursion*) has the same expressions as PCF with two changes: fix is replaced with crec as discussed above and the only variables allowed are those of orders 0 and 1.

***Types*** The $\mathsf{ATR}_1$ types consist of *labeled base types* ($T_0$ from Figure 6) and the order 1 and 2 simple types over these base types. We first consider labels ($L$ from Figure 6).

LABELS. Labels are strings of alternating $\diamond$'s and $\Box$'s in which the rightmost symbol of a nonempty label is always $\diamond$. A label $\mathbf{a}_k \ldots \mathbf{a}_0$ can be thought of as describing program-oracle interactions: each symbol $\mathbf{a}_i$ represents an action ($\Box$ = an oracle action, $\diamond$ = a program action) with the ordering in time being $\mathbf{a}_0$ through $\mathbf{a}_k$. *Terminology:* $\varepsilon$ = the empty label, $L \leq L'$ means label $L$ is a suffix of label $L'$, and $L \vee L'$ is the $\leq$-maximum of $L$ and $L'$. Also let *succ*$(L)$ = the successor of $L$ in the $\leq$-ordering, *depth*$(L)$ = the number of $\Box$'s in $L$, and, for each $d \in \omega$, $\Box_d = (\Box\diamond)^d$ and $\diamond_d = \diamond(\Box\diamond)^d$. Note: *depth*$(\Box_d)$ = *depth*$(\diamond_d) = d$.

LABELED BASE TYPES. The $\mathsf{ATR}_1$ base types are all of the form $\mathsf{N}_L$, where $L$ is a label. We subtype-order these types by: $\mathsf{N}_L \leq: \mathsf{N}_{L'} \iff L \leq L'$, and thus have the linear ordering:

[7] Clocking the fixed point process is a strong restriction. However, there are results ([25, Chapter 4]) that suggest that clocking, whether explicit or implicit, is needed to produce programs for which one can effectively determine explicit run-time bounds.

[8] Were two uses allowed, the recurrences would be of the form $T(m, \vec{n}) \leq 2 \cdot T(m-1, \vec{n}) + q(\vec{n})$ and such $T$'s can grow exponentially in $m$.

$$E \ ::= \ \ldots \mid (\mathsf{crec}\ K\ (\lambda_r V\,.\,E)) \qquad L \ ::= \ (\Box\diamond)^* \mid \diamond(\Box\diamond)^*$$
$$T_0 \ ::= \ \mathsf{N}_L \qquad T \ ::= \ \text{the order 0, 1, and 2 simple types over } T_0$$

**Figure 6.** $\mathsf{ATR}_1$ syntax

$$\textit{Zero-I:}\ \frac{}{\Gamma;\Delta \vdash \epsilon\colon \mathsf{N}_\varepsilon} \qquad \textit{Const-I:}\ \frac{}{\Gamma;\Delta \vdash K\colon \mathsf{N}_\diamond} \qquad \textit{Aff-Id-I:}\ \frac{}{\Gamma; v\colon\gamma \vdash v\colon\gamma}$$

$$\textit{Int-Id-I:}\ \frac{}{\Gamma,\ v\colon\sigma;\Delta \vdash v\colon\sigma} \qquad \mathsf{op}\textit{-I:}\ \frac{\Gamma;\Delta \vdash E\colon \mathsf{N}_{\diamond_d}}{\Gamma;\Delta \vdash (\mathsf{op}\ E)\colon \mathsf{N}_{\diamond_d}}$$

$$\textit{Shift:}\ \frac{\Gamma;\Delta \vdash E\colon\sigma}{\Gamma;\Delta \vdash E\colon\tau}\ (\sigma \propto \tau) \qquad \textit{Subsumption:}\ \frac{\Gamma;\Delta \vdash E\colon\sigma}{\Gamma;\Delta \vdash E\colon\tau}\ (\sigma \leq\colon \tau)$$

$$\mathsf{down}\textit{-I:}\ \frac{\Gamma;\Delta_0 \vdash E_0\colon \mathsf{N}_{L_0} \quad \Gamma;\Delta_1 \vdash E_1\colon \mathsf{N}_{L_1}}{\Gamma;\Delta_0,\Delta_1 \vdash (\mathsf{down}\ E_0\ E_1)\colon \mathsf{N}_{L_1}}$$

$$\to\textit{-I:}\ \frac{\Gamma,\ v\colon\sigma;\Delta \vdash E\colon\tau}{\Gamma;\Delta \vdash (\lambda v\,.\,E)\colon\sigma\to\tau} \qquad \to\textit{-E:}\ \frac{\Gamma;\Delta \vdash E_0\colon\sigma\to\tau\ \ \Gamma;\_ \vdash E_1\colon\sigma}{\Gamma;\Delta \vdash (E_0\ E_1)\colon\tau}$$

$$\mathsf{if}\textit{-I:}\ \frac{\Gamma;\_ \vdash E_0\colon \mathsf{N}_L \quad \Gamma;\Delta_1 \vdash E_1\colon \mathsf{N}_{L'} \quad \Gamma;\Delta_2 \vdash E_2\colon \mathsf{N}_{L'}}{\Gamma;\Delta_1 \cup \Delta_2 \vdash (\mathsf{if}\ \ E_0\ \ \mathsf{then}\ \ E_1\ \ \mathsf{else}\ \ E_2)\colon \mathsf{N}_{L'}}$$

$$\mathsf{crec}\textit{–I:}\ \frac{\vdash K\colon \mathsf{N}_\diamond \quad \Gamma; f\colon\gamma \vdash E\colon\gamma}{\Gamma;\_ \vdash (\mathsf{crec}\ K\ (\lambda_r f\,.\,E))\colon\gamma}\ \big(\gamma \in \mathcal{R} \text{ and } \textit{TailPos}(f,E)\big)$$

where:

$$\mathcal{R} \stackrel{\text{def}}{=} \left\{ (\mathsf{N}_{\Box_d}, \mathbf{b}_2, \ldots, \mathbf{b}_k) \to \mathbf{b} \ \middle|\ \begin{array}{l}\text{if } \mathbf{b}_i \leq\colon \mathsf{N}_{\Box_d}\text{, then}\\ \mathbf{b}_i \text{ is also oracular}\end{array} \right\}.$$

$$\textit{TailPos}(f,E) \stackrel{\text{def}}{=} \left[\begin{array}{l}\text{Each occurrence of } f \text{ in } E\\ \text{is as the head of a tail call}\end{array}\right].$$

**Figure 7.** $\mathsf{ATR}_1$ typing rules

```
reverse: N_ε → N_◇ =                    // reverse a_1 ... a_k = a_k ... a_1.
  λw . letrec f: N_ε → N_◇ → N_◇ =
         λb, x, r . if (t_0 x)      then f b (d x) (c_0 r)
                    else if (t_1 x) then f b (d x) (c_1 r)
                    else r
       in f w w

prn: (N_◇ → N_◇ → N_◇) → N_ε → N_◇ =    // prn e y = (See (1).)
  λe, y . letrec f: N_ε → N_◇ → N_◇ → N_◇ → N_◇ =
             λb, x, z, r .
               if (t_0 x)      then f b (d x) (c_0 z) (e (c_0 z) r)
               else if (t_1 x) then f b (d x) (c_1 z) (e (c_1 z) r)
               else r
           in f y (reverse y) ε (e ε ε)

cat: N_ε → N_◇ → N_◇ =                  // cat w x = w ⊕ x.
  λw, x . let f: N_◇ → N_◇ → N_◇ =
            λy, z . if (t_0 y) then (c_0 z)
                    else if (t_1 y) then (c_1 z)
                    else x
          in prn f w

// fcat f a_1 ... a_k = (f a_1 ... a_k) ⊕ (f a_2 ... a_k) ⊕ ··· ⊕ (f ε).
fcat: (N_◇ → N_□◇) → N_ε → N_◇□◇ =
  λf, x . let e: N_◇ → N_◇□◇ → N_◇□◇ = λy, r . (cat (f y) r)
          in prn e x

findk: (N_◇ → N_□◇) → N_◇ → N_◇ =       // findk f x = (See (5).)
  λf, x . letrec h: N_□◇ → N_◇ → N_◇ =
             // Invariant: k ≤ len(m) and |m| ≤ |f|(|x|)
             λm, k . if k == x then k
                     else if k == (len m) then k
                     else h (max (f (k+1)) m) (k+1)
          in h (f ε) ε
```

where the above uses the syntactic sugar:

$$(\mathsf{let}\ x = D\ \mathsf{in}\ E) \stackrel{\text{def}}{=} E[x \leftarrow D].$$
$$(\mathsf{letrec}\ \ f = D\ \ \mathsf{in}\ \ E) \stackrel{\text{def}}{=} E[f \leftarrow (\mathsf{crec}\ \underline{0}\ (\lambda_r f\,.\,D))]$$

**Figure 8.** $\mathsf{ATR}_1$ versions of *reverse*, *prn*, *cat*, *fcat*, and *findk*

$\mathsf{N}_\varepsilon \leq: \mathsf{N}_\diamond \leq: \mathsf{N}_{\Box\diamond} \leq: \mathsf{N}_{\diamond\Box\diamond} \leq: \cdots$, or equivalently, $\mathsf{N}_{\Box_0} \leq: \mathsf{N}_{\diamond_0} \leq: \mathsf{N}_{\Box_1} \leq: \mathsf{N}_{\diamond_1} \leq: \cdots$. Define $\mathit{depth}(\mathsf{N}_L) = \mathit{depth}(L)$, $\mathit{side}(\mathsf{N}_{\Box_d}) = \Box$, and $\mathit{side}(\mathsf{N}_{\diamond_d}) = \diamond$. $\mathsf{N}_{\Box_d}$ is the depth-$d$-normal type discussed above and $\mathsf{N}_{\diamond_d}$ is the depth-$d$-safe type.

THE $\mathsf{ATR}_1$ ARROW TYPES. These are just the order 1 and 2 simple types over the $\mathsf{N}_L$'s. We extend $\leq:$ to the arrow types via:

$$\sigma_1 \leq: \sigma_0 \;\;\&\;\; \tau_0 \leq: \tau_1 \quad\Longleftrightarrow\quad \sigma_0 \to \tau_0 \leq: \sigma_1 \to \tau_1. \qquad (4)$$

*Terminology:* We write $\tau :\geq \sigma$ for $\sigma \leq: \tau$ and $\sigma \lneq: \tau$ for $[\,\sigma \leq: \tau$ and $\sigma \neq \tau]$. Let $\mathit{shape}(\sigma) =$ the simple type over $\mathsf{N}$ resulting from erasing all the labels. The *tail* of a type is given by:

$$\mathit{tail}(\mathsf{N}_L) \;=\; \mathsf{N}_L. \qquad \mathit{tail}(\sigma \to \tau) \;=\; \mathit{tail}(\tau).$$

Let $\mathit{depth}(\sigma) = \mathit{depth}(\mathit{tail}(\sigma))$ and $\mathit{side}(\sigma) = \mathit{side}(\mathit{tail}(\sigma))$. When $\mathit{side}(\sigma) = \Box$, we call $\sigma$ an *oracular* type, and when $\mathit{side}(\sigma) = \diamond$ we call $\sigma$ a *computational* type.

DEFINITION 1 (Predicative, impredicative, flat, and strict types). An $\mathsf{ATR}_1$ type $\gamma$ is *predicative* when $\gamma$ is a base type or else $\gamma = \sigma \to \tau$ with $\tau$ predicative and $\mathit{tail}(\sigma) \leq: \mathit{tail}(\tau)$. A type is *impredicative* when it fails to be predicative. An $\mathsf{ATR}_1$ type $(\sigma_1, \ldots, \sigma_k) \to \mathsf{N}_L$ is *flat* when $\mathit{tail}(\sigma_i) = \mathsf{N}_L$ for some $i$. A type is *strict* when it fails to be flat.

*Examples:* $\mathsf{N}_\varepsilon \to \mathsf{N}_\diamond$ is predicative whereas $\mathsf{N}_\diamond \to \mathsf{N}_\varepsilon$ is impredicative, and both are strict. Both $\mathsf{N}_\diamond \to \mathsf{N}_\diamond$ and $\mathsf{N}_\diamond \to \mathsf{N}_{\Box\diamond} \to \mathsf{N}_\diamond$ are flat, but the first is predicative and the second impredicative. Recursive definitions tend to involve flat types.

Example 11 below illustrates that values of both impredicative and flat types require special semantic restrictions. These restrictions will be detailed in §4.3 and §4.4 below. Here we give a quick sketch of these restrictions as they figure in definition of $\propto$, the *shifts-to* relation, used in the typing rules. *For each impredicative type* $(\vec{\sigma}) \to \mathsf{N}_L$: if $f\colon (\vec{\sigma}) \to \mathsf{N}_L$, then the value of $|f(\vec{x})|$ is essentially independent of the values of the $|x_i|$'s with $\mathit{tail}(\sigma_i) :\gtrsim \mathsf{N}_L$. *For each flat type* $(\vec{\sigma}) \to \mathsf{N}_L$ (that for simplicity here we further restrict to be a type-level-1 computational type): if $\vdash f\colon (\vec{\sigma}) \to \mathsf{N}_L$, then $|f(\vec{x})| \leq p + \bigvee\{\, |x_i| \;:\; \mathit{tail}(\sigma_i) = \mathsf{N}_L \,\}$, where $p$ is a second-order polynomial over elements of $\{\, |x_i| \;:\; \mathit{tail}(\sigma_i) \lneq: \mathsf{N}_L \,\}$.

