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

Bruce M. Kapron
bmkapron@uvic.ca
University of Victoria
Canada

Jean-Yves Marion
jean-yves.marion@loria.fr
Université de Lorraine, CNRS, LORIA, F-54000 Nancy, France

Romain Péchoux
romain.pechoux@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

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 restrain strongly the expressive power of the language.

Keywords: Feasible functionals, BFF,implicit computational complexity, tiering, type-2, type system.

1 Introduction

Type-2 computational complexity aims to study classes of functions that take type-1, i.e. function, arguments. The notion of feasibility for type-2 functionals was first studied in [Constable 1973] and in [Mehlhorn 1976] using subrecursive formalisms. Later, [Cook and Kapron 1989; Cook and Urquhart 1993] 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 [Kapron and Cook 1991, 1996] 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 lended 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 [Férée et al. 2015; Kapron and Cook 1991, 1996] or a bounding function within the class [Constable 1973; Mehlhorn 1976]. This is analogous to a shortcoming in Cobham's characterization of the class of (type 1) polynomial time computable functions FP [Cobham 1965]. 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 [Cook and Kapron 1989] or from including unnatural constructs or type-2 recursion patterns [Cook and Urquhart 1993; Danner and Royer 2006; Irwin et al. 2001] 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 [Kawamura and Steinberg 2017] by constraining Cook's definition of Oracle Polynomial Time (OPT) [Cook 1992], 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 [Kawamura and Steinberg 2017], 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 [Kapron and Steinberg 2018], which considers 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 works [Bellantoni and Cook 1992] and [Leivant and Marion 1993]. 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 [Marion 2011] characterizing FP. Consequently, it relies on a non-interference principle and is also inspired by the type system of [Volpano et al. 1996] 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 4.3) 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 1.1. $\lambda([\![\mathrm{ST}]\!])_2 = \mathsf{BFF}_2$.

Soundness ($\lambda([\![\mathrm{ST}]\!])_2 \subseteq \mathsf{BFF}_2$, Theorem 7.6) is demonstrated by showing that each function of $[\![\mathrm{ST}]\!]$ is in Kapron-Steinberg's MPT class [Kapron and Steinberg 2018]. The type system makes use of several tiers and is designed to enforce a tier-based non-interference result (Theorem 5.3) and generalizes the operator type discipline of [Marion 2011] to ensure the polynomial step count property (Corollary 5.6) and the finite lookahead revision property (Theorem 5.8), 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 [Kapron and Steinberg 2018] 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 ($\mathsf{BFF}_2 \subseteq \lambda([\![\mathrm{ST}]\!])_2$, Theorem 8.6) is shown using an alternative characterization: $\lambda(\mathsf{FP} \cup \{\mathcal{I}'\})_2 = \mathsf{BFF}_2$, where $\mathcal{I}'$ is a bounded iterator that is polynomially equivalent to the recursor $\mathcal{R}$ of [Cook and Urquhart 1993], as demonstrated in [Kapron and Steinberg 2019]. The simulation of FP is performed by showing that our type system strictly embeds the tier-based type system of [Marion and Péchoux 2014]. Consequently, our type system also provides a characterization of FP (Theorem 8.4) 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 [Kapron and Steinberg 2018], 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 9.2, 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 [Mitchell 1991], 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 [Cook et al. 2006; Lee et al. 2001]. 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 1.1 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 6.4. 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^0_2-complete problem in the arithmetical hierarchy [Hájek 1979].

Outline. §4 is devoted to presenting the type system technical developments and main intuitions. §5 states the type system main properties. §6 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.

2 Related work

Implicit Computational Complexity (ICC). has lead to the development of several techniques such as interpretations [Bonfante et al. 2011], light logics [Girard 1998], mwp-bounds [Ben-Amram et al. 2008; Jones and Kristiansen 2009], and tiering [Hainry and Péchoux 2015; Leivant 1995; Leivant and Marion 2013; Marion 2011]. 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 [Baillot and Mazza 2010; Baillot and Terui 2004] or polynomial space [Gaboardi et al. 2008]. Interpretations were extended to higher-order polynomials in [Baillot and Lago 2016] to study FP and adapted in [Férée et al. 2015; Hainry and Péchoux 2017] 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 [Cook and Kapron 1989; Irwin et al. 2001] 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 [Cook and Urquhart 1993; Kapron and Steinberg 2019]: the recursion schemes are not natural and cannot be used in practice by a programmer. Several characterizations [Kapron and Cook 1991, 1996] using type-2 polynomials were also developed but they focus on machines rather than programs.

3 Imperative programming language with oracles

Syntax and semantics. Consider a set $\mathbb{V}$ of variables and a set $\mathbb{O}$ of operators op of fixed arity $ar(\mathsf{op})$. 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 and commands are defined by the grammar of Figure 1, where $\mathsf{x}, \mathsf{y} \in \mathbb{V}$, $\mathsf{op}, \restriction \in \mathbb{O}$, and ϕ is a single oracle symbol. Let $\mathcal{V}(\mathsf{p}_\phi)$ be the set of variables occurring in the program p_ϕ. An expression of the shape $\phi(\mathsf{e}_1 \restriction \mathsf{e}_2)$ is called an *oracle call*. e_1 is called the *input data*, e_2 is called the *input bound* and $\mathsf{e}_1 \restriction \mathsf{e}_2$ is called the *input*. We write $\phi \notin \mathsf{p}_\phi$ in the special case where no oracle call appears in p_ϕ.

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 $|w|$ ($|\overline{t}|$, 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 $[\![\mathsf{op}]\!] : \mathbb{W}^{ar(\mathsf{op})} \to \mathbb{W}$ 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 $w_{\restriction n}$ 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, $1001_{\restriction 0} = 1$, $1001_{\restriction 1} = 11$, $1001_{\restriction 2} = 101$, and $1001_{\restriction 6} = 1001100$. Define $\forall v, w \in \mathbb{W}$, $[\![\restriction]\!](v, w) = v_{\restriction |w|}$. Padding ensures that $|[\![\restriction]\!](v, w)| = |w| + 1$. 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.

The oracle symbol ϕ computes a total function from $\mathbb{W}$ to $\mathbb{W}$, called an oracle function. In order to lighten notation, 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 $\mu[\mathsf{x}_1 \leftarrow w_1, \ldots, \mathsf{x}_n \leftarrow w_n]$ be a notation for the store μ' satisfying $\forall \mathsf{x} \in dom(\mu) - \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$, $\mu'(\mathsf{x}) = \mu(\mathsf{x})$ and $\forall \mathsf{x}_i \in \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$, $\mu'(\mathsf{x}_i) = w_i$. Let μ_0 be the store defined by $\forall \mathsf{x} \in dom(\mu_0)$, $\mu_0(\mathsf{x}) = \epsilon$. The size of a store μ is defined by $|\mu| = \Sigma_{\mathsf{x} \in dom(\mu)} |\mu(\mathsf{x})|$.

The operational semantics of the language is deterministic and is given in Figure 2. The judgment $\mu \vDash \mathsf{e} \to w$ means that the expression e is evaluated to the word $w \in \mathbb{W}$ with respect to the store μ The judgment $\mu \vDash \mathsf{c} \to \mu'$ expresses that, under the input store μ, the command c terminates and outputs the store μ'. In rule (Seq) of Figure 2, it is implicitly assumed that c_1 is not a sequence.

Given a derivation $\pi_\phi : \mu \vDash \mathsf{p}_\phi \to w$, let $|\pi_\phi|$ denote the size of the derivation, that is the number of nodes in the derivation tree rooted at $\mu \vDash \mathsf{p}_\phi \to w$. Note that $|\pi_\phi|$ corresponds to the number of steps in a sequential execution of p_ϕ, initialized with store μ. On the other hand, with no restriction on operators this measure is too coarse to correspond, even asymptotically, to running time. With suitable restriction, there is a correspondence, given in Proposition 7.5 below.

