Complete and tractable machine-independent characterizations of second-order polytime

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

Complete and tractable machine-independent characterizations of second-order polytime

Emmanuel Hainry[1], Bruce M. Kapron[2], Jean-Yves Marion[1], and Romain Péchoux[1]✉

[1] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
[2] University of Victoria, Victoria, BC, Canada
{hainry,marion,pechoux}@inria.fr bmkapron@uvic.ca

Abstract. The class of Basic Feasible Functionals BFF is the second-order counterpart of the class of first-order functions computable in polynomial time. We present several implicit characterizations of BFF based on a typed programming language of terms. These terms may perform calls to imperative procedures, which are not recursive. The type discipline has two layers: the terms follow a standard simply-typed discipline and the procedures follow a standard tier-based type discipline. BFF consists exactly of the second-order functionals that are computed by typable and terminating programs. The completeness of this characterization surprisingly still holds in the absence of lambda-abstraction. Moreover, the termination requirement can be specified as a completeness-preserving instance, which can be decided in time quadratic in the size of the program. As typing is decidable in polynomial time, we obtain the first tractable (*i.e.*, decidable in polynomial time), sound, complete, and implicit characterization of BFF, thus solving a problem opened for more than 20 years.

Keywords: Basic feasible functionals · Type 2 · Second-order · Polynomial time · Tiering · Safe recursion

1 Introduction

Motivations. The class of second-order functions computable in polynomial time was introduced and studied by Mehlhorn [27], building on an earlier proposal by Constable [10]. Kapron and Cook characterized this class using oracle Turing machines, giving it the name Basic Feasible Functionals (BFF):

Definition 1 ([19]). *A functional F is in* BFF*, if there are an oracle Turing machine M and a second-order polynomial*[3] *P such that M computes F in time bounded by $P(|f|, |x|)$, for any oracle f and any input x.*[4]

Since then, BFF was consensually considered as the natural extension to second-order of the well-known class of (first-order) polynomial time computable functions, FP. Notions of second-order polynomial time, while of intrinsic interest,

[3] Second-order polynomials are a type-2 analogue of ordinary polynomials.
[4] The size of an oracle f is a first-order function defined by $|f|(n) = \max_{|y| \leq n} |f(y)|$.

have also been applied in a range of areas, including structural complexity theory [27], resource-bounded topology [29], complexity of total search problems [5], feasible real analysis [21], and verification [14].

Starting with Cobham's seminal work [9], there have been several attempts to provide *machine-independent* characterizations of complexity classes such as (P and) FP, that is, characterizations based on programming languages rather than on machines. Beyond the purely theoretical aspects, the practical interest of such characterizations is to be able to automatically guarantee that a program can be executed efficiently and in a secure environment. For these characterizations to hold, some restrictions are placed on a given programming language. They ensure that a program can be simulated by a Turing machine in polynomial time and, therefore, corresponds to a function in FP. This property is called *soundness*. Conversely, we would like any function in FP to be computable by a program satisfying the restrictions. This property is called (extensional) *completeness*. For automation to be possible, it is necessary that the characterizations studied be *tractable*; that is, decidable in polynomial time. Moreover, they should preferably not require a prior knowledge of the program complexity. One speaks then of *implicit* characterization insofar as the programmer does not have to know an explicit bound on the complexity of the analyzed programs.

In the first-order setting, different restrictions and techniques have been developed to characterize the complexity class FP. One can think, among others, of the safe recursion and ramified recursion techniques for function algebras [6,24], of interpretation methods for term rewrite systems [8], or of light and soft linear logics typing-discipline for lambda-calculi [15,4,3].

