A tier-based typed programming language characterizing Feasible Functionals

Emmanuel Hainry, Bruce Kapron, Jean-Yves Marion, Romain Péchoux

A TIER-BASED TYPED PROGRAMMING LANGUAGE CHARACTERIZING FEASIBLE FUNCTIONALS

EMMANUEL HAINRY, BRUCE M. KAPRON, JEAN-YVES MARION, AND ROMAIN PÉCHOUX

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
e-mail address: emmanuel.hainry@loria.fr

University of Victoria, Victoria, BC, Canada
e-mail address: bmkapron@uvic.ca

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
e-mail address: jean-yves.marion@loria.fr

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
e-mail address: romain.pechoux@loria.fr

ABSTRACT. The class of Basic Feasible Functionals BFF_2 is the type-2 counterpart of the class FP of type-1 functions computable in polynomial time. Several characterizations have been suggested in the literature, but none of these present a programming language with a type system guaranteeing this complexity bound. We give a characterization of BFF_2 based on an imperative language with oracle calls using a tier-based type system whose inference is decidable. Such a characterization should make it possible to link higher-order complexity with programming theory. The low complexity (cubic in the size of the program) of the type inference algorithm contrasts with the intractability of the aforementioned methods and does not overly constrain the expressive power of the language.

1. INTRODUCTION

Type-2 computational complexity aims to study classes of functions that take type-1 arguments. The notion of feasibility for type-2 functionals was first studied in [Con73] and in [Meh76] using subrecursive formalisms. Later, [CK89, CU93] provided characterizations of polynomial time complexity at all finite types based on programming languages with explicit bounds and applied typed lambda-calculi, respectively. The class characterized in these works was christened the Basic Feasible Functionals, BFF for short.

It was shown in [KC91, KC96] that, similarly to type-1, feasible type-2 functions correspond to the programs computed in time polynomial in the size of their input. In this setting, the polynomial bound is a type-2 function as the size of a type-1 input is itself a type-1 object. This characterization lent support to the notion that at type level 2, the Basic Feasible Functionals (BFF_2) are the correct generalization of FP to type-2.

Nevertheless, these characterizations are faced by at least two problems:

---

(1) Characterizations using a general model of computation (whether machine- or program-based) require externally imposed and explicit resource bounding, either by a type-2 polynomial [KC91, KC96, FHHP15] or a bounding function within the class of [Con73, Meh76]. This is analogous to a shortcoming in Cobham's characterization of the class of (type 1) polynomial time computable functions FP [Cob65]. Such bounding requires either a prior knowledge of program complexity or a check on type-2 polynomial time constraints, which is highly intractable;

(2) There is no natural programming language for these characterizations as they rely on machines or function algebras and cannot be adapted directly to programs. Some attempts have been made to provide programming languages for characterizing BFF_2. These languages are problematic either due to a need to provide some form of explicit external bounding [CK89] or from including unnatural constructs or type-2 recursion patterns [CU93, IRK01, DR06] which severely constrain the way in which type-2 programs may be written. All these distinct approaches would make it difficult for a non-expert programmer to use these formalisms as programming languages.

A solution to Problem (1) was suggested in [KS17] by constraining Cook's definition of Oracle Polynomial Time (OPT) [Coo92], which allows type-1 polynomials to be substituted for type-2 polynomials. To achieve this, oracle Turing machines are required to have a *polynomial step count*: on any input, the length of their computations is bounded by a type-1 polynomial in the size of their input and the maximal size of any answer returned by the oracle. However BFF_2 is known to be strictly included in OPT. In [KS17], OPT is constrained by only allowing computations in which oracle return values increase in size a constant number of times, resulting in a class they called SPT (*strong polynomial time*). This class is strictly contained in BFF_2. BFF_2 is recovered in [KS18] by putting a dual restriction, called *finite lookahead revision*, on machines: on any input, the number of oracle calls on input of increasing size is bounded by a constant. The class of functions computed by machines having polynomial step count and finite lookahead revision is called MPT. The type-2 restriction of the simply-typed lambda closure of functions in MPT (and SPT) characterizes exactly BFF_2.

Problem (2) has been extensively tackled by the Implicit Computational Complexity community for type-1 complexity. This line of work provides machine independent characterizations that eliminate the external explicit bound and was initiated by the seminal works [BC92] and [LM93]. However, none of these works has been adapted to the case of type-2 complexity in a tractable approach. To this day, tractable implicit characterizations of type-2 complexity classes are still missing.

Our contribution. We provide the first tractable characterization of type-2 polynomial time using a typed imperative language with oracle calls. Each oracle call comes with an associated *input bound* which aims at bounding the size of the oracle input. However the size of the oracle answer, which is unpredictable, remains unbounded and, consequently, the language can be used in practice.

The characterization is inspired by the tier-based type system of [Mar11] characterizing FP. Consequently, it relies on a non-interference principle and is also inspired by the type system of [VIS96] guaranteeing confidentiality and integrity policies by ensuring that values of high level variables do not depend on values of low level variables during a program execution. In our context, the level is called a tier.

Let $[\![\mathrm{ST}]\!]$ be the set of functions computed by typable (also called *safe*, see Definition 6) and terminating programs and let $\lambda(X)_2$ be the type-2 restriction of the simply-typed lambda closure of terms with constants in X. The characterization of BFF_2 is as follows:

Theorem 1. $\lambda([\![\mathrm{ST}]\!])_2 = \mathrm{BFF}_2$.

Soundness ($\lambda([\![\mathrm{ST}]\!])_2 \subseteq \mathrm{BFF}_2$, Theorem 33) is demonstrated by showing that each function of $[\![\mathrm{ST}]\!]$ is in Kapron-Steinberg's MPT class [KS18]. The type system makes use of several tiers and is designed to enforce a tier-based non-interference result (Theorem 17) and generalizes the operator type discipline of [Mar11] to ensure the polynomial step count property (Corollary 22) and the finite lookahead revision property (Theorem 25), two non-trivial semantic properties. Two important points to stress are that: i) these properties are enforced statically on programs as consequences of being typable (whereas they were introduced in [KS18] as pure semantic requirements on machines); ii) the enforcement of finite lookahead revision through the use of tiering is a new non-trivial result.

Completeness ($\mathrm{BFF}_2 \subseteq \lambda([\![\mathrm{ST}]\!])_2$, Theorem 41) is shown using an alternative characterization: $\lambda(\mathrm{FP} \cup \{\mathcal{I}'\})_2 = \mathrm{BFF}_2$, where $\mathcal{I}'$ is a bounded iterator that is polynomially equivalent to the recursor $\mathcal{R}$ of [CU93], as demonstrated in [KS19]. The simulation of FP is performed by showing that our type system strictly embeds the tier-based type system of [MP14]. Consequently, our type system also provides a characterization of FP (Theorem 39) with strictly more expressive power when restricted to type-1 programs. Finally, a typable and terminating program computing the bounded iterator functional $\mathcal{I}'$ is exhibited. As in [KS18], the simply-typed lambda-closure is mandatory to achieve completeness as oracle composition is not allowed by the syntax of the language.

The tractability of the type system is proved in Theorem 44, where type inference is shown to be to be solvable in cubic time in the size of the program. As a consequence of the decidability of type inference for simply typed lambda-calculus [Mit91], we obtain the first decidable (up to a termination assumption) programming language based characterization of type-2 polynomial. While the termination assumption is obviously not decidable, it is the most general condition for the result to hold. However, it can be replaced without loss of completeness by combining our type system with automatic termination provers for imperative programs, for example [CPR06, LJB01]. The price to pay is a loss of expressive power. Hence this paper provides a new approach for reasoning about type-2 feasibility automatically, in contrast to related works.

The characterization of Theorem 1 is extensionally complete: all functions of BFF_2 are computed by a typable and terminating program. It is not intensionally complete: there are false negatives as discussed in Example 11. This incompleteness is a consequence of the decidability of type inference as providing intensionally complete descriptions of polynomial time is known to be a Σ_2^0-complete problem in the arithmetical hierarchy [Háj79].

Outline. §4 is devoted to presenting the type system technical developments and main intuitions. §6 states the type system main properties. §5 presents several examples that will help the reader to understand the underlying subtle mechanisms. Soundness and completeness are proved in §7 and §8, respectively. The decidability of type inference is shown in §9. Future work is discussed in §10.

This paper is an extended and improved version of the paper [HKMP20] presented at Logic In Computer Science 2020, including complete proofs.

Expressions	$e, e_1, \ldots$	$::=$	$\mathtt{x} \mid \mathtt{op}(\overline{\mathtt{e}}) \mid \phi(\mathtt{e}_1 \upharpoonright \mathtt{e}_2)$
Commands	c, c_1, c_2	$::=$	$\mathtt{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{c}_1; \mathtt{c}_2 \mid \mathtt{if(e)}\{\mathtt{c}_1\} \; \mathtt{else} \; \{\mathtt{c}_2\}$
			$\mid \mathtt{while(e)}\{\mathtt{c}\}$
Programs	$\mathtt{p}_\phi$	$::=$	$\mathtt{c} \; \mathtt{return} \; \mathtt{x}$

Figure 1. Syntax of imperative programs with oracles

2. Related work

Implicit Computational Complexity (ICC). has lead to the development of several techniques such as interpretations [BMM11], light logics [Gir98], mwp-bounds [BAJK08, JK09], and tiering [Mar11, LM13, HP15]. These tools are restricted to type-1 complexity. Whereas the light logic approach can deal with programs at higher types, its applications are restricted to type-1 complexity classes such as FP [BT04, BM10] or polynomial space [GMR08]. Interpretations were extended to higher-order polynomials in [BL16] to study FP and adapted in [FHHP15, HP17] to BFF_2. However, by essence, all these characterizations use (at least) type-2 polynomials and cannot be considered as tractable.

Other characterizations of BFF_2**.** The characterizations of [CK89, IRK01] are based on a simple imperative programming language that enforces an explicit external bound on the size of oracle outputs within loops. This restriction is impractical from a programming perspective as the size of oracle outputs cannot be predicted. In this paper, the bound is programmer friendly by its implicit nature and because it only constraints the size of the oracle input. Function algebra characterizations were developed in [KS19, CU93]: the recursion schemes are not natural and cannot be used in practice. Several characterizations [KC91, KC96] using type-2 polynomials were also developed but they focus on machines rather than programs.

3. Imperative programming language with oracles

3.1. **Syntax and semantics.** Consider a set $\mathbb{V}$ of variables and a set $\mathbb{O}$ of operators $\mathtt{op}$ of fixed arity $ar(\mathtt{op})$. For notational convenience, operators are used both in infix and prefix notations. Let $\overline{t}$ denote a tuple of n elements (variables, expressions, words, ...) $t_1, \ldots, t_n$, where n is given by the context.

Expressions, commands and programs are defined by the grammar of Figure 1, where $\mathtt{x}, \mathtt{y} \in \mathbb{V}, \mathtt{op}, \upharpoonright \in \mathbb{O}$, and ϕ is an oracle symbol. There can be only one oracle per program. Consequently, each program is indexed by its oracle as subscript.[1]

Let $\mathcal{V}(\mathtt{p}_\phi)$ be the set of variables occurring in the program $\mathtt{p}_\phi$. An expression of the shape $\phi(\mathtt{e}_1 \upharpoonright \mathtt{e}_2)$ is called an *oracle call*. $\mathtt{e}_1$ is called the *input data*, $\mathtt{e}_2$ is called the *input bound* and $\mathtt{e}_1 \upharpoonright \mathtt{e}_2$ is called the *input*. We write $\phi \notin \mathtt{p}_\phi$ in the special case where no oracle call appears in $\mathtt{p}_\phi$.

Let $\mathbb{W} = \Sigma^*$ be the set of words over a finite alphabet Σ such that $\{0, 1\} \subseteq \Sigma$. The symbol ϵ denotes the empty word. The length of a word w (tuple $\overline{t}$) is denoted $|w|$ ($|\overline{t}|$,

---

[1]The results can be generalised naturally to a constant number of oracles. However, this is of no particular interest with respect to the complexity class BFF_2.

respectively). Given two words w and v in $\mathbb{W}$ let $v.w$ denote the concatenation of v and w. For a given symbol $a \in \Sigma$, let a^n be defined inductively by $a^0 = \epsilon$ and $a^{n+1} = a.a^n$. Let $\trianglelefteq$ be the sub-word relation over $\mathbb{W}$, which is defined by $v \trianglelefteq w$, if there are u and u' such that $w = u.v.u'$.

A total function $\llbracket \mathtt{op} \rrbracket : \mathbb{W}^{ar(\mathtt{op})} \to \mathbb{W}$ is associated to each operator. Constants may be viewed as operators of arity zero.

For a given word $w \in \mathbb{W}$ and an integer n, let $w_{\upharpoonright n}$ be the word obtained by truncating w to its first $\min(n, |w|)$ symbols and then padding with a word of the form 10^k to obtain a word of size exactly $n+1$. For example, $1001_{\upharpoonright 0} = 1$, $1001_{\upharpoonright 1} = 11$, $1001_{\upharpoonright 2} = 101$, and $1001_{\upharpoonright 6} = 1001100$. Define $\forall v, w \in \mathbb{W}$, $\llbracket \upharpoonright \rrbracket(v, w) = v_{\upharpoonright |w|}$. Padding ensures that $|\llbracket \upharpoonright \rrbracket(v, w)| = |w| + 1$. The syntax of programs enforces that oracle calls are always performed on input data padded by the input bound. Combined with the above property, this ensures that oracle calls are always performed on input data whose size does not exceed the size of the input bound plus one. Consequently, no oracle call can be performed on the empty word.

The oracle symbol ϕ computes a total function from $\mathbb{W}$ to $\mathbb{W}$, called an oracle function. In order to lighten notations, we will make no distinction between the oracle symbol ϕ and the oracle function it represents.

A store μ is a partial map from $\mathbb{V}$ to $\mathbb{W}$. Let $dom(\mu)$ be the domain of μ. Let $\mu[\mathtt{x}_1 \leftarrow w_1, \ldots, \mathtt{x}_n \leftarrow w_n]$ be a notation for the store μ' satisfying $\forall \mathtt{x} \in dom(\mu) - \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, $\mu'(\mathtt{x}) = \mu(\mathtt{x})$ and $\forall \mathtt{x}_i \in \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, $\mu'(\mathtt{x}_i) = w_i$. Let μ_0 be the store defined by $dom(\mu_0) = \mathbb{V}$ and $\forall \mathtt{x} \in dom(\mu_0)$, $\mu_0(\mathtt{x}) = \epsilon$. The size of a store μ is defined by $|\mu| = \Sigma_{\mathtt{x} \in dom(\mu)} |\mu(\mathtt{x})|$.

The judgment $\mu \vDash_\phi \mathtt{e} \to w$ means that the expression $\mathtt{e}$ is evaluated to the word $w \in \mathbb{W}$ with respect to the store μ and the oracle ϕ. The judgment $\mu \vDash_\phi \mathtt{c} \to \mu'$ expresses that, under the store μ and the oracle ϕ, the command $\mathtt{c}$ terminates and outputs the store μ'. As the oracle is fixed for each program, we will omit it throughout the paper in the judgments subscript, $e.g.$, writing $\mu \vDash \mathtt{e} \to w$ for $\mu \vDash_\phi \mathtt{e} \to w$. The operational semantics of the language is deterministic and is given in Figure 2. In rule (Seq) of Figure 2, it is implicitly assumed that $\mathtt{c}_1$ is not a sequence.

A *derivation* $\pi_\phi : \mu \vDash \mathtt{p}_\phi \to w$ is a tree rooted at $\mu \vDash \mathtt{p}_\phi \to w$, where children of each node are obtained by applying the rules of Figure 2. Let $|\pi_\phi|$ denote the size of the derivation π_ϕ. Note that $|\pi_\phi|$ corresponds to the number of steps in a sequential execution of $\mathtt{p}_\phi$, initialized with store μ. Hence, $|\pi_\phi|$ can be infinite. With no restriction on operators, this measure is too coarse to correspond, even asymptotically, to running time. With suitable restrictions, there is a correspondence, given in Proposition 30 below.