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

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

$$
\begin{array}{llll}
\texttt{Expressions} & \mathsf{e}, \mathsf{e}_1, \ldots & ::= & \mathsf{x} \mid \mathsf{op}(\overline{\mathsf{e}}) \mid \phi(\mathsf{e}_1 \restriction \mathsf{e}_2) \\
\texttt{Commands} & \mathsf{c}, \mathsf{c}_1, \mathsf{c}_2 & ::= & \texttt{skip} \mid \mathsf{x} := \mathsf{e} \mid \mathsf{c}_1; \mathsf{c}_2 \mid \texttt{if(e)\{c}_1\texttt{\} else \{c}_2\texttt{\}} \mid \texttt{while(e)\{c\}} \\
\texttt{Programs} & \mathsf{p}_\phi & ::= & \mathsf{c}\ \texttt{return}\ \mathsf{x}
\end{array}
$$

Figure 1. Syntax of imperative programs with oracles

$$
\frac{}{\mu \vDash \mathsf{x} \to \mu(\mathsf{x})}\ (\text{Var}) \qquad \frac{\forall i \leq ar(\mathsf{op}),\ \mu \vDash \mathsf{e}_i \to w_i}{\mu \vDash \mathsf{op}(\overline{\mathsf{e}}) \to [\![\mathsf{op}]\!](\overline{w})}\ (\text{Op}) \qquad \frac{\mu \vDash \mathsf{e}_1 \to v \quad \mu \vDash \mathsf{e}_2 \to w \quad \phi([\![\restriction]\!](v,w)) = u}{\mu \vDash \phi(\mathsf{e}_1 \restriction \mathsf{e}_2) \to u}\ (\text{Orc})
$$

$$
\frac{}{\mu \vDash \texttt{skip} \to \mu}\ (\text{Skip}) \qquad \frac{\mu \vDash \mathsf{c}_1 \to \mu_1 \quad \mu_1 \vDash \mathsf{c}_2 \to \mu_2}{\mu \vDash \mathsf{c}_1; \mathsf{c}_2 \to \mu_2}\ (\text{Seq}) \qquad \frac{\mu \vDash \mathsf{e} \to w}{\mu \vDash \mathsf{x} := \mathsf{e} \to \mu[\mathsf{x} \leftarrow w]}\ (\text{Asg})
$$

$$
\frac{\mu \vDash \mathsf{e} \to w \quad \mu \vDash \mathsf{c}_w \to \mu' \quad w \in \{0,1\}}{\mu \vDash \texttt{if(e)\{c}_1\texttt{\} else \{c}_0\texttt{\}} \to \mu'}\ (\text{Cond}) \qquad \frac{\mu \vDash \mathsf{e} \to 0}{\mu \vDash \texttt{while(e)\{c\}} \to \mu}\ (\text{Wh}_0)
$$

$$
\frac{\mu \vDash \mathsf{e} \to 1 \quad \mu \vDash \mathsf{c}; \texttt{while(e)\{c\}} \to \mu'}{\mu \vDash \texttt{while(e)\{c\}} \to \mu'}\ (\text{Wh}_1) \qquad \frac{\mu \vDash \mathsf{c} \to \mu'}{\mu \vDash \mathsf{c}\ \texttt{return}\ \mathsf{x} \to \mu'(\mathsf{x})}\ (\text{Prg})
$$

Figure 2. Big step operational semantics

for operators will depend on their category in the type system.

Definition 3.1 (Neutral and positive operators)**.**

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

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 3.2. The operator == tests whether or not its arguments are equal and the operator pred computes the predecessor.

$$
[\![==]\!](w,v) = \begin{cases} 1 & \text{if } v = w \\ 0 & \text{otherwise} \end{cases}
$$

$$
[\![\mathsf{pred}]\!](v) = \begin{cases} \epsilon & \text{if } v = \epsilon \\ u & \text{if } v = a.u,\ a \in \Sigma \end{cases}
$$

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

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 ([Kapron and Cook 1991, 1996; Mehlhorn 1976]).

4.1 Tiers and typing judgments

Atomic types are elements of the totally ordered set $(\mathbf{N}, \leq, \mathbf{0}, \vee, \wedge)$ 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 [Leivant 1995], $\leq$ is the usual ordering on integers and $\vee$ and $\wedge$ are the max and min operators over integers. Let $\prec$ be defined by $\prec := \leq \cap \neq$. We use the symbols $\mathbf{k}, \mathbf{k}', \ldots, \mathbf{k}_1, \mathbf{k}_2, \ldots$ to denote tier variables. For a finite set of tiers, $\{\mathbf{k}_1, \ldots, \mathbf{k}_n\}$, let $\vee_{i=1}^n \mathbf{k}_i$ ($\wedge_{i=1}^n \mathbf{k}_i$, respectively) denote $\mathbf{k}_1 \vee \ldots \vee \mathbf{k}_n$ ($\mathbf{k}_1 \wedge \ldots \wedge \mathbf{k}_n$, 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 $\mathbf{k} \in \mathbf{N}$ a set of admissible operator types $\Delta(\mathsf{op})(\mathbf{k})$, where the operator types corresponding to the operator op are of the shape $\mathbf{k}_1 \to \ldots \mathbf{k}_{ar(\mathsf{op})} \to \mathbf{k}'$, with $\mathbf{k}_i, \mathbf{k}' \in \mathbf{N}$.

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 $\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$ or *expression typing judgments* of the shape $\Gamma, \Delta \vdash \mathsf{e} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$. The meaning of such a typing judgment is that the *expression tier* or *command tier* is $\mathbf{k}$, the *innermost tier* is $\mathbf{k}_{in}$, and the *outermost tier* is $\mathbf{k}_{out}$. The innermost (resp. outermost) tier is the tier of the innermost (resp. outermost) while loop guard where the expression or command is located.

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

A typing derivation $\rho \rhd \Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$ is a tree whose root is the typing judgment $\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$

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 because of the non-overlapping requirements for $\mathbf{k}_{out}$ in Figure 3. The notation ρ will be used whenever mentioning the root of a typing derivation is not explicitly needed.

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 6.2). The current subsection introduces such a restriction, called *safe*.

Definition 4.1 (Safe operator typing environment)**.** An operator typing environment Δ is *safe* if for each $\mathsf{op} \in dom(\Delta)$, op is neutral or positive and $[\![\mathsf{op}]\!]$ is a polynomial time computable function, and for each $\mathbf{k}_{in} \in \mathbf{N}$, and for each $\mathbf{k}_1 \to \ldots \mathbf{k}_{ar(\mathsf{op})} \to \mathbf{k} \in \Delta(\mathsf{op})(\mathbf{k}_{in})$, the two conditions below hold:

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

Example 4.2. Consider the operators ==, $\mathtt{pred}$ and suc_i of Example 3.2. For a safe typing environment Δ, it holds that $\Delta(==)(\mathbf{1}) = \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}\} \cup \{\mathbf{k} \to \mathbf{k}' \to \mathbf{0} \mid \mathbf{k}, \mathbf{k}' \preceq \mathbf{1}\}$, as == is neutral. However $\mathbf{0} \to \mathbf{1} \to \mathbf{1} \notin \Delta(==)(\mathbf{1})$ as it breaks Condition (1) of Definition 4.1 since the operator output tier has to be smaller than each of its operand tier (i.e. $\mathbf{1} \not\preceq \mathbf{0} \wedge \mathbf{1}$).

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

For the positive operator suc_i, we have $\Delta(\mathsf{suc}_i)(\mathbf{1}) = \{\mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$. $\mathbf{1} \to \mathbf{1} \notin \Delta(\mathsf{suc}_i)(\mathbf{1})$ as the operator output tier has to be strictly smaller than $\mathbf{1}$, due to Condition (2) of Definition 4.1. Applying the same restriction, it holds that $\Delta(\mathsf{suc}_i)(\mathbf{2}) = \{\mathbf{2} \to \mathbf{1}, \mathbf{2} \to \mathbf{0}, \mathbf{1} \to \mathbf{1}, \mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$.