***Typing rules*** The $\mathsf{ATR}_1$-typing rules are given in Figure 7. The rules *Zero-I*, *Const-I*, *Int-Id-I*, *Subsumption*, op-*I*, $\to$-*I*, and $\to$-*E* are fairly standard (with one subtlety in $\to$-*E* discussed below). The if-*I* and down-*I* rules are designed to allow definitions for functions like the $F_2$ example of §3.1.[9] The remaining three rules *Shift* (that coerces types) and *Aff-Id-I* and crec-*I* (that relate to recursions and the split type contexts) require some discussion.

SHIFT. The *Shift* rule covariantly coerces the type of a term to be deeper. For types of shape $\mathsf{N} \to \mathsf{N}$, the core idea in the definition of the shifts-to relation ($\propto$) is: $(\mathsf{N}_{L_1} \to \mathsf{N}_{L_0}) \propto (\mathsf{N}_{L'_1} \to \mathsf{N}_{L'_0})$ when $\mathit{depth}(\mathsf{N}_{L'_0}) = \mathit{depth}(\mathsf{N}_{L_0}) + \big(\mathit{depth}(\mathsf{N}_{L'_1}) - \mathit{depth}(\mathsf{N}_{L_1})\big)$. The motivation for this is that if $p$ and $q$ are second-order polynomials of depths $d_p$ and $d_q$, respectively, and $x$ is a base-type variable appearing in $p$ that is treated as representing a depth-$d_x$ value (with $d_x \leq d_q$), then $p[x \leftarrow q]$ is of depth $\leq d_p + (d_q - d_x)$. The full story for $\propto$ has to account of the sides ($\Box$ and $\diamond$) of component types, impredicative and flat types, and more. In the interest of space, we omit the fussy details of the definition of $\propto$ in this version of the paper. Here is a key example of how the Shift rule is used. Consider the problem: $\Gamma;\_ \vdash f(f(x))\colon ?$, where $\Gamma = f\colon \mathsf{N}_\diamond \to \mathsf{N}_{\Box\diamond},\ x\colon \mathsf{N}_\diamond$. Using $\to$-*E* and *Subsumption*, we derive $\Gamma;\_ \vdash f(x)\colon \mathsf{N}_{\diamond\Box\diamond}$. Using *Shift* we derive $\Gamma;\_ \vdash f\colon \mathsf{N}_{\diamond\Box\diamond} \to \mathsf{N}_{\Box\diamond\Box\diamond}$. Using $\to$-*E* again we obtain $\Gamma;\_ \vdash f(f(x))\colon \mathsf{N}_{\Box\diamond\Box\diamond}$ as desired.

AFFINELY RESTRICTED VARIABLES AND crec. Each $\mathsf{ATR}_1$ type judgment is of the form $\Gamma;\Delta \vdash E\colon\gamma$ where each type context is separated into two parts: a *intuitionistic zone* ($\Gamma$) and an *affine zone* ($\Delta$). $\Gamma$ and $\Delta$ are simply finite maps (with disjoint preimages) from variables to $\mathsf{ATR}_1$-types. By convention, "$\_$" denotes an empty zone. Also by convention we shall restrict our attention to $\mathsf{ATR}_1$ type judgments in which each affine zone consists of at most one type assignment. In reading the rules of Figure 7, think of a variable in an affine zone as destined to be the recursor variable in some crec expression. An intuitionistic zone assigns types to each of the mundane variables. *Terminology:* A variable $f$ is said to be *affinely restricted* in $\Gamma;\Delta \vdash E\colon\sigma$ when $f$ is assigned a type by $\Delta$ or else is $\lambda_r$-abstracted over in $E$.

The split type contexts is adapted from Barber and Plotkin's DILL [1]. The key appropriation from DILL is $\to$-*E*; it forbids free occurrences of affinely restricted variables in the operand position of any intuitionistic application. This precludes the typing of crec-expressions containing subterms such as $\lambda_r f\,.\,(\lambda g\,.\,(g\,(g\,\epsilon))\,f) \equiv_\beta \lambda_r f\,.\,(f\,(f\,\epsilon))$ where $f$ is used multiple times.

The crec-*I* rule forbids any free occurrence of an affinely restricted variable; if such an occurrence was allowed, it could be used any number of times via the crec-recursion. The crec-*I* rule requires that the recursor variable have a type in $\mathcal{R}$ (as defined in Figure 7). When $\gamma = (\mathsf{N}_{\Box_d}, \mathbf{b}_2, \ldots, \mathbf{b}_k) \to \mathbf{b} \in \mathcal{R}$, it turns out that $\mathcal{R}$'s restrictions limit a type-$\gamma$ crec-expression to at most $p$-many recursions, where $p$ is some fixed, depth-$d$ second-order polynomial. Excluding $\mathsf{N}_\diamond, \ldots, \mathsf{N}_{\diamond_{d-1}}$ in $\gamma$ forbids depth $0, \ldots, d-1$ analogues of safe-values from figuring in the recursion, consequently, the recursion cannot accumulate information that could unboundedly change $p$'s value.

SCHOLIUM 2. $\mathsf{ATR}_1$ has no explicit $\multimap$-types. *Implicitly*, a subexpression $(\lambda_r f\,.\,E)$ is of type $\gamma \multimap \gamma$ and crec-*I* plays that roles of both $\multimap$-*I* and $\multimap$-*E*. $\mathsf{ATR}_1$'s very restricted use of affinity permits this $\multimap$-bypass. Dropping the $\mathit{TailPos}(f, E)$ side condition in crec-*I* fails to add many interesting typable programs because of the empty affine zone restriction in $\to$-*E*. If $\mathsf{ATR}_1$ had explicit $\multimap$-types, then dropping the $\mathit{TailPos}(f, E)$ becomes more interesting.

***Some examples*** Figure 8 contains five sample programs.[10] For the typing of *fcat*, *cat*'s type is shifted to $\mathsf{N}_{\Box_1} \to \mathsf{N}_{\diamond_1} \to \mathsf{N}_{\diamond_1}$ and *prn*'s type is shifted to $(\mathsf{N}_{\diamond_0} \to \mathsf{N}_{\diamond_1} \to \mathsf{N}_{\diamond_1}) \to \mathsf{N}_{\Box_0} \to \mathsf{N}_{\diamond_1}$. The final program computes

$$\lambda f \in (\mathbb{N} \to \mathbb{N}), x \in \mathbb{N}\,. \qquad (5)$$
$$\begin{cases} (\mu k < x)\,\big[k = \max_{i \leq k} \mathit{len}(f(i))\big], & \text{if such a } k \text{ exists;} \\ x, & \text{otherwise;} \end{cases}$$

where $\mathit{len}(z)$ = the dyadic representation of $z$'s length. This is a surprising and subtle example of a BFF due to Kapron [14] and was a key example that lead to the Kapron-Cook Theorem [15]. In *findk*, we assume we have: a type-$(\mathsf{N}_{\Box_1} \to \mathsf{N}_{\Box_1} \to \mathsf{N}_{\Box_1})$ definition of $(x, y) \mapsto [\underline{1}$, if $x = y$; $\epsilon$, otherwise*]*, a type-$(\mathsf{N}_{\Box_1} \to \mathsf{N}_{\Box_1})$ definition of *len*, a type-$(\mathsf{N}_{\Box_1} \to \mathsf{N}_{\Box_1} \to \mathsf{N}_{\Box_1})$ definition of *max*, and a type-$(\mathsf{N}_{\diamond_0} \to \mathsf{N}_{\diamond_0})$ definition of $x \mapsto x + 1$. It is an easy exercise to fill in these details. A much more challenging exercise is to define (5) via *prn*'s.

[9] E.g.: Suppose $\Gamma = f\colon \mathsf{N}_{\diamond_0} \to \mathsf{N}_{\Box_1},\ x\colon \mathsf{N}_{\diamond_0},\ y\colon \mathsf{N}_{\diamond_0},\ g\colon \mathsf{N}_{\diamond_1} \to \mathsf{N}_{\diamond_1} \to \mathsf{N}_{\diamond_1}$ and that $g$ denotes $\lambda x, y \in \mathbb{N}\,.\,(x \bmod (y+1))$. Then $\Gamma;\_ \vdash g(f(x), y)\colon \mathsf{N}_{\diamond_1}$, but $\Gamma;\_ \vdash \mathsf{down}(g(f(x), y), y)\colon \mathsf{N}_{\diamond_0}$ and this term is equivalent to $f(x) \bmod (y+1)$ since $(x \bmod (y+1)) \leq y$. From this point is it clear how one can construct a type $(\mathsf{N}_{\diamond_0} \to \mathsf{N}_{\Box_1}) \to \mathsf{N}_{\diamond_0} \to \mathsf{N}_{\diamond_0}$ definition of $F_2$. The mild bit of impreditivity in if-*I* and down-*I* gets around much of the usual irritations of ramified type systems.

[10] For the sake of readability, we use the let and letrec constructs as syntactic sugar, where in this paper $(\mathsf{let}\ x = D\ \mathsf{in}\ E) \;\equiv\; E[x \leftarrow D]$ and $(\mathsf{letrec}\ f = D\ \mathsf{in}\ E) \;\equiv\; E[f \leftarrow (\mathsf{crec}\ \underline{0}\ (\lambda_r f\,.\,D))]$.

$$\textit{Zero-I:}\ \frac{}{\Sigma \vdash \underline{0}\colon \mathsf{T}_\varepsilon} \qquad \textit{Const-I:}\ \frac{}{\Sigma \vdash \underline{k}\colon \mathsf{T}_\diamond}$$

$$\textit{Subsumption:}\ \frac{\Sigma \vdash S\colon \sigma}{\Sigma \vdash S\colon \tau}\ (\sigma \leq\colon \tau) \qquad \textit{Shift:}\ \frac{\Sigma \vdash S\colon \sigma}{\Sigma \vdash S\colon \tau}\ (\sigma \propto \tau)$$

$$\vee\textit{-I:}\ \frac{\{\, \Sigma_i \vdash p_i\colon \mathsf{T}_{\diamond_d} \,\}_{i=1,2}}{\Sigma_0 \cup \Sigma_1 \vdash (\vee\ S_0\ S_1)\colon \mathsf{T}_L}$$

$$+\textit{-I:}\ \frac{\{\, \Sigma_i \vdash p_i\colon \mathsf{T}_{\diamond_d} \,\}_{i=1,2}}{\Sigma_1 \cup \Sigma_2 \vdash p_1 + p_2\colon \mathsf{T}_{\diamond_d}} \qquad *\textit{-I:}\ \frac{\{\, \Sigma_i \vdash p_i\colon \mathsf{T}_{\diamond_d} \,\}_{i=1,2}}{\Sigma_1 \cup \Sigma_2 \vdash p_1 * p_2\colon \mathsf{T}_{\diamond_d}}$$

**Figure 9.** The additional size typing rules

***Semantics*** The CEK machine for PCF also provides an operational semantics of $\mathsf{ATR}_1$. For a denotational semantics we *provisionally* take the obvious modification of PCF's $\mathcal{V}$-semantics (introduced in §2). We discuss some serious troubles with this semantics in §4.2.

***Some syntactic properties*** The number of *uses* of variable $v$ in expression $E$ (written: *uses*$(v, E)$) is defined similarly to the count of free occurrences of $v$ in $E$ except that *uses*$(v, (\mathsf{if}\ E_0\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2))$ = *uses*$(v, E_0) +$ *uses*$(v, E_1) \vee$ *uses*$(v, E_2)$ and *uses*$(v, (\mathsf{crec}\ k\ (\lambda_r f\,.\,E_0))) =$ *unbounded* if $v$ occurs free in the crec-expression.

LEMMA 3 (One-use). *For* $\Gamma; f\colon \gamma \vdash E\colon \gamma$ *and for* $\Gamma; \_ \vdash (\mathsf{crec}\ k\ (\lambda_r f\,.\,E))\colon \gamma$, *we have uses*$(f, E) \leq 1$.

LEMMA 4 (Subject reduction). *If* $\Gamma; \Delta \vdash E\colon \gamma$ *and* $E$ $\beta\eta$*-reduces to* $E'$, *then* $\Gamma; \Delta \vdash E'\colon \gamma$.

LEMMA 5. $\Gamma; \Delta \vdash \lambda\vec{x}\,.\,E\colon (\vec{\sigma}) \to \mathsf{N}_L \iff \Gamma, \vec{x}\colon \vec{\sigma}; \Delta \vdash E\colon \mathsf{N}_L$.