A program $\mathtt{p}_\phi$ such that $\mathcal{V}(\mathtt{p}_\phi) = \{\overline{\mathtt{x}}\}$ computes the partial function $\llbracket \mathtt{p}_\phi \rrbracket \in \mathbb{W}^{|\overline{\mathtt{x}}|} \to \mathbb{W}$, defined by $\llbracket \mathtt{p}_\phi \rrbracket(\overline{w}) = w$ if $\exists \pi_\phi$, $\pi_\phi : \mu_0[\mathtt{x}_1 \leftarrow w_1, \ldots, \mathtt{x}_{|\overline{\mathtt{x}}|} \leftarrow w_{|\overline{\mathtt{x}}|}] \vDash \mathtt{p}_\phi \to w$. In the special case where, for any oracle ϕ, $\llbracket \mathtt{p}_\phi \rrbracket$ is a total function, the program $\mathtt{p}_\phi$ is said to be terminating.

A second order function $f : (\mathbb{W} \to \mathbb{W}) \to (\mathbb{W} \to \mathbb{W})$ is computed by a program $\mathtt{p}_\phi$ if for any oracle function $\phi \in \mathbb{W} \to \mathbb{W}$ and word $w \in \mathbb{W}$, $f(\phi)(w) = \llbracket \mathtt{p}_\phi \rrbracket(w)$.

3.2. **Neutral and positive operators.** We define two classes of operators called neutral and positive. This categorization of operators will be used in §4.2 where the admissible types for operators will depend on their category in the type system.

Definition 2 (Neutral and positive operators)**.**
- *An operator* $\mathtt{op}$ *is* neutral *if:*
  (1) *either* $\mathtt{op}$ *is a constant operator,* i.e.*,* $ar(\mathtt{op}) = 0$;

$$\frac{}{\mu \vDash \mathtt{x} \to \mu(\mathtt{x})} \ (\text{Var}) \qquad \frac{\forall i \leq ar(\mathtt{op}), \ \mu \vDash \mathtt{e}_i \to w_i}{\mu \vDash \mathtt{op}(\overline{\mathtt{e}}) \to [\![\mathtt{op}]\!](\overline{w})} \ (\text{Op})$$

$$\frac{\mu \vDash \mathtt{e}_1 \to v \quad \mu \vDash \mathtt{e}_2 \to w \quad \phi([\![\upharpoonright]\!](v, w)) = u}{\mu \vDash \phi(\mathtt{e}_1 \upharpoonright \mathtt{e}_2) \to u} \ (\text{Orc})$$

$$\frac{}{\mu \vDash \mathtt{skip} \to \mu} \ (\text{Skip}) \qquad \frac{\mu \vDash \mathtt{e} \to w}{\mu \vDash \mathtt{x} := \mathtt{e} \to \mu[\mathtt{x} \leftarrow w]} \ (\text{Asg})$$

$$\frac{\mu \vDash \mathtt{c}_1 \to \mu_1 \quad \mu_1 \vDash \mathtt{c}_2 \to \mu_2}{\mu \vDash \mathtt{c}_1; \ \mathtt{c}_2 \to \mu_2} \ (\text{Seq}) \qquad \frac{\mu \vDash \mathtt{e} \to w \quad \mu \vDash \mathtt{c}_w \to \mu' \quad w \in \{0,1\}}{\mu \vDash \mathtt{if}(\mathtt{e})\{\mathtt{c}_1\} \ \mathtt{else} \ \{\mathtt{c}_0\} \to \mu'} \ (\text{Cond})$$

$$\frac{\mu \vDash \mathtt{e} \to 0}{\mu \vDash \mathtt{while}(\mathtt{e})\{\mathtt{c}\} \to \mu} \ (\text{Wh}_0) \qquad \frac{\mu \vDash \mathtt{e} \to 1 \quad \mu \vDash \mathtt{c}; \ \mathtt{while}(\mathtt{e})\{\mathtt{c}\} \to \mu'}{\mu \vDash \mathtt{while}(\mathtt{e})\{\mathtt{c}\} \to \mu'} \ (\text{Wh}_1)$$

$$\frac{\mu \vDash \mathtt{c} \to \mu'}{\mu \vDash \mathtt{c} \ \mathtt{return} \ \mathtt{x} \to \mu'(\mathtt{x})} \ (\text{Prg})$$

FIGURE 2. Big step operational semantics

(2) $[\![\mathtt{op}]\!] : \mathbb{W}^{ar(\mathtt{op})} \to \{0,1\}$, i.e., $[\![\mathtt{op}]\!]$ *is a predicate;*
(3) *or* $\forall \overline{w} \in \mathbb{W}^{ar(\mathtt{op})}, \ \exists i \leq ar(\mathtt{op}), \ [\![\mathtt{op}]\!](\overline{w}) \trianglelefteq w_i$;
- *An operator* $\mathtt{op}$ *is* positive *if there is a constant* $c_{\mathtt{op}}$ *such that:*
  $\forall \overline{w}^{ar(\mathtt{op})} \in \mathbb{W}, \ |[\![\mathtt{op}]\!](\overline{w})| \leq \max_i |w_i| + c_{\mathtt{op}}.$

A neutral operator is always a positive operator but the converse is not true. In the remainder, we name positive operators those operators that are positive but not neutral.

Example 3. *The operator* $==$ *tests whether or not its arguments are equal and the operator* `pred` *computes the predecessor.*

$$[\![==]\!](w, v) = \begin{cases} 1 & \textit{if } v = w \\ 0 & \textit{otherwise} \end{cases} \qquad [\![\mathtt{pred}]\!](v) = \begin{cases} \epsilon & \textit{if } v = \epsilon \\ u & \textit{if } v = a.u, \ a \in \Sigma \end{cases}$$

Both operators are neutral. $[\![\mathtt{suc}_i]\!](v) = i.v, \textit{ for } i \in \{0,1\}, \textit{ is a positive operator since}$ $|[\![\mathtt{suc}_i]\!](v)| = |i.v| = |v| + 1.$

4. TYPE SYSTEM

In this section, we introduce a tier based type system, the main contribution of the paper, that allows to provide a characterization of type-2 polynomial time complexity ([Meh76, KC91, KC96]).

4.1. **Tiers and typing judgments.** Atomic types are elements of the totally ordered set $(\mathbf{N}, \preceq, \mathbf{0}, \vee, \wedge)$ where $\mathbf{N} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$ is the set of natural numbers, called *tiers*, in accordance with the data ramification principle of [Lei95], $\preceq$ is the usual ordering on integers and $\vee$ and $\wedge$ are the max and min operators over integers. Let $\prec$ be defined by $\prec := \preceq \cap \neq$. We use the symbols $\mathbf{t}, \mathbf{t}', \ldots, \mathbf{t}_1, \mathbf{t}_2, \ldots$ to denote tier variables. For a finite set of tiers, $\{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$, let $\vee_{i=1}^n \mathbf{t}_i$ ($\wedge_{i=1}^n \mathbf{t}_i$, respectively) denote $\mathbf{t}_1 \vee \ldots \vee \mathbf{t}_n$ ($\mathbf{t}_1 \wedge \ldots \wedge \mathbf{t}_n$, respectively).

A variable typing environment Γ is a finite mapping from $\mathbb{V}$ to $\mathbf{N}$, which assigns a single tier to each variable.

An operator typing environment Δ is a mapping that associates to each operator op and each tier $\mathbf{t} \in \mathbf{N}$ a set of admissible operator types $\Delta(\mathsf{op})(\mathbf{t})$, where the operator types corresponding to the operator op are of the shape $\mathbf{t}_1 \to \ldots \to \mathbf{t}_{ar(\mathsf{op})} \to \mathbf{t}'$, with $\mathbf{t}_i, \mathbf{t}' \in \mathbf{N}$.

Let $dom(\Gamma)$ (resp. $dom(\Delta)$) denote the set of variables typed by Γ (resp. operators typed by Δ).

Typing judgments are either *command typing judgments* of the shape $\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$ or *expression typing judgments* of the shape $\Gamma, \Delta \vdash \mathsf{e} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$. The intended meaning of such a typing judgment is that the *expression tier* or *command tier* is $\mathbf{t}$, the *innermost tier* is $\mathbf{t}_{in}$, and the *outermost tier* is $\mathbf{t}_{out}$. The innermost (resp. outermost) tier is the tier of the guard of the innermost (resp. outermost) while loop containing the expression or command in question. In the case of a single non-nested while loop, the innermost and outermost tiers are equal (as illustrated by rule (W_0) of Figure 3). These two tiers are irrelevant for an expression or a command not appearing inside a while loop.

The type system preventing flows from $\mathbf{t}_2$ to $\mathbf{t}_1$, whenever $\mathbf{t}_2 \prec \mathbf{t}_1$ holds, is presented in Figure 3.

A typing derivation $\rho \triangleright \Gamma, \Delta \vdash \mathsf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$ is a tree whose root is the typing judgment $\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$ and whose children are obtained by applications of the typing rules. Due to the rule (OP) of Figure 3, that allows several admissible types for operators, typing derivations are, in general, not unique. However the two typing rules for while loops (W) and (W_0) are mutually exclusive (when read bottom-up) because of the non-overlapping requirements for $\mathbf{t}_{out}$ in Figure 3. The notation ρ will be used whenever mentioning the root of a typing derivation is not explicitly needed. We use the notation $\rho \triangleright \Gamma, \Delta \vdash \mathsf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$ (R) to denote the typing derivation $\rho \triangleright \Gamma, \Delta \vdash \mathsf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$ whose children are obtained by application of a typing rule labelled by (R).

Given two typing derivations ρ and ρ', we write $\rho' \leq \rho$ (respectively $\rho' < \rho$) if ρ' is a (strict) subtree of ρ. Let $\mathcal{D}(\rho)$ be defined by $\mathcal{D}(\rho) = \{\rho' \mid \rho' \leq \rho\}$ and let $\mathring{\mathcal{D}}(\rho)$ be defined by $\mathring{\mathcal{D}}(\rho) = \mathcal{D}(\rho) - \{\rho\}$.

4.2. **Safe environments and programs.** The typing rules of Figure 3 are not restrictive enough in themselves to guarantee polynomial time computation, even for type-1. Indeed operators need to be restricted to prevent exponential programs from being typable (see counter-Example 9). The current subsection introduces such a restriction, called *safe*.

Definition 4 (Safe operator typing environment). *An operator typing environment Δ is safe if for each $\mathsf{op} \in dom(\Delta)$ of arity $ar(\mathsf{op}) > 0$, op is neutral or positive and $[\![\mathsf{op}]\!]$ is a polynomial time computable function, and for each $\mathbf{t}_{in} \in \mathbf{N}$, and for each $\mathbf{t}_1 \to \ldots \mathbf{t}_{ar(\mathsf{op})} \to \mathbf{t} \in \Delta(\mathsf{op})(\mathbf{t}_{in})$, the two conditions below hold:*

(1) $\mathbf{t} \preceq \wedge_{i=1}^{ar(\mathsf{op})} \mathbf{t}_i \preceq \vee_{i=1}^{ar(\mathsf{op})} \mathbf{t}_i \preceq \mathbf{t}_{in}$,
(2) *if the operator op is positive then $\mathbf{t} \prec \mathbf{t}_{in}$.*

$$\frac{\Gamma(\mathtt{x}) = \mathbf{t}}{\Gamma, \Delta \vdash \mathtt{x} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{V})$$

$$\frac{\mathbf{t}_1 \to \cdots \to \mathbf{t}_{ar(\mathtt{op})} \to \mathbf{t} \in \Delta(\mathtt{op})(\mathbf{t}_{in}) \quad \forall i \le ar(\mathtt{op}), \ \Gamma, \Delta \vdash \mathtt{e}_i : (\mathbf{t}_i, \mathbf{t}_{in}, \mathbf{t}_{out})}{\Gamma, \Delta \vdash \mathtt{op}(\overline{\mathtt{e}}) : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{OP})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e}_1 : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{e}_2 : (\mathbf{t}_{out}, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \mathbf{t} \prec \mathbf{t}_{in} \wedge \mathbf{t} \preceq \mathbf{t}_{out}}{\Gamma, \Delta \vdash \phi(\mathtt{e}_1 \upharpoonright \mathtt{e}_2) : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{OR})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})}{\Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t+1}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{SUB}) \qquad \frac{}{\Gamma, \Delta \vdash \mathtt{skip} : (\mathbf{0}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{SK})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{x} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{e} : (\mathbf{t}_2, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \mathbf{t}_1 \preceq \mathbf{t}_2}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{e} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{A})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c}_1 : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{c}_2 : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})}{\Gamma, \Delta \vdash \mathtt{c}_1; \mathtt{c}_2 : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{S})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{c}_1 : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{c}_0 : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})}{\Gamma, \Delta \vdash \mathtt{if}(\mathtt{e})\{\mathtt{c}_1\} \ \mathtt{else} \ \{\mathtt{c}_0\} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{C})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t}, \mathbf{t}, \mathbf{t}_{out}) \quad \mathbf{1} \preceq \mathbf{t} \preceq \mathbf{t}_{out}}{\Gamma, \Delta \vdash \mathtt{while}(\mathtt{e})\{\mathtt{c}\} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (\mathrm{W})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}) \quad \Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t}, \mathbf{t}, \mathbf{t}) \quad \mathbf{1} \preceq \mathbf{t}}{\Gamma, \Delta \vdash \mathtt{while}(\mathtt{e})\{\mathtt{c}\} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{0})} \ (\mathrm{W}_0)$$

FIGURE 3. Tier-based type system

Example 5. *Consider the operators* ==, pred *and* $\mathtt{suc}_i$ *of Example 3. For a safe typing environment* Δ, *it holds that* $\Delta(==)(\mathbf{1}) = \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}\} \cup \{\mathbf{t} \to \mathbf{t}' \to \mathbf{0} \mid \mathbf{t}, \mathbf{t}' \preceq \mathbf{1}\}$, *as* == *is neutral. However* $\mathbf{0} \to \mathbf{1} \to \mathbf{1} \notin \Delta(==)(\mathbf{1})$ *as it breaks Condition (1) of Definition 4 since the operator output tier has to be smaller than each of its operand tier (i.e.,* $\mathbf{1} \npreceq \mathbf{0} \wedge \mathbf{1}$).

It also holds that $\Delta(\mathtt{pred})(\mathbf{2}) = \{\mathbf{2} \to \mathbf{t} \mid \mathbf{t} \preceq \mathbf{2}\} \cup \{\mathbf{1} \to \mathbf{t} \mid \mathbf{t} \preceq \mathbf{1}\} \cup \{\mathbf{0} \to \mathbf{0}\}$.

For the positive operator $\mathtt{suc}_i$, *we have* $\Delta(\mathtt{suc}_i)(\mathbf{1}) = \{\mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$. $\mathbf{1} \to \mathbf{1} \notin \Delta(\mathtt{suc}_i)(\mathbf{1})$ *as the operator output tier has to be strictly smaller than* $\mathbf{1}$, *due to Condition (2) of Definition 4. Applying the same restriction, it holds that* $\Delta(\mathtt{suc}_i)(\mathbf{2}) = \{\mathbf{2} \to \mathbf{1}, \mathbf{2} \to \mathbf{0}, \mathbf{1} \to \mathbf{1}, \mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$.

Definition 6 (Safe program). *Given* Γ *a variable typing environment and* Δ *a safe operator typing environment, the program* $\mathtt{p}_\phi = \mathtt{c} \ \mathtt{return} \ \mathtt{x}$ *is a safe program if there are* $\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out}$ *such that* $\rho \rhd \Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$.

Definition 7. *Let* ST *be the set of safe and terminating programs and* $[\![\text{ST}]\!]$ *be the set of functionals computed by programs in* ST*:*

$$[\![\text{ST}]\!] = \{\lambda\phi.\lambda w_1.\cdots\lambda w_n.[\![\mathbf{p}_\phi]\!](w_1,\ldots,w_n) \mid \mathbf{p}_\phi \in \text{ST}\}.$$

4.3. **Some intuitions.** Before providing a formal treatment of the type system's main properties in §6, we provide the reader with a brief intuition of types, that are triplets of tiers $(\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$, in a typing derivation obtained by applying the typing rules of Figure 3:

- $\mathbf{t}$ is the tier of the expression or command under consideration. It is used to prevent data flows from lower tiers to higher tiers in control flow commands and assignments. By safety and by rules (OP) and (OR), expression tiers are structurally decreasing. Consequently, rule (A) ensures that data can only flow from higher tiers to lower tiers. Command tiers are structurally increasing and, consequently, an assignment of a higher tier variable can never be controlled by a lower tier in a conditional or while command. The subtyping rule (SUB) for commands follows this discipline by allowing a command of tier $\mathbf{t}$ to be considered as a tier $\mathbf{t+1}$ command and, hence, controlled by an expression of tier $\mathbf{t+1}$. However subtyping is strictly prohibited for expressions as this would break the flow.