Definition 4.3 (Safe program)**.** Given Γ a variable typing environment and Δ a safe operator typing environment, the program $\mathsf{p}_\phi = \mathsf{c}\ \mathtt{return}\ \mathsf{x}$ is a *safe program* if there are $\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}$ such that $\rho \rhd \Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$.

Definition 4.4. Let ST be the set of safe and terminating programs and $[\![\mathrm{ST}]\!]$ be the set of functionals computed by programs in ST:

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

4.3 Some intuitions

Before providing a formal treatment of the type system main properties in §5, we provide the reader with a brief intuition of types, that are triplets of tiers $(\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$, in a typing derivation obtained by applying the typing rules of Figure 3:

- $\mathbf{k}$ 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 statements 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 statement.
- $\mathbf{k}_{in}$ 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 $\mathbf{k}$ is strictly smaller than $\mathbf{k}_{in}$,
  - in rule (OP), for a positive operator, the return type $\mathbf{k}$ is strictly smaller than $\mathbf{k}_{in}$.

  This forbids programs from iterating on a data whose size can increase during the iteration.
- $\mathbf{k}_{out}$ 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 worse expressive power.

5 Properties of safe programs

We now show the main properties of safe programs:

- a standard non-interference property in §5.2 ensuring that computations on higher tiers do not depend on lower tiers (Theorem 5.3).
- a polynomial time property in §5.3 ensuring that terminating programs terminate in time polynomial in the input size and maximal oracle's output size (Theorem 5.5).

$$\frac{\Gamma(\mathsf{x}) = \mathbf{k}}{\Gamma, \Delta \vdash \mathsf{x} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{V}) \qquad \frac{\mathbf{k}_1 \to \cdots \to \mathbf{k}_{ar(\mathsf{op})} \to \mathbf{k} \in \Delta(\mathsf{op})(\mathbf{k}_{in}) \qquad \forall i \leq ar(\mathsf{op}),\ \Gamma, \Delta \vdash \mathsf{e}_i : (\mathbf{k}_i, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathsf{op}(\overline{\mathsf{e}}) : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{OP})$$

$$\frac{\Gamma, \Delta \vdash \mathsf{e}_1 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathsf{e}_2 : (\mathbf{k}_{out}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \mathbf{k} \prec \mathbf{k}_{in} \wedge \mathbf{k} \preceq \mathbf{k}_{out}}{\Gamma, \Delta \vdash \phi(\mathsf{e}_1 \restriction \mathsf{e}_2) : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{OR}) \qquad \frac{}{\Gamma, \Delta \vdash \mathtt{skip} : (\mathbf{0}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{SK})$$

$$\frac{\Gamma, \Delta \vdash \mathsf{c}_1 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathsf{c}_2 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathsf{c}_1; \mathsf{c}_2 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{S}) \qquad \frac{\Gamma, \Delta \vdash \mathsf{x} : (\mathbf{k}_1, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathsf{e} : (\mathbf{k}_2, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \mathbf{k}_1 \preceq \mathbf{k}_2}{\Gamma, \Delta \vdash \mathsf{x} := \mathsf{e} : (\mathbf{k}_1, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{A})$$

$$\frac{\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k{+}1}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{SUB}) \qquad \frac{\Gamma, \Delta \vdash \mathsf{e} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathsf{c}_1 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathsf{c}_0 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathtt{if(e)\{c_1\}\ else\ \{c_0\}} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}\ (\mathrm{C})$$

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

Figure 3. Tier-based type system

- a finite lookahead revision property in §5.4 ensuring that, for any oracle and any input, the number of oracle calls on input of increasing size is bounded by a constant (Theorem 5.8).

5.1 Notations

Let us first introduce some preliminary notations. Let $\mathcal{E}(a)$ (res. $C(a)$) 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.

5.2 Non-interference

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

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

Lemma 5.1 (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 $\Gamma, \Delta \vdash \mathsf{e} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$, then for all $\mathsf{x} \in \mathcal{V}(\mathsf{e})$, $\mathbf{k} \preceq \Gamma(\mathsf{x})$.

Proof. By structural induction on expressions. □

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

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

Lemma 5.2 (Confinement)**.** Given a safe program p_ϕ with respect to the typing environments Γ, Δ, for any $\mathsf{c} \in C(\mathsf{p}_\phi)$, if $\Gamma, \Delta \vdash \mathsf{c} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$, then for all $\mathsf{x} \in \mathcal{A}(\mathsf{c})$, $\Gamma(\mathsf{x}) \preceq \mathbf{k}$.

Proof. By contradiction. □

For a given variable typing environment Γ and a given tier $\mathbf{k}$, we define an equivalence relation on stores by: $\mu \approx^{\Gamma}_{\mathbf{k}} \mu'$ if $dom(\mu) = dom(\mu') = dom(\Gamma)$ and for each $\mathsf{x} \in dom(\Gamma)$, if $\mathbf{k} \preceq \Gamma(\mathsf{x})$ then $\mu(\mathsf{x}) = \mu'(\mathsf{x})$.

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

Theorem 5.3 (Non-interference)**.** Given a safe program c `return` x with respect to the typing environments Γ, Δ. For any stores μ_1 and μ_2 if $\mu_1 \approx^{\Gamma}_{\mathbf{k}} \mu_2$, $\mu_1 \vDash \mathsf{c} \to \mu_1'$ and $\mu_2 \vDash \mathsf{c} \to \mu_2'$ then $\mu_1' \approx^{\Gamma}_{\mathbf{k}} \mu_2'$.

Proof. By structural induction on program derivations, using Lemma 5.1 and Lemma 5.2. □

5.3 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.

Definition 5.4. Let $m_\mu^{\mathsf{p}_\phi}$ be the maximum of $|\mu|$ and the maximum size of an oracle answer in the derivation $\pi_\phi : \mu \vDash \mathsf{p}_\phi \to u$. Formally,

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

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

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

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

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

Theorem 5.5. Given a safe program p_ϕ with respect to the typing environments Γ, Δ, there is a polynomial P such that for any derivation $\pi_\phi : \mu \vDash \mathsf{p}_\phi \to w$, $|\pi_\phi| = O(P(m_\mu^{\mathsf{p}_\phi}))$.

Proof. By induction on the tier of a command using non-interference Theorem (5.3) and the stratification properties of the type system. □

The proof of the above Theorem is similar to proofs of polynomiality in [Marion 2011] and [Marion and Péchoux 2014]. There are only two main distinctions:

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

Corollary 5.6. Given a program p_ϕ, if $\mathsf{p}_\phi \in \mathrm{ST}$ then p_ϕ has a polynomial step count.

5.4 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.

Definition 5.7. Given a program p_ϕ, let $(l_n^{\pi_\phi})$ be the sequence of oracle input values $[\![\uparrow]\!](v, w)$ in a rule (OR) obtained by a left-to-right depth-first search of the derivation $\pi_\phi : \mu \vDash \mathsf{p}_\phi \to w$. Let $lr((l_n^{\pi_\phi})) = \#\{i \mid |l_i^{\pi_\phi}| > \max_{j<i}(|l_j^{\pi_\phi}|)\}$.

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

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

Theorem 5.8 (Finite lookahead revision)**.** Given a program p_ϕ, if p_ϕ is safe with respect to the typing environments Γ, Δ then it has finite lookahead revision.

Proof. Using simple security (5.1) and confinement (5.2) Lemmata, and the stratification properties of the type system. □

6 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 6.4. 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, $\mathsf{e}^{\mathbf{k}}$ means that e is of tier $\mathbf{k}$. 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 v denotes 1^{v}, and 0 denotes ϵ. Also, with this convention, $[\![\mathtt{pred}]\!](\mathsf{v}) = \max\{0, \mathsf{v}-1\}$ and $[\![\mathtt{suc}_1]\!](\mathsf{v}) = \mathsf{v}+1$.

Example 6.1 (Addition)**.** Consider the simple program of Figure 4a, with no oracle, computing the unary addition. This program is safe with respect to the typing derivation of Figure 5.