In the second-order setting, a machine-independent characterization of BFF was provided in [16]. This characterization uses the tier-based (*i.e.*, safe/ramified recursion-based) type discipline introduced in [26] on imperative programs for characterizing FP and can be restated as follows:

$$\mathtt{BFF} = \lambda(\llbracket \mathrm{ST} \rrbracket)_2,$$

$\llbracket \mathrm{ST} \rrbracket$ denotes the set of functions computed by typable and terminating programs; λ denotes the lambda closure, that is, for a given set of functionals X, $\lambda(X)$ is the set of functionals denoted by simply-typed lambda-terms using constants in X; X_2 is the restriction of X to second-order functionals. Type inference for $\llbracket \mathrm{ST} \rrbracket$ is fully automatic and can be performed in time cubic in the size of the analyzed program. However the above characterization has two main weaknesses:

- It is not complete: As $\llbracket \mathrm{ST} \rrbracket \subsetneq \mathtt{BFF}$, the typed language alone is not complete for BFF and a lambda closure (*i.e.*, $\lambda(X)$) of functionals computed by typable and terminating programs is required to ensure completeness.
- It is not tractable: the set $\llbracket \mathrm{ST} \rrbracket$ relies on a termination assumption and it is unclear whether the characterization sill holds for a decidable or, for that matter, tractable termination technique.

Thus, providing a tractable, implicit, sound, and complete programming language for characterizing second-order polynomial time is still an open problem.

Contributions. Our paper provides the first solution to this problem, open for more than 20 years. To this end, we introduce a higher-order programming language and design a suitable typing discipline that address the two weaknesses described above. The lambda closure requirement for completeness is removed by designing a suitable programming language that consists of a layer of simply-typed terms that can perform calls to a layer of imperative and non-recursive procedures following a tier-based type discipline. This language allows for some restricted forms of procedure composition that are handled by the simply-typed terms and also allows for some restricted forms of oracle composition that are managed through the use of *closures*, syntactic elements playing the role of first-order abstractions with free variables. The termination criterion is specified as a completeness-preserving instance, called SCP_S, of a variant of Size Change Termination [23] introduced in [7] that can be checked in time quadratic in the size of the analyzed program. The main contributions of this paper are:

- A programming language in which typable (SAFE) and terminating (SN) programs capture exactly BFF (Theorem 2).
- A restriction to lambda-free programs, called rank-0 programs, such that typable (SAFE_0) and terminating (SN) programs still capture exactly BFF (Theorem 3); hence showing that lambda-abstraction only provides a syntactic relaxation, and corresponds to a conservative extension in terms of computable functions.
- A proof that type inference for SAFE is P-complete, and a type inference procedure running in time cubic in the program size for SAFE_0 (Theorem 4).
- A simple termination criterion, called SCP_S, preserving soundness and completeness of the characterizations both for SAFE and for SAFE_0 (Theorem 5) that can be checked in quadratic time.
- A complete characterization of BFF in terms of typable (SAFE) and terminating (SCP_S) programs (Theorem 6) that captures strictly more programs (Example 1) than [16], and is decidable in P-time.

The contributions of the paper are a non-trivial extension of existing works:

- The critical Programming Language design decisions rely mostly on the notion of *continuation*, that fixes a given oracle (closure) for once in the imperative layer. If the oracle were allowed to be updated inside a while loop, depending on some local value, then the language would yield a class beyond BFF, by computing exponential functions.
- It is a surprising result that the characterization of BFF still holds in the absence of lambda-abstraction as a basic construct of the proposed programming language, in particular that completeness does not rely on lambda-abstractions. This is an important improvement over [16] and [20], both of which required external lambda-closure.
- The type system is designed so that each procedure is typed exactly once. Types are not unique, but this does not prevent type inference from being polytime, as exhibiting one type is sufficient. The tractability of type inference is obtained by combining the tractabilities of type inference in the tier-based layer and in the simply-typed layer [25].

– The particular choice of the termination criterion SCP_S was made to show that termination can be specified as a tractable/feasible criterion while preserving completeness. This is also a new result. SCP_S may include nested loops (as described in [7]) and can be replaced by any termination criterion capturing the programs of our completeness proof. SCP_S was chosen for its tractability, but not only: the SCP criterion of [7] ensures termination by using an error state which breaks the control flow. This control-flow break damages the non-interference property needed for tier-based typing to guarantee time complexity bounds.