- $\mathbf{t}_{in}$ is the tier of the innermost while loop containing the expression or command under consideration, provided it exists. It is used to allow declassification (*i.e.*, a release of some information at a lower tier to a higher tier) to occur in the program by allowing an operator to have types depending on the context. Moreover, the innermost tier restricts the return types of operators and oracle calls:
  - in rule (OR), the return type $\mathbf{t}$ is strictly smaller than $\mathbf{t}_{in}$,
  - in rule (OP), for a positive operator, the return type $\mathbf{t}$ is strictly smaller than $\mathbf{t}_{in}$.
  This forbids programs from iterating on a data whose size can increase during the iteration.

- $\mathbf{t}_{out}$ is the tier of the outermost while loop containing the expression or command under consideration, provided it exists. Its purpose is to bound by a constant the number of lookahead revisions (that is the number of times a query to the oracle may increase in size) allowed in oracle calls. By rule (OR), all oracle input bounds have a tier equal to the tier of the outermost while loop where they are called. Hence, the size of the data stored in the input bound cannot increase in a fixed while loop and it can increase at most a constant number of times.

There are two rules (W) and (W$_0$) for while loops. (W) is the standard rule and updates the innermost tier with the tier of the while loop guard under consideration. (W$_0$) is an initialization rule that allows the programmer to instantiate by default the main command with outermost tier $\mathbf{0}$ as it has no outermost while. It could be sacrificed for simplicity but at the price of a worst expressive power.

5. Examples

In this section, we provide several examples and counter-examples, starting with programs with no oracle calls in order to illustrate how the type system works. Some of its restrictions in terms of expressive power are also discussed in Example 11. In the typing derivations, we sometimes omit the environments, writing $\vdash$ instead of $\Gamma, \Delta \vdash$ in order to lighten the notations. Moreover, for notational convenience, we will use labels for expression tiers. For example, $\mathbf{e^t}$ means that $\mathbf{e}$ is of tier $\mathbf{t}$. Also, to make the presentation of the examples lighter, we will work over the unary integers rather than all of $\mathbb{W}$. In particular, a value $\mathbf{v}$ denotes

$1^{\mathbf{v}}$, and in particular 0 denotes ϵ. Also, with this convention, $[\![\mathtt{pred}]\!](\mathtt{v}) = \max\{0, \mathtt{v} - 1\}$ and $[\![\mathtt{suc}_1]\!](\mathtt{v}) = \mathtt{v} + 1$.

Example 8 (Addition). *Consider the simple program below, with no oracle, computing the unary addition.*

```
while(x > 0)¹{
   x¹ := pred(x)¹;
   y⁰ := suc₁(y)⁰
}
return  y
```

This program is safe with respect to the following typing derivation:

$$
\cfrac{\cfrac{\cfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (V)}{\vdash \mathtt{x} > 0 : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (OP) \qquad \cfrac{\rho_1 \rhd \vdash \mathtt{x} := \mathtt{pred}(\mathtt{x}) : (\mathbf{1}, \mathbf{1}, \mathbf{1}) \quad \rho_2 \rhd \vdash \mathtt{y} := \mathtt{suc}_1(\mathtt{y}) : (\mathbf{0}, \mathbf{1}, \mathbf{1})}{\vdash \mathtt{x} := \mathtt{pred}(\mathtt{x}); \ \mathtt{y} := \mathtt{suc}_1(\mathtt{y}) : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (S)}{\vdash \mathtt{while}(\mathtt{x} > 0)\{\mathtt{x} := \mathtt{pred}(\mathtt{x}); \ \mathtt{y} := \mathtt{suc}_1(\mathtt{y})\} : (\mathbf{1}, \mathbf{1}, \mathbf{0})}\ (W_0)
$$

The while loop is guarded by $\mathtt{x} > 0$. If the main command is typed by $(\mathbf{1}, \mathbf{1}, \mathbf{0})$ then the expression $\mathtt{x} > 0$ is of tier $\mathbf{1}$ by the typing rule (W_0). Consequently, the variable $\mathtt{x}$ is forced to be of tier $\mathbf{1}$ using the type $\mathbf{1} \to \mathbf{1}$ for the operator > 0 in the (OP) rule. $\mathbf{1} \to \mathbf{1} \in \Delta(> 0)(\mathbf{1})$ holds as the operator > 0 is neutral. One application of the subtyping rule (SUB) is performed for the sequence to be typed as the subcommands are required to have homogeneous types.

The typing derivation ρ_1 is as follows:

$$
\cfrac{\cfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (V) \quad \cfrac{\cfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (V)}{\vdash \mathtt{pred}(\mathtt{x}) : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (OP)}{\rho_1 \rhd \vdash \mathtt{x} := \mathtt{pred}(\mathtt{x}) : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (A)
$$

In ρ_1, the $\mathtt{pred}$ operator is used with the type $\mathbf{1} \to \mathbf{1}$ in the (OP) rule. This use is authorized as, $\mathtt{pred}$ is neutral and, consequently, $\mathbf{1} \to \mathbf{1} \in \Delta(\mathtt{pred})(\mathbf{1})$. As a consequence, the rule (A) in ρ_1 can be derived as the tier of the assigned variable $\mathtt{x}$ (equal to $\mathbf{1}$) is smaller than the tier of the expression $\mathtt{pred}(\mathtt{x})$ (also equal to $\mathbf{1}$).

The second typing derivation ρ_2 is as follows:

$$
\cfrac{\cfrac{\Gamma(\mathtt{y}) = \mathbf{0}}{\vdash \mathtt{y} : (\mathbf{0}, \mathbf{1}, \mathbf{1})}\ (V) \quad \cfrac{\cfrac{\Gamma(\mathtt{y}) = \mathbf{0}}{\vdash \mathtt{y} : (\mathbf{0}, \mathbf{1}, \mathbf{1})}\ (V)}{\vdash \mathtt{suc}_1(\mathtt{y}) : (\mathbf{0}, \mathbf{1}, \mathbf{1})}\ (OP)}{\rho_2 \rhd \vdash \mathtt{y} := \mathtt{suc}_1(\mathtt{y}) : (\mathbf{0}, \mathbf{1}, \mathbf{1})}\ (A)
$$

The only distinction between ρ_2 and ρ_1 is that the operator $\mathtt{suc}_1$ is positive. Consequently, with an innermost tier of $\mathbf{1}$, the type $\mathbf{1} \to \mathbf{1}$ is not authorized for such an operator (since $\mathbf{1} \to \mathbf{1} \notin \Delta(\mathtt{suc}_1)(\mathbf{1})$). Indeed, by Example 5, $\Delta(\mathtt{suc}_1)(\mathbf{1}) = \{\mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$. The type $\mathbf{1} \to \mathbf{0}$ is ruled out as it would require a non-homogeneous type for $\mathtt{y}$. Consequently, the rule (OP) is applied on type $\mathbf{0} \to \mathbf{0}$ and the variable $\mathtt{y}$ must be of tier $\mathbf{0}$. Notice that the program could also be typed by assigning higher tiers $\mathbf{t}$ and $\mathbf{t}'$ such that $\mathbf{t}' \prec \mathbf{t}$, to $\mathtt{x}$ and $\mathtt{y}$, respectively.

Example 9 (Exponential)**.** *The program below, computing the exponential, is not safe.*

```
while(x > 0){
   z := y;
   while(z > 0){
      z := pred(z);
      y := suc₁(y)
   };
   x := pred(x)
}
return  y
```

By contradiction, suppose that it can be typed with respect to the typing environments Γ *and* Δ*. Let* $\Gamma(\mathbf{x}), \Gamma(\mathbf{y})$ *and* $\Gamma(\mathbf{z})$ *be* $\mathbf{t_x}, \mathbf{t_y}$ *and* $\mathbf{t_z}$*, respectively.*

The subcommand $\mathbf{z} := \mathbf{y}$ *enforces* $\mathbf{t_z} \preceq \mathbf{t_y}$ *to be satisfied for the following typing derivation to hold.*

$$\cfrac{\cfrac{\Gamma(\mathbf{z}) = \mathbf{t_z}}{\vdash \mathbf{z} : (\mathbf{t_z}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (V) \qquad \cfrac{\Gamma(\mathbf{y}) = \mathbf{t_y}}{\vdash \mathbf{y} : (\mathbf{t_y}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (V)}{\rho_1 \rhd\, \vdash \mathbf{z} := \mathbf{y} : (\mathbf{t_z}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (A)$$

The subcommand $\mathbf{y} := \mathbf{suc_1}(\mathbf{y})$ *enforces the constraint* $\mathbf{t_y} \prec \mathbf{t}_{in}$*,* $\mathbf{t}_{in}$ *being the command innermost tier, for the typing derivation to hold.*

$$\cfrac{\cfrac{\Gamma(\mathbf{y}) = \mathbf{t_y}}{\vdash \mathbf{y} : (\mathbf{t_y}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (V) \qquad \cfrac{\mathbf{t_y} \to \mathbf{t_y} \in \Delta(\mathbf{suc_1})(\mathbf{t}_{in}) \qquad \cfrac{\Gamma(\mathbf{y}) = \mathbf{t_y}}{\vdash \mathbf{y} : (\mathbf{t_y}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (V)}{\vdash \mathbf{suc_1}(\mathbf{y}) : (\mathbf{t_y}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (OP)}{\rho_2 \rhd\, \vdash \mathbf{y} := \mathbf{suc_1}(\mathbf{y}) : (\mathbf{t_y}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (A)$$

Indeed, as $\mathbf{suc_1}$ *is a positive operator, by Condition 2 of Definition 4,* $\mathbf{t_y} \prec \mathbf{t}_{in}$ *has to be satisfied for* $\mathbf{t_y} \to \mathbf{t_y} \in \Delta(\mathbf{suc_1})(\mathbf{t}_{in})$ *to hold.*

The innermost while loop enforces the constraint $\mathbf{t}_{in} \preceq \mathbf{t_z}$ *in the following typing derivation.*

$$\cfrac{\cfrac{\cfrac{\Gamma(\mathbf{z}) = \mathbf{t_z}}{\vdash \mathbf{z} : (\mathbf{t_z}, \mathbf{t}'_{in}, \mathbf{t}_{out})}\ (V)}{\vdash \mathbf{z} > 0 : (\mathbf{t}_{in}, \mathbf{t}'_{in}, \mathbf{t}_{out})}\ (OP) \qquad \cfrac{\vdots}{\vdash \mathbf{z} := \mathbf{pred}(\mathbf{z}); \mathbf{y} := \mathbf{suc_1}(\mathbf{y}) : (\mathbf{t}_{in}, \mathbf{t}_{in}, \mathbf{t}_{out})}\ (S)}{\rho_3 \rhd\, \vdash \mathbf{while}(\mathbf{z} > 0)\{\mathbf{z} := \mathbf{pred}(\mathbf{z}); \mathbf{y} := \mathbf{suc_1}(\mathbf{y})\} : (\mathbf{t}_{in}, \mathbf{t}'_{in}, \mathbf{t}_{out})}\ (W)$$

First, notice that only the rule (W) can be applied to this typing derivation as the corresponding subcommand is already contained inside a while loop and, consequently, $\mathbf{1} \preceq \mathbf{t}_{out}$ *is enforced by the outermost while loop using rule (W) or rule* (W_0)*. Second, the tier of this subcommand is equal to the innermost tier* $\mathbf{t}_{in}$ *of subcommand* $\mathbf{y} := \mathbf{suc_1}(\mathbf{y})$ *(in* ρ_2*). Indeed, rules (W) and* (W_0) *are the only typing rules updating the innermost tier and there is no while loop in between. Finally, in the rule (OP), as* > 0 *is neutral, Condition 1 of Definition 4 enforces that* $\mathbf{t}_{in} \preceq \mathbf{t_z} \preceq \mathbf{t}'_{in}$ *holds for the program to be typed.*

Putting all the above constraints together, we obtain the contradiction $\mathbf{t_z} \preceq \mathbf{t_y} \prec \mathbf{t}_{in} \preceq \mathbf{t_z}$*. Consequently, the program cannot be typed.*

Example 10 (Multiple tiers). *Consider the following program illustrating the use of multiple tiers.*

```
c₁ : while(x > 0)²{
        x² := pred(x)²;
        y¹ := suc₁(suc₁(y))¹
    } ;
c₂ : while(y > 0)¹{
        y¹ := pred(y)¹;
        z⁰ := suc₁(suc₁(z))⁰
    }
    return  z
```

The program is safe with respect to the variable typing environment Γ such that $\Gamma(\mathtt{x}) = \mathbf{2}$, $\Gamma(\mathtt{y}) = \mathbf{1}$ and $\Gamma(\mathtt{z}) = \mathbf{0}$. The main command can be typed by $(\mathbf{2}, \mathbf{2}, \mathbf{0})$ as illustrated below, provided that $\mathtt{c_1}$ and $\mathtt{c_2}$ are the commands corresponding to the first while loop and second while loop, respectively.

$$\cfrac{\cfrac{\begin{matrix}\rho_1 \\ \vdots\end{matrix}}{\rho_1 \rhd\, \vdash \mathtt{c_1} : (\mathbf{2}, \mathbf{2}, \mathbf{0})}\ (W_0) \qquad \cfrac{\begin{matrix}\rho_2 \\ \vdots\end{matrix}}{\rho_2 \rhd\, \vdash \mathtt{c_2} : (\mathbf{2}, \mathbf{2}, \mathbf{0})}\ (W_0)}{\vdash \mathtt{c_1};\ \mathtt{c_2} : (\mathbf{2}, \mathbf{2}, \mathbf{0})}\ (S)$$

The typing derivation ρ_1 corresponds to the first while loop $\mathtt{c_1}$ and is described below.

$$\cfrac{\cfrac{\cfrac{\Gamma(\mathtt{x}) = \mathbf{2}}{\vdash \mathtt{x} : (\mathbf{2}, \mathbf{2}, \mathbf{2})}\ (V)}{\vdash \mathtt{x} > 0 : (\mathbf{2}, \mathbf{2}, \mathbf{2})}\ (OP) \quad \cfrac{\cfrac{\cfrac{\begin{matrix}\rho_1^1 \\ \vdots\end{matrix}}{\vdash \mathtt{x} := \mathtt{pred(x)} : (\mathbf{2}, \mathbf{2}, \mathbf{2})}\ (A) \quad \cfrac{\cfrac{\cfrac{\begin{matrix}\rho_2^1 \\ \vdots\end{matrix}}{\vdash \mathtt{y} := \mathtt{suc_1(suc_1(y))} : (\mathbf{1}, \mathbf{2}, \mathbf{2})}\ (A)}{\vdash \mathtt{y} := \mathtt{suc_1(suc_1(y))} : (\mathbf{2}, \mathbf{2}, \mathbf{2})}\ (SUB)}{\vdash \mathtt{x} := \mathtt{pred(x)};\ \mathtt{y} := \mathtt{suc_1(suc_1(y))} : (\mathbf{2}, \mathbf{2}, \mathbf{2})}\ (S)}{}}{\vdash \mathtt{c_1} : (\mathbf{2}, \mathbf{2}, \mathbf{0})}\ (W_0)$$

The typing derivation ρ_1^1 can be built easily using rules (A), (OP), and (V) as $\mathtt{pred}$ is neutral and can be 0.9given the type $\mathbf{2} \to \mathbf{2}$ in $\Delta(\mathtt{pred})(\mathbf{2})$ (see Example 5). The typing derivation ρ_2^1 can be built using the same rules as $\mathtt{suc_1}$ is positive and can be given the type $\mathbf{1} \to \mathbf{1}$ in $\Delta(\mathtt{suc_1})(\mathbf{2})$ (see Example 5 again). ρ_2^1 requires the prior application of subtyping rule (SUB) as the tier of the assignment is equal to $\Gamma(\mathtt{y}) = \mathbf{1}$.