Lemma 5 acts as a reality check on $\propto$'s definition.

## 4. Polynomial-size boundedness

The goal of this section is to establish that every $\mathsf{ATR}_1$ expression has a second-order polynomial size bound, where the form of the size bound is determined by the type of the $\mathsf{ATR}_1$ expression (Theorem 24). We show in §4.2 that if this size bound is to be true at all, then $\mathcal{V}$, the naïve denotational semantics for $\mathsf{ATR}_1$ inherited from PCF, must be trimmed.[11] This trimming is done in §4.3 for impredicative types and in §4.4 for flat types. The result of this trimming is $\mathcal{V}_{\mathrm{wt}}$, the *well-tempered semantics* of Definition 20, under which we can prove the polynomial-size boundedness theorem in §4.5. We do not have the evidence to claim that $\mathcal{V}_{\mathrm{wt}}$ is in any way "canonical." However, we strongly suspect that *any* ramified type system that characterizes the type-level 2 basic feasible functionals will have to include restrictions analogous to those for $\mathcal{V}_{\mathrm{wt}}$.

### 4.1 Size bounds

***The size types*** To work with size bounds, we introduce the *size types* and a typing of second-order polynomials under these types. The size types parallel the intuitionistic part of $\mathsf{ATR}_1$'s type system.

DEFINITION 6.

(a) For each $\mathsf{ATR}_1$ type $\sigma$, let $|\sigma| = \sigma[\mathsf{N} \to \mathsf{T}]$. (E.g., $|\mathsf{N}_\epsilon \to \mathsf{N}_\diamond| = \mathsf{T}_\epsilon \to \mathsf{T}_\diamond$.) These $|\sigma|$'s are the *size types*. All the $\mathsf{ATR}_1$-types terminology and operations (e.g., *shape*, *tail*, $\leq\colon$, $\propto$, etc.) are defined analogously for size types.

[11] As noted in footnote 4, this sort of tirmming is not new. Arguably, the simpliest denotational semantics for the simply-typed $\lambda$-calculus over base type $\mathsf{N}$ is $\mathcal{H}$, the full set-theoretic hierarchy over $\{\,\mathbf{0}, \mathbf{1}\,\}^*$. However, for PCF ($\approx$ the simply-typed $\lambda$-calculus + a fixed-point combinator), $\mathcal{H}$ must be trimmed in order for fixed-points to have a definite meaning.

$$\begin{array}{l}
E_1\colon \mathsf{N}_\varepsilon \to \mathsf{N}_\diamond\ = \qquad\qquad \textit{// Assume } g_1\colon \mathsf{N}_\diamond \to \mathsf{N}_\varepsilon.\\
\quad \lambda w\,.\ \mathsf{let}\ h_1\colon \mathsf{N}_\diamond \to \mathsf{N}_\diamond \to \mathsf{N}_\diamond =\\
\qquad\quad \lambda x, y\,.\ \mathsf{if}\ x \neq \epsilon\ \mathsf{then}\ (\mathit{dup}\ (g_1\ y)\ (g_1\ y))\ \mathsf{else}\ w\\
\qquad \mathsf{in}\ \mathit{prn}\ h_1\ w\\
\\
E_2\colon \mathsf{N}_\varepsilon \to \mathsf{N}_\diamond\ = \qquad\qquad \textit{// Assume } g_2\colon \mathsf{N}_\diamond \to \mathsf{N}_\diamond.\\
\quad \lambda w\,.\ \mathsf{let}\ h_2\colon \mathsf{N}_\diamond \to \mathsf{N}_\diamond \to \mathsf{N}_\diamond =\\
\qquad\quad \lambda x, y\,.\ \mathsf{if}\ x \neq \epsilon\ \mathsf{then}\ (g_2\ y)\ \mathsf{else}\ w\\
\qquad \mathsf{in}\ \mathit{prn}\ h_2\ w
\end{array}$$

**Figure 10.** Two problematic programs

(b) The typing rules for the second-order polynomials under the size types consist of *Id-I*, $\to$*-I*, and $\to$*-E* from Figure 2 and the rules of Figure 9.

We *provisionally* take $\mathcal{L}[\![\sigma]\!] = \mathcal{L}[\![\mathit{shape}(\sigma)]\!]$ and $\mathcal{L}[\![\Sigma \vdash p\colon \sigma]\!] =$ as before. Later a pruned version of the $\mathcal{L}$-semantics will end up as our intended semantics for the second-order polynomials to parallel our pruning of the $\mathcal{V}$-semantics for $\mathsf{ATR}_1$.

We note a few basic properties of the second-order polynomials under the size types. In particular, Lemma 8 connects the depth of a second-order polynomial $p$ and the depths of the types assignable to $p$. *Terminology:* Inductively define $\underline{0}_\gamma$ by: $\underline{0}_{\mathsf{T}_L} = \underline{0}$ and $\underline{0}_{\sigma\to\tau} = \lambda x\,.\,\underline{0}_\tau$. We often write $\underline{0}_\gamma$ for $\mathcal{L}[\![\vdash \underline{0}_\gamma\colon \gamma]\!]\rho$.

LEMMA 7 (Subject Reduction). *Suppose* $\Sigma \vdash p\colon \sigma$ *and* $p$ $\beta\eta$*-reduces to* $p'$. *Then* $\Sigma \vdash p'\colon \sigma$.

LEMMA 8 (Label Soundness). *Suppose* $\Sigma \vdash p\colon \sigma$ *has a derivation in which the only types assigned by contexts are from* $\{\, \mathsf{N}_\varepsilon \,\} \cup \{\, (\mathsf{N}_\diamond^k) \to \mathsf{N}_{\Box\diamond} \mid k > 1 \,\}$. *Then depth*$(p) \leq$ *depth*$(\sigma)$.

LEMMA 9. $\underline{0}_\gamma$ *is the* $\leq_\gamma$*-least element of* $\mathcal{L}[\![\gamma]\!]$.

The following definition formalizes what it means for an $\mathsf{ATR}_1$ expression to be polynomially size-bounded. **N.B.** If $v$ is an $\mathsf{ATR}_1$ variable, we treat $|v|$ as a size-expression variable.

DEFINITION 10. Suppose $\sigma$ is a $\mathsf{ATR}_1$ type, $\Gamma; \Delta$ is a $\mathsf{ATR}_1$ type context, and $\rho \in \mathcal{V}[\![\Gamma; \Delta]\!]$.

(a) $|\Gamma; \Delta| \stackrel{\text{def}}{=} \{\, |v| \mapsto |\sigma| \mid (\Gamma; \Delta)(v) = \sigma \,\}$.

(b) Define $|\rho| \in \mathcal{L}[\![|\Gamma; \Delta|]\!]$ by $|\rho|(|v|) = |\rho(v)|$.[12]

(c) Suppose $\Gamma; \Delta \vdash E\colon \sigma$ and $|\Gamma; \Delta| \vdash p\colon |\sigma|$. We say that $p$ *bounds the size of* $E$ (or, $p$ is a *size-bound for* $E$) with respect to $\Gamma; \Delta$ when for all $\rho \in \mathcal{V}[\![\Gamma; \Delta]\!]$, $|\mathcal{V}[\![E]\!]\,\rho| \leq \mathcal{L}[\![p]\!]\,|\rho|$.

### 4.2 Semantic troubles

A naïve (and *false!!*) statement of polynomial-size boundedness for $\mathsf{ATR}_1$ would be: *For each* $\Gamma; \Delta \vdash E\colon \sigma$, *there is a second-order polynomial* $p_E$ *that bounds the size of* $E$ *with respect to* $\Gamma; \Delta$. The following illustrates the problems here.

EXAMPLE 11. *Let* $E_1$ *and* $E_2$ *be as in Figure 10, let* prn *be as in Figure 8, and let* dup *be an* $\mathsf{ATR}_1$*-program such that* $(\mathrm{dup}\ w\ x) = |w|$*-many* $x$*'s concatenated together; so* $|\mathrm{dup}\ w\ x| = |w| * |x|$.

*(a) Suppose* $\Gamma_1 = g_1\colon \mathsf{N}_\diamond \to \mathsf{N}_\varepsilon$ *and* $\rho_1 = \{\, g_1 \mapsto \lambda z \in \mathbb{N}\,.\,z \,\}$. *Then* $|\mathcal{V}[\![E_1]\!]\,\rho_1| = \lambda n \in \omega\,.\,n^{2^n}$. *Note* $|\rho_1(g_1)| = \lambda n \in \omega\,.\,n$ *is a polynomial function. The problem is that* $\rho_1(g_1) = \lambda x \in \mathbb{N}\,.\,x$ *subverts the intent of the type-system by allowing an unrestricted flow of information about "safe" values into "normal" values.*

*(b) Suppose* $\Gamma_2 = g_2\colon \mathsf{N}_\diamond \to \mathsf{N}_\diamond$ *and* $\rho_2 = \{\, g_2 \mapsto \lambda z \in \mathbb{N}\,.\,z \oplus z \,\}$. *Then* $|\mathcal{V}[\![E_2]\!]\,\rho_2| = \lambda n \in \omega\,.\,n \cdot 2^n$. *Note* $|\rho_2(g_2)| =$

[12] **N.B.** The $|\cdot|$ in "$|v|$" is syntactic, whereas the $|\cdot|$ in "$|\rho|$" and "$|\rho(v)|$" are semantic.

*$\lambda n \in \omega \,.\, 2n$ is a polynomial function. The problem is that $\rho_2(g_2) = \lambda y \in \mathbb{N} \,.\, y \oplus y$ subverts the "poly-max restriction" on the sizes of "safe" values discussed in §4.4.*

The problem of Example 11(a) is addressed in §4.3 by pruning the $\mathcal{L}$- and $\mathcal{V}$-semantics to restrict impredicative-type values. The problem of Example 11(b) is addressed in §4.4 by further pruning to restrict flat-type values.

### 4.3 Impredicative types and nearly well-foundedness

Failing to restrict impredicative-type values leads to problems like that of Example 11(a). These problems can be avoided by requiring that each such value have a length that is *nearly well-founded*.

DEFINITION 12. An $\ell \in \mathcal{L}[\![\gamma]\!]$ is *$\gamma$-well-founded* when $\gamma = \mathsf{T}_L$ or else $\gamma = (\sigma_1, \ldots, \sigma_k) \to \mathsf{T}_L$ and, for each $i$ with $tail(\sigma_i) :\gtrsim \mathsf{T}_L$, $\ell$ has no dependence on its $i$-th argument. An $\ell$ is *nearly $\gamma$-well-founded* when there is a $\gamma$-well-founded $\ell'$ such that $\ell \leq \ell'$.

Why *nearly* well-founded? The natural sources of $\mathsf{ATR}_1$-terms with impredicative types are the if-then-else and down constructs. Let $c = \lambda x, y, z \,.\, (\mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z)$ and $d = \lambda x, y \,.\, (\mathsf{down}\, x\, y)$, where $\vdash c\colon (\mathsf{N}_L, \mathsf{N}^2_{L'}) \to \mathsf{N}_{L'}$, $\vdash d\colon (\mathsf{N}_L, \mathsf{N}_{L'}) \to \mathsf{N}_{L'}$, and $L > L'$. Thus $|c| \in \mathcal{L}[\![|(\mathsf{N}_L, \mathsf{N}^2_{L'}) \to \mathsf{N}_{L'}|]\!]$ and $|d| \in \mathcal{L}[\![|(\mathsf{N}_L, \mathsf{N}_{L'}) \to \mathsf{N}_{L'}|]\!]$. Neither $|c|$ nor $|d|$ is well-founded since $|c| = \lambda k, m, n \,.\, [m,\ \text{if } k = \underline{0};\ n, \text{otherwise}]$ and $|d| = \lambda k, m \,.\, \min(k, m)$. However, both $|c|$ and $|d|$ are nearly well-founded as $|c| \leq \lambda k, m, n \,.\, (m \vee n)$ and $|d| \leq \lambda k, m \,.\, m$.

LEMMA 13. *Suppose $\Sigma \vdash p\colon \sigma$, $\rho \in \mathcal{L}[\![\Sigma]\!]$, and $\rho(x)$ is nearly $\Sigma(x)$-well-founded for each $x \in \mathrm{preimage}(\Sigma)$. Then $\mathcal{L}[\![p]\!]\,\rho$ is nearly $\sigma$-well-founded.*

Lemma 13 indicates that a semantics for the second-order polynomials based on nearly well-foundedness will be well defined. *Terminology:* The *restriction* of $f \in (X_1, \ldots, X_k) \to Y$ to $(X'_1, \ldots, X'_k) \to Y$ (where $X'_1 \subseteq X_1, \ldots, X'_k \subseteq X_k$) is $\lambda x_1 \in X'_1, \ldots, x_k \in X'_k \,.\, f(x_1, \ldots, x_k)$.

DEFINITION 14 (The nearly well-founded semantics).