The while loop is guarded by $\mathsf{x} > 0$. If the main command is typed by $(\mathbf{1}, \mathbf{1}, \mathbf{0})$ then the expression $\mathsf{x} > 0$ is of tier $\mathbf{1}$ by the typing rule (W_0). Consequently, the variable 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.

In the subderivation ρ_1 of Figure 5, 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 x (equal to $\mathbf{1}$) is smaller than the tier of the expression $\mathtt{pred}(\mathsf{x})$ (also equal to $\mathbf{1}$).

Now consider the subderivation ρ_2 of Figure 5. 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 4.2, $\Delta(\mathtt{suc}_1)(\mathbf{1}) = \{\mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$. The type $\mathbf{1} \to \mathbf{0}$ is ruled out as it would require a non-homogeneous type for y. Consequently, the rule (OP) is applied on type $\mathbf{0} \to \mathbf{0}$ and the variable y must be of tier $\mathbf{0}$. Notice that the program could also be typed by assigning higher tiers $\mathbf{k}$ and $\mathbf{k}'$ such that $\mathbf{k}' \prec \mathbf{k}$, to x and y, respectively.

Example 6.2 (Exponential)**.** The program of Figure 4b, computing the exponential, is not safe.

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

The subcommand $\mathsf{z} := \mathsf{y}$ enforces $\mathbf{k}_\mathsf{z} \preceq \mathbf{k}_\mathsf{y}$ by the typing derivation of Figure 6a.

The subcommand $\mathsf{y} := \mathtt{suc}_1(\mathsf{y})$ enforces the constraint $\mathbf{k}_\mathsf{y} \prec \mathbf{k}_{in}$, $\mathbf{k}_{in}$ being the command innermost tier, by the typing derivation of Figure 6b. Indeed, as $\mathtt{suc}_1$ is a positive operator, by Condition 2 of Definition 4.1, $\mathbf{k}_\mathsf{y} \prec \mathbf{k}_{in}$ has to be satisfied for $\mathbf{k}_\mathsf{y} \to \mathbf{k}_\mathsf{y} \in \Delta(\mathtt{suc}_1)(\mathbf{k}_{in})$ to hold.

The innermost while loop enforces the constraint $\mathbf{k}_{in} \preceq \mathbf{k}_\mathsf{z}$ in the typing derivation of Figure 6c. First, notice that

```
while(x > 0)^1{
   x^1 := pred(x)^1;
   y^0 := suc_1(y)^0
}
return  y
```

(a) Unary addition

```
while(x > 0){
   z := y;
   while(z > 0){
      z := pred(z);
      y := suc_1(y)
   };
   x := pred(x)
}
return  y
```

(b) Exponential

```
c_1 : while(x > 0)^2{
        x^2 := pred(x)^2;
        y^1 := suc_1(suc_1(y))^1
      } ;
c_2 : while(y > 0)^1{
        y^1 := pred(y)^1;
        z^0 := suc_1(suc_1(z))^0
      }
      return  z
```

(c) Multiple tiers

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

(d) Oracle

```
x := 0; z := 0;
while(y >= x){
   if(φ(y ↾ x) == 0){z := 1}
   else {skip};
   x := suc_1(x)
}
return  z
```

(e) No finite lookahead revision

```
x^3 := n; y^2 := x^3; z^2 := 0;
while(x^3 >= 0){
   z^2 := max(φ(y^2 ↾ x^3)^2, z^2);
   x^3 := pred(x^3)
}; v^1 := z^2; u^0 := 0;
while(z^2 >= 0){
   w^1 := φ(v^1 ↾ z^2)^1;
   while(w^1 >= 0){
      u^0 := suc_1(u)^0; w^1 := pred(w^1)
   }; z^2 := pred(z)^2
}
return  u
```

(f) Multiple tiers and oracle

Figure 4. Examples

only the rule (W) can be applied to this subderivation as the corresponding subcommand is already contained inside a while loop and, consequently, $\mathbf{1} \preceq \mathbf{k}_{out}$ 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 $\mathbf{k}_{in}$ of subcommand $y := suc_1(y)$ (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 4.1 enforces that $\mathbf{k}_{in} \preceq \mathbf{k}_z \preceq \mathbf{k}'_{in}$ holds for the program to be typed.

Putting all the above constraints together, we obtain the contradiction $\mathbf{k}_z \preceq \mathbf{k}_y \prec \mathbf{k}_{in} \preceq \mathbf{k}_z$. Consequently, the program cannot be typed.

Example 6.3 (Multiple tiers)**.** Consider the program of Figure 4c illustrating the use of multiple tiers.

The program is safe with respect to the variable typing environment Γ such that $\Gamma(x) = \mathbf{2}$, $\Gamma(y) = \mathbf{1}$ and $\Gamma(z) = \mathbf{0}$. The main command can be typed by $(\mathbf{2}, \mathbf{2}, \mathbf{0})$ in the derivation of Figure 7a, provided that c_1 and c_2 are the commands corresponding to the first while loop and second while loop, respectively.

The derivation ρ_1 corresponding to the first while loop is described in Figure 7b. The subderivation ρ_1^1 can be built easily using rules (A), (OP), and (V) as pred is neutral and can be given the type $\mathbf{2} \to \mathbf{2}$ in $\Delta(\mathsf{pred})(\mathbf{2})$ (see Example 4.2). The subderivation ρ_2^1 can be built using the same rules as suc_1 is positive and can be given the type $\mathbf{1} \to \mathbf{1}$ in $\Delta(suc_1)(\mathbf{2})$ (see Example 4.2 again). ρ_2^1 requires the prior application of subtyping rule (SUB) as the tier of the assignment is equal to $\Gamma(y) = \mathbf{1}$.

The derivation ρ_2, described in Figure 7c, can be obtained in a similar way by taking the type $\mathbf{1} \to \mathbf{1}$ for the neutral operator pred in $\Delta(\mathsf{pred})(\mathbf{1})$ and the type $\mathbf{0} \to \mathbf{0}$ for the positive operator suc_1 in $\Delta(suc_1)(\mathbf{1})$. The initial subtyping rule is required as it is not possible to derive $\vdash y > 0 : (\mathbf{2}, \mathbf{2}, \mathbf{2})$ with the requirement that $\Gamma(y) = \mathbf{1}$.

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 $\Gamma(y) \prec \Gamma(x)$ and the second while loop enforces that $\Gamma(z) \prec \Gamma(y)$. More generally, the program can be typed by $(\mathbf{k{+}2}, \mathbf{k{+}2}, \mathbf{0})$ or $(\mathbf{k{+}2}, \mathbf{k{+}2}, \mathbf{k{+}2})$, for any tier $\mathbf{k}$.

Example 6.4 (Oracle)**.** For a given input x and a given oracle ϕ, the program of Figure 4d computes whether there exists a unary integer n of size smaller than $|x|$ such that $\phi(n) = 0$.

This program is safe and can be typed by $(\mathbf{1}, \mathbf{1}, \mathbf{0})$ under the variable typing environment Γ such that $\Gamma(x) = \mathbf{1}$ and $\Gamma(y) = \Gamma(z) = \mathbf{0}$. 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 $z := 1$, skip, and $x := \mathsf{pred}(x)$ 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 in Figure 8.

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 x and y in the oracle input (i.e. $\phi(\mathsf{x} \restriction \mathsf{y})$) is not typable as the tier of 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 $\mathtt{while(x > 0)\{x := \phi(x \restriction x)\}}$ are rejected by the type system.

Note that the program of Figure 4d is typable as the oracle calls are performed in a decreasing order and, hence, does not break the finite lookahead revision property presented in §5.4.

Now consider the equivalent program of Figure 4e where oracle calls are performed in increasing order. 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 $\mathbf{k}$ of the commands under the while will be equal to the tier of the guard y >= x, independently of whether rule (W) or rule (W_0) is used to type the while command.