The typing derivation ρ_2, described below,

$$\cfrac{\cfrac{\cfrac{\Gamma(\mathtt{y}) = \mathbf{1}}{\vdash \mathtt{y} : (\mathbf{1}, \mathbf{2}, \mathbf{1})}\ (V)}{\vdash \mathtt{y} > 0 : (\mathbf{1}, \mathbf{2}, \mathbf{1})}\ (OP) \quad \cfrac{\cfrac{\cfrac{\begin{matrix}\rho_1^2 \\ \vdots\end{matrix}}{\vdash \mathtt{y} := \mathtt{pred(y)} : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (A) \quad \cfrac{\cfrac{\cfrac{\begin{matrix}\rho_2^2 \\ \vdots\end{matrix}}{\vdash \mathtt{z} := \mathtt{suc_1(suc_1(z))} : (\mathbf{0}, \mathbf{1}, \mathbf{1})}\ (A)}{\vdash \mathtt{z} := \mathtt{suc_1(suc_1(z))} : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (SUB)}{\vdash \mathtt{y} := \mathtt{pred(y)};\ \mathtt{z} := \mathtt{suc_1(suc_1(z))} : (\mathbf{1}, \mathbf{1}, \mathbf{1})}\ (S)}{\cfrac{\vdash \mathtt{c_2} : (\mathbf{1}, \mathbf{2}, \mathbf{0})}{\vdash \mathtt{c_2} : (\mathbf{2}, \mathbf{2}, \mathbf{0})}\ (SUB)}\ (W_0)}{}}$$

can be obtained in a similar way by taking the type $\mathbf{1} \to \mathbf{1}$ for the neutral operator $\mathtt{pred}$ *in* $\Delta(\mathtt{pred})(\mathbf{1})$ *and the type* $\mathbf{0} \to \mathbf{0}$ *for the positive operator* $\mathtt{suc}_1$ *in* $\Delta(\mathtt{suc}_1)(\mathbf{1})$. *The initial subtyping rule is required as it is not possible to derive* $\vdash \mathtt{y} > 0 : (\mathbf{2}, \mathbf{2}, \mathbf{2})$ *with the requirement that* $\Gamma(\mathtt{y}) = \mathbf{1}$.

It is worth noticing that the above program cannot be typed with only two tiers $\{\mathbf{0}, \mathbf{1}\}$. *Indeed, the first while loop enforces that* $\Gamma(\mathtt{y}) \prec \Gamma(\mathtt{x})$ *and the second while loop enforces that* $\Gamma(\mathtt{z}) \prec \Gamma(\mathtt{y})$. *More generally, the program can be typed by* $(\mathbf{t}{+}\mathbf{2}, \mathbf{t}{+}\mathbf{2}, \mathbf{0})$ *or* $(\mathbf{t}{+}\mathbf{2}, \mathbf{t}{+}\mathbf{2}, \mathbf{t}{+}\mathbf{2})$, *for any tier* $\mathbf{t}$.

Example 11 (Oracle). *For a given input* $\mathtt{x}$ *and a given oracle* ϕ, *the program below computes whether there exists a unary integer n of size smaller than $|\mathtt{x}|$ such that $\phi(n) = 0$.*

```
y⁰ := x¹;
z⁰ := 0;
while(x >= 0)¹{
    if(ϕ(y⁰ ↾ x¹) == 0)⁰{
        z⁰ := 1
    }
    else {skip};
    x¹ := pred(x)¹
}
return  z
```

This program is safe and can be typed by $(\mathbf{1}, \mathbf{1}, \mathbf{0})$ *under the variable typing environment* Γ *such that* $\Gamma(\mathtt{x}) = \mathbf{1}$ *and* $\Gamma(\mathtt{y}) = \Gamma(\mathtt{z}) = \mathbf{0}$. *The constants 0 and 1 can be considered to be neutral operators of zero arity and, hence, can be given any tier smaller than the innermost tier. It is easy to verify that the commands* $\mathtt{z} := 1$, $\mathtt{skip}$, *and* $\mathtt{x} := \mathtt{pred}(\mathtt{x})$ *can be typed by* $(\mathbf{0}, \mathbf{1}, \mathbf{1})$, $(\mathbf{0}, \mathbf{1}, \mathbf{1})$, *and* $(\mathbf{1}, \mathbf{1}, \mathbf{1})$, *respectively, using typing rules (OP), (SK), and (A).*

The conditional subcommand can be typed as described below.

$$
\cfrac{\mathbf{0} \to \mathbf{1} \to \mathbf{0} \in \Delta(\text{==})(\mathbf{1}) \quad \cfrac{\cfrac{\Gamma(\mathtt{y}) = \mathbf{0}}{\vdash \mathtt{y} : (\mathbf{0},\mathbf{1},\mathbf{1})}\ (V) \quad \cfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : (\mathbf{1},\mathbf{1},\mathbf{1})}\ (V)}{\vdash \phi(\mathtt{y} \restriction \mathtt{x}) : (\mathbf{0},\mathbf{1},\mathbf{1})}\ (OR) \quad \cfrac{}{\vdash 0 : (\mathbf{1},\mathbf{1},\mathbf{1})}\ (OP)}{\cfrac{\cfrac{\vdash \phi(\mathtt{y} \restriction \mathtt{x}) == 0 : (\mathbf{0},\mathbf{1},\mathbf{1})}{\vdash \mathtt{if}(\phi(\mathtt{y} \restriction \mathtt{x}) == 0)\{\mathtt{z} := 1\}\ \mathtt{else}\ \{\mathtt{skip}\} : (\mathbf{0},\mathbf{1},\mathbf{1})}}{}}\ (OP) \quad \vdots\ (C)
$$

The while loop will be typed using rule (W_0). *Consequently, the inner command can be typed by* $(\mathbf{1}, \mathbf{1}, \mathbf{1})$ *after applying subtyping once.*

Notice that the equivalent program obtained by swapping $\mathtt{x}$ *and* $\mathtt{y}$ *in the oracle input (i.e., $\phi(\mathtt{x} \restriction \mathtt{y})$) is not typable as the tier of $\mathtt{x}$ has to be strictly smaller than the innermost tier in typing rule (OR). Although this requirement restricts the expressive power of the type system, it is strongly needed as it prevents uncontrolled loops on oracle outputs to occur. In particular, commands of the shape* $\mathtt{while}(\mathtt{x} > 0)\{\mathtt{x} := \phi(\mathtt{x} \restriction \mathtt{x})\}$ *are rejected by the type system.*

Note that the above program is typable as the oracle calls are performed in a decreasing order and, hence, does not break the finite lookahead revision property, which will be presented in §6.4.

Now consider the equivalent program where oracle calls are performed in increasing order.

```
  x := 0;
  z := 0;
  while(y >= x){
     if(φ(y ↾ x) == 0){
        z := 1
     }
     else {skip};
     x := suc₁(x)
  }
  return  z
```

This program is not a safe program. Suppose, by contradiction, that it can be typed with respect to a safe operator typing environment. The innermost tier $\mathbf{t}$ of the commands under the while will be equal to the tier of the guard $\mathtt{y} >= \mathtt{x}$, independently of whether rule (W) or rule (W_0) is used to type the while command. Moreover, $\mathtt{x}$ has a tier $\mathbf{t_x}$ such that $\mathbf{t} \preceq \mathbf{t_x}$, using rule (OP) on the guard and, by definition of safe typing environments. Now $\mathtt{suc}_1$ is a positive operator and, consequently, by rule (OP) and, by definition of safe typing environments again, $\mathtt{suc}_1(\mathtt{x})$ has a tier $\mathbf{t_{suc_1(x)}}$ strictly smaller than the innermost tier, i.e., $\mathbf{t_{suc_1(x)}} \prec \mathbf{t}$. By typing rule (A), in order to be typed, the command $\mathtt{x} := \mathtt{suc}_1(\mathtt{x})$ enforces $\mathbf{t_x} \preceq \mathbf{t_{suc_1(x)}}$. Hence, we obtain a contradiction: $\mathbf{t} \prec \mathbf{t}$.

Example 12 (Multiple tiers with oracle). *The following program computes the function $\Sigma_{i=0}^{\max_{x=0}^n \phi(x)} \phi(i)$.*

```
  x³ := n; y² := x³; z² := 0;
  while(x³ >= 0){
     z² := max(φ(y² ↾ x³)², z²);
     x³ := pred(x³)
  }; v¹ := z²; u⁰ := 0;
  while(z² >= 0){
     w¹ := φ(v¹ ↾ z²)¹;
     while(w¹ >= 0){
        u⁰ := suc₁(u)⁰; w¹ := pred(w¹)
     }; z² := pred(z)²
  }
  return  u
```

This program can be typed by $(\mathbf{3}, \mathbf{3}, \mathbf{0})$ under the variable type assignment Γ such that $\Gamma(\mathtt{x}) = \mathbf{3}$, $\Gamma(\mathtt{y}) = \Gamma(\mathtt{z}) = \mathbf{2}$, $\Gamma(\mathtt{v}) = \Gamma(\mathtt{w}) = \mathbf{1}$, and $\Gamma(\mathtt{u}) = \mathbf{0}$.

The first while loop will be typed using rule (W_0). Consequently, its inner command is typed by $(\mathbf{3}, \mathbf{3}, \mathbf{3})$. As the $\max$ operator is neutral, it can be given the type $\mathbf{2} \to \mathbf{2} \to \mathbf{2} \in \Delta(\max)(\mathbf{3})$. The oracle call is typable as the input data $\mathtt{y}$ has a tier strictly small than the innermost tier $(\mathbf{3})$ and the input bound has tier equal to the outermost tier $(\mathbf{3})$.

The second while loop can be typed using rule (W_0) after applying subtyping rule (SUB). Consequently, its inner command is typed by $(\mathbf{2}, \mathbf{2}, \mathbf{2})$. The oracle call is performed on input data of strictly smaller tier $(\mathbf{1})$ and on input bound of tier equal to the outermost tier $(\mathbf{2})$.

The inner while loop can be typed using rule (W) and thus updates the innermost tier to $\mathbf{1}$. Consequently, suc_1 is enforced to be of tier $\mathbf{0} \to \mathbf{0}$ in the inner command.

6. Properties of safe programs

We now show the main properties of safe programs:
- a standard non-interference property in §6.2 ensuring that computations on higher tiers do not depend on lower tiers (Theorem 17).
- a polynomial time property in §6.3 ensuring that terminating programs terminate in time polynomial in the input size and maximal oracle output size (Theorem 21).
- a finite lookahead revision property in §6.4 ensuring that, for any oracle and any input, the number of oracle calls on input of increasing size is bounded by a constant (Theorem 25).

6.1. **Notation.** Let us first introduce some preliminary notation. Let $\mathcal{E}(a)$ (res. $\mathcal{C}(a)$) be the set of expressions (respectively commands) occurring in a, for $a \in \{\mathsf{p}_\phi, \mathsf{c}\}$. Let $\mathcal{A}(\mathsf{c})$ be the set of variables that are assigned to in c, *e.g.*, $\mathcal{A}(\mathsf{x} := \mathsf{y}; \mathsf{y} := \mathsf{z}) = \{\mathsf{x}, \mathsf{y}\}$. Let $Op(\mathsf{e})$ and $\mathcal{V}(\mathsf{e})$ be the set of operators in expression e and the set of variables in expression e, respectively.

6.2. **Non-interference.** We now show that the type system provides classical non-interference properties.

In a safe program, only variables of tier higher than $\mathbf{t}$ can be accessed to evaluate an expression of tier $\mathbf{t}$.

Lemma 13 (Simple security)**.** *Given a safe program p_ϕ with respect to the typing environments Γ, Δ, for any expression $\mathsf{e} \in \mathcal{E}(\mathsf{p}_\phi)$, if $\Gamma, \Delta \vdash \mathsf{e} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$, then for all $\mathsf{x} \in \mathcal{V}(\mathsf{e})$, $\mathbf{t} \preceq \Gamma(\mathsf{x})$.*

Proof. By structural induction on expressions. $\square$

There is no equivalent lemma for commands because of the subtyping rule (SUB).

Corollary 14. *Given a safe program p_ϕ with respect to the typing environments Γ, Δ, for any $\mathsf{x} := \mathsf{e} \in \mathcal{C}(\mathsf{p}_\phi)$, $\Gamma(\mathsf{x}) \preceq \wedge_{\mathsf{y} \in \mathcal{V}(\mathsf{e})}\Gamma(\mathsf{y})$.*

Proof. Given a command $\mathsf{x} := \mathsf{e} \in \mathcal{C}(\mathsf{p}_\phi)$ of a safe program with respect to the typing environments Γ, Δ, there are $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_{in}, \mathbf{t}_{out}$ such that $\Gamma, \Delta \vdash \mathsf{x} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out})$, $\Gamma, \Delta \vdash \mathsf{e} : (\mathbf{t}_2, \mathbf{t}_{in}, \mathbf{t}_{out})$ and $\mathbf{t}_1 \preceq \mathbf{t}_2$, by typing rule (A). Applying Lemma 13, $\forall \mathsf{y} \in \mathcal{V}(\mathsf{e})$, $\mathbf{t}_2 \preceq \Gamma(\mathsf{y})$. By typing rule (V) $\mathbf{t}_1 = \Gamma(\mathsf{x})$ and, consequently, $\forall \mathsf{y} \in \mathcal{V}(\mathsf{e})$, $\Gamma(\mathsf{x}) \preceq \Gamma(\mathsf{y})$. $\square$

The following Lemma states that command tiers are monotonic in their subcommand tier in a given typing derivation.

Lemma 15. *Let ρ be a typing derivation of a safe program with respect to the typing environments Γ, Δ. For any typing derivations $\rho_1 \rhd \Gamma, \Delta \vdash \mathsf{c}_1 : (\mathbf{t}^1, \mathbf{t}^1_{in}, \mathbf{t}^1_{out}) \in \mathcal{D}(\rho)$ and $\rho_2 \rhd \Gamma, \Delta \vdash \mathsf{c}_2 : (\mathbf{t}^2, \mathbf{t}^2_{in}, \mathbf{t}^2_{out}) \in \mathcal{D}(\rho_1)$, $\mathbf{t}^2 \preceq \mathbf{t}^1$.*

Proof. Suppose by contradiction that $\rho_1 < \rho_2$ and $\mathbf{t}^1 \prec \mathbf{t}^2$ hold. As all typing rules for commands in Figure 3 are monotonic in the command tier, $\rho_2 \rhd \Gamma, \Delta \vdash \mathsf{c}_2 : (\mathbf{t}^2, \mathbf{t}^2_{in}, \mathbf{t}^2_{out})$ cannot be derived from ρ_1. $\square$

The confinement Lemma expresses the fact that commands of tier $\mathbf{t}$ cannot write in variables of strictly higher tier.

Lemma 16 (Confinement). *Given a safe program $\mathbf{p}_\phi$ with respect to the typing environments Γ, Δ, for any $\mathbf{c} \in \mathcal{C}(\mathbf{p}_\phi)$, if $\Gamma, \Delta \vdash \mathbf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$, then for all $\mathbf{x} \in \mathcal{A}(\mathbf{c})$, $\Gamma(\mathbf{x}) \preceq \mathbf{t}$.*

Proof. By contradiction. Suppose that $\rho_1 \rhd \Gamma, \Delta \vdash \mathbf{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})$ holds. Consider a variable $\mathbf{x} \in \mathcal{A}(\mathbf{c})$ and suppose that $\mathbf{t} \prec \Gamma(\mathbf{x})$. By typing rule (A), there is an expression $\mathbf{e}$ and there are tiers $\mathbf{t}'_{in}, \mathbf{t}'_{out}$ such that $\rho_2 \rhd \Gamma, \Delta \vdash \mathbf{x} := \mathbf{e} : (\Gamma(\mathbf{x}), \mathbf{t}'_{in}, \mathbf{t}'_{out})$ and $\rho_2 \in \mathcal{D}(\rho_1)$. Consequently, by Lemma 15, $\Gamma(\mathbf{x}) \preceq \mathbf{t}$, which contradicts the assumption. $\square$

For a given variable typing environment Γ and a given tier $\mathbf{t}$, we define an equivalence relation on stores by: $\mu \approx_{\mathbf{t}}^\Gamma \mu'$ if $dom(\mu) = dom(\mu') = dom(\Gamma)$ and for each $\mathbf{x} \in dom(\Gamma)$, if $\mathbf{t} \preceq \Gamma(\mathbf{x})$ then $\mu(\mathbf{x}) = \mu'(\mathbf{x})$.