(a) Inductively define $\mathcal{L}_{\mathrm{nwf}}[\![\gamma]\!]$ by: $\mathcal{L}_{\mathrm{nwf}}[\![\mathsf{T}_L]\!] = \omega$. For $\gamma = (\sigma_1, \ldots, \sigma_k) \to \mathsf{T}_L$, $\mathcal{L}_{\mathrm{nwf}}[\![\gamma]\!]$ is the restriction to $(\mathcal{L}_{\mathrm{nwf}}[\![\sigma_1]\!], \ldots, \mathcal{L}_{\mathrm{nwf}}[\![\sigma_k]\!]) \to \mathcal{L}_{\mathrm{nwf}}[\![\mathsf{T}_L]\!]$ of the $\gamma$-nearly well-founded elements of $\mathcal{L}[\![\gamma]\!]$. Define $\mathcal{L}_{\mathrm{nwf}}[\![\Sigma]\!]$ and $\mathcal{L}_{\mathrm{nwf}}[\![\Sigma \vdash p\colon \gamma]\!]$ in the obvious way.

(b) Inductively define $\mathcal{V}_{\mathrm{nwf}}[\![\gamma]\!]$ by: $\mathcal{V}_{\mathrm{nwf}}[\![\mathsf{N}_L]\!] = \mathbb{N}$. For $\gamma = (\sigma_1, \ldots, \sigma_k) \to \mathsf{N}_L$, $\mathcal{V}_{\mathrm{nwf}}[\![\gamma]\!]$ is the restriction to $(\mathcal{V}_{\mathrm{nwf}}[\![\sigma_1]\!], \ldots, \mathcal{V}_{\mathrm{nwf}}[\![\sigma_k]\!]) \to \mathcal{V}_{\mathrm{nwf}}[\![\mathsf{N}_L]\!]$ of the $f \in \mathcal{V}[\![\gamma]\!]$ with $|f| \in \mathcal{L}_{\mathrm{nwf}}[\![|\gamma|]\!]$. Define $\mathcal{V}_{\mathrm{nwf}}[\![\Gamma; \Delta]\!]$ and $\mathcal{V}_{\mathrm{nwf}}[\![\Gamma; \Delta \vdash E\colon \gamma]\!]$ in the obvious way.

(c) $p =_{\mathrm{nwf}} p'$ means $\mathcal{L}_{\mathrm{nwf}}[\![\Sigma \vdash p\colon \gamma]\!]\,|\rho| = \mathcal{L}_{\mathrm{nwf}}[\![\Sigma \vdash p'\colon \gamma]\!]\,|\rho|$ for all $|\rho| \in \mathcal{L}_{\mathrm{nwf}}[\![\Sigma]\!]$. Define $\leq_{\mathrm{nwf}}, \geq_{\mathrm{nwf}}, \ldots$ analogously.

There is still a problem with impredicative-type values. In giving closed form upper bounds on recursions, we often need a well-founded upper bound on the value of a variable of an impredicative type. There is no effective way to obtain such bound. To deal with this we introduce a combinator $\mathbf{p}$ such that $(\mathbf{p}\, p)$ stands for an arbitrarily chosen well-founded upper bound on $p$.[13] In most uses, $(\mathbf{p}\, p)$-expressions are destined to be substituted for by concrete, well-founded terms. As this version of the paper omits the proofs in which $\mathbf{p}$ plays a role, we omit formally defining $\mathbf{p}$ here.

The following terminology is useful in working with terms involving impredicative types.

DEFINITION 15 (Shadowing). Suppose $\Sigma \vdash p\colon \sigma$. An occurrence of a subterm $r$ of $p$ is *shadowed* when it properly appears within another shadowed occurrence or else it has an enclosing subexpression $(t\ r)$ where the $t$ is of an impredicative type $\sigma \to \tau$ with $tail(\sigma) :\gtrsim tail(\tau)$. A variable $v$ is a *shadowed free variable* for $s$ when all of $v$'s free occurrences in $p$ are shadowed; otherwise $v$ is an *unshadowed free variable* for $p$.

[13] This is analogous to the situation where one knows $f \in O(n)$ and picks an arbitrary $a \in \omega$ such that $f(n) \leq a \cdot (n + 1)$ for all $n \in \omega$.

### 4.4 Flat types and well-temperedness

***Safe upper bounds*** The restriction to the $\mathcal{V}_{\mathrm{nwf}}$-semantics solves the problem with impredicative types, but *not* the problem with flat types. Below we work towards addressing the flat-types problem by noting that $\mathsf{ATR}_1$ expressions that do not involve flat-type variables have upper bounds that are *safe* (Definition 16). The next section shows that the problem with flat-types can be solved by requiring each flat-type length to have a safe upper bound.

Safe second-order polynomial bounds are a generalization of the *poly-max* bounds of Bellantoni and Cook [2]. In their formalism if one has a type-level 1 function $f$ defined over normal variables $x_1, \ldots, x_j$ and safe variables $y_1, \ldots, y_k$, then for all values of $x_1, \ldots, y_k$, we have

$$|f(\vec{x}, \vec{y})| \ \leq\ p + \max(|y_1|, \ldots, |y_k|) \tag{6}$$

where $p$ is polynomial over $|x_1|, \ldots, |x_j|$. Such poly-max bounds play a key role in the complexity-theoretic aspects of systems inspired by Bellantoni-Cook's formalism. Definition 16 lifts these poly-max bounds to type-level 2. *Convention:* In writing $p = (v\ s_1\ \ldots\ s_k)$, we mean $v$ is a variable and, when $k = 0$, $p = v$.

DEFINITION 16 (Strictness, chariness, and safety). Suppose $\mathbf{b}$, $\gamma$, $\sigma$, and $\tau$ range over size types and that $\Sigma \vdash p\colon \gamma$.

(a) We say that $p$ is $\mathbf{b}$-*strict* with respect to $\Sigma$ when $tail(\gamma) \leq \mathbf{b}$ and every unshadowed free-variable occurrence in $p$ has a type with tail $\lesssim \mathbf{b}$.

(b) We say that $p$ is $\mathbf{b}$-*chary* with respect to $\Sigma$ when $\gamma = \mathbf{b}$ and either (i) $p = (v\ q_1 \cdots q_k)$ with $k \geq 0$, $q_1, \ldots, q_k$ $\mathbf{b}$-strict, and $\mathbf{b}$ is the $\leq$-minimal type assignable to $p$ with respect to $\Sigma$, or (ii) $p = p_1 \vee \cdots \vee p_m$, where each $p_i$ satisfies (i). (Note that $\underline{0}$ sneaks in as $\mathbf{b}$-chary; take $m = 0$ in (ii).)

(c) A $p$ is *$\gamma$-safe* with respect to $\Sigma$ if and only if $\Sigma \vdash s\colon \gamma$ and: (i) if $\gamma = \mathsf{T}_{\Box_d}$, then $p =_{\mathrm{nwf}} q \vee r$ where $q$ is $\gamma$-strict and $r$ is $\gamma$-chary; (ii) if $\gamma = \mathsf{T}_{\diamond_d}$, then $p =_{\mathrm{nwf}} q + r$ where $q$ is a $\gamma$-strict polynomial and $r$ is $\gamma$-chary $r$; and (iii) if $\gamma = \sigma \to \tau$, then $(p\, v)$ is $\tau$-safe with respect to $\Sigma, v\colon \sigma$.

*Example:* In the right-hand side of (6), the $p$ is equivalent to a $\mathsf{T}_\diamond$-strict term, $\bigvee_{j=1}^{k} |y_j|$ is equivalent to a $\mathsf{T}_\diamond$-chary term, and, hence, the entire right-hand side is equivalent to a $\mathsf{T}_\diamond$-safe term.

*Terminology:* We say that a type judgment is *flat-type variable free* when no variable is explicitly or implicitly assigned a flat type by the judgment. We note:

LEMMA 17. *If $\Gamma; \Delta \vdash E\colon \sigma$ is flat-type variable free, then there is a $|\gamma|$-safe $p_E$ with respect to $|\Gamma; \Delta|$ such that $p_E$ bounds the size of $E$ with respect to $\Gamma; \Delta$.*

So if $\Gamma; \Delta \vdash E\colon \sigma$ is flat-type variable free, but nonetheless $\sigma$ is a flat type, then by Lemma 17 $E$ has a $|\sigma|$-safe size bound $p_E$. (Moreover, this $p_E$ turns out to be flat-type variable free.)

***Well-temperedness*** To avoid problems like the one of Example 11(b), flat-type values need to be restricted. Lemma 17 tells us that every flat-type variable free $\mathsf{ATR}_1$ term has a safe upper bound. This suggests that the solution to the flat-type problem is to require all flat-type values to have safe size bounds. We call this property *well-temperedness*, meaning: all things are in the right proportions.

DEFINITION 18. An $\ell \in \mathcal{L}_{\mathrm{nwf}}[\![\gamma]\!]$ is *$\gamma$-well-tempered* when $\gamma$ is strict or else when $\gamma$ is flat and there is a closed, $\gamma$-safe $p$ with $\ell \leq \mathcal{L}_{\mathrm{nwf}}[\![p]\!]$.

LEMMA 19. *Suppose* $\Sigma \vdash p\colon \sigma$, $\rho \in \mathcal{L}_{\rm nwf}[\![\Sigma]\!]$, *and* $\rho(x)$ *is* $\Sigma(x)$*-well-tempered for each* $x \in \mathrm{preimage}(\Sigma)$. *Then* $\mathcal{L}_{\rm nwf}[\![p]\!]\,\rho$ *is* $\sigma$*-well-tempered.*

Lemma 19 indicates that a semantics for the size expressions based on well-temperedness will be well-defined.

DEFINITION 20 (The well-tempered semantics).

(a) Inductively define $\mathcal{L}_{\rm wt}[\![\sigma]\!]$ by: $\mathcal{L}_{\rm wt}[\![\mathsf{T}_L]\!] = \omega$ and, for $\sigma = (\sigma_1, \ldots, \sigma_k) \to \mathsf{T}_L$, $\mathcal{L}_{\rm wt}[\![\sigma]\!]$ is the restriction to $(\mathcal{L}_{\rm wt}[\![\sigma_1]\!], \ldots, \mathcal{L}_{\rm wt}[\![\sigma_k]\!]) \to \mathcal{L}_{\rm wt}[\![\mathsf{T}_L]\!]$ of the $\sigma$-well-tempered elements of $\mathcal{L}_{\rm nwf}[\![\sigma]\!]$. $\mathcal{L}_{\rm wt}[\![\Sigma]\!]$ and $\mathcal{L}_{\rm wt}[\![\Sigma \vdash p\colon \sigma]\!]$ are defined in the obvious way.

(b) Inductively define $\mathcal{V}_{\rm wt}[\![\sigma]\!]$ by: $\mathcal{V}_{\rm wt}[\![\mathsf{N}_L]\!] = \mathbb{N}$ and, for $\sigma = (\sigma_1, \ldots, \sigma_k) \to \mathsf{N}_L$, $\mathcal{V}_{\rm wt}[\![\sigma]\!]$ is the restriction to $(\mathcal{V}_{\rm wt}[\![\sigma_1]\!], \ldots, \mathcal{V}_{\rm wt}[\![\sigma_k]\!]) \to \mathcal{V}_{\rm wt}[\![\mathsf{N}_L]\!]$ of the $f \in \mathcal{V}_{\rm nwf}[\![\sigma]\!]$ with $|f| \in \mathcal{L}_{\rm wt}[\![|\sigma|]\!]$. $\mathcal{V}_{\rm wt}[\![\Gamma; \Delta]\!]$ and $\mathcal{V}_{\rm wt}[\![\Gamma; \Delta \vdash E\colon \sigma]\!]$ are defined in the obvious way.

(c) We write $p =_{\rm wt} p'$ when $\mathcal{L}_{\rm wt}[\![\Sigma \vdash p\colon \sigma]\!]\,|\rho| = \mathcal{L}_{\rm wt}[\![\Sigma \vdash p'\colon \sigma]\!]\,|\rho|$ for all $|\rho| \in \mathcal{L}_{\rm wt}[\![\Sigma]\!]$. We define $\leq_{\rm wt}$, $\geq_{\rm wt}$, … analogously.

There is still a problem with flat-type values. To give closed form upper bounds on recursions, we sometimes need to decompose a flat-type expression into its strict and chary parts. To help with this we introduce two new combinators, $\mathbf{q}$ and $\mathbf{r}$, which are roughly analogous to the $\mathbf{p}$ combinator of §4.3. Since we do not need to use $\mathbf{q}$ and $\mathbf{r}$ explicitly in this version of the paper, we omit their definitions and development. However, thanks in part to $\mathbf{q}$ and $\mathbf{r}$ we have the following second-order polynomial decomposition.