Moreover, x has a tier $\mathbf{k}_\mathsf{x}$ such that $\mathbf{k} \preceq \mathbf{k}_\mathsf{x}$, using rule (OP) on the guard and, by definition of safe typing environments.

Now suc_1 is a positive operator and, consequently, by rule (OP) and, by definition of safe typing environments again, $\mathsf{suc}_1(\mathsf{x})$ has a tier $\mathbf{k}_{\mathsf{suc}_1(\mathsf{x})}$ strictly smaller than the innermost tier, i.e. $\mathbf{k}_{\mathsf{suc}_1(\mathsf{x})} \prec \mathbf{k}$. By typing rule (A), in order to be typed, the command $\mathsf{x} := \mathsf{suc}_1(\mathsf{x})$ enforces $\mathbf{k}_\mathsf{x} \preceq \mathbf{k}_{\mathsf{suc}_1(\mathsf{x})}$. Hence, we obtain a contradiction: $\mathbf{k} \prec \mathbf{k}$.

Example 6.5 (Multiple tiers with oracle)**.** The program of Figure 4f computes the function $\Sigma_{i=0}^{\max_{x=0}^{n}\phi(x)}\phi(i)$.

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

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 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.

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 [Kapron and Cook 1991, 1996; Mehlhorn 1976]. For that purpose, we use the characterization of [Kapron and Steinberg 2018] based on moderately polynomial time (MPT) 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 7.1. Given an oracle TM M_ϕ 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 $P(m_\mathbf{a}^{M_\phi})$.
- a 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 7.2 (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

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathsf{x}) = \mathbf{1}}{\vdash \mathsf{x} : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{V})}{\vdash \mathsf{x} > 0 : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{OP}) \qquad \dfrac{\dfrac{\dfrac{\Gamma(\mathsf{x}) = \mathbf{1}}{\vdash \mathsf{x} : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{V}) \quad \dfrac{\Gamma(\mathsf{x}) = \mathbf{1}}{\vdash \mathsf{x} : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{V})}{\vdash \mathsf{pred}(\mathsf{x}) : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{OP})}{\rho_1 \rhd \vdash \mathsf{x} := \mathsf{pred}(\mathsf{x}) : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{A}) \qquad \dfrac{\dfrac{\dfrac{\Gamma(\mathsf{y}) = \mathbf{0}}{\vdash \mathsf{y} : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{V})}{\vdash \mathsf{suc}_1(\mathsf{y}) : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{OP})}{\rho_2 \rhd \vdash \mathsf{y} := \mathsf{suc}_1(\mathsf{y}) : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{A})}{\vdash \mathsf{y} := \mathsf{suc}_1(\mathsf{y})\ (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{SUB})}{\vdash \mathsf{x} := \mathsf{pred}(\mathsf{x}); \mathsf{y} := \mathsf{suc}_1(\mathsf{y}) : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{S})}{\vdash \mathsf{while}(\mathsf{x} > 0)\{\mathsf{x} := \mathsf{pred}(\mathsf{x}); \mathsf{y} := \mathsf{suc}_1(\mathsf{y})\} : (\mathbf{1},\mathbf{1},\mathbf{0})}\,(\mathrm{W}_0)$$

Figure 5. Typing derivation of Example 6.1 (Figure 4a)

$$\dfrac{\dfrac{\Gamma(\mathsf{z}) = \mathbf{k}_\mathsf{z}}{\vdash \mathsf{z} : (\mathbf{k}_\mathsf{z}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{V}) \quad \dfrac{\Gamma(\mathsf{y}) = \mathbf{k}_\mathsf{y}}{\vdash \mathsf{y} : (\mathbf{k}_\mathsf{y}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{V})}{\rho_1 \rhd \vdash \mathsf{z} := \mathsf{y} : (\mathbf{k}_\mathsf{z}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{A}) \quad \text{provided that } \mathbf{k}_\mathsf{z} \preceq \mathbf{k}_\mathsf{y}$$

(a) Derivation ρ_1

$$\dfrac{\dfrac{\Gamma(\mathsf{y}) = \mathbf{k}_\mathsf{y}}{\vdash \mathsf{y} : (\mathbf{k}_\mathsf{y}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{V}) \quad \dfrac{\mathbf{k}_\mathsf{y} \to \mathbf{k}_\mathsf{y} \in \Delta(\mathsf{suc}_1)(\mathbf{k}_{in}) \qquad \dfrac{\Gamma(\mathsf{y}) = \mathbf{k}_\mathsf{y}}{\vdash \mathsf{y} : (\mathbf{k}_\mathsf{y}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{V})}{\vdash \mathsf{suc}_1(\mathsf{y}) : (\mathbf{k}_\mathsf{y}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{OP})}{\rho_2 \rhd \vdash \mathsf{y} := \mathsf{suc}_1(\mathsf{y}) : (\mathbf{k}_\mathsf{y}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{A})$$

(b) Derivation ρ_2

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathsf{z}) = \mathbf{k}_\mathsf{z}}{\vdash \mathsf{z} : (\mathbf{k}_\mathsf{z}, \mathbf{k}'_{in}, \mathbf{k}_{out})}\,(\mathrm{V})}{\vdash \mathsf{z} > 0 : (\mathbf{k}_{in}, \mathbf{k}'_{in}, \mathbf{k}_{out})}\,(\mathrm{OP}) \quad \dfrac{\vdots}{\vdash \mathsf{z} := \mathsf{pred}(\mathsf{z}); \mathsf{y} := \mathsf{suc}_1(\mathsf{y}) : (\mathbf{k}_{in}, \mathbf{k}_{in}, \mathbf{k}_{out})}\,(\mathrm{S})}{\rho_3 \rhd \vdash \mathsf{while}(\mathsf{z} > 0)\{\mathsf{z} := \mathsf{pred}(\mathsf{z}); \mathsf{y} := \mathsf{suc}_1(\mathsf{y})\} : (\mathbf{k}_{in}, \mathbf{k}'_{in}, \mathbf{k}_{out})}\,(\mathrm{W})$$

(c) Derivation ρ_3

Figure 6. Typing derivation of Example 6.2 (Figure 4b)

$lev(\tau \to \tau') = \max(lev(\tau) + 1, lev(\tau'))$. For a given class of functionals X, let X_2 be the set of functionals of level 2.

Lemma 7.3 (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 7.4 ([Kapron and Steinberg 2018])**.**

$$\lambda(\mathrm{MPT})_2 = \mathtt{BFF}_2.$$

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 p_ϕ and the running time of a corresponding sequential execution of p_ϕ. 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 7.5. Suppose that p_ϕ is a safe program. There are an oracle TM M_ϕ and a polynomial P such that for any derivation $\pi_\phi : \mu \vDash \mathsf{p}_\phi \to w$, M_ϕ on input a_μ simulates the execution of p_ϕ with initial store μ in time $O(P(|\pi_\phi|))$.

Proof. By induction on the structure of the derivation. □

Theorem 7.6 (Soundness)**.** $\lambda([\![\mathrm{ST}]\!])_2 \subseteq \mathtt{BFF}_2$.