We introduce a non-interference Theorem ensuring that the values of tier $\mathbf{t}$ variables during the evaluation of a program do not depend on values of tier $\mathbf{t}'$ variables for $\mathbf{t}' \prec \mathbf{t}$.

Theorem 17 (Non-interference). *Given a safe program $\mathbf{c}$ return $\mathbf{x}$ with respect to the typing environments Γ, Δ. For any stores μ_1 and μ_2 if $\mu_1 \approx_{\mathbf{t}}^\Gamma \mu_2$, $\mu_1 \vDash \mathbf{c} \to \mu'_1$ and $\mu_2 \vDash \mathbf{c} \to \mu'_2$ then $\mu'_1 \approx_{\mathbf{t}}^\Gamma \mu'_2$.*

Proof. By structural induction on derivations. The base case of a derivation consisting in only one node is straightforward as only the rule (Skip) of Figure 2 can be fired. Consequently, $\mathbf{c} = \mathtt{skip}$ and $\mu'_1 = \mu_1 \approx_{\mathbf{t}}^\Gamma \mu'_2 = \mu_2$.

Now consider the two derivations $\pi_\phi^1 : \mu_1 \vDash \mathbf{c} \to \mu'_1$ and $\pi_\phi^2 : \mu_2 \vDash \mathbf{c} \to \mu'_2$ such that $\mu_1 \approx_{\mathbf{t}}^\Gamma \mu_2$. We perform a case analysis on commands.

- if $\mathbf{c} = \mathbf{x} := \mathbf{e}$ then the rule at the root of derivations π_ϕ^1 and π_ϕ^2 is the rule (Asg) of Figure 2. There are two cases to consider.
  - If $\mathbf{t} \preceq \Gamma(\mathbf{x})$ then, by Corollary 14, $\forall \mathbf{y} \in \mathcal{V}(\mathbf{e})$, $\Gamma(\mathbf{x}) \preceq \Gamma(\mathbf{y})$ and consequently, $\forall \mathbf{y} \in \mathcal{V}(\mathbf{e})$, $\mu_1(\mathbf{y}) = \mu_2(\mathbf{y})$. This implies that there exists $w \in \mathbb{W}$ such that $\mu_1 \vDash \mathbf{e} \to w$ and $\mu_2 \vDash \mathbf{e} \to w$. Consequently $\mu_1 \vDash \mathbf{c} \to \mu_1[\mathbf{x} \leftarrow w]$, $\mu_2 \vDash \mathbf{c} \to \mu_2[\mathbf{x} \leftarrow w]$ and $\mu_1[\mathbf{x} \leftarrow w] \approx_{\mathbf{t}}^\Gamma \mu_2[\mathbf{x} \leftarrow w]$.
  - If $\Gamma(\mathbf{x}) \prec \mathbf{t}$ then $\mu_1 \vDash \mathbf{c} \to \mu_1[\mathbf{x} \leftarrow w]$, $\mu_2 \vDash \mathbf{c} \to \mu_2[\mathbf{x} \leftarrow v]$ and $\mu_1[\mathbf{x} \leftarrow w] \approx_{\mathbf{t}}^\Gamma \mu_2[\mathbf{x} \leftarrow v]$.
- if $\mathbf{c} = \mathbf{c}_1; \mathbf{c}_2$ then the rule at the root of derivations π_ϕ^1 and π_ϕ^2 is the rule (Seq) of Figure 2. Consequently, there exist two stores μ''_1 and μ''_2, such that, for $i \in \{1, 2\}$, $\pi_\phi^{i'} : \mu_i \vDash \mathbf{c}_1 \to \mu''_i$ and $\pi_\phi^{i''} : \mu''_i \vDash \mathbf{c}_2 \to \mu'_i$ are subderivations of π_ϕ^i. Therefore, if $\mu_1 \approx_{\mathbf{t}}^\Gamma \mu_2$ then $\mu''_1 \approx_{\mathbf{t}}^\Gamma \mu''_2$ and $\mu'_1 \approx_{\mathbf{t}}^\Gamma \mu'_2$, by applying the induction hypothesis twice.
- if $\mathbf{c} = \mathtt{while}(\mathbf{e})\{\mathbf{c}'\}$ then the rule at the root of derivations π_ϕ^1 and π_ϕ^2 can be either rule (Wh$_0$) or rule (Wh$_1$) of Figure 2. If $\Gamma, \Delta \vdash \mathtt{while}(\mathbf{e})\{\mathbf{c}'\} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out})$ then $\Gamma, \Delta \vdash \mathbf{e} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out})$ and there are two cases to consider:
  - If $\mathbf{t} \preceq \mathbf{t}_1$ then, by Lemma 13, $\forall \mathbf{x} \in \mathcal{V}(\mathbf{e})$, $\mathbf{t}_1 \preceq \Gamma(\mathbf{x})$. Consequently, $\forall \mathbf{x} \in \mathcal{V}(\mathbf{e})$, $\mu_1(\mathbf{x}) = \mu_2(\mathbf{x})$. It follows that $\mu_1 \vDash \mathbf{e} \to w$ and $\mu_2 \vDash \mathbf{e} \to w$, with $w \in \{0, 1\}$. Notice that, it excludes the possibility to apply (Wh$_1$) on one derivation and (Wh$_0$) on the other derivation. For the non-trivial case where $w = 1$, we obtain that $\pi_\phi^{1'} : \mu_1 \vDash \mathbf{c}'; \mathbf{c} \to \mu'_1$, $\pi_\phi^{2'} : \mu_2 \vDash \mathbf{c}'; \mathbf{c} \to \mu'_2$, and $\mu'_1 \approx_{\mathbf{t}}^\Gamma \mu'_2$, by induction on the subderivations $\pi_\phi^{1'}$ and $\pi_\phi^{2'}$. Consequently, $\mu_1 \vDash \mathbf{c} \to \mu'_1$, $\mu_2 \vDash \mathbf{c} \to \mu'_2$ and $\mu'_1 \approx_{\mathbf{t}}^\Gamma \mu'_2$.
  - If $\mathbf{t}_1 \prec \mathbf{t}$ then, by Lemma 16, $\forall \mathbf{x} \in \mathcal{A}(\mathbf{c}')$, $\Gamma(\mathbf{x}) \preceq \mathbf{t}_1$. Consequently, if $\mu_1 \vDash \mathbf{c} \to \mu'_1$ and $\mu_2 \vDash \mathbf{c} \to \mu'_2$ then $\mu'_1 \approx_{\mathbf{t}}^\Gamma \mu'_2$.

All the other cases can be treated in a similar manner. $\qquad\square$

6.3. **Polynomial step count.** In this section, we show that terminating and safe programs have a runtime polynomially bounded by the size of the input store and the maximal size of answers returned by the oracle in the course of execution.

The following Lemma shows that the innermost tier of a while loop subcommand is always an upper bound on the tier of this loop.

Lemma 18. *Let ρ be a typing derivation of a safe program with respect to the typing environments Γ, Δ. For any typing derivations $\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}^1_{in}, \mathbf{t}^1_{out}) \in \mathcal{D}(\rho)$ and $\rho_2 \rhd \Gamma, \Delta \vdash \mathtt{c_2} : (\mathbf{t}^2, \mathbf{t}^2_{in}, \mathbf{t}^2_{out}) \in \mathring{\mathcal{D}}(\rho_1)$, $\mathbf{t}^2_{in} \preceq \mathbf{t}^1$.*

Proof. Given a typing derivation $\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}^1_{in}, \mathbf{t}^1_{out})$, we show by induction that the property *Inv* defined on command typing derivations by:

$$Inv(\rho \rhd \Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t}, \mathbf{t}_{in}, \mathbf{t}_{out})) \triangleq (\mathbf{t}_{in} \preceq \mathbf{t}^1)$$

is invariant on command typing derivations strictly smaller than ρ_1, *i.e.*, typing derivations ρ such that $\rho < \rho_1$.

Base case. Suppose that the typing derivation ρ_1 is of the shape:

$$\frac{\Gamma, \Delta \vdash \mathtt{e_1} : \ldots \qquad \rho_2 \rhd \Gamma, \Delta \vdash \mathtt{c_1} : (\mathbf{t}^2, \mathbf{t}^2_{in}, \mathbf{t}^2_{out})}{\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}^1_{in}, \mathbf{t}^1_{out})} \text{ (R)}.$$

Rule (R) can be either rule (W) or rule (W$_0$) of Figure 3. In both cases, it holds that $\mathbf{t}^2 = \mathbf{t}^2_{in} = \mathbf{t}^1$ and, consequently, $Inv(\rho_2)$ holds.

General case. Consider the typing derivation $\rho_3 \in \mathring{\mathcal{D}}(\rho_2)$ of the shape:

$$\frac{\ldots \qquad \rho_4 \rhd \Gamma, \Delta \vdash \mathtt{c_4} : (\mathbf{t}^4, \mathbf{t}^4_{in}, \mathbf{t}^4_{out})}{\rho_3 \rhd \Gamma, \Delta \vdash \mathtt{c_3} : (\mathbf{t}^3, \mathbf{t}^3_{in}, \mathbf{t}^3_{out})} \text{ (R')},$$

for some typing rule (R'). By induction hypothesis, $Inv(\rho_3)$ holds, and, consequently, $\mathbf{t}^3_{in} \preceq \mathbf{t}^1$. Moreover, by Lemma 15, $\mathbf{t}^3 \preceq \mathbf{t}^1$ holds.

By a case analysis on typing rules of Figure 3, (R') can be either (S), (A), (SUB), (C), or (W). Indeed, rule (W$_0$) cannot be applied under a while loop as it requires the outermost tier to be equal to $\mathbf{0}$. This is impossible under a while loop as, by looking at the constraints in rules (W) and (W$_0$) of Figure 3, one can check that the outermost tier of a while loop strict subcommand is greater than $\mathbf{1}$.

These five rules imply that either $\mathbf{t}^4_{in} = \mathbf{t}^3_{in}$ (rules (S), (A), (SUB), and (C)) or $\mathbf{t}^4_{in} = \mathbf{t}^3$ (rule (W)). Consequently, $\mathbf{t}^4_{in} \preceq \mathbf{t}^1$, and $Inv(\rho_4)$ holds. $\qquad\square$

Consequently, the innermost tier of a while loop provides a lower bound on the tier of the loop guard expression.

We now show that within a while loop of tier $\mathbf{t}$, if an expression is assigned to a tier $\mathbf{t}$ variable then this expression does not contain positive operators. Moreover, there are no oracle calls in tier $\mathbf{t}$ subcommands of a tier $\mathbf{t}$ while loop.

Lemma 19. *Let ρ be a typing derivation of a safe program with respect to the typing environments Γ, Δ. For any $\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}^1_{in}, \mathbf{t}^1_{out}) \in \mathcal{D}(\rho)$ and $\rho_2 \rhd \Gamma, \Delta \vdash \mathtt{x := e_2} : (\mathbf{t}^2, \mathbf{t}^2_{in}, \mathbf{t}^2_{out}) \in \mathcal{D}(\rho_1)$, if $\mathbf{t}^1 = \mathbf{t}^2$ then:*

$$\dfrac{\dfrac{\Gamma(\mathbf{x}) = \mathbf{t}^2}{\Gamma, \Delta \vdash \mathbf{x} : (\mathbf{t}^2, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)} \text{ (V)} \qquad \dfrac{\mathbf{t}^4 \to \mathbf{t}^3 \in \Delta(\mathtt{op})(\mathbf{t}_{in}^2) \qquad \dfrac{\Gamma(\mathbf{y}) = \mathbf{t}^4}{\Gamma, \Delta \vdash \mathbf{y} : (\mathbf{t}^4, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)} \text{ (V)}}{\mathtt{op}(\mathbf{y}) : (\mathbf{t}^3, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)} \text{ (OP)}}{\Gamma, \Delta \vdash \mathbf{x} := \mathtt{op}(\mathbf{y}) : (\mathbf{t}^2, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)} \text{ (A)}$$

FIGURE 4. Proof of Lemma 19 (1)

$$\dfrac{\dfrac{\Gamma(\mathbf{x}) = \mathbf{t}^2}{\Gamma, \Delta \vdash \mathbf{x} : (\mathbf{t}^2, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)} \text{ (V)} \qquad \dfrac{\vdots \qquad\qquad \vdots}{\dfrac{\Gamma, \Delta \vdash \mathsf{e}' : (\mathbf{t}^3, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2) \qquad \Gamma, \Delta \vdash \mathsf{e}'' : (\mathbf{t}_{out}^2, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)}{\Gamma, \Delta \vdash \phi(\mathsf{e}' \restriction \mathsf{e}'') : (\mathbf{t}^3, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)}}}{\Gamma, \Delta \vdash \mathbf{x} := \phi(\mathsf{e}' \restriction \mathsf{e}'') : (\mathbf{t}^2, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2)} \text{ (A)}$$

FIGURE 5. Proof of Lemma 19 (2)

(1) *for any* $\mathtt{op} \in Op(\mathsf{e}_2)$, $\mathtt{op}$ *is a neutral operator;*
(2) *there is no oracle call in* e_2.

Proof. Both (1) and (2) are proved by contradiction using Lemma 18.

(1) By contradiction, suppose that there exists an assignment $\mathbf{x} := \mathsf{e}_2$ such that $\mathbf{t}^1 = \mathbf{t}^2$ and there exists an operator in $Op(\mathsf{e}_2)$ that is positive. For simplicity, suppose that $ar(\mathtt{op}) = 1$ and that $\mathsf{e}_2 = \mathtt{op}(\mathbf{y})$. We obtain the typing derivation of Figure 4, with the constraints that $\mathbf{t}^2 \preceq \mathbf{t}^3 \preceq \mathbf{t}^4 \preceq \mathbf{t}_{in}^2$. $\mathbf{t}^2 \preceq \mathbf{t}^3$ is enforced by rule (A) and $\mathbf{t}^3 \preceq \mathbf{t}^4 \preceq \mathbf{t}_{in}^2$ is enforced by definition of safe typing environment applied to $\Delta(\mathtt{op})(\mathbf{t}_{in}^2)$ in rule (OP). Moreover as $\mathtt{op}$ is positive, the constraint $\mathbf{t}^3 \prec \mathbf{t}_{in}^2$ is enforced and, consequently, $\mathbf{t}^2 \prec \mathbf{t}_{in}^2$. However, by Lemma 18 applied to the typing derivation sub-tree of root $\mathbf{x} := \mathtt{op}(\mathbf{y})$, $\mathbf{t}_{in}^2 \preceq \mathbf{t}^1$, hence $\mathbf{t}_{in}^2 \preceq \mathbf{t}^2$, by hypothesis ($\mathbf{t}^2 = \mathbf{t}^1$), and we obtain a contradiction. The general case can be treated by a structural induction on expressions.

(2) By contradiction suppose that $\mathbf{t}^1 = \mathbf{t}^2$ and that there is an oracle call in e_2. Suppose for simplicity that $\mathsf{e}_2 = \phi(\mathsf{e}' \restriction \mathsf{e}'')$. We obtain the typing derivation of Figure 5. By the constraint of typing rule (A), $\mathbf{t}^2 \preceq \mathbf{t}^3$. By the constraints of typing rule (OR), $\mathbf{t}^3 \prec \mathbf{t}_{in}^2$ and $\mathbf{t}^3 \preceq \mathbf{t}_{out}^2$. Consequently, $\mathbf{t}^2 \prec \mathbf{t}_{in}^2$ is enforced. By Lemma 18, $\mathbf{t}_{in}^2 \preceq \mathbf{t}^1$ and, consequently, we obtain a contradiction. The general case can be treated in a similar manner by a structural induction on expressions. □

Definition 20. *Let* $m_\mu^{\mathtt{p}_\phi}$ *be the maximum of* $|\mu|$ *and the maximum size of an oracle answer in the derivation* $\pi_\phi : \mu \vDash \mathtt{p}_\phi \to u$. *Formally,*

$$m_\mu^{\mathtt{p}_\phi} = \max_{(v,w) \in C(\pi_\phi)} (|\mu|, \max\{|\phi([\![\restriction]\!](v,w))|\}),$$

where $C(\pi_\phi)$ *is the set of pairs* (v, w) *such that*

$$\dfrac{\mu \vDash \mathsf{e}_1 \to v \quad \mu \vDash \mathsf{e}_2 \to w}{\mu \vDash \phi(\mathsf{e}_1 \restriction \mathsf{e}_2) \to \phi([\![\restriction]\!](v,w))} \in \pi_\phi.$$

   A program $\mathtt{p}_\phi$ *has a* polynomial step count *if there is a polynomial P such that for any store* μ *and any oracle* ϕ, $\pi_\phi : \mu \vDash \mathtt{p}_\phi \to w$, $|\pi_\phi| \leq P(m_\mu^{\mathtt{p}_\phi})$.