LEMMA 21 (Safe polynomial decomposition). *Suppose* $\Sigma \vdash p\colon \mathbf{b}$, *where* $\{ y_1, \ldots, y_k \} = \{ v \mid \Sigma(v) = \mathbf{b} \}$. *Then we can effectively find a* $\mathbf{b}$*-strict* $q$ *and a* $\mathbf{b}$*-chary* $r$ *such that* $p \leq_{\rm wt} q \odot r \vee \bigvee_{i=1}^{k} y_i$, *where* $\odot = \vee$, *if* $\mathbf{b}$ *is oracular, and* $\odot = +$, *if* $\mathbf{b}$ *is computational. Moreover,* $r$ *can be chosen to have no free occurrence of any* $y_i$.

We state another key property of safe terms. Note that the $\leq_{\rm wt}$ in the conclusion *cannot*, in general, be improved to $=_{\rm wt}$.

LEMMA 22 (Safe polynomial substitution). *Fix* $\Sigma$. *Given a* $\gamma$*-safe polynomial* $p_0$, *a* $\sigma$*-safe polynomial* $p_1$, *and a variable* $v$ *with* $\Sigma(v) = \sigma$, *we can effectively find a* $\gamma$*-safe polynomial* $p_0'$ *such that* $p_0[v \leftarrow p_1] \leq_{\rm wt} p_0'$.

### 4.5 The polynomial size boundedness theorem

We now have a reasonable semantics for $\mathsf{ATR}_1$ and the tools to establish a *safe polynomial boundedness* result for $\mathsf{ATR}_1$, where:

DEFINITION 23. Suppose $\Gamma; \Delta \vdash E\colon \sigma$. We say that $p$ is a $|\sigma|$*-safe second-order polynomial size bound* for $E$ with respect to $\Gamma; \Delta$ when $p$ is a $|\sigma|$-safe polynomial with respect to $|\Gamma; \Delta|$ such that, for all $\rho \in \mathcal{V}_{\rm wt}[\![\Gamma; \Delta]\!]$, $|\mathcal{V}_{\rm wt}[\![E]\!]\,\rho| \leq \mathcal{L}_{\rm wt}[\![s]\!]\,|\rho|$.

THEOREM 24 (Polynomial Boundedness). *Given* $\Gamma; \Delta \vdash E\colon \gamma$, *we can effectively find* $p_E$, *a* $|\gamma|$*-safe second-order size-bound for* $E$ *with respect to* $\Gamma; \Delta$.

This theorem's proof is a structural induction on the derivation of $\Gamma; \Delta \vdash E\colon \gamma$. Handling each of the $\mathsf{ATR}_1$ constructs, other that crec, is reasonably straightforward. Handling crec is more work, but the argument is clean and fairly direct.

## 5. Polynomial-time boundedness

### 5.1 Time bounds

The next major goal is to show that every $\mathsf{ATR}_1$ expression is CEK-computable within a second-order polynomial time bound (Theorem 44). The first step towards this goal is to set up a formal framework for working with time bounds. We start by noting the obvious: *Run time is* ***not*** *an extensional property of programs.* That is, $\mathcal{V}_{\rm wt}$-equivalent expressions can have quite distinct run time properties. We thus introduce $\mathcal{T}$, a new semantics for $\mathsf{ATR}_1$ that provides upper bounds on the time complexity of expressions.

***The setting for time complexities***

*CEK costs.* As previously stated, our cost model for $\mathsf{ATR}_1$ computations is the CEK cost model.

*Worst-case bounds.* $\mathcal{T}[\![E]\!]$ will provide an upper bound on the CEK cost of evaluating $E$, but not necessarily a tight upper bound.

*No free lunch.* All evaluations have positive costs. This even applies to "immediately evaluating" expressions (e.g., $\lambda$-expressions), since checking if something "immediate-evaluates" counts as a computation with costs.

*Inputs as oracles.* We treat each type-level 1 input $f$ as an oracle. In a time-complexity context this means that an $f$ is thought of answering any query in one time step, or equivalently, any computation involved in determining the reply to a query happens unobserved off-stage. Thus the cost of a query to $f$ involves only (i) the time to write down a query, $x$, and (ii) the time to read the reply, $f(x)$. The times (i) and (ii) are roughly $|x|$ and $|f|(|x|)$, respectively. Thus our time bounds will ultimately be expressed in terms of the *lengths* of the values of free and input variables.

***Currying and time complexity*** In common usage, "the time complexity of $E$" can mean one of two things. First, when $E$ is of base type, the phrase usually refers to the time required to compute the value of $E$. We might think of this as *time past*—the time it took to arrive at $E$'s value. Second, when $E$ is of an arrow type, the phrase usually refers to the function that, given the sizes of arguments, returns the time the procedure described by $E$ will take when run on arguments of the specified sizes. We might think of this as *time in possible futures* in which $E$'s value is applied. An expression can have both a past and futures of interest. Consider $(E_0\ E_1)$ where $E_0$ is of type $\mathsf{N}_\varepsilon \to \mathsf{N}_\varepsilon \to \mathsf{N}_\diamond$ and $E_1$ is of type $\mathsf{N}_\varepsilon$. Then $(E_0\ E_1)$ has a time complexity in the first sense as it took time to evaluate the expression, and, since $(E_0\ E_1)$ is of type $\mathsf{N}_\varepsilon \to \mathsf{N}_\diamond$, it also has a time complexity in the second sense. Now consider just $E_0$ itself. It too can have a nontrivial time complexity in the first sense and the potential/futures part of $E_0$'s time complexity must account for the multiple senses of time complexity just attributed to $(E_0\ E_1)$. Type-level-2 expressions add further twists to the story. Our treatment of time complexity takes into account these extended senses.

***Costs and potentials*** In the following the time complexity of an expression $E$ always has two components: a *cost* and a *potential*. A cost is always a positive (tally) integer and is intended to be an upper bound on the time it takes time to evaluate $E$. The form of a potential depends on the type of $E$. Suppose $E$ is of a base (i.e., $\mathsf{N}_L$) type. Then $E$'s potential is intended to be an upper bound on the length of its value, an element of $\omega$. The length of $E$'s value describes the potential of $E$ in the sense that when $E$'s value is used, its length is the only facet of the value that plays a role in determining time complexities. Now suppose $E$ is of type, say, $\mathsf{N}_\varepsilon \to \mathsf{N}_\diamond$. Then $E$'s potential will be an $f_E \in (\omega \to \omega \times \omega)$ that maps a $p \in \omega$ (the length/potential of the value of an argument of $E$) to a $(c_r, p_r) \in \omega \times \omega$ where $c_r$ is the cost of applying the value of $E$ to something of length $p$ and $p_r$ is the length/potential of the result. Note that $(c_r, p_r)$ is a time complexity for something of base type. Generalizing from this, our motto will be:

> *The potential of a type-*$(\sigma \to \tau)$ *thing is a map from potentials of type-*$\sigma$ *things to time complexities of type-*$\tau$ *things.*

Our first task in making good on this motto is to situate time complexities in a suitable semantic model.

***A model for time complexities*** The *time types* are the result of the following translations of $\mathsf{ATR}_1$ types: $\|\sigma\| \stackrel{\text{def}}{=} \mathsf{T} \times \langle\!\langle\sigma\rangle\!\rangle$, $\langle\!\langle \mathsf{N}_L \rangle\!\rangle \stackrel{\text{def}}{=} \mathsf{T}_L$, and $\langle\!\langle \sigma \to \tau \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle\sigma\rangle\!\rangle \to \|\tau\|$. So, $\|\mathsf{N}_{L_1} \to \mathsf{N}_{L_2} \to \mathsf{N}_{L_0}\| = \mathsf{T} \times (\mathsf{T}_{L_1} \to \mathsf{T} \times (\mathsf{T}_{L_2} \to \mathsf{T} \times \mathsf{T}_{L_0}))$ and $\|(\mathsf{N}_{L_1} \to \mathsf{N}_{L_2}) \to \mathsf{N}_{L_0}\| = \mathsf{T} \times ((\mathsf{T}_{L_1} \to \mathsf{T} \times \mathsf{T}_{L_2}) \to \mathsf{T} \times \mathsf{T}_{L_0})$. The time types are thus a subset of the simple product types over $\{\, \mathsf{T},\ \mathsf{T}_\epsilon,\ \mathsf{T}_\diamond,\ \mathsf{T}_{\Box\diamond},\ \ldots \,\}$. The intent is that $\mathsf{T}$ is the type of costs, the $\mathsf{T}_L$'s help describe lengths, $\|\gamma\|$ is the type of complexity bounds of type-$\gamma$ objects, and $\langle\!\langle\gamma\rangle\!\rangle$ is the type of potentials of type-$\gamma$ objects. (Note: $\langle\!\langle \sigma \to \tau \rangle\!\rangle$'s definition parallels the motto.)

Our proof of polynomial time boundedness for $\mathsf{ATR}_1$ (Theorem 44) needs to intertwine the size estimates implicit in potentials and the size bounds of Theorem 24. The semantics for the time types thus needs to be an extension of the $\mathcal{L}_{\text{wt}}$-semantics (Definition 20(a)). To define this extension we use a combinator, Pot, defined in Definition 35 below. For the moment it is enough to know that, for each $\mathsf{ATR}_1$-type $\sigma$ and $p \in \mathcal{L}_{\text{wt}}[\![\langle\!\langle\sigma\rangle\!\rangle]\!]$, $\mathsf{Pot}(p) \in \mathcal{L}_{\text{wt}}[\![|\sigma|]\!]$ is a canonical projection of $p$ to a type-$|\sigma|$ size bound. Following the definition of Pot, Lemma 36 notes that all of the notions introduced between here and there mesh properly.

DEFINITION 25 ($\mathcal{L}_{\text{wt}}$ extended to the time types). Suppose $\sigma$ and $\tau$ are $\mathsf{ATR}_1$ types. Then $\mathcal{L}_{\text{wt}}[\![\|\sigma\|]\!] \stackrel{\text{def}}{=} \omega \times \mathcal{L}_{\text{wt}}[\![\langle\!\langle\sigma\rangle\!\rangle]\!]$, $\mathcal{L}_{\text{wt}}[\![\langle\!\langle \mathsf{N}_L \rangle\!\rangle]\!] \stackrel{\text{def}}{=} \omega$, and $\mathcal{L}_{\text{wt}}[\![\langle\!\langle \sigma \to \tau \rangle\!\rangle]\!] \stackrel{\text{def}}{=}$ the set of all $f$, a monotone Kleene-Kreisel function from $\mathcal{L}_{\text{wt}}[\![\langle\!\langle\sigma\rangle\!\rangle]\!]$ to $\mathcal{L}_{\text{wt}}[\![\|\tau\|]\!]$, such that: (i) $\mathsf{Pot}(f) \in \mathcal{L}_{\text{wf}}[\![|\sigma \to \tau|]\!]$ and (ii) $\mathsf{Pot}(f(p_1)) = \mathsf{Pot}(f(p_2))$ whenever $\mathsf{Pot}(p_1) = \mathsf{Pot}(p_2)$.

Condition (i) above restricts $\mathcal{L}_{\text{wt}}[\![\langle\!\langle \sigma \to \tau \rangle\!\rangle]\!]$ so that the projection Pot acts as advertised. Condition (ii) restricts each $f \in \mathcal{L}_{\text{wt}}[\![\langle\!\langle \sigma \to \tau \rangle\!\rangle]\!]$ so that the size information in $f(p)$ depends only on the size information in $p$.

We can now define the time-complexity and potential interpretations of the $\mathsf{ATR}_1$ types. ($\mathcal{P}[\![\,\cdot\,]\!]$ is a notational convenience.)

DEFINITION 26. Suppose $\sigma$ is an $\mathsf{ATR}_1$-type. Then $\mathcal{T}[\![\sigma]\!] \stackrel{\text{def}}{=} \mathcal{L}_{\text{wt}}[\![\|\sigma\|]\!]$ and $\mathcal{P}[\![\sigma]\!] \stackrel{\text{def}}{=} \mathcal{L}_{\text{wt}}[\![\langle\!\langle\sigma\rangle\!\rangle]\!]$.

***The $\mathcal{T}$-interpretation of constants and oracles*** The following two definitions introduce a translation from the $\mathcal{V}_{\text{wt}}$ model (Definition 20(b)) into the $\mathcal{T}$ model. We use this translation to assign time complexities to inputs: string/numeric constants and oracles.

DEFINITION 27. Let $\|a\| \stackrel{\text{def}}{=} (\underline{1} \vee |a|, \langle\!\langle a \rangle\!\rangle)$ and $\langle\!\langle a \rangle\!\rangle \stackrel{\text{def}}{=} |a|$ for each $a \in \mathcal{V}_{\text{wt}}[\![\mathsf{N}_L]\!]$.

By Lemma 36(a) below, $\|a\| \in \mathcal{T}[\![\mathsf{N}_L]\!]$. We view $\|a\|$ as the time complexity of the string/numeric constant $a$. The interpretation of the cost component of $\|a\|$ is that cost of evaluating the constant $a$ is the cost of writing down $a$ character by character. (When $a = \epsilon$, we still charge $\underline{1}$.)