Proof. First, we can show that $[\![\mathrm{ST}]\!] \subseteq$ MPT using Theorem 5.5 and Theorem 5.8, and observing that MPT is stable through oracle padding. By Lemma 7.3 and Theorem 7.4, $\lambda([\![\mathrm{ST}]\!])_2 \subseteq \lambda(\mathrm{MPT})_2 = \mathtt{BFF}_2$. □

Notice that the lambda-closure of $[\![\mathrm{ST}]\!]$ is mandatory for characterizing $\mathtt{BFF}_2$ as it is well-known that MPT is strictly

$$\dfrac{\dfrac{\rho_1}{\vdots}}{\dfrac{}{\rho_1 \rhd \vdash \mathtt{c}_1 : (\mathbf{2},\mathbf{2},\mathbf{0})}\,(\mathrm{W}_0)} \quad \dfrac{\dfrac{\rho_2}{\vdots}}{\dfrac{}{\rho_2 \rhd \vdash \mathtt{c}_2 : (\mathbf{2},\mathbf{2},\mathbf{0})}\,(\mathrm{W}_0)}}{\vdash \mathtt{c}_1;\mathtt{c}_2 : (\mathbf{2},\mathbf{2},\mathbf{0})}\,(\mathrm{S})$$

(a) Main derivation

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{x}) = \mathbf{2}}{\vdash \mathtt{x} : (\mathbf{2},\mathbf{2},\mathbf{2})}\,(\mathrm{V})}{\vdash \mathtt{x} > 0 : (\mathbf{2},\mathbf{2},\mathbf{2})}\,(\mathrm{OP}) \quad \dfrac{\dfrac{\rho_1^1}{\vdots}}{\vdash \mathtt{x} := \mathtt{pred}(\mathtt{x}) : (\mathbf{2},\mathbf{2},\mathbf{2})}\,(\mathrm{A}) \quad \dfrac{\dfrac{\rho_2^1}{\vdots}}{\vdash \mathtt{y} := \mathtt{suc}_1(\mathtt{suc}_1(\mathtt{y})) : (\mathbf{1},\mathbf{2},\mathbf{2})}\,(\mathrm{A})}{\vdash \mathtt{y} := \mathtt{suc}_1(\mathtt{suc}_1(\mathtt{y})) : (\mathbf{2},\mathbf{2},\mathbf{2})}\,(\mathrm{SUB})}{\vdash \mathtt{x} := \mathtt{pred}(\mathtt{x}); \mathtt{y} := \mathtt{suc}_1(\mathtt{suc}_1(\mathtt{y})) : (\mathbf{2},\mathbf{2},\mathbf{2})}\,(\mathrm{S})}{\vdash \mathtt{c}_1 : (\mathbf{2},\mathbf{2},\mathbf{0})}\,(\mathrm{W}_0)$$

(b) Derivation ρ_1

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{y}) = \mathbf{1}}{\vdash \mathtt{y} : (\mathbf{1},\mathbf{2},\mathbf{1})}\,(\mathrm{V})}{\vdash \mathtt{y} > 0 : (\mathbf{1},\mathbf{2},\mathbf{1})}\,(\mathrm{OP}) \quad \dfrac{\dfrac{\rho_1^2}{\vdots}}{\vdash \mathtt{y} := \mathtt{pred}(\mathtt{y}) : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{A}) \quad \dfrac{\dfrac{\rho_2^2}{\vdots}}{\vdash \mathtt{z} := \mathtt{suc}_1(\mathtt{suc}_1(\mathtt{z})) : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{A})}{\vdash \mathtt{z} := \mathtt{suc}_1(\mathtt{suc}_1(\mathtt{z})) : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{SUB})}{\vdash \mathtt{y} := \mathtt{pred}(\mathtt{y}); \mathtt{z} := \mathtt{suc}_1(\mathtt{suc}_1(\mathtt{z})) : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{S})}{\dfrac{\vdash \mathtt{c}_2 : (\mathbf{1},\mathbf{2},\mathbf{0})}{\vdash \mathtt{c}_2 : (\mathbf{2},\mathbf{2},\mathbf{0})}\,(\mathrm{SUB})}\,(\mathrm{W}_0)$$

(c) Derivation ρ_2

Figure 7. Typing derivation of Example 6.3 (Figure 4c)

$$\dfrac{\dfrac{\mathbf{0} \to \mathbf{1} \to \mathbf{0} \in \Delta(==)(\mathbf{1})}{\vdash \phi(\mathtt{y} \restriction \mathtt{x}) : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{OR}) \quad \dfrac{}{\vdash 0 : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{OP})}{\vdash \phi(\mathtt{y} \restriction \mathtt{x}) == 0 : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{OP}) \quad \dfrac{\Gamma(\mathtt{y}) = \mathbf{0}}{\vdash \mathtt{y} : (\mathbf{0},\mathbf{1},\mathbf{1})}\,(\mathrm{V}) \quad \dfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : (\mathbf{1},\mathbf{1},\mathbf{1})}\,(\mathrm{V})$$

$$\vdots$$

$$\vdash \mathtt{if}(\phi(\mathtt{y} \restriction \mathtt{x}) == 0)\{\mathtt{z} := 1\}\ \mathtt{else}\ \{\mathtt{skip}\} : (\mathbf{0},\mathbf{1},\mathbf{1})\,(\mathrm{C})$$

Figure 8. Typing derivation of example 6.4 (Figure 4d)

included in BFF_2. In particular, the following example taken from [Kapron and Steinberg 2018] and computing a function of BFF_2 cannot be typed as all our oracle calls have input bounds.

Example 7.7. The functional F defined below is in BFF_2 but not in MPT.

$$F(\phi, \epsilon) = \epsilon$$
$$F(\phi, \mathsf{suc}_1(n)) = \phi \circ \phi(F(\phi, n) \restriction \phi(\epsilon))$$

Consequently, F is not in $[\![\mathrm{ST}]\!]$, since $[\![\mathrm{ST}]\!] \subseteq$ MPT. Indeed, the outermost oracle call is performed without any oracle input bound and clearly cannot 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 [Marion and Péchoux 2014], characterizing FP, is a strict subsystem of the type system of Figure 3. Second, we show that the bounded iterator functional $\mathcal{I}'$ of [Kapron and Steinberg 2019] 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 [Marion and Péchoux 2014], restricted to one single thread. Let α, β be tier variables ranging over $\{\mathbf{0}, \mathbf{1}\}$. The type system is provided in Figure 9, where $\overline{\alpha}$ stands for $\alpha_1 \to \ldots \to \alpha_{ar(\mathsf{op})}$, with $\alpha_i \in \{\mathbf{0}, \mathbf{1}\}$.

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 op a set of operator types $\Delta(\mathsf{op})$, of the shape $\overline{\alpha} \to \alpha$, with $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Definition 8.1. A program is 2-tier safe if it can be typed using 2-tier operator typing environment Δ satisfying, for any $\mathsf{op} \in dom(\Delta)$, $[\![op]\!] \in \mathsf{FP}$, op is either positive or neutral, and for each $\alpha_1 \to \ldots \to \alpha_{ar(\mathsf{op})} \to \alpha \in \Delta(\mathsf{op})$:

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

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

Theorem 8.2 (Theorem 7 of [Marion and Péchoux 2014])**.**

$$[\![2\mathrm{ST}]\!] = \mathsf{FP}.$$

Now we show the following inclusion.

Lemma 8.3. $2\mathrm{ST} \subsetneq \mathrm{ST}$.

Proof. By an easy induction on the type derivation. □

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

Theorem 8.4. $\mathsf{FP} = [\![\mathrm{ST}]\!]_1$.

Proof. By Theorem 8.2, for any function f in FP is computable by a 2-tier safe and terminating program p_ϕ with no oracle calls, i.e. $f = \lambda w.[\![\mathsf{p}_\phi]\!](w)$. By Lemma 8.3, p_ϕ is in ST and, consequently, $f \in [\![\mathrm{ST}]\!]_1$. Conversely, by Corollary 5.6, if $\mathsf{p}_\phi \in \mathrm{ST}$ then p_ϕ has a polynomial step count. More precisely, if $\phi \notin \mathsf{p}_\phi$ and $\mathsf{p}_\phi \in \mathrm{ST}$ has a polynomial time step count and runs in time bounded by $P(|w|)$, for some polynomial P, as there is no oracle call. Consequently, $[\![\mathrm{ST}]\!]_1 \subseteq \mathsf{FP}$. □

Note that the completeness part of the above Theorem ($\mathsf{FP} \subseteq [\![\mathrm{ST}]\!]_1$) 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 [Marion 2011].

8.2 Type two iteration

[Kapron and Steinberg 2019] 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

$$lmin(a, b) = \begin{cases} a, \text{ if } |a| < |b|, \\ b, \text{ otherwise.} \end{cases}$$

They use Cook's notion [Cook 1992] of polynomial time reducibility to show that this functional is polynomial time-equivalent to the recursor $\mathcal{R}$ of [Cook and Urquhart 1993]. As a consequence of Cook-Urquhart Theorem, the following characterization is obtained.