   We show that a safe program has a polynomial step count on terminating computations.

Theorem 21. *Given a safe program* $\mathtt{p}_\phi$ *with respect to the typing environments* Γ, Δ*, there is a polynomial P such that for any derivation* $\pi_\phi : \mu \vDash \mathtt{p}_\phi \to w$, $|\pi_\phi| \leq P(m_\mu^{\mathtt{p}_\phi})$.

Proof. By induction on the tier of a command using non-interference Theorem (17) and Lemma 19.

   The case **0** is trivial as there are no while loops (see rules (W) and (W$_0$) of Figure 3). Now consider a safe program, whose command is of tier **t** strictly greater than **0**. For any variable of tier **t**, by Lemma 19, only neutral operators may be applied in assignments to such a variable in a while. Combining Definition 2 and non-interference (Theorem 17), the word computed by a neutral operator is either a subword of the initial values stored in tier **t** variables or a constant in $\{0,1\}$. Moreover, the number of times a positive operator is applied to such a variable is at most constant. Indeed, such an assignment occurs outside a while loop. Putting it altogether, the number of distinct values in a store for each tier **t** variable is in $O((m_\mu^{\mathtt{p}_\phi})^2)$ as strict subword operations can be iterated at most a quadratic number of time.

   The evaluation of while loops of tier **t** unfolds at most a polynomial number of commands of tier at most **t** − **1**. The degree of the polynomial depends on the number of variables of tier $\geq$ **t**, by non-interference (Theorem 17), and on the number of nested tier **t** while loops and, as a consequence, is bounded by a constant: the size of the program. By induction, all these commands of tier strictly smaller are evaluated in a polynomial number of steps in their input. Each of their input is polynomially bounded by $m_\mu^{\mathtt{p}_\phi}$ as it consists in oracle calls combined with at most a polynomial number of positive operators.

   It just remains to observe that we compose a constant number (**t**+**1**) of polynomials and that polynomials are closed under composition.   □

   The proof of the above Theorem is similar to proofs of polynomiality in [Mar11] and [MP14], except for two distinctions:

- As strictly more than 2 tiers are allowed, the innermost tier $\mathbf{t}_{in}$ is used to ensure that operators and oracle calls are stratified (Lemma 19): in a while loop of innermost tier $\mathbf{t}_{in}$ the return type of an oracle or positive operator is always strictly smaller than $\mathbf{t}_{in}$. Hence the results of such computations cannot be assigned to variables whose tier is equal to $\mathbf{t}_{in}$.
- Oracle calls may return a value whose size is not bounded by the program input. This is the reason why $m_\mu^{\mathtt{p}_\phi}$ has to be considered as an input of the time bound.

Corollary 22. *Given a program* $\mathtt{p}_\phi$, *if* $\mathtt{p}_\phi \in \mathrm{ST}$ *then* $\mathtt{p}_\phi$ *has a polynomial step count.*

6.4. **Finite lookahead revision.** In this section, we show that, whereas terminating and safe programs may perform a polynomial number (in the size of the input and the maximal size of the oracle answers) of oracle calls during their execution, they may only perform a constant number of oracle calls on input data of increasing size.

   We first start to show that the outermost tier of a command of a safe program is an upper bound on the tiers of while loop expressions guarding this command.

Lemma 23. *Let ρ be a typing derivation of a safe program with respect to the typing environments Γ, Δ. For any $\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}_{in}^1, \mathbf{t}_{out}^1)(R) \in \mathcal{D}(\rho)$ and $\rho_2 \rhd \Gamma, \Delta \vdash \mathtt{c_2} : (\mathbf{t}^2, \mathbf{t}_{in}^2, \mathbf{t}_{out}^2) \in \mathring{\mathcal{D}}(\rho_1)$, if $R \in \{W, W_0\}$ then $\mathbf{t}^1 \preceq \mathbf{t}_{out}^2$.* [2]

Proof. The typing rule (W_0) is the only rule changing the outermost tier. It is straightforward to observe that this rule can only be applied to an outermost while loop as the outermost tier of the command is updated from $\mathbf{0}$ to $\mathbf{t}$, for some $\mathbf{t}$ such that $\mathbf{1} \preceq \mathbf{t}$. There are two cases to consider depending on R:

- if R=W then

$$\frac{\dots \quad \rho_1' \rhd \Gamma, \Delta \vdash \mathtt{c_1} : (\mathbf{t}^1, \mathbf{t}^1, \mathbf{t}_{out}^1) \quad \mathbf{1} \preceq \mathbf{t}^1 \preceq \mathbf{t}_{out}^1}{\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}_{in}^1, \mathbf{t}_{out}^1)} \ (W).$$

  Clearly, $\mathbf{t}^1 \preceq \mathbf{t}_{out}^1 = \mathbf{t}_{out}^2$ by the guard condition and as $\rho_2 \leq \rho_1'$ and all the rules under a while preserve the outermost tier.

- if R=W_0 then

$$\frac{\dots \quad \rho_1' \rhd \Gamma, \Delta \vdash \mathtt{c_1} : (\mathbf{t}^1, \mathbf{t}^1, \mathbf{t}^1) \quad \mathbf{1} \preceq \mathbf{t}^1}{\rho_1 \rhd \Gamma, \Delta \vdash \mathtt{while(e_1)\{c_1\}} : (\mathbf{t}^1, \mathbf{t}_{in}^1, \mathbf{0})} \ (W_0).$$

  It is straightforward that $\mathbf{t}^1 = \mathbf{t}_{out}^2$ as $\rho_2 \leq \rho_1'$ ($\rho_2 \in \mathring{\mathcal{D}}(\rho_1)$) and all the rules under a while preserve the outermost tier. $\square$

Definition 24. *Given a program $\mathtt{p}_\phi$, let $(l_n^{\pi_\phi})$ be the sequence of oracle input values $[\![\lceil]\!](v, w)$ in a rule (OR) obtained by a left-to-right depth-first traversal of the derivation $\pi_\phi : \mu \vDash \mathtt{p}_\phi \to w$. Let $lr((l_n^{\pi_\phi})) = \#\{i \mid |l_i^{\pi_\phi}| > \max_{j<i}(|l_j^{\pi_\phi}|)\}$.*

  $\mathtt{p}_\phi$ has finite lookahead revision *if there is a constant r such that for any oracle ϕ and for any derivation π_ϕ (i.e., for all program inputs), we have $lr((l_n^{\pi_\phi})) \leq r$.*

Note that the left-to-right depth-first traversal in a derivation exactly corresponds to the order of a sequential execution of a command.

Theorem 25 (Finite lookahead revision)**.** *Given a program $\mathtt{p}_\phi$, if $\mathtt{p}_\phi$ is safe with respect to the typing environments Γ, Δ then it has finite lookahead revision.*

Proof. By Lemma 23, the outermost tier $\mathbf{t}_{out}$ of any command within a while loop command $\mathtt{c}$ is an upper bound on the tier of any while loop guard in $\mathtt{c}$. By typing rule (OR) of Figure 3, the tier of the input bound $\mathtt{e_2}$ in any oracle call $\phi(\mathtt{e_1} \lceil \mathtt{e_2})$ in $\mathtt{c}$ must be equal to the outermost tier $\mathbf{t}_{out}$. By Lemma 13, only variables of tier greater than or equal to $\mathbf{t}_{out}$ can occur in $\mathtt{e_2}$. Consider such a variable $\mathtt{x}$ of tier $\mathbf{t}$. For simplicity, we assume that $\mathtt{x}$ is the only variable of tier greater than or equal to $\mathbf{t}_{out}$ in $\mathtt{c}$. If $\mathbf{t}_{out} \prec \mathbf{t}$ then $\mathtt{x}$ cannot be assigned to in $\mathtt{c}$, by Lemma 16. If $\mathbf{t} = \mathbf{t}_{out}$ then $\mathtt{x}$ can only be assigned to by expressions that contain neither oracle calls nor positive operators, by Lemma 19. It means that in any assignment $\mathtt{x} = \mathtt{e}$ of the loop, the expression $\mathtt{e}$ can only contain variable $\mathtt{x}$ and neutral operators. Hence the length of the value stored by $\mathtt{x}$ cannot increase between two iterations of any loop in $\mathtt{c}$. Following the same reasoning, the length of the oracle's input bound $\mathtt{e_2}$ cannot increase between two iterations of a loop in $\mathtt{c}$. As $|[\![\lceil]\!](v, w)| = |w|+1$, the length of

---

[2]The assumption for the last rule of ρ_1 to be in {W,W_0} ensures that $\mathbf{t}^1$ is exactly the tier of the guard expression $\mathtt{e_1}$. Otherwise typing rule (SUB) can be applied arbitrarily and the result does not hold.

the oracle's input values cannot increase within a loop in c and, consequently, the number of lookahead revisions in a sequence $(l_n^{\pi_\phi})$ is bounded by a constant.[3]

The general case, where several variables of tier greater than or equal to $\mathbf{t}_{out}$ occur inside a while loop, only differs by a constant factor as only a finite number of different assignments on these variables may happen. □

7. Soundness

In this section, we show a soundness result: the type-2 simply typed lambda-closure of programs in ST is included in the class of basic feasible functionals BFF_2 [Meh76, KC91, KC96]. For that purpose, we use the characterization of [KS18] based on moderately polynomial time functionals. We show that terminating and safe program can be simulated by oracle Turing machines with a polynomial step count and a finite lookahead revision. We discuss briefly the requirement of the lambda-closure in §7.2.

7.1. **Moderately Polynomial Time Functionals.**
We consider oracle Turing machines M_ϕ with one query tape and one answer tape for oracle calls. If a query is written on the query tape and the machine enters a *query state*, then the oracle's answer appears on the answer tape in one step.

Definition 26. *Given an oracle TM M_ϕ, computing a total function, and an input $\mathbf{a}$, let $m_{\mathbf{a}}^{M_\phi}$ be the maximum of the size of the input $\mathbf{a}$ and of the biggest oracle answer in the run of machine on input $\mathbf{a}$ with oracle ϕ. A machine M_ϕ has:*

- *a polynomial step count if there is a polynomial P such that for any input $\mathbf{a}$ and oracle ϕ, M runs in time bounded by $P(m_{\mathbf{a}}^{M_\phi})$.*
- *finite lookahead revision if there exists a natural number $r \in \mathbb{N}$ such that for any oracle and any input, in the run of the machine, it happens at most r times that a query is posed whose size exceeds the size of all previous queries.*

Definition 27 (Moderately Polynomial Time)**.** MPT *is the class of second order functionals computable by an oracle TM with a polynomial step count and finite lookahead revision.*

The set of functionals over words $\mathbb{W}$ is defined to be the set of functions of type $\tau_1 \to \ldots \to \tau_n \to \mathbb{W}$, where the each type τ_i is defined inductively by $\tau ::= \mathbb{W} \mid \tau \to \tau$.

Suppose given a countably infinite number of variables $x^\tau, y^\tau, \ldots$, for each type τ. For a given class of functionals X, let $\lambda(X)$ be the set of closed simply typed lambda-terms generated inductively as follows:

- for each type τ, variables $x^\tau, y^\tau, \ldots$ are terms,
- each functional $F \in X$ of type τ is a term,
- for any term t of type τ' and variable x^τ, $\lambda x.t$ is a term of type $\tau \to \tau'$,
- for any terms t of type $\tau \to \tau'$ and s of type τ, $t\ s$ is a term of type τ'.

Each lambda-term of type τ represents a functional of type τ and terms are considered up to β and η equivalence. The level of a type is defined inductively by $lev(\mathbb{W}) = 0$ and $lev(\tau \to \tau') = \max(lev(\tau) + 1, lev(\tau'))$. For a given class of functionals X, let X_2 be the set of functionals of level 2.

---

[3]Intuitively, this constant corresponds to the number of consecutive non-nested while loops.

Lemma 28 (Monotonicity)**.** *Given two classes* X, Y *of functionals, if* $X \subseteq Y$ *then* $\lambda(X)_2 \subseteq \lambda(Y)_2$.

For a given functional F of type $\tau_1 \to \ldots \to \tau_n \to \mathbb{W}$ and variables X_i of type τ_i, we will use the notation $F(X_1, \ldots, X_n)$ as a shorthand notation for $F \ (X_1) \ \ldots \ (X_n)$.

We are now ready to state the characterization of Basic Feasible Functionals in terms of moderately polynomial time functions.

Theorem 29 ([KS18])**.** $\lambda(\mathrm{MPT})_2 = \mathrm{BFF}_2$.

7.2. **Proof of soundness.** At this point we are able to give a clearer statement of the relationship between the size of a derivation for a safe program $\mathbf{p}_\phi$ and the running time of a corresponding sequential execution of $\mathbf{p}_\phi$. To make this precise, the running time of M_ϕ for a given input $a \in \mathbb{W}$ is just the number of steps that it takes to terminate on with oracle ϕ, starting with a on its input tape (or undefined if the computation does not terminate). Given a store μ, this may appropriately be encoded by a single input a_μ. We then have

Proposition 30. *Suppose that* $\mathbf{p}_\phi$ *is a safe program. There are an oracle TM* M_ϕ *and a polynomial* P *such that for any derivation* $\pi_\phi : \mu \vDash \mathbf{p}_\phi \to w$, M_ϕ *on input* a_μ *simulates the execution of* $\mathbf{p}_\phi$ *with initial store* μ *in time* $O(P(|\pi_\phi|))$.

Proof. By induction on the structure of the derivation of $\mathbf{p}_\phi$. $\square$

An oracle ϕ' is padded if there exists an oracle ϕ such that for any binary word w and integer n, $\phi'(w10^n) = \phi(w)$. Let $\tilde{\phi}$ denote the padded version of ϕ.

Before showing Theorem 33, we first show that the complexity class MPT is invariant through padding.

Proposition 31. $f \in \mathrm{MPT}$ *if and only if there exists* $f' \in \mathrm{MPT}$ *such that for any input* a *and any oracle* ϕ, $f(\phi)(a) = f'(\tilde{\phi})(a)$.

Proof. Both directions can be proved through simple rewriting of the functions.
- It is trivial that if f on ϕ has finite lookahead revision k, replacing oracle calls $\phi(w)$ to calls to the padded oracle $\tilde{\phi}(w1)$ does not modify the lookahead revision.
- For the other direction, assume f' works on padded oracle $\tilde{\phi}$. Then we can design f working on ϕ such that each call to $\tilde{\phi}(w10^n)$ is replaced by 2 successive calls: $\phi(w10^n)$ and $\phi(w)$, the first being unused. This preserves the lookahead revision. Note that this construction may change the maximum of the lengths of the inputs and oracle answers. However, it may only increase this value, hence, if f' has a polynomial step count, then so has f. $\square$

Now we can show a soundness result stating that any terminating and safe program computes a second order function in MPT.

Proposition 32. $[\![\mathrm{ST}]\!] \subseteq \mathrm{MPT}$.