Leading example. The program `ce` of Example 1 will be our leading example, as it computes a function known to be in $\mathtt{BFF} - [\![\mathrm{ST}]\!]$ (*i.e.*, it computes a function in `BFF` and not in $[\![\mathrm{ST}]\!]$, see [20]). This program will be shown to be in SAFE_0 and, consequently, in SAFE and to terminate with SCP_S.

Example 1 (Program `ce`*).* Let $\mathbb{W}$ be the set of words. Let the operator ε of arity 0 represent the empty word constant, let the operator != test whether or not its arguments are distinct, and let the operator `pred` remove the first letter of a word. The binary operator $\restriction$ truncates and pads the size of its first operand to the size of its second operand plus 1. When the boxed variables `X` and `y` are fed with the inputs $f \in \mathbb{W} \to \mathbb{W}$ and $w \in \mathbb{W}$, respectively, program `ce` calls procedure `KS` in the term `t`. Program `ce` computes $|w|$ (*i.e.*, the size of the word w) bounded iterations of $f \circ f$ through the iteration of the assignment $\mathtt{z} := \mathtt{X}_2(\mathtt{z} \restriction \mathtt{w})$ in procedure `KS`. The bound on the output size of each iteration is computed by the first assignment $\mathtt{w} := \mathtt{X}_1(\varepsilon \restriction \varepsilon)$ of `KS` and is equal to $f(1)$ (that is, $f([\![\varepsilon \restriction \varepsilon]\!])$, with $[\![\varepsilon \restriction \varepsilon]\!]$=1; $[\![\mathtt{e}]\!]$ being the result of evaluating the expression `e`).

```
box [X,y] in                                                          ce
    declare                                                           }
        KS(X1, X2, v) {                                               }
            var w,z;                                                  }
                                                                      }
            w  :=  X1(ε ↾ ε);          } Statement st                 }
                                                                      }
            z  :=  ε;                  }                              }
            while (v != ε) {           }                              }
                v  :=  pred(v);        } Statement st'   } Procedure p }
                z  :=  X2(z ↾ w)       }                              }
            }                          }                              }
            return z                   }                              }
        }                                                             }
                                                                      }
  in call KS({x → X @ x}, {x → X @ (X @ x)}, y)  } Term t             }
```

Related work. Several tools providing machine-independent characterizations of distinct complexity classes have been developed in the field of Implicit Com-

putational Complexity (ICC). Most of these tools are restricted to the study of first-order complexity classes. Whereas light logic approaches can deal with programs at higher types, their applications are restricted to first-order complexity classes such as FP [15,4,3]. Interpretation methods were extended to higher-order polynomials in [2] to study FP and adapted in [13] and [17] to characterize BFF. However, these characterizations are not decidable as they require checking of second-order polynomial inequalities. [12] and [18] study characterizations of BFF in terms of a simple imperative programming language that enforces an explicit external bound on the size of oracle outputs within loops. The corresponding restriction is not implicit by nature and is impractical from a programming perspective as the size of oracle outputs cannot be predicted. In this paper, the bound is programmer friendly because it is implicit and it only constrains the size of the oracle input.

2 A second-order language with imperative procedures

The syntax and semantics of the programming language designed to capture the complexity class BFF are introduced in this section. Programs of this language consist in second-order terms in which imperative procedures are declared and called. These procedures have no global variables, are not recursive, and their parameters can be of order 1 (oracles) or 0 (local variables). Oracles are in read-only mode: they cannot be declared and, hence, modified inside a procedure. Oracles can only be composed at the term level through the use of closures, first-order abstractions that can be passed as parameters in a procedure call.

Syntax. When we refer to a *type-i* syntactic element e (a variable, an expression, a statement, ...), for $i \in \mathbb{N}$, we implicitly assume that the element e denotes some function of order i over words as basic type. We will sometimes write e^i in order to make the order explicit. For example, e^0 denotes a word. This notion will be formally defined in Section 3. Let $\overline{e}$ denote a (possibly empty) tuple of n elements $e_1, \ldots, e_n$, where n is given by the context. Let $|\overline{e}|$ denote the length of tuple $\overline{e}$, *i.e.*, $|\overline{e}| \triangleq n$. Let π_i, $i \leq |\overline{e}|$, denote the projectors on tuples, *i.e.*, $\pi_i(\overline{e}) \triangleq e_i$.