DEFINITION 28. Let $\|f\| \stackrel{\text{def}}{=} (\underline{1}, \langle\!\langle f \rangle\!\rangle)$ and $\langle\!\langle f \rangle\!\rangle \stackrel{\text{def}}{=} \lambda p \in \mathcal{P}[\![\sigma]\!]\,.\ \max\{\, \|(f\,x)\| \;:\; \langle\!\langle x \rangle\!\rangle \le p \,\}$ for each $f \in \mathcal{V}_{\text{wt}}[\![\sigma \to \tau]\!]$.

By Lemma 36(a) below, $\|f\| \in \mathcal{T}[\![\sigma \to \tau]\!]$. We view $\|f\|$ as the time complexity of $f$ as an oracle: the only time costs associated with applying $f$ are those involved in setting up applications of $f$ and reading off the results. Recall that under call-by-value, a $\lambda$-expression immediately evaluates to itself. The function $f$ will be treated analogously to a $\lambda$-term. Hence, the cost component of $\|f\|$ is $\underline{1}$. The definition of $\langle\!\langle f \rangle\!\rangle$ parallels both our informal discussion of the notion of the potential of a type-level 1 function and the definition of the length of functions in §2. One can show that when $f$ is a type-level 2, $\langle\!\langle f \rangle\!\rangle$ is total. (The argument is similar to the proof of the totality of the type-level 2 notion of length defined by (3) in §2.) The following unpacks Definition 28.

LEMMA 29. *For $f \in \mathcal{V}_{\text{wt}}[\![(\mathsf{N}_{L_1}, \ldots, \mathsf{N}_{L_k}) \to \mathsf{N}_{L_0}]\!]$, $\langle\!\langle f \rangle\!\rangle = q_1$ where $q_i = \lambda p_i \in \omega\,.\ (\underline{1}, q_{i+1})$ (for $1 \le i < k$) and $q_k = \lambda p_k \in \omega\,.\ \big(\underline{1} \vee |f|(\vec{p}), |f|(\vec{p})\big)$.*

***$\mathcal{T}$-Applications***

DEFINITION 30.

(a) Suppose $t_0 \in \mathcal{T}[\![\sigma \to \tau]\!]$ and $t_1 \in \mathcal{T}[\![\sigma]\!]$, where $t_0 = (c_0, p_0)$, $t_1 = (c_1, p_1)$, and $(c_r, p_r) = p_0(p_1)$. Then $t_0 \star t_1 \stackrel{\text{def}}{=} (c_0 + c_1 + c_r + \underline{3},\ p_r)$.

(b) Suppose $t_0 \in \mathcal{T}[\![(\sigma_1, \ldots, \sigma_k) \to \tau]\!], t_1 \in \mathcal{T}[\![\sigma_1]\!], \ldots, t_k \in \mathcal{T}[\![\sigma_k]\!]$. Then $t_0 \star \vec{t} \stackrel{\text{def}}{=} t_0 \star t_1 \star \cdots \star t_k$. (The $\star$ left associates.)

By Lemma 36(b) below, $t_0 \star t_1 \in \mathcal{T}[\![\tau]\!]$ when $t_0 \in \mathcal{T}[\![\sigma \to \tau]\!]$ and $t_1 \in \mathcal{T}[\![\sigma]\!]$. Suppose $t_0$ (resp., $t_1$) is the time complexity of a type-$(\sigma \to \tau)$ expression $E_0$ (resp., type-$\sigma$ expression $E_1$). Then $t_0 \star t_1$ is intended to be the time complexity of $(E_0\ E_1)$. The cost component of $t_0 \star t_1$ is: (the cost of evaluating $E_0$) + (the cost of evaluating $E_1$) + (the cost of applying $E_0$'s value to $E_1$'s value) + $\underline{3}$, where $\underline{3}$ is the CEK-overhead of an application. The potential component is simply the potential of the result of the application.

***$\mathcal{T}$-Environments*** As a companion to $\mathcal{T}$-application we shall define an analogue currying in $\mathcal{T}$. Before doing that we need to introduce $\mathcal{T}$-environments. Recall that in a call-by-value language, variables name *values* [23], i.e., the end result of a (terminating) evaluation. Thus, a value does not need to be evaluated again, at least no more than an input value does. Hence, if a $\mathcal{T}$-environment maps a variable to a type-$\gamma$ time complexity $(c, p)$, then $c$ should be: $\underline{1} \vee p$, when $\gamma$ is a base type, and $\underline{1}$, when $\gamma$ is an arrow type.

DEFINITION 31. Suppose $\sigma$ and $\tau$ vary over $\mathsf{ATR}_1$ types and $\Gamma; \Delta$ is an $\mathsf{ATR}_1$ is type context.

(a) $\|\Gamma; \Delta\| \stackrel{\text{def}}{=} \{\, v \mapsto \|\sigma\| \;:\; (\Gamma; \Delta)(v) = \sigma \,\}$.

(b) For $p \in \mathcal{P}[\![\mathbf{b}]\!]$, $\mathsf{val}(p) \stackrel{\text{def}}{=} (\underline{1} \vee p, p)$.

(c) For $p \in \mathcal{P}[\![\sigma \to \tau]\!]$, $\mathsf{val}(p) \stackrel{\text{def}}{=} (\underline{1}, p)$.

(d) $\mathcal{T}_{\mathsf{val}}[\![\sigma]\!] \stackrel{\text{def}}{=} \{\, \mathsf{val}(p) \;:\; p \in \mathcal{P}[\![\sigma]\!] \,\}$.

(e) $\mathcal{T}[\![\Gamma; \Delta]\!]$ is the set of all finite maps of the form $\{\, v_1 \mapsto t_1, \ldots, v_k \mapsto t_k \,\}$, where $\{\, v_1, \ldots, v_k \,\} = \text{preimage}(\Gamma; \Delta)$ and $t_1 \in \mathcal{T}_{\mathsf{val}}[\![(\Gamma; \Delta)(v_1))]\!], \ldots, t_k \in \mathcal{T}_{\mathsf{val}}[\![(\Gamma; \Delta)(v_k))]\!]$.

(f) For each $\rho \in \mathcal{V}_{\text{wt}}[\![\Gamma; \Delta]\!]$, define $\|\rho\| \in \mathcal{T}[\![\Gamma; \Delta]\!]$ by $\|\rho\|\,(v) = \|\rho(v)\|$. Such at $\|\rho\|$ is called an *oracle environment*.

We use $\varrho$ as a variable over $\mathcal{T}[\![\Gamma; \Delta]\!]$. **N.B.** Not every $\mathcal{T}$-environment of interest is an oracle environment.

***$\mathcal{T}$-currying*** Here then is our time-complexity analogue to currying. Recall that $\mathcal{T}[\![\Gamma; \Delta \vdash E\colon \tau]\!]$ will be (when we get around to defining it) a function from $\mathcal{T}[\![\Gamma; \Delta]\!]$ to $\mathcal{T}[\![\tau]\!]$.

DEFINITION 32. Suppose that $\Gamma; \Delta$ is a $\mathsf{ATR}_1$ type context with $(\Gamma; \Delta)(v_i) = \sigma_i$, for $i = 1, \ldots, k$; that $\Gamma'; \Delta'$ is the result of removing $v_1\colon\sigma_1$ from $\Gamma; \Delta$; and that $X$ is a function from $\mathcal{T}[\![\Gamma; \Delta]\!]$ to $\mathcal{T}[\![\tau]\!]$. Then $\Lambda_\star(v_1, X)$ is the function from $\mathcal{T}[\![\Gamma'; \Delta']\!]$ to $\mathcal{T}[\![\sigma_1 \to \tau]\!]$ given by:

$$\Lambda_\star(v_1, X)\,\varrho' \stackrel{\text{def}}{=} \big(\,\underline{1},\ \lambda p \in \mathcal{P}[\![\sigma_1]\!]\,.\,(X\ \varrho'_p)\,\big)\,, \qquad (7)$$

where $\varrho' \in \mathcal{T}[\![\Gamma'; \Delta']\!]$ and $\varrho'_p = \varrho' \cup \{\, v_1 \mapsto \mathsf{val}(p) \,\}$. Also, for $k > 1$, $\Lambda_\star(v_1, v_2, \ldots, v_k, X) \stackrel{\text{def}}{=} \Lambda_\star(v_1, \Lambda_\star(v_2, \ldots, v_k, X))$.

Note the complementary roles of $\Lambda_\star$ and $\star$: $\Lambda_\star$ shifts the past (the cost) into the future (the potential) and $\star$ shifts part of the future (the potential) into the past (the cost). This being complexity theory, there are carrying charges on all this shifting. This is illustrated by the next lemma that shows how $\Lambda_\star$ and $\star$ interact. First, we introduce:

DEFINITION 33. *dally*$(d,(c,p)) \stackrel{\text{def}}{=} (c+d,p)$ for $d \in \omega$ and $(c,p)$, a time complexity.

LEMMA 34 (Almost the $\eta$-law). *Suppose $\Gamma;\Delta$, $X$, $\vec{v}$, $\vec{\sigma}$, and $\tau$ are as in Definition 32. Let $\Gamma';\Delta'$ be the result of removing $v_1{:}\,\sigma_1,\ldots,v_k{:}\,\sigma_k$ from $\Gamma;\Delta$. Let $\varrho \in \mathcal{T}[\![\Gamma;\Delta]\!]$ and let $\varrho'$ be the restriction of $\varrho$ to the domain of definition of $\Gamma';\Delta'$. Then*

$$\big(\Lambda_\star(v_1,\ldots,v_k,X)\,\varrho'\big)\star\varrho(v_1)\star\cdots\star\varrho(v_k) \;=\; \textit{dally}(5\cdot k+4+\textstyle\sum_{i=1}^{k}c_i,\ X\,\varrho), \tag{8}$$

*where* $(c_1,p_1)=\varrho(v_1),\ldots,(c_k,p_k)=\varrho(v_k)$.

***Projections*** The next definition introduces a way of recovering more conventional bounds from time complexities. Note, by Definitions 27 and 28, and Lemma 29, when $x$ is a string/numeric constant or a type-1 oracle we can treat $\|x\|$ as a function of $|x|$.

DEFINITION 35. Suppose $\sigma$ and $(\sigma_1,\ldots,\sigma_k)\to\mathsf{N}_L$ are $\mathsf{ATR}_1$ types.

(a) For each $t\in\mathcal{T}[\![\sigma]\!]$, let $\textit{cost}(t)\stackrel{\text{def}}{=}\pi_1(t)$ and $\textit{pot}(t)\stackrel{\text{def}}{=}\pi_2(t)$. *(So, $t=(\textit{cost}(t),\textit{pot}(t))$.)*

(b) For each $t\in\mathcal{T}[\![\mathsf{N}_L]\!]$, let $\mathsf{Cost}(t)=\textit{cost}(t)$ and $\mathsf{Pot}(t)=\textit{pot}(t)$ and, for each $t\in\mathcal{T}[\![(\sigma_1,\ldots,\sigma_k)\to\mathsf{N}_L]\!]$, let $\mathsf{Cost}(t)\stackrel{\text{def}}{=}\lambda\overrightarrow{|x|}\,.\,\textit{cost}(t\star\overrightarrow{\|x\|})$ and $\mathsf{Pot}(t)\stackrel{\text{def}}{=}\lambda\overrightarrow{|x|}\,.\,\textit{pot}(t\star\overrightarrow{\|x\|})$, where $\overrightarrow{|x|}$ abbreviates $|x_1|\in\mathcal{L}_{\mathrm{wt}}[\![\sigma_1]\!],\ldots,|x_k|\in\mathcal{L}_{\mathrm{wt}}[\![\sigma_k]\!]$ and $\overrightarrow{\|x\|}$ abbreviates $\|x_1\|,\ldots,\|x_k\|$. *(So, $t\star\overrightarrow{\|x\|}=(\mathsf{Cost}(t)(\overrightarrow{|x|}),\mathsf{Pot}(t)(\overrightarrow{|x|}))$.)*

(c) For each $p\in\mathcal{P}[\![\sigma]\!]$, let $\mathsf{Pot}(p)\stackrel{\text{def}}{=}\mathsf{Pot}(\,(\underline{1},p)\,)$.

Suppose $t$ is the time complexity of an expression $E$ of type $(\vec{\sigma})\to\mathsf{N}_L$. Then both $\mathsf{Cost}(t)$ and $\mathsf{Pot}(t)$ are functions of the sizes of possible arguments of $E$. The intent is that $\mathsf{Cost}(t)(\overrightarrow{|x|})$ is an upper bound on the time cost of first evaluating $E$ and then applying its value to arguments of the specified sizes and that $\mathsf{Pot}(t)$ is an upper bound on the length of $E$'s value.