Proof. Given an ST program $\mathbf{p}_\phi$, by Theorem 21, $\mathbf{p}_\phi$ has a polynomial step count. By Theorem 25, $\mathbf{p}_\phi$ has finite lookahead revision and, consequently, it computes a function f' in MPT, however this function needs its oracles to be padded. By Proposition 31, f' is equivalent to a function f of MPT on standard oracles. $\square$

Theorem 33 (Soundness)**.** $\lambda([\![\mathrm{ST}]\!])_2 \subseteq \mathrm{BFF}_2$.

$$\frac{\Gamma(\mathtt{x}) = \alpha}{\Gamma, \Delta \vdash_2 \mathtt{x} : \alpha} \; (\mathrm{V}_2) \qquad \frac{\forall i \leq n, \; \Gamma, \Delta \vdash_2 \mathtt{e}_i : \alpha_i \quad \overline{\alpha} \rightarrow \alpha \in \Delta(\mathtt{op})}{\Gamma, \Delta \vdash_2 \mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_{ar(\mathtt{op})}) : \alpha} \; (\mathrm{OP}_2)$$

$$\frac{\Gamma, \Delta \vdash_2 \mathtt{x} : \alpha \quad \Gamma, \Delta \vdash_2 \mathtt{e} : \beta \quad \alpha \preceq \beta}{\Gamma, \Delta \vdash_2 \mathtt{x} := \mathtt{e} : \alpha} \; (\mathrm{A}_2) \qquad \frac{}{\Gamma, \Delta \vdash_2 \mathtt{skip} : \alpha} \; (\mathrm{SK}_2)$$

$$\frac{\Gamma, \Delta \vdash_2 \mathtt{c} : \alpha \quad \Gamma, \Delta \vdash_2 \mathtt{c}' : \beta}{\Gamma, \Delta \vdash_2 \mathtt{c}; \mathtt{c}' : \alpha \vee \beta} \; (\mathrm{S}_2) \qquad \frac{\Gamma, \Delta \vdash_2 \mathtt{e} : \alpha \quad \Gamma, \Delta \vdash_2 \mathtt{c} : \alpha \quad \Gamma, \Delta \vdash_2 \mathtt{c}' : \alpha}{\Gamma, \Delta \vdash_2 \mathtt{if}(\mathtt{e})\{\mathtt{c}\} \; \mathtt{else} \; \{\mathtt{c}'\} : \alpha} \; (\mathrm{C}_2)$$

$$\frac{\Gamma, \Delta \vdash_2 \mathtt{e} : \mathbf{1} \quad \Gamma, \Delta \vdash_2 \mathtt{c} : \alpha}{\Gamma, \Delta \vdash_2 \mathtt{while}(\mathtt{e})\{\mathtt{c}\} : \mathbf{1}} \; (\mathrm{W}_2)$$

FIGURE 6. 2-tier-based type system for 2ST

Proof. By Proposition 32, $[\![\mathrm{ST}]\!] \subseteq \mathrm{MPT}$. By Lemma 28 and Theorem 29, $\lambda([\![\mathrm{ST}]\!])_2 \subseteq \lambda(\mathrm{MPT})_2 = \mathrm{BFF}_2$. □

Notice that the lambda-closure of $[\![\mathrm{ST}]\!]$ is mandatory for characterizing BFF_2 as it is known that MPT is strictly included in BFF_2. In particular, the following counter-example taken from [KS18] and computing a function of BFF_2 cannot be typed as all our oracle calls have input bounds.

Example 34. *The functional F defined below is in BFF_2 but not in MPT.*

$$F(\phi, \epsilon) = \epsilon$$
$$F(\phi, \mathtt{suc}_1(n)) = \phi \circ \phi(F(\phi, n) \upharpoonright \phi(\epsilon))$$

Consequently, F is not in $[\![\mathrm{ST}]\!]$, since $[\![\mathrm{ST}]\!] \subseteq \mathrm{MPT}$. Indeed, the outermost oracle call is performed without any oracle input bound and can clearly not be captured by typable programs.

8. COMPLETENESSES AT TYPE-1 AND TYPE-2

Completeness is demonstrated in two steps. First, we show that each type 1 polynomial time computable function FP can be computed by a terminating program in ST, with no oracle calls. For that purpose, we show that the 2-tier sequential version of the type system of [MP14], characterizing FP, is a strict subsystem of the type system of Figure 3. Second, we show that the bounded iterator functional $\mathcal{I}'$ of [KS19] can be simulated by a terminating and typable program in ST. The completeness follows as the type-2 simply typed lambda-closure of the bounded iterator $\mathcal{I}'$ and the functions of FP provides an alternative characterization of BFF_2.

8.1. **A characterization of** FP. For that purpose, we consider the 2-tier based characterization of FP in [MP14], restricted to one single thread. Let α, β be tier variables ranging over $\{\mathbf{0}, \mathbf{1}\}$. The type system is provided in Figure 6, where $\overline{\alpha}$ stands for $\alpha_1 \rightarrow \ldots \rightarrow \alpha_{ar(\mathtt{op})}$.

In this particular context, the notion of *2-tier safe program* is defined as follows. A 2-tier operator typing environment Δ is a mapping that associates to each operator $\mathtt{op}$ a set of operator types $\Delta(\mathtt{op})$, of the shape $\alpha_1 \rightarrow \ldots \rightarrow \alpha_{ar(\mathtt{op})} \rightarrow \alpha$, with $\alpha_i, \alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Definition 35. *A program is 2-tier safe if it can be typed using 2-tier operator typing environment Δ satisfying, for any $\mathtt{op} \in dom(\Delta)$, $[\![op]\!] \in \mathrm{FP}$, $\mathtt{op}$ is either positive or neutral, and for each $\alpha_1 \to \ldots \to \alpha_{ar(\mathtt{op})} \to \alpha \in \Delta(\mathtt{op})$:*

- $\alpha \preceq \wedge_{i=1,n}\alpha_i$ *and*
- *if $\mathtt{op}$ is positive but not neutral then $\alpha = \mathbf{0}$.*

Let $2\mathrm{ST}$ *be the set of 2-tier safe and terminating programs and* $[\![2\mathrm{ST}]\!]$ *be the set of functions computed by these programs.*

Theorem 36 (Theorem 7 of [MP14]). $[\![2\mathrm{ST}]\!] = \mathrm{FP}$.

We first show that the set of 2-tier safe programs is (strictly) embedded in the set of safe programs. We define a naive translation $()^\star$ from 2-tier operator typing environments to typing environments as follows: $(\Delta)^\star$ is the operator typing environment defined by $\forall \mathtt{op}, (\Delta)^\star(\mathtt{op})(\mathbf{t}) = \Delta(\mathtt{op})$, if $\mathbf{t} = \mathbf{1}$, and $\emptyset$, otherwise.

Lemma 37. *For any command or expression b, if $\Gamma, \Delta \vdash_2 b : \alpha$ then $\Gamma, (\Delta)^\star \vdash b : (\alpha, \mathbf{1}, \mathbf{1})$.*

Proof. By an easy induction on the typing derivation of $\Gamma, \Delta \vdash_2 b : \alpha$:

- If the last rule is (S_2), then $b = \mathtt{c_1; c_2}$ and $\Gamma, (\Delta)^\star \vdash \mathtt{c_i} : (\alpha_i, \mathbf{1}, \mathbf{1})$ by the induction hypothesis, for α_i such that $\alpha = \alpha_1 \vee \alpha_2$. Consequently, $\Gamma, (\Delta)^\star \vdash b : (\alpha, \mathbf{1}, \mathbf{1})$ can be derived applying rule (C) and at most once rule (SUB) (in the case where $\alpha_1 \neq \alpha_2$).
- If the last rule is (W_2) then $\Gamma, \Delta \vdash_2 b : \mathbf{1}$ for $b = \mathtt{while(e)\{c'\}}$ and, by the induction hypothesis, $\Gamma, (\Delta)^\star \vdash \mathtt{e} : (\mathbf{1}, \mathbf{1}, \mathbf{1})$ and $\Gamma, (\Delta)^\star \vdash \mathtt{c'} : (\alpha, \mathbf{1}, \mathbf{1})$, for some α. Consequently, $\Gamma, (\Delta)^\star \vdash b : (\mathbf{1}, \mathbf{1}, \mathbf{1})$ can be derived.
- If the last rule is (OP_2), then $b = \mathtt{op(\bar{e})}$, for some operator $\mathtt{op}$ and expressions $\bar{\mathtt{e}} = \mathtt{e_1}, \ldots, \mathtt{e}_{ar(\mathtt{op})}$ such that $\Gamma, \Delta \vdash_2 \mathtt{e_i} : \alpha_i$ and $\alpha_1 \to \ldots \alpha_n \to \alpha \in \Delta(\mathtt{op})$. By the induction hypothesis, $\Gamma, (\Delta)^\star \vdash \mathtt{e_i} : (\alpha_i, \mathbf{1}, \mathbf{1})$. Moreover, $\alpha_1 \to \ldots \alpha_n \to \alpha \in (\Delta)^\star(\mathtt{op})(\mathbf{1}) = \Delta(\mathtt{op})$ and, consequently, $\Gamma, (\Delta)^\star \vdash b : (\alpha, \mathbf{1}, \mathbf{1})$ can be derived using rule (OP).
- the cases where the last rule is (SK_2), (C_2), (V_2) and (A_2) can be straightforwardly simulated by (SK)+(SUB), (C), (V) and (A), respectively. □

Lemma 38. $2\mathrm{ST} \subsetneq \mathrm{ST}$.

Proof. Consider a tier-2 safe program $\mathtt{p}_\phi = \mathtt{c \ return \ x}$ in $2\mathrm{ST}$. There exist typing environments Γ, Δ such that $\Gamma, \Delta \vdash_2 \mathtt{c} : \alpha$. Moreover, for each $\alpha \to \ldots \to \alpha_n \to \alpha \in \Delta(\mathtt{op})$:

- $\alpha \preceq \wedge_{i=1,n}\alpha_i$ and
- if $\mathtt{op}$ is positive but not neutral then $\alpha = \mathbf{0}$.

By Lemma 37, $\Gamma, (\Delta)^\star \vdash \mathtt{c} : (\alpha, \mathbf{1}, \mathbf{1})$, and $\forall \mathtt{op}, (\Delta)^\star(\mathtt{op})(\mathbf{1}) = \Delta(\mathtt{op})$. Consequently, for all $\alpha_1 \to \alpha_n \to \alpha \in (\Delta)^\star(\mathtt{op})(\mathbf{1})$, the following hold:

- $\alpha \preceq \wedge_{i=1,n}\alpha_i \preceq \vee_{i=1,n}\alpha_i \preceq \mathbf{1}$ as $\forall i, \ \alpha_i \in \{\mathbf{0}, \mathbf{1}\}$ and
- if $\mathtt{op}$ is positive but not neutral then $\alpha \prec \mathbf{1}$, as $\alpha = \mathbf{0}$.

Hence $(\Delta)^\star$ is safe and $\mathtt{p}_\phi$ is in ST. The inclusion is strict as $2\mathrm{ST}$ programs have no oracle call. □

Let $[\![\mathrm{ST}]\!]_1$ be defined as the set of type-1 functions computed by safe and terminating programs with no oracle calls, $[\![\mathrm{ST}]\!]_1 = \{\lambda w.[\![\mathtt{p}_\phi]\!](w) \mid \phi \notin \mathtt{p}_\phi \text{ and } \mathtt{p}_\phi \in \mathrm{ST}\}$.

Theorem 39. $\mathrm{FP} = [\![\mathrm{ST}]\!]_1$.

Proof. By Theorem 36, any function f in FP is computable by a 2-tier safe and terminating program $\mathtt{p}_\phi$ with no oracle calls, *i.e.*, $f = \lambda w.[\![\mathtt{p}_\phi]\!](w)$. By Lemma 38, $\mathtt{p}_\phi$ is in ST and, consequently, $f \in [\![\mathrm{ST}]\!]_1$. Conversely, by Corollary 22, if $\mathtt{p}_\phi \in \mathrm{ST}$ then $\mathtt{p}_\phi$ has a polynomial step count. Since there is no oracle call, it means that the runtime of $\mathtt{p}_\phi$ is bounded by $P(|w|)$ for some polynomial P. Consequently, $[\![\mathrm{ST}]\!]_1 \subseteq \mathrm{FP}$. $\square$

Note that the completeness part of the above Theorem (FP $\subseteq [\![\mathrm{ST}]\!]_1$) can also be proved directly by simulating polynomials over unary numbers and Turing Machines with a program in ST as in the completeness proof of [Mar11].

8.2. **Type two iteration.** [KS19] introduces a bounded iterator functional $\mathcal{I}'$ of type $(\mathbb{W} \to \mathbb{W}) \to \mathbb{W} \to \mathbb{W} \to \mathbb{W} \to \mathbb{W}$ defined by:

$$\mathcal{I}'(F, a, b, c) = (\lambda x.F(lmin(x, a)))^{|c|}(b),$$

where $lmin$ is a functional of type $\mathbb{W} \to \mathbb{W} \to \mathbb{W}$ defined by:

$$lmin(a, b) = \left\{ \begin{array}{ll} a, & \text{if } |a| < |b|, \\ b, & \text{otherwise.} \end{array} \right.$$

In [KS19], using Cook's notion [Coo92] of polynomial time reducibility, it is shown that this functional is polynomial time-equivalent to the recursor $\mathcal{R}$ of [CU93]. As a consequence of the Cook-Urquhart Theorem, the following characterization is obtained.

Theorem 40 ([KS19]). $\lambda(\mathrm{FP} \cup \{\mathcal{I}'\})_2 = \mathrm{BFF}_2$.

Our proof of type-2 completeness will mostly rely on the use of this latter characterization of BFF_2.

Theorem 41 (Type-2 completeness). $\mathrm{BFF}_2 \subseteq \lambda([\![\mathrm{ST}]\!])_2$.

Proof. By Theorem 39, $\mathrm{FP} = [\![\mathrm{ST}]\!]_1$, hence $\mathrm{FP} \subseteq \lambda([\![\mathrm{ST}]\!])_1$ by definition of lambda-closure.
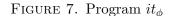
Now we show that $\mathcal{I}'$ can be computed by a terminating program in ST. For that purpose, assume that $lmin$ is an operator of our language. $lmin$ is neutral, by definition. The program it_ϕ, written in Figure 7, computes the functional $\lambda\phi.\lambda a.\lambda b.\lambda c.\mathcal{I}'(\phi, a, b, c)$ and can be typed by $(\mathbf{1}, \mathbf{1}, \mathbf{0})$, as described in Figure 8, under the typing environment Γ such that $\Gamma(c) = \Gamma(a) = \mathbf{1}$, $\Gamma(\mathtt{x}) = \Gamma(b) = \mathbf{0}$ and operator typing environment Δ such that $\mathbf{0} \to \mathbf{1} \to \mathbf{0} \in \Delta(lmin)(\mathbf{1})$ and $\mathbf{1} \to \mathbf{1} \in \Delta(> 0)(\mathbf{1})$, and $\mathbf{1} \to \mathbf{1} \in \Delta(\mathtt{pred})(\mathbf{1})$. Note that the simulation uses the padded oracle variant $\tilde{\phi}$ of ϕ. Recall that for each word w and each integer n, $\tilde{\phi}(w10^n) = \phi(w)$. Consequently, the iteration of $\tilde{\phi}$ in the program it_ϕ simulates the iteration of ϕ, when provided as input of $\mathcal{I}'$ (see also Proposition 31).

As $\mathrm{FP} \subseteq \lambda([\![\mathrm{ST}]\!])_1$ and $\mathcal{I}' \in [\![\mathrm{ST}]\!]$. We have that $\lambda(\mathrm{FP} \cup \{\mathcal{I}'\})_2 \subseteq \lambda([\![\mathrm{ST}]\!])_2$ and the result follows by Theorem 40. $\square$

To illustrate the need of the type-2 lambda closure for achieving completeness, consider a variant of Example 34:

$$F'(\phi, \epsilon) = \epsilon$$
$$F'(\phi, \mathtt{suc}_1(n)) = \phi \circ \phi(lmin(F'(\phi, n), \phi(\epsilon)))$$

This functional is in BFF_2 but neither in MPT nor in ST as, by essence, it has no finite lookahead revision. Indeed, the outermost oracle call input data is not bounded and iterated

```
x⁰ := b⁰;
while(c ≠ ε)¹{
    x⁰ := φ̃(lmin(x,a)⁰ ↾ a¹)⁰;
    c¹ := pred(c)¹}
}
return  x
```

FIGURE 7. Program it_ϕ

FIGURE 8. Typing derivation for program it_ϕ

linearly in the input. However it can be computed by $\lambda\phi.\lambda n.(\mathcal{I}' \ (\lambda x.\phi \ (\phi \ x)) \ \phi(\epsilon) \ \epsilon \ n)$ and is in $\lambda(\llbracket ST \rrbracket)_2$, as $\mathcal{I}' = \llbracket it_\phi \rrbracket \in \llbracket ST \rrbracket$, it_ϕ being the program in the proof of Theorem 41, and computes the functional $\lambda\phi.\lambda n.F'(\phi, n)$.

9. Other properties

9.1. Intensional and extensional properties of tiers.
The type system of Figure 3 enjoys several other properties of interest. First, completeness can be achieved using only 2 tiers (at the price of worse expressive power). Second, type inference is decidable in polynomial time in the size of the program.