Let $\mathbb{V}$ be a set of variables that can be split into three disjoint sets $\mathbb{V} = \mathbb{V}_0 \uplus \mathbb{V}_1 \uplus \mathbb{V}_{\geq 2}$. The type-0 variables in $\mathbb{V}_0$ will be denoted by lower case letters $\mathtt{x}, \mathtt{y}, \ldots$ and the type-1 variables in $\mathbb{V}_1$ will be denoted by upper case letters $\mathtt{X}, \mathtt{Y}, \ldots$ Variables in $\mathbb{V}$ of arbitrary type will be denoted by letters $\mathtt{a}, \mathtt{b}, \mathtt{a}_1, \mathtt{a}_2, \ldots$.

Let $\mathbb{O}$ be a set of (type-1) operators $\mathtt{op}$ of fixed arity $ar(\mathtt{op})$ that will be used both in infix and prefix notations for notational convenience and that are always fully applied, *i.e.*, applied to a number $ar(\mathtt{op})$ of operands.

The programs are defined by the grammar of Figure 1. A program is either a term $\mathtt{t}^0$, a *procedure declaration* `declare p in prog`, or the declaration of a *boxed variable* `a`, called *box*, followed by a program: `box [a] in prog`. Boxed variables will represent the program inputs.

In Figure 1, there are three constructor/destructor pairs for abstraction and application; each of them playing a distinct rôle:

- $\lambda \mathtt{a}.\mathtt{t}$ and $\mathtt{t}_1@\mathtt{t}_2$ are the standard abstraction and application on terms.
- The application of a type-1 variable $\mathtt{X}$ within a statement is called an *oracle call*, written $\mathtt{X}(\mathtt{e}_1 \upharpoonright \mathtt{e}_2)$, where $\mathtt{e}_1$ 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*. The corresponding abstraction is called a *closure*, a type-1 map of the shape $\{\mathtt{x} \to \mathtt{t}^0\}$, where the type-0 term $\mathtt{t}$ may contain free variables.
- A procedure declaration $\mathtt{P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})\{[\mathtt{var}\ \overline{\mathtt{y}};]\ \mathtt{st}\ \mathtt{return}\ \mathtt{x}\}$ is an abstraction that computes type-2 functions mapping type-1 and type-0 inputs ($\overline{\mathtt{X}}$ and $\overline{\mathtt{x}}$, respectively) to a type-0 output ($\mathtt{x}$). The *procedure calls* of the shape $\mathtt{call}\ \mathtt{P}(\overline{\mathtt{c}}, \overline{\mathtt{t}^0})$ are the corresponding applications and take closures as type-1 inputs and terms as type-0 inputs.

Type-0 var.	$\mathtt{x}, \mathtt{y}, \mathtt{u}, \mathtt{v}, \mathtt{w} \ldots$	$\in \mathbb{V}_0$
Type-1 var.	$\mathtt{X}, \mathtt{Y}, \mathtt{X}_1, \mathtt{X}_2, \ldots$	$\in \mathbb{V}_1$
Variables	$\mathtt{a}, \mathtt{b}, \mathtt{a}_1, \mathtt{a}_2, \ldots$	$\in \mathbb{V} = \mathbb{V}_0 \uplus \mathbb{V}_1 \uplus \mathbb{V}_{\geq 2}$
Operators	$\mathtt{op}, \upharpoonright$	$\in \mathbb{O}$
Expressions	$\mathtt{e}, \mathtt{e}_1, \mathtt{e}_2, \ldots$	$::= \mathtt{x} \mid \mathtt{op}(\overline{\mathtt{e}}) \mid \mathtt{X}(\mathtt{e}_1 \upharpoonright \mathtt{e}_2)$
Statements	$\mathtt{st}, \mathtt{st}_1, \ldots$	$::= \mathtt{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{st}_1; \mathtt{st}_2 \mid \mathtt{if(e)\{st_1\}\ else\ \{st_2\}} \mid \mathtt{while(e)\{st\}}$
Procedures	$\mathtt{p}, \mathtt{p}_1, \mathtt{p}_2, \ldots$	$::= \mathtt{P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})\{[\mathtt{var}\ \overline{\mathtt{y}};]\ \mathtt{st}\ \mathtt{return}\ \mathtt{x}\}$
Terms	$\mathtt{t}, \mathtt{t}_1, \mathtt{t}_2, \ldots$	$::= \mathtt{a} \mid \lambda \mathtt{a}.\mathtt{t} \mid \mathtt{t}_1@\mathtt{t}_2 \mid \mathtt{call}\ \mathtt{P}(\overline{\mathtt{c}}, \overline{\mathtt{t}^0})$
Closures	$\mathtt{c}, \mathtt{c}_1, \mathtt{c}_2, \ldots$	$::= \{\mathtt{x} \to \mathtt{t}^0\}$
Programs	$\mathtt{prog}$	$::= \mathtt{t}^0 \mid \mathtt{declare}\ \overline{\mathtt{p}}\ \mathtt{in\ prog} \mid \mathtt{box}\ [\overline{\mathtt{a}}]\ \mathtt{in\ prog}$