LEMMA 36 (The consistency check). *Suppose $\sigma$ and $\sigma\to\tau$ are $\mathsf{ATR}_1$ types.*

*(a) For each $x\in\mathcal{V}_{\mathrm{wt}}[\![\sigma]\!]$, $\|x\|\in\mathcal{T}[\![\sigma]\!]$ and $\mathsf{Pot}(x)=|x|$.*

*(b) For each $t_0\in\mathcal{T}[\![\sigma\to\tau]\!]$ and $t_1\in\mathcal{T}[\![\sigma]\!]$, $t_0\star t_1\in\mathcal{T}[\![\tau]\!]$.*

*(c) $\Lambda_\star$ is well-defined in the sense that the left-hand side of (7) is in $\mathcal{T}[\![\sigma_1\to\tau]\!]$ as asserted in Definition 36*

***Time-complexity polynomials*** To complete the basic time-complexity framework, we define an extension of the second-order polynomials for the simple product types over $\mathsf{T}$, $\mathsf{T}_\varepsilon$, $\mathsf{T}_\diamond,\ldots$ under the $\mathcal{L}$-semantics. The restriction of these to the time types under the $\mathcal{L}_{\mathrm{wt}}$-semantics are the *time-complexity polynomials*. First we extend the grammar for raw expressions to include: $P ::= (P,P) \mid \pi_1(P)\mid\pi_2(P)$. Then we add the following typing rules:

$$\frac{\Sigma\vdash P{:}\,\sigma_1\times\sigma_2}{\Sigma\vdash\pi_i(P){:}\,\sigma_i}\qquad\frac{\{\,\Sigma\vdash P_i{:}\,\sigma_i\,\}_{i=1,2}}{\Sigma\vdash P_1\odot P_2{:}\,\sigma}\qquad\frac{\{\,\Sigma\vdash P_i{:}\,\sigma_i\,\}_{i=1,2}}{\Sigma\vdash(P_1,P_2){:}\,\sigma_1\times\sigma_2}$$

for second-order polynomials, where $\sigma$, $\sigma_1$, and $\sigma_2$ simple product types over $\mathsf{T}$, $\mathsf{T}_\varepsilon$, $\mathsf{T}_\diamond,\ldots$ and $\odot$ stands for any of $*$, $+$, or $\vee$. Next we extend the arithmetic operations to all types by recursively defining, for each $\gamma$ and each $x,y\in\mathcal{L}[\![\gamma]\!]$, $x\odot y\stackrel{\text{def}}{=}$

$$\begin{cases}\star, & \text{if } \gamma=();\\ \text{the usual thing}, & \text{if } \gamma \text{ is a base type};\\ (\pi_1(x)\odot\pi_1(y),\pi_2(x)\odot\pi_2(y)), & \text{if } \gamma=\sigma\times\tau;\\ \lambda z\in\mathcal{L}[\![\sigma]\!]\,.\,(x(z)\odot y(z)), & \text{if } \gamma=\sigma\to\tau.\end{cases}$$

The $\mathcal{L}$-interpretation of the polynomials is just the obvious thing. Note that $q_1$ of Lemma 29 and the right-hand side of (8) are well-typed, time-complexity polynomials. Also note that by Definition 30(a), if $q_1$ and $q_2$ time-complexity polynomials with $\|\Gamma;\Delta\|\ \vdash\ q_1{:}\,\|\sigma\to\tau\|$ and $\|\Gamma;\Delta\|\ \vdash\ q_2{:}\,\|\sigma\|$, then $q_1\star q_2$ is a time-complexity polynomial with $\|\Gamma;\Delta\|\vdash q_1\star q_2{:}\,\|\tau\|$.

$$\mathcal{T}[\![c]\!]\,\varrho=\|c\|\qquad \mathcal{T}[\![(\mathtt{c_a}\ E_0)]\!]\,\varrho=(c_0+\underline{2},\,p_0+\underline{1}).$$
$$\mathcal{T}[\![v]\!]\,\varrho=\varrho(v).\qquad \mathcal{T}[\![(\mathtt{t_a}\ E_0)]\!]\,\varrho=(c_0+\underline{2},\,\underline{1}).$$
$$\mathcal{T}[\![(\mathtt{d}\ E_0)]\!]\,\varrho=(c_0+\underline{2},\,(p_0-1)\vee\underline{0}).$$
$$\mathcal{T}[\![(\mathtt{down}\ E_0\ E_1)]\!]\,\varrho=(c_0+c_1+p_0+p_1+\underline{3},\,\min(p_0,p_1)).$$
$$\mathcal{T}[\![(\lambda v\,.\,E_0)]\!]\,\varrho=\Lambda_\star(v,\mathcal{T}[\![E_0]\!])\,\varrho.$$
$$\mathcal{T}[\![(E_0\ E_1)]\!]\,\varrho=(\mathcal{T}[\![E_0]\!]\,\varrho)\star(\mathcal{T}[\![E_1]\!]\,\varrho).$$
$$\mathcal{T}[\![(\mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2)]\!]\,\varrho=(c_0+\underline{2},\underline{0})+(c_1,p_1)\vee(c_2,p_2).$$
*Above:* $c$ is a constant, $\varrho\in\mathcal{T}[\![\Gamma;\Delta]\!]$, and $(c_i,p_i)=\mathcal{T}[\![\Gamma;\Delta\vdash E_i{:}\,\sigma_i]\!]\,\varrho$ for $i=0,1,2$.

**Figure 11.** The $\mathcal{T}$-interpretation of $\mathsf{ATR}_1^-$.

### 5.2 The time-complexity interpretation of $\mathsf{ATR}_1^-$

We here establish a poly-time boundedness result for $\mathsf{ATR}_1^-$, the subsystem of $\mathsf{ATR}_1$ obtained by dropping the crec construct. Definition 37 introduces the $\mathcal{T}$-interpretation of $\mathsf{ATR}_1^-$ and the proof of Theorem 41 shows that $\mathsf{ATR}_1^-$-expressions have time complexities that are polynomial bounded and well-behaved in other ways. All of this turns out to be reasonably straightforward. *Convention:* Through out this section suppose that $\gamma$, $\sigma$, and $\tau$ are $\mathsf{ATR}_1$ types and $\Gamma;\Delta$ is an $\mathsf{ATR}_1$ type context.

DEFINITION 37. Figure 11 provides the $\mathcal{T}$-interpretation for each $\mathsf{ATR}_1^-$ construct.

There are three key things to establish about the time complexities assigned by $\mathcal{T}$, that they are: (i) not too big, (ii) not too small, and (iii) monotone. "Not too big" means that the time complexities are polynomially bounded in the sense of Definition 38 below. "Not too small" means that $\textit{cost}(\mathcal{T}[\![E]\!]\ \|\rho\|)\ge\mathrm{CEK\text{-}cost}(E,\rho)$ and $\mathsf{Pot}(\mathcal{T}[\![E]\!]\ \|\rho\|)$ is at least as large as $|\mathcal{V}_{\mathrm{wt}}[\![E]\!]\,\rho|$, but no larger than $\mathsf{Cost}(\mathcal{T}[\![E]\!]\ \|\rho\|)$. This "not too small" property (*soundness*) is introduced in Definition 39. Finally, "monotone" means that $\mathcal{T}[\![E]\!]\,\varrho\le\mathcal{T}[\![E]\!]\,\varrho'$ when $\varrho\le\varrho'$[14] and that when $\mathcal{T}[\![E]\!]\,\varrho$ is a function, it is point-wise, monotone nondecreasing. Monotonicity is introduced in Definition 40 and plays an important role in dealing with crec. Theorem 41 establishes that the $\mathcal{T}$-interpretation of $\mathsf{ATR}_1^-$ satisfies these three properties. *Convention:* Below let $\mathcal{F}$ range over programming formalisms (e.g., $\mathsf{ATR}_1^-$ and $\mathsf{ATR}_1$).

DEFINITION 38 (Constructive polynomial-time boundedness). A $\mathcal{T}$-interpretation of $\mathcal{F}$ is *constructively polynomial-time bounded* when, for each $\mathcal{F}$-judgment $\Gamma;\Delta\ \vdash\ E{:}\,\sigma$, we can effectively find a time-complexity polynomial $q$ with $|\Gamma;\Delta|\ \vdash\ q{:}\,\|\sigma\|$ such that $\mathcal{T}[\![E]\!]\,\|\rho\|\le\mathcal{L}_{\mathrm{wt}}[\![q]\!]\,|\rho|$ for each $\rho\in\mathcal{V}_{\mathrm{wt}}[\![\Gamma;\Delta]\!]$.

DEFINITION 39 (Soundness). A $\mathcal{T}$-interpretation of $\mathcal{F}$ is *sound* when, for each $\mathcal{F}$-judgment $\Gamma;\Delta\vdash E{:}\,\gamma$ and each $\rho\in\mathcal{V}_{\mathrm{wt}}[\![\Gamma;\Delta]\!]$, we have $\mathrm{CEK\text{-}cost}(E,\rho)\le\textit{cost}(\mathcal{T}[\![E]\!]\,\|\rho\|)$ and $\big|\mathcal{V}_{\mathrm{wt}}[\![E]\!]\,\rho\big|\le\mathsf{Pot}(\mathcal{T}[\![E]\!]\,\|\rho\|)\le\mathsf{Cost}(\mathcal{T}[\![E]\!]\,\|\rho\|)$.

DEFINITION 40 (Monotonicity). A $\mathcal{T}$-interpretation of $\mathcal{F}$ is *monotone* when, for each $\mathcal{F}$-judgment $\Gamma;\Delta\vdash E{:}\,\gamma$: (i) $\mathcal{T}[\![E]\!]$ is a point-wise, monotone nondecreasing function from $\mathcal{T}[\![\Gamma;\Delta]\!]$ to $\mathcal{T}[\![\gamma]\!]$,

[14] *Convention:* For $\varrho,\ \varrho'\in\mathcal{T}[\![\Gamma;\Delta]\!]$, we write $\varrho\le\varrho'$ when $\varrho(v)\le\varrho'(v)$ for each $v\in\mathrm{preimage}(\Gamma;\Delta)$.

and (ii) if $\gamma = (\sigma_0, \ldots, \sigma_k) \to \mathbf{b}$, then the function $\mathcal{T}[\![\Gamma; \Delta]\!] \times \mathcal{T}[\![\sigma_0]\!] \times \cdots \times \mathcal{T}[\![\sigma_k]\!]$ to $\mathcal{T}[\![\mathbf{b}]\!]$ given by $(\varrho, x_0, \ldots, x_k) \mapsto ((\mathcal{T}[\![E]\!]\,\varrho)\,x_0\ \ldots\ x_k)$ is pointwise, monotone nondecreasing.

THEOREM 41. *The $\mathcal{T}$-interpretation of $\mathsf{ATR}_1^-$ is monotone, sound, and constructively polynomial-time bounded.*

The proof is a logical relations argument.

### 5.3 An affine decomposition of time complexities

When analyzing a program's run time, one often must decompose its time complexity into pieces that may have little to do with the program's apparent syntactic structure. Theorem 43 below is a general time-complexity decomposition result for $\mathsf{ATR}_1$ expressions. The $\mathsf{ATR}_1$ typing rules for affinely restricted variables are critical in ensuring this decomposition. The decomposition is used to obtain the recurrences needed to analyze the time complexity of crec expressions.[15] To help in the theorem's statement, we introduce:

DEFINITION 42.

(a) $(c_1, p_1) \uplus (c_2, p_2) \stackrel{\text{def}}{=} (c_1 + c_2, p_1 \vee p_2)$, where $(c_1, p_1)$, $(c_2, p_2) \in \mathcal{T}[\![\gamma]\!]$. *(Clearly, $(c_1, p_1) \uplus (c_2, p_{h2}) \in \mathcal{T}[\![\gamma]\!]$.)*

(b) For each $\mathsf{ATR}_1$-type $\gamma$, define $\epsilon_\gamma$ inductively by: $\epsilon_{\mathsf{N}_L} = \epsilon$ and $\epsilon_{\sigma\to\tau} = \lambda x\,.\,\epsilon_\tau$. *(Clearly, $\vdash \epsilon_\gamma : \gamma$ and $|\mathcal{V}_{\mathrm{wt}}[\![\epsilon_\gamma]\!]\,\rho_\emptyset| = \underline{0}_{|\gamma|}$.)*

(c) Given $f : (\sigma_1, \ldots, \sigma_k) \to \mathsf{N}_L$, an expression of the form $(f\ E_1\ \ldots\ E_k)$ is called a *full application* of $f$.