Let $\llbracket ST^{\mathbf{t}} \rrbracket$ be the subset of functionals of $\llbracket ST \rrbracket$ computable by terminating and typable programs using tiers bounded by $\mathbf{t}$. Formally, $\mathbf{p}_\phi \in ST^{\mathbf{t}}$ if and only if $\mathbf{p}_\phi$ is terminating and $\Gamma, \Delta \vdash \mathbf{p}_\phi : (\mathbf{t}', \mathbf{t}'_{in}, \mathbf{t}'_{out})$ for a safe operator typing environment Δ and a variable typing environment Γ such that $\forall \mathbf{x} \in \mathcal{V}(\mathbf{p}_\phi)$, $\Gamma(\mathbf{x}) \preceq \mathbf{t}$.

We can show that tiers allow strictly more expressive power in terms of captured programs. However tiers greater than $\mathbf{1}$ are equivalent from an extensional point of view.

Proposition 42. *The following properties hold:*
(1) $\forall \mathbf{t} \succeq \mathbf{0}$, $ST^{\mathbf{t}} \subsetneq ST^{\mathbf{t}+1}$,
(2) $\forall \mathbf{t} \succeq \mathbf{1}$, $\lambda(\llbracket ST^{\mathbf{t}} \rrbracket)_2 = BFF_2$.

Proof. (1) The inclusion is trivial. For any tier $\mathbf{t}+1$, it is easy to enforce the tier of one variable of a safe and terminating program to be $\mathbf{t}+1$ using $\mathbf{t}$ sequential while loops of the shape:

```
while(x_{t+1} > 0){x_{t+1} := pred(x_{t+1}); x_t := suc_1(x_t)};
...
while(x_1 > 0){x_1 := pred(x_1); x_0 := suc_1(x_0)}
```

Consequently, the inclusion is strict.

(2) The proof of Theorem 41, only makes use of programs of tier smaller than $\mathbf{1}$. Consequently, $\lambda(\llbracket \mathrm{ST}^{\mathbf{1}} \rrbracket)_2 = \mathrm{BFF}_2$. By Proposition 42, $\mathrm{ST}^{\mathbf{t}} \subsetneq \mathrm{ST}^{\mathbf{t}+1}$ and, consequently, $\forall \mathbf{t},\ \llbracket \mathrm{ST}^{\mathbf{t}} \rrbracket \subseteq \llbracket \mathrm{ST}^{\mathbf{t}+1} \rrbracket$. We obtain that $\forall \mathbf{t},\ \mathrm{BFF}_2 \subseteq \lambda(\llbracket \mathrm{ST}^{\mathbf{t}} \rrbracket)_2 \subseteq \lambda(\llbracket \mathrm{ST} \rrbracket)_2 = \mathrm{BFF}_2$ and so the result. $\qquad\square$

Proposition 42 implies that the use of exactly 2 tiers is sufficient to achieve completeness but weakens the type system expressive power.

9.2. **Decidability of type inference.**

Proposition 43. *Given a program p_ϕ of size n and a safe operator typing environment Δ, deciding if there exists a variable typing environment Γ such that $\mathsf{p}_\phi \in \mathrm{ST}^{\mathbf{t}}$ can be done in time $\mathcal{O}(n^2 \times \mathbf{t})$.*

Proof. The proof follows the type inference proof of [HMP13]: the tier of each variable x is encoded using $3(\mathbf{t}+1)$ boolean variables $x_{\mathbf{0}}^{tier}$, $x_{\mathbf{1}}^{tier}$, $\ldots$, $x_{\mathbf{t}}^{tier}$, $x_{\mathbf{0}}^{in}$, $x_{\mathbf{1}}^{in}$, $\ldots$, $x_{\mathbf{t}}^{in}$, $x_{\mathbf{0}}^{out}$, $x_{\mathbf{1}}^{out}$, $\ldots$, $x_{\mathbf{t}}^{out}$. The same encoding can be done for an expression e and a command c using the boolean variables e and c. These variables can be interpreted as follows: $x_{\mathbf{t}}^{in}$ is true whenever the innermost tier of variable x is at most $\mathbf{t}$ and $c_{\mathbf{1}}^{tier}$ is false whenever the tier of command c is strictly more than $\mathbf{1}$. For example, if the tier of x is $\mathbf{1}$, then $x_{\mathbf{0}}^{tier}$ is false and $x_{\mathbf{1}}^{tier}, \ldots, x_{\mathbf{t}}^{tier}$ are true.

The tiers of each boolean variable $a \in \{x, e, c\}$ is enforced to be correctly encoded by the following propositional formula $\bigwedge_{k \in \{tier, in, out\}} \bigwedge_{i \prec \mathbf{t}} (a_i^k \implies a_{i+1}^k)$, which is equivalent to $\bigwedge_{k \in \{tier, in, out\}} \bigwedge_{i \prec \mathbf{t}} (\neg a_i^k \vee a_{i+1}{}^k)$. This accounts for $3\mathbf{t}$ clauses for each variable.

Equality of the k-tier of variables a and the l-tier of variable b (with $k, l \in \{tier, in, out\}$) can be expressed as:

$$\bigwedge_i (\neg a_i^k \vee b_i{}^l) \wedge \bigwedge_i (a_i^k \vee \neg b_i{}^l).$$

This accounts for $2(\mathbf{t}+1)$ clauses.

Strict inequality of tiers, for example the tier of e strictly less than the innermost tier of d, can be encoded as:

$$\bigwedge_{i \prec \mathbf{t}} (\neg d_{i+1}^{in} \vee e_i^{tier}) \wedge \neg d_0^{in} \wedge e_{\mathbf{t}}^{tier}.$$

This accounts for $\mathbf{t}+2$ clauses.

Open inequality of tiers, for example the tier of variable x is at most the tier of variable y, can be encoded as:

$$\bigwedge_i (\neg y_i^{tier} \vee x_i^{tier}).$$

This accounts for $\mathbf{t}+1$ clauses.

Now we inspect each of the program constructs relatively to the corresponding typing rule in Figure 3:

- Consider typing rule (A).

$$\frac{\Gamma, \Delta \vdash \mathtt{x} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{e} : (\mathbf{t}_2, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \mathbf{t}_1 \preceq \mathbf{t}_2}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{e} \ : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (A).$$

The typing of command $\mathtt{c} \triangleq \mathtt{x} := \mathtt{e}$ translates to 1 open inequality and 5 equalities, accounting for $11(\mathbf{t}+1)$ clauses:

$$\begin{aligned}
&\bigwedge_i (\neg e_i^{tier} \vee x_i^{tier}) \wedge \\
&\bigwedge_i (\neg c_i^{tier} \vee x_i^{tier}) \wedge \bigwedge_i (c_i^{tier} \vee \neg x_i^{tier}) \wedge \\
&\bigwedge_i (\neg c_i^{in} \vee x_i^{in}) \wedge \bigwedge_i (c_i^{in} \vee \neg x_i^{in}) \wedge \\
&\bigwedge_i (\neg c_i^{in} \vee e_i^{in}) \wedge \bigwedge_i (c_i^{in} \vee \neg e_i^{in}) \wedge \\
&\bigwedge_i (\neg c_i^{out} \vee x_i^{out}) \wedge \bigwedge_i (c_i^{out} \vee \neg x_i^{out}) \wedge \\
&\bigwedge_i (\neg c_i^{out} \vee e_i^{out}) \wedge \bigwedge_i (c_i^{out} \vee \neg e_i^{out})
\end{aligned}$$

- Similarly, consider typing rule (W):

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out}) \quad \Gamma, \Delta \vdash \mathtt{c} : (\mathbf{t}_1, \mathbf{t}_1, \mathbf{t}_{out}) \quad \mathbf{1} \preceq \mathbf{t}_1 \preceq \mathbf{t}_{out}}{\Gamma, \Delta \vdash \mathtt{while(e)\{c\}} \ : (\mathbf{t}_1, \mathbf{t}_{in}, \mathbf{t}_{out})} \ (W).$$

The typing of command $\mathtt{w} \triangleq \mathtt{while(e)\{c\}}$ translates to 2 inequalities ($\mathbf{1} \preceq \mathbf{t}_1$ which is trivial and $\mathbf{t}_1 \preceq \mathbf{t}_{out}$) and 6 equalities, accounting for $13(\mathbf{t}+1)+1$ clauses:

$$\begin{aligned}
&\neg w_0^{tier} \wedge \\
&\bigwedge_i \neg w_i^{out} \vee w_i^{tier} \wedge \\
&\bigwedge_i \neg e_i^{tier} \vee w_i^{tier} \wedge \bigwedge_i e_i^{tier} \vee \neg w_i^{tier} \wedge \\
&\bigwedge_i \neg c_i^{tier} \vee w_i^{tier} \wedge \bigwedge_i c_i^{tier} \vee \neg w_i^{tier} \wedge \\
&\bigwedge_i \neg c_i^{in} \vee w_i^{tier} \wedge \bigwedge_i c_i^{in} \vee \neg w_i^{tier} \wedge \\
&\bigwedge_i \neg e_i^{in} \vee w_i^{in} \wedge \bigwedge_i e_i^{in} \vee \neg w_i^{in} \wedge \\
&\bigwedge_i \neg c_i^{out} \vee w_i^{out} \wedge \bigwedge_i c_i^{out} \vee \neg w_i^{out} \wedge \\
&\bigwedge_i \neg e_i^{out} \vee w_i^{out} \wedge \bigwedge_i e_i^{out} \vee \neg w_i^{out} \wedge
\end{aligned}$$

- The number of equalities used when encoding a rule (OP) is $3 \times ar(op) + 1$, which needs in total $(3 \times ar(op) + 1) \times (2(\mathbf{t}+1))$ clauses. Hence $\mathcal{O}(n \times \mathbf{t})$ clauses.

Finally, the type inference problem can be reduced to 2-SAT with $\mathcal{O}(n^2 \times \mathbf{t})$ clauses, which can be solved in time linear in the number of clauses [EIS76, APT79]. □

Theorem 44. *Given a program $\mathtt{p}_\phi$ and a safe operator typing environment Δ, deciding if there exists a variable typing environment Γ such that $\mathtt{p}_\phi \in \mathrm{ST}$ can be done in time cubic in the size of the program.*

Proof. The maximal tier needed to type a program can be bounded by the size of the program as the number of strict inequalities on tiers is fixed by the number of rules (OP) (in the case of a positive operator) and (OR) needed to type a program. Consequently, with n the size of $\mathtt{p}_\phi$, we can simply check if $\mathtt{p}_\phi \in \mathrm{ST}^n$. By Proposition 43, this means that we can decide if $\mathtt{p}_\phi \in \mathrm{ST}$ in time $\mathcal{O}(n^3)$. □

10. Conclusion and future work

We have presented a first tractable characterization of the class of type-2 polynomial time computable functionals BFF_2 based on a simple imperative programming language. This characterization does not require any explicit and external resource bound and its restriction to type-1 provides an alternative characterization of the class FP.

The presented type system can be generalized to programs with a constant number of oracles (the typing rule for oracles remains unchanged). However the lambda closure is mandatory for completeness as illustrated by Example 34. An open issue of interest is to get rid of this closure in order to obtain a characterization of BFF_2 in terms of a pure imperative programming language. Indeed, in our context, programs can be viewed as a simply typed lambda-terms with typable and terminating imperative procedure calls. One suggestion is to study to which extent oracle composition can be added directly to the program syntax.

Another issue of interest is to study whether this non-interference based approach could be extended (or adapted within the context of light logics) to characterize BFF_2 on a pure functional language. We leave these open issues as future work.

Acknowledgements.

References

[APT79]   Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[BAJK08] Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In *Logic and Theory of Algorithms*, pages 67–76. Springer, 2008.

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

[BL16]   Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Information and Computation*, 248:56–81, 2016.

[BM10]   Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theoretical Computer Science*, 411(2):470–503, 2010.

[BMM11]   Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

[BT04]   Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *Logic in Computer Science, LICS 2004*, pages 266–275. IEEE, 2004.

[CK89]   Stephen A. Cook and Bruce M. Kapron. Characterizations of the basic feasible functionals of finite type. In *Symposium on Foundations of Computer Science, FOCS 1989*, pages 154–159. IEEE, 1989.

[Cob65]   Alan Cobham. The intrinsic computational difficulty of functions. In *International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1965.

[Con73]   Robert L. Constable. Type two computational complexity. In *Symposium on Theory of Computing, STOC 1973*, pages 108–121. ACM, 1973.

[Coo92]   Stephen A. Cook. Computability and complexity of higher type functions. In *Logic from Computer Science*, pages 51–72. Springer, 1992.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: beyond safety. In *International Conference on Computer Aided Verification, CAV 2006*, pages 415–418. Springer, 2006.

[CU93] Stephen A. Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, 1993.

[DR06] Norman Danner and James S. Royer. Adventures in time and space. In *Symposium on Principles of Programming Languages, POPL 2006*, pages 168–179. ACM, 2006.

[EIS76] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

[FHHP15] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 585:41–54, 2015.

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[GMR08] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. A logical account of PSPACE. In *Symposium on Principles of Programming Languages, POPL 2008*, pages 121–131. ACM, 2008.

[Háj79] Petr Hájek. Arithmetical hierarchy and complexity of computation. *Theoretical Computer Science*, 8:227–237, 1979.

[HKMP20] Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. A tier-based typed programming language characterizing feasible functionals. In *Symposium on Logic in Computer Science, LICS 2020*, pages 535–549, 2020.

[HMP13] Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *International Conference on Foundations of Software Science and Computational Structures, FoSSaCS 2013*, pages 305–320. Springer, 2013.

[HP15] Emmanuel Hainry and Romain Péchoux. Objects in polynomial time. In *Asian Symposium on Programming Languages and Systems, APLAS 2015*, Lecture Notes in Computer Science, pages 387–404. Springer, 2015.

[HP17] Emmanuel Hainry and Romain Péchoux. Higher order interpretation for higher order complexity. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2017*, pages 269–285. EasyChair, 2017.

[IRK01] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. On characterizations of the basic feasible functionals (part I). *Journal of Functional Programming*, 11(1):117–153, 2001.

[JK09] Neil D. Jones and Lars Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *ACM Transactions on Computational Logic*, 10(4):28:1–28:41, 2009.

[KC91] Bruce M. Kapron and Stephen A. Cook. A new characterization of Mehlhorn's polynomial time functionals (extended abstract). In *Symposium on Foundations of Computer Science, FOCS 1991*, pages 342–347. IEEE, 1991.

[KC96] Bruce M. Kapron and Stephen A. Cook. A new characterization of type-2 feasibility. *SIAM Journal on Computing*, 25(1):117–132, 1996.

[KS17] Akitoshi Kawamura and Florian Steinberg. Polynomial running times for polynomial-time oracle machines. In *International Conference on Formal Structures for Computation and Deduction, FSCD 2017*, pages 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[KS18] Bruce M. Kapron and Florian Steinberg. Type-two polynomial-time and restricted lookahead. In *Logic in Computer Science, LICS 2018*, pages 579–588. ACM, 2018.

[KS19] Bruce M. Kapron and Florian Steinberg. Type-two iteration with bounded query revision. In *Joint Workshops on Developments in Implicit Computational complExity and Foundational & Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2019*, EPTCS, pages 61–73, 2019.

[Lei95] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhäuser, Boston, MA, 1995.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages, POPL 2001*, pages 81–92. ACM, 2001.

[LM93] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1/2):167–184, 1993.

[LM13]   Daniel Leivant and Jean-Yves Marion. Evolving graph-structures and their implicit computational complexity. In *International Colloquium on Automata, Languages, and Programming, ICALP 2013, Part II*, Lecture Notes in Computer Science, pages 349–360. Springer, 2013.

[Mar11]   Jean-Yves Marion. A type system for complexity flow analysis. In *Logic in Computer Science, LICS 2011*, pages 123–132. IEEE Computer Society, 2011.

[Meh76]   Kurt Mehlhorn. Polynomial and abstract subrecursive classes. *Journal of Computer and System Sciences*, 12(2):147–178, 1976.

[Mit91]   John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[MP14]   Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *Theory and Applications of Models of Computation, TAMC 2014*, Lecture Notes in Computer Science, pages 124–140. Springer, 2014.

[VIS96]   Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.