Fig. 1: Syntax of type-2 programs

For some syntactic element e of the language, let $\mathbb{V}(e) \subseteq \mathbb{V}$ be the set of all variables occurring in e. A variable is free if it is not under the scope of an abstraction and it is not boxed. A program is *closed* if it has no free variable.

For a given procedure declaration $\mathtt{p} = \mathtt{P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})\{[\mathtt{var}\ \overline{\mathtt{y}};]\ \mathtt{st}\ \mathtt{return}\ \mathtt{x}\}$, define the *procedure name* of $\mathtt{p}$ by $\mathtt{n}(\mathtt{p}) \triangleq \mathtt{P}$. Define also $\mathtt{body}(\mathtt{P}) \triangleq \mathtt{st}$, $\mathtt{local}(\mathtt{P}) \triangleq \{\overline{\mathtt{y}}\}$, and $\mathtt{param}(\mathtt{P}) \triangleq \{\overline{\mathtt{X}}, \overline{\mathtt{x}}\}$. $\mathtt{body}(\mathtt{P})$ is called the *body* of procedure $\mathtt{P}$. The variables in $\mathtt{local}(\mathtt{P})$ are called *local variables* and the variables in $\mathtt{param}(\mathtt{P})$ are called *parameters*. Finally, define $\mathtt{Proc}(\mathtt{t})$ (and $\mathtt{Proc}(\mathtt{prog})$) to be the set of procedure names that are called within the term $\mathtt{t}$ (respectively program $\mathtt{prog}$).

Throughout the paper, we will restrict our study to closed programs in *normal form*. These consist of programs with no free variable that can be written as follows $\mathtt{box}\ [\overline{\mathtt{X}}, \overline{\mathtt{x}}]\ \mathtt{in\ declare}\ \overline{\mathtt{p}}\ \mathtt{in\ t}$, for some term $\mathtt{t}$ such that the following well-formedness conditions hold: (i) There are no name clashes. (ii) There are no free variables in a given procedure. (iii) Any procedure call has a corresponding procedure declaration. A closed program in normal form of the shape $\mathtt{box}\ [\overline{\mathtt{X}}, \overline{\mathtt{x}}]\ \mathtt{in\ declare}\ \overline{\mathtt{p}}\ \mathtt{in}\ \mathtt{t}^0$, for some type-0 term $\mathtt{t}$, will compute a type-2

functional. The typing discipline presented in Section 3 will restrict the analysis to such programs.

Operational semantics. 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 is denoted $|w|$. 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 $\exists u, u' \in \mathbb{W},\ w = u.v.u'$.

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 and, consequently, oracle calls are always performed on input data whose size does not exceed the size of the input bound plus one.

A total function $[\![\mathtt{op}]\!] : \mathbb{W}^{ar(\mathtt{op})} \to \mathbb{W}$ is associated with each operator $\mathtt{op}$ of arity $ar(\mathtt{op})$. Constants may be viewed as operators of arity zero. We define two classes of operators called neutral and positive depending on the total function they compute. This categorization of operators will be used by our type system as the admissible types for operators will depend on their category.