THEOREM 43 (Affine decomposition). *Suppose $\Gamma; f : \gamma \vdash E : \mathsf{N}_{L_0}$, where $\gamma = (\mathsf{N}_{L_1}, \ldots, \mathsf{N}_{L_k}) \to \mathsf{N}_{L_0} \in \mathcal{R}$ and TailPos$(f, E)$. Let $\zeta$ denote the substitution $[f \leftarrow \epsilon_\gamma]$. Then*

$$\mathcal{T}[\![E]\!]\,\varrho \ \le\ \mathcal{T}[\![E\,\zeta]\!]\,\varrho \ \uplus\ ((\mathcal{T}[\![f]\!]\,\varrho) \star \vec{t}\,), \tag{9}$$

*for each $\varrho \in \mathcal{T}[\![\Gamma; f : \gamma]\!]$, where $(f\ E_1^1\ \ldots\ E_k^1), \ldots, (f\ E_1^\ell\ \ldots\ E_k^\ell)$ are the full applications of $f$ occurring in $E$ and $t_j = \bigvee_{i=1}^{\ell} \mathsf{val}(\mathcal{T}[\![E_j^i]\!])\,\varrho$, for $j = 1, \ldots, k$.*

By Lemma 3 we know that there is at most one use of an affinely restricted variable in an expression. In terms of costs, (9) says that the cost of evaluating $E$ can be bounded by the sum of: (i) the cost of evaluating $E\,\zeta$, which includes the all of the costs of $E$ except for the possible application of the value of $f$ to the values of its arguments, and (ii) *cost*$((\mathcal{T}[\![f]\!]\,\varrho) \star \vec{t}\,)$, which clearly bounds the cost of any such $f$ application. In terms of potentials, (9) says that $E$ is size bounded by the maximum of (i) the size of the value of $E\,\zeta$, which covers all the cases where $f$ is not applied, and (ii) *pot*$((\mathcal{T}[\![f]\!]\,\varrho) \star \vec{t}\,)$, which covers all the cases where $f$ is applied.

If (9) solely concerned CEK costs, the above remarks would almost constitute a proof. However, (9) is about $\mathcal{T}$-interpretations of expressions and $\mathcal{T}[\![E]\!]$ is an approximation to the true time complexities involved in evaluating $E$. The theorem asserts that our $\mathcal{T}$-interpretation of $\mathsf{ATR}_1$ is verisimilar enough to capture this property of time complexities. This later requires a little work.

### 5.4 The time-complexity interpretation of $\mathsf{ATR}_1$

We are now consider the time complexity properties of crec expressions. This version of the paper omits the details of this, but the idea is simply that we use Theorem 43 to define $\mathcal{T}[\![(\mathsf{crec}\ a\ (\lambda_r f\,.\,A))]\!]$ in terms of $\mathcal{T}[\![(\mathsf{crec}\ (\mathbf{0} \oplus a)\ (\lambda_r f\,.\,A))]\!]$. This gives us a recurrence equation for each crec expression so that we can now prove:

THEOREM 44. *The $\mathcal{T}$-interpretation of $\mathsf{ATR}_1$ is monotone, sound, and constructively polynomial-time bounded.*

[15] The theorem presupposes that $\mathcal{T}[\![\,\cdot\,]\!]$ is defined on crec expressions. But, since no affinely restricted variable can occur free in a well-typed crec-expression and since the application of the theorem will be within an induction, this presupposition does not add any difficulties.

The proof of this is a direct extension of the proof of Theorem 41 in which we solve the recurrences given by the definition of $\mathcal{T}[\![(\mathsf{crec}\ a\ (\lambda_r f\,.\,A))]\!]$. The presence of higher-type functions in the recurrences is a manageable complication here.

### 5.5 Polynomial-time completeness

Finally, we note that each type-1 and type-2 BFF is computable by some $\mathsf{ATR}_1$ program. *Terminology:* An $\mathsf{ATR}_1$-type $\gamma$ is *unhindered* when $\gamma$ is a base type or else $\gamma = (\gamma_1, \ldots, \gamma_k) \to \mathsf{N}_L$ is strict and predicative with each $\gamma_1$ unhindered. By the definition of the $\mathcal{V}_{\mathrm{wt}}$-semantics, $\mathcal{V}_{\mathrm{wt}}[\![\gamma]\!] = \mathcal{V}[\![\mathit{shape}(\gamma)]\!]$ exactly when $\gamma$ is unhindered.

THEOREM 45. *Suppose $\sigma$ is a simple type over $\mathsf{N}$ of level 1 or 2. Then the class of type-$\sigma$ BFFs = $\{\, \mathcal{V}_{\mathrm{wt}}[\![\vdash E : \gamma]\!] \ :\ \sigma = \mathit{shape}(\gamma)$ and $\gamma$ is unhindered $\}$.*

The $\subseteq$ containment follows by straightforward programming. The $\supseteq$ containment follows by Theorem 44.

## 6. Related work

Ramified types based on Bellantoni and Cook's ideas, higher types, and linear types are common features of work on implicit complexity (see Hofmann's survey [10]), but most of that work has focused on guaranteeing complexity of type-level 1 programs. The $\mathsf{ATR}_1$ type system is roughly a refinement of the type systems of [12, 13] which were constructed to help study higher-type complexity classes. Also, the type systems of this paper and [12, 13] were greatly influenced by Leivant's ramified type systems [17, 16].

The time-complexity cost/potential distinction appears in prior work. A version of this distinction can be found in Sands' Ph.D. thesis [26]. Shultis [28] sketched how to use the distinction in order to give time-complexity semantics for reasoning about the run-time programs that involve higher types. Van Stone [29] gives a much more detailed and sophisticated semantics using this distinction. Very roughly, Sands', Shultis', and Van Stone's work was targeted toward giving static analyses to extract time-bounds for functional programs that compute first-order functions. The time-complexity semantics of this paper was developed independently of Shultis' and Van Stone's work.

## 7. Possible extensions

***Recursions with type-level 1 parameters*** In work extending the results of this paper we have a programming formalism, $\mathsf{ATR}_2$, that extends $\mathsf{ATR}_1$ to permit type-level 1 parameters in crec-recursions. In particular, $\mathsf{ATR}_2$ is expressive enough to give a continuation-passing-style definition of *prn*. The proofs of polynomial-size boundedness and of polynomial-time boundedness for $\mathsf{ATR}_2$ are more involved than those for $\mathsf{ATR}_1$, but not excessively so.

***Type checking, type inference, time-bound inference*** We have not studied the problem of $\mathsf{ATR}_1$ type checking. But since $\mathsf{ATR}_1$ is just an applied simply typed lambda calculus with subtyping, standard type-checking tools should suffice. Type inference is a much more interesting problem. We suspect that a useful type inference algorithm could be based on Frederiksen and Jones' [7] work on applying size-change analysis to detect whether programs run in polynomial time. Another interesting problem would be to start with a well-typed $\mathsf{ATR}_1$ program and to then extract reasonably tight size and time bounds.

***Lazy evaluation*** To handle a call-by-name or a call-by-need version of $\mathsf{ATR}_1$, one would: (i) construct an abstract machine for this lazy-$\mathsf{ATR}_1$, (ii) modify the $\mathcal{T}$-semantics a bit to accommodate the lazy constructs; and (iii) rework the $\mathcal{T}$-interpretation of $\mathsf{ATR}_1$

which would then have to be shown monotone, sound, and constructively polynomial-time bounded. If our lazy-$\mathsf{ATR}_1$ allowed infinite strings, then the $\mathcal{V}_{\mathrm{wt}}$-semantics would also have to be modified. Note that Sands [26] and Van Stone [29] both consider lazy evaluation in their work.

***Lists and streams*** There are multiple senses of the "size" of a list. For example, the run-time of *reverse* should depend on just a list's length, whereas the run-time of a search depends on both the list's length and the sizes of the list's elements. Any useful extension of $\mathsf{ATR}_1$ that includes lists needs to account for these multiple senses of size in both the type system and the $\mathcal{V}_{\mathrm{wt}}$- and $\mathcal{T}$-semantics. If lists are combined with laziness, then we also have the problem of handling infinite lists. However, $\mathsf{ATR}_1$ and its semantics already handle one flavor of infinite object, i.e., type-level 1 inputs, so handling a second flavor of infinite object many not be too hard.

***More general recursions*** The fact that $\mathsf{ATR}_2$ can express at least some linear continuations is indicative that our basic type system and semantics can be extended to handle a reasonably general class of linear recursion schemes. Nonlinear recursions (e.g. tree recursions) are trickier to handle because there must be independent clocks on each branch of the recursion that together guarantee certain global upper bounds.

***Beyond type-level 2*** There are semantic and complexity-theoretic issues to be resolved in order to extend the semantics of $\mathsf{ATR}_1$ to type-levels 3 and above. The key problem is that (3), our definition of the length of a type-2 function, does not generalize to type-level 3 because for total, continuous $\Psi\colon((\mathsf{N} \to \mathsf{N}) \to \mathsf{N}) \to \mathsf{N}$ and $G\colon(\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$, we can have $\sup\{\, |\psi(F)| \mid |F| \leq |G| \,\} = \infty$, even if $G$ is 0-1 valued. To fix this problem one can introduce difference notion of length that incorporates information about a function's modulus of continuity. It appears that $\mathsf{ATR}_1$ and the $\mathcal{V}_{\mathrm{wt}}$- and $\mathcal{T}$-semantics extend to this new setting. However, it also appears that this new notion of length gives us a new notion of higher-type feasibility that goes beyond the BFFs. Sorting out what is going on here should be fun.

## Acknowledgments

Thanks to Susan Older and Bruce Kapron for repeatedly listening to me describe this work along its evolution. Thanks to Neil Jones and Luke Ong for inviting me to Oxford for a visit and for some extremely helpful comments on an early draft of this paper. Thanks also to the anonymous POPL referees for many helpful comments. Finally many thanks to Peter O'Hearn, Josh Berdine, and the Queen Mary theory group for hosting my visit in the Autumn of 2005 and for repeatedly raking my poor type-systems over the coals until something reasonably simple and civilized survived the ordeals. This work was partially supported by EPSRC grant GR/T25156/01 and NSF grant CCR-0098198.

## References

[1] A. Barber and G. Plotkin, *Dual intuitionistic linear logic*, Tech. report, LFCS, Univ of Edinburgh, 1997.

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

[3] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg, *Characterising polytime through higher type recursion*, Annals of Pure and Applied Logic (2000), 17–30.

[4] 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.

[5] S. Cook and A. Urquhart, *Functional interpretations of feasibly constructive arithmetic*, Annals of Pure and Applied Logic **63** (1993), 103–200.

[6] M. Felleisen and D. Friedman, *Control operators, the SECD-machine, and the lambda calculus*, Formal Descriptions of Programming Concepts III, 1987, pp. 193–217.

[7] C. Frederiksen and N. Jones, *Recognition of polynomial-time programs*, Tech. Report TOPPS/D-501, DIKU, University of Copenhagen, 2004.

[8] O. Goldreich, *Foundations of cryptography, Vol. I: Basic tools*, Cambridge University Press, 2001.

[9] D. J. Gurr, *Semantic frameworks for complexity*, Ph.D. thesis, University of Edinburgh, 1990.

[10] M. Hofmann, *Programming languages capturing complexity classes*, SIGACT News **31** (2000), 31–42.

[11] ———, *Linear types and non-size increasing polynomial time computation*, Information and Computation **183** (2003), 57–85.

[12] R. Irwin, B. Kapron, and J. Royer, *On characterizations of the basic feasible functional, Part I*, Journal of Functional Programming **11** (2001), 117–153.

[13] ———, *On characterizations of the basic feasible functional, Part II*, unpublished manuscript, 2002.

[14] B. Kapron, *Feasible computation in higher types*, Ph.D. thesis, Department of Computer Science, University of Toronto, 1991.

[15] B. Kapron and S. Cook, *A new characterization of type 2 feasibility*, SIAM Journal on Computing **25** (1996), 117–132.

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

[17] ———, *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.

[18] D. Leivant and J.-Y. Marion, *Lambda calculus characterizations of polytime*, FundamentæInformaticæ**19** (1993), 167–184.

[19] J. Longley, *On the ubiquity of certain total type structures (Extended abstract)*, Proceedings of the Workshop on Domains VI (M. Escardó and A. Jung, eds.), Electronic Notes in Theoretical Computer Science, vol. 73, Elsevier Science Publishers, 2004, pp. 87–109.

[20] ———, *Notions of computability at higher types, I*, Logic Colloquium 2000 (R. Cori, A. Razborov, S. Torcevic, and C. Wood, eds.), Lecture Notes in Logic, vol. 19, A. K. Peters, 2005.

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

[22] B. Pierce, *Types and programming languages*, MIT Press, 2002.

[23] G. Plotkin, *Call-by-name, call-by-value and the λ-calculus*, Theoretical Computer Science **1** (1975), 125–159.

[24] ———, *LCF considered as a programming language*, Theoretical Computer Science **5** (1977), 223–255.

[25] J. Royer and J. Case, *Subrecursive programming systems: Complexity & succinctness*, Birkhäuser, 1994.

[26] D. Sands, *Calculi for time analysis of functional programs*, Ph.D. thesis, University of London, 1990.

[27] A. Schönhage, *Storage modification machines*, SIAM Journal on Computing **8** (1980), 490–508.

[28] J. Shultis, *On the complexity of higher-order programs*, Tech. Report CU-CS-288-85, University of Colorado, Boulder, 1985.

[29] K. Van Stone, *A denotational approach to measuring complexity in functional programs*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, 2003.

[30] G. Winskel, *Formal semantics*, MIT Press, 1993.