Theorem 8.5 ([Kapron and Steinberg 2019])**.**

$$\lambda(\mathsf{FP} \cup \{\mathcal{I}'\})_2 = \mathsf{BFF}_2.$$

Theorem 8.6 (Type-2 completeness)**.** $\mathsf{BFF}_2 \subseteq \lambda([\![\mathrm{ST}]\!])_2$.

Proof. By Theorem 8.4, $\mathsf{FP} = [\![\mathrm{ST}]\!]_1 \subseteq \lambda([\![\mathrm{ST}]\!])_1$ as any function $f \in [\![\mathrm{ST}]\!]_1$ is of the shape $\lambda w.[\![\mathsf{p}_\phi]\!](w)$, for $\mathsf{p}_\phi \in \mathrm{ST}$, and, consequently $f = (\lambda\phi.f)\ \phi \in \lambda([\![\mathrm{ST}]\!])_1$.

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 11, 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 10, under the typing environment Γ such that $\Gamma(c) = \Gamma(a) = \mathbf{1}$, $\Gamma(\mathsf{x}) = \Gamma(b) = \mathbf{0}$ 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 $\mathbf{1} \to \mathbf{1} \in \Delta(\mathsf{pred})(\mathbf{1})$. Note that the simulation uses the padded oracle variant $\tilde{\phi}$ of ϕ.

As $\mathsf{FP} \subseteq \lambda([\![\mathrm{ST}]\!])_1$ and $\mathcal{I}' \in [\![\mathrm{ST}]\!]$. We have that $\lambda(\mathsf{FP} \cup \{\mathcal{I}'\})_2 \subseteq \lambda([\![\mathrm{ST}]\!])_2$ and the result follows by Theorem 8.5. □

To illustrate the need of the type-2 lambda closure for achieving completeness, consider a variant of Example 7.7:

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

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 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 $\lambda([\![\mathrm{ST}]\!])_2$, as $\mathcal{I}' = [\![it_\phi]\!] \in [\![\mathrm{ST}]\!]$, it_ϕ being the program in the proof of Theorem 8.6, and computes the functional $\lambda\phi.\lambda n.F'(\phi, n)$.

9 Other properties

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 a worst expressive power). Second, type inference is decidable in polynomial time in the size of the program.

Let $[\![\mathrm{ST}^{\mathbf{k}}]\!]$ be the subset of functional of $[\![\mathrm{ST}]\!]$ computable by terminating and typable programs using tiers bounded by $\mathbf{k}$. Formally, $\mathsf{p}_\phi \in \mathrm{ST}^{\mathbf{k}}$ if and only if p_ϕ is terminating and $\Gamma, \Delta \vdash \mathsf{p}_\phi : (\mathbf{k}', \mathbf{k}'_{in}, \mathbf{k}'_{out})$ for a safe operator typing environment Δ and a variable typing environment Γ such that $\forall \mathsf{x} \in \mathcal{V}(\mathsf{p}_\phi)$, $\Gamma(\mathsf{x}) \preceq \mathbf{k}$.

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.

$$\frac{\Gamma(\mathsf{x}) = \alpha}{\Gamma, \Delta \vdash_2 \mathsf{x} : \alpha}\ (\mathrm{V}_2) \qquad \frac{\forall i \le n,\ \Gamma, \Delta \vdash_2 \mathsf{e}_i : \alpha_i \quad \overline{\alpha} \to \alpha \in \Delta(\mathsf{op})}{\Gamma, \Delta \vdash_2 \mathsf{op}(\mathsf{e}_1, \ldots, \mathsf{e}_{ar(\mathsf{op})}) : \alpha}\ (\mathrm{OP}_2) \qquad \frac{\Gamma, \Delta \vdash_2 \mathsf{x} : \alpha \quad \Gamma, \Delta \vdash_2 \mathsf{e} : \beta \quad \alpha \preceq \beta}{\Gamma, \Delta \vdash_2 \mathsf{x} := \mathsf{e} : \alpha}\ (\mathrm{A}_2) \qquad \frac{}{\Gamma, \Delta \vdash_2 \mathtt{skip} : \alpha}\ (\mathrm{SK}_2)$$

$$\frac{\Gamma, \Delta \vdash_2 \mathsf{c} : \alpha \quad \Gamma, \Delta \vdash_2 \mathsf{c}' : \beta}{\Gamma, \Delta \vdash_2 \mathsf{c}; \mathsf{c}' : \alpha \vee \beta}\ (\mathrm{S}_2) \qquad \frac{\Gamma, \Delta \vdash_2 \mathsf{e} : \alpha \quad \Gamma, \Delta \vdash_2 \mathsf{c} : \alpha \quad \Gamma, \Delta \vdash_2 \mathsf{c}' : \alpha}{\Gamma, \Delta \vdash_2 \mathtt{if(e)\{c\}\ else\ \{c'\}} : \alpha}\ (\mathrm{C}_2) \qquad \frac{\Gamma, \Delta \vdash_2 \mathsf{e} : \mathbf{1} \quad \Gamma, \Delta \vdash_2 \mathsf{c} : \alpha}{\Gamma, \Delta \vdash_2 \mathtt{while(e)\{c\}} : \mathbf{1}}\ (\mathrm{W}_2)$$

Figure 9. 2-tier-based type system for 2ST

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(\mathsf{x}) = \mathbf{0}}{\vdash \mathsf{x} : (\mathbf{0},\mathbf{1},\mathbf{1})}\ (\mathrm{V}) \quad \dfrac{\dfrac{\Gamma(a) = \mathbf{1}}{\vdash a : (\mathbf{1},\mathbf{1},\mathbf{1})}\ (\mathrm{V})}{\vdash lmin(\mathsf{x}, a) : (\mathbf{0},\mathbf{1},\mathbf{1})}\ (\mathrm{OP}) \quad \dfrac{\Gamma(a) = \mathbf{1}}{\vdash a : (\mathbf{1},\mathbf{1},\mathbf{1})}\ (\mathrm{V})}{\vdash \tilde{\phi}(lmin(\mathsf{x}, a) \upharpoonright a) : (\mathbf{0},\mathbf{1},\mathbf{1})}\ (\mathrm{OR})}{\vdash \mathsf{x} := \tilde{\phi}(lmin(\mathsf{x}, a) \upharpoonright a) : (\mathbf{0},\mathbf{1},\mathbf{1})}\ (\mathrm{A})}{\vdash \mathsf{x} := \tilde{\phi}(lmin(\mathsf{x}, a) \upharpoonright a) : (\mathbf{1},\mathbf{1},\mathbf{1})}\ (\mathrm{SUB})$$

$$\dfrac{\vdots \qquad \dfrac{\vdash \mathsf{x} := \tilde{\phi}(lmin(\mathsf{x}, a) \upharpoonright a) : (\mathbf{1},\mathbf{1},\mathbf{1}) \qquad \vdots}{\vdash \mathsf{x} := \tilde{\phi}(lmin(\mathsf{x}, a) \upharpoonright a); c := \mathsf{pred}(c) : (\mathbf{1},\mathbf{1},\mathbf{1})}\ (\mathrm{S})}{\vdash \mathtt{while}(c > 0)\{\mathsf{x} := \tilde{\phi}(lmin(\mathsf{x}, a) \upharpoonright a); c := \mathsf{pred}(c)\} : (\mathbf{1},\mathbf{1},\mathbf{0})}\ (\mathrm{W}_0)$$

Figure 10. Program simulation $\mathcal{I}'$

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

Figure 11. Program it_ϕ

Proposition 9.1. The following properties hold:

1. $\forall \mathbf{k} \geq \mathbf{0},\ \mathrm{ST}^{\mathbf{k}} \subsetneq \mathrm{ST}^{\mathbf{k+1}}$,
2. $\forall \mathbf{k} \geq \mathbf{1},\ \lambda(\llbracket \mathrm{ST}^{\mathbf{k}} \rrbracket)_2 = \mathsf{BFF}_2$.

Theorem 9.2. Given a program p_ϕ and a safe operator typing environment Δ, deciding if there exists a variable typing environment Γ such that $\mathsf{p}_\phi \in \mathrm{ST}$ can be done in time cubic in the size of the program.

Proof. Using a reduction to 2-SAT. □

10 Conclusion and future work

We have presented the 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 or 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 7.7. 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 in a purely functional language. We leave these open issues as future work.

Acknowledgments

The authors would like to thank the anonymous referees for their feedback. Kapron was supported in part by a NSERC Discovery Grant.

References

Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inform. Process. Lett.* 8, 3 (1979), 121–123. https://doi.org/10.1016/0020-0190(79)90002-4

Patrick Baillot and Ugo Dal Lago. 2016. Higher-order interpretations and program complexity. *Inf. Comput.* 248 (2016), 56–81. https://doi.org/10.1016/j.ic.2015.12.008

Patrick Baillot and Damiano Mazza. 2010. Linear logic by levels and bounded time complexity. *Theor. Comput. Sci.* 411, 2 (2010), 470–503. https://doi.org/10.1016/j.tcs.2009.09.015

Patrick Baillot and Kazushige Terui. 2004. Light Types for Polynomial Time Computation in Lambda-Calculus. In *Logic in Computer Science, LICS 2004*. IEEE, 266–275. https://doi.org/10.1109/LICS.2004.1319621

Stephen Bellantoni and Stephen Cook. 1992. A New Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity* 2 (1992), 97–110. https://doi.org/10.1007/BF01201998

Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. 2008. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Logic and Theory of Algorithms*. Springer, 67–76.

Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. 2011. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.* 412, 25 (2011), 2776–2796. https://doi.org/10.1016/j.tcs.2011.02.007

Alan Cobham. 1965. The intrinsic computational difficulty of functions. In *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, Y. Bar-Hillel (Ed.). North-Holland, Amsterdam, 24–30.

Robert L. Constable. 1973. Type two computational complexity. In *Proc. 5th annual ACM Symposium on Theory of Computing*. ACM, 108–121. https://doi.org/10.1145/800125.804041

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. TERMINATOR: beyond safety. In *International Conference on Computer Aided Verification*. Springer, 415–418. https://doi.org/10.1007/11817963_37

Stephen A Cook. 1992. Computability and complexity of higher type functions. In *Logic from Computer Science*. Springer, 51–72. https://doi.org/10.1007/978-1-4612-2822-6_3

Stephen A. Cook and Bruce M. Kapron. 1989. Characterizations of the basic feasible functionals of finite type. In *30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*. IEEE, 154–159. https://doi.org/10.1109/SFCS.1989.63471

Stephen A. Cook and Alasdair Urquhart. 1993. Functional interpretations of feasibly constructive arithmetic. *Ann. Pure Appl. Logic* 63, 2 (1993), 103–200. https://doi.org/10.1016/0168-0072(93)90044-E

Norman Danner and James S. Royer. 2006. Adventures in time and space. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*. ACM, 168–179. https://doi.org/10.1145/1111037.1111053

Shimon Even, Alon Itai, and Adi Shamir. 1976. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM J. Comput.* 5, 4 (1976), 691–703. https://doi.org/10.1137/0205048

Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. 2015. Characterizing polynomial time complexity of stream programs using interpretations. *Theor. Comput. Sci.* 585 (2015), 41–54. https://doi.org/10.1016/j.tcs.2015.03.008

Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. 2008. A logical account of pspace. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*. ACM, 121–131. https://doi.org/10.1145/1328438.1328456

Jean-Yves Girard. 1998. Light Linear Logic. *Inf. Comput.* 143, 2 (1998), 175–204. https://doi.org/10.1006/inco.1998.2700

Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. 2013. Type-based complexity analysis for fork processes. In *International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2013)*. Springer, 305–320. https://doi.org/10.1007/978-3-642-37075-5_20

Emmanuel Hainry and Romain Péchoux. 2015. Objects in Polynomial Time. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015 (Lecture Notes in Computer Science)*. Springer, 387–404. https://doi.org/10.1007/978-3-319-26529-2_21

Emmanuel Hainry and Romain Péchoux. 2017. Higher order interpretation for higher order complexity. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair, 269–285. https://doi.org/10.29007/1tkw

Petr Hájek. 1979. Arithmetical Hierarchy and Complexity of Computation. *Theor. Comput. Sci.* 8 (1979), 227–237.

Robert J. Irwin, James S. Royer, and Bruce M. Kapron. 2001. On characterizations of the basic feasible functionals (Part I). *J. Funct. Program.* 11, 1 (2001), 117–153. https://doi.org/10.1017/S0956796800003841

Neil D. Jones and Lars Kristiansen. 2009. A Flow Calculus of Mwp-bounds for Complexity Analysis. *ACM Trans. Comput. Logic* 10, 4 (2009), 28:1–28:41. https://doi.org/10.1145/1555746.1555752

Bruce M. Kapron and Stephen A. Cook. 1991. A New Characterization of Mehlhorn's Polynomial Time Functionals (Extended Abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE, 342–347. https://doi.org/10.1109/SFCS.1991.185389

Bruce M. Kapron and Stephen A. Cook. 1996. A New Characterization of Type-2 Feasibility. *SIAM J. Comput.* 25, 1 (1996), 117–132. https://doi.org/10.1137/S0097539794263452

Bruce M. Kapron and Florian Steinberg. 2018. Type-two polynomial-time and restricted lookahead. In *Logic in Computer Science, LICS 2018*. ACM, 579–588. https://doi.org/10.1145/3209108.3209124

Bruce M. Kapron and Florian Steinberg. 2019. Type-two iteration with bounded query revision. In *Proceedings Third Joint Workshop on Developments in Implicit Computational complExity and Foundational & Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2019 (EPTCS)*. 61–73. https://doi.org/10.4204/EPTCS.298.5

Akitoshi Kawamura and Florian Steinberg. 2017. Polynomial Running Times for Polynomial-Time Oracle Machines. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 23:1–23:18. https://doi.org/10.4230/LIPIcs.FSCD.2017.23

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 81–92. https://doi.org/10.1145/360204.360210

Daniel Leivant. 1995. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, Peter Clote and Jeffrey B. Remmel (Eds.). Birkhäuser, Boston, MA, 320–343. https://doi.org/10.1007/978-1-4612-2566-9_11

Daniel Leivant and Jean-Yves Marion. 2013. Evolving Graph-Structures and Their Implicit Computational Complexity. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Proceedings, Part II (Lecture Notes in Computer Science)*. Springer, 349–360. https://doi.org/10.1007/978-3-642-39212-2_32

Daniel Leivant and Jean-Yves Marion. 1993. Lambda Calculus Characterizations of Poly-Time. *Fundam. Inform.* 19, 1/2 (1993), 167–184.

Jean-Yves Marion. 2011. A Type System for Complexity Flow Analysis. In *Logic in Computer Science, LICS 2011*. IEEE Computer Society, 123–132. https://doi.org/10.1109/LICS.2011.41

Jean-Yves Marion and Romain Péchoux. 2014. Complexity Information Flow in a Multi-threaded Imperative Language. In *Theory and Applications of Models of Computation, TAMC 2014 (Lecture Notes in Computer Science)*. Springer, 124–140. https://doi.org/10.1007/978-3-319-06089-7_9

Kurt Mehlhorn. 1976. Polynomial and abstract subrecursive classes. *J. Comp. Sys. Sci.* 12, 2 (1976), 147–178. https://doi.org/10.1016/S0022-0000(76)80035-9

John C Mitchell. 1991. Type inference with simple subtypes. *J. Funct. Program.* 1, 3 (1991), 245–285.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187. https://doi.org/10.3233/JCS-1996-42-304