An operator $\mathtt{op}$, computing the total function $[\![\mathtt{op}]\!] : \mathbb{W}^{ar(\mathtt{op})} \to \mathbb{W}$, is:

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

As neutral operators are always positive, in the sequel, we reserve the name positive for those operators that are positive but not neutral.

In what follows, let $f, g, \ldots$ denote total functions in $\mathbb{W} \to \mathbb{W}$. A *store* μ consists of the disjoint union of a map μ_0 from $\mathbb{V}_0$ to $\mathbb{W}$ and a map μ_1 from $\mathbb{V}_1$ to total functions in $\mathbb{W} \to \mathbb{W}$. For $i \in \{0,1\}$, μ_i is called a *type-i store*. Let $dom(\mu)$ be the domain of the store μ. Let $\mu[\mathtt{x} \leftarrow w]$ denote the store μ' satisfying $\mu'(b) = \mu(b)$, for all $b \neq \mathtt{x}$, and $\mu'(\mathtt{x}) = w$. This notation is extended naturally to type-1 variables $\mu[\mathtt{X} \leftarrow f]$ and to sequences of variables $\mu[\overline{\mathtt{x}} \leftarrow \overline{w}, \overline{\mathtt{X}} \leftarrow \overline{f}]$. Finally, let $\mu_\emptyset$ denote the empty store.

Let $\downarrow$ denote the standard big-step call-by-name reduction relation on terms defined by: if $\mathtt{t}_1 \downarrow \lambda \mathtt{a}.\mathtt{t}$ and $\mathtt{t}\{\mathtt{t}_2/\mathtt{a}\} \downarrow v$ then $\mathtt{t}_1 @ \mathtt{t}_2 \downarrow v$, where $\{\mathtt{t}_2/\mathtt{a}\}$ is the standard substitution and where v can be a type-0 variable $\mathtt{x}$, a lambda-abstraction $\lambda \mathtt{a}.\mathtt{t}$, a type-1 variable application $\mathtt{X}@\mathtt{t}$, or a procedure call $\mathtt{call}\ \mathtt{P}(\overline{\mathtt{c}}, \overline{\mathtt{t}^0})$.

A *continuation* is a map ϕ from $\mathbb{V}_1$ to $\mathtt{Closures}$, *i.e.*, $\phi(\mathtt{X}) = \{\mathtt{x} \to \mathtt{t}^0\}$ for some type-1 variable $\mathtt{X}$, some type-0 variable $\mathtt{x}$, and some type-0 term $\mathtt{t}^0$. Let $\overline{\mathtt{X}} \mapsto \overline{\mathtt{c}}$ with $|\overline{\mathtt{X}}| = |\overline{\mathtt{c}}|$, be a notation for the continuation mapping each $\mathtt{X}_i \in \mathbb{V}_1$ to the closure $\mathtt{c}_i$.

Given a set of procedures σ, a store μ, and a continuation ϕ, we define three distinct kinds of judgments: $(\sigma, \mu, \phi, \mathtt{e}) \rightarrow_{\mathtt{exp}} w$ for expressions, $(\sigma, \mu, \phi, \mathtt{st}) \rightarrow_{\mathtt{st}} \mu'$ for statements, and $(\sigma, \mu, \mathtt{prog}) \rightarrow_{\mathtt{env}} w$ for programs. The big-step operational semantics of the language is described in Figure 2.

A program $\mathtt{prog} = \mathtt{box}\ [\overline{\mathtt{X}}, \overline{\mathtt{x}}]\ \mathtt{in}\ \mathtt{declare}\ \overline{\mathtt{p}}\ \mathtt{in}\ \mathtt{t}^0$ computes the second-order partial functional $[\![\mathtt{prog}]\!] \in (\mathbb{W} \to \mathbb{W})^{|\overline{\mathtt{X}}|} \to \mathbb{W}^{|\overline{\mathtt{x}}|} \to \mathbb{W}$, defined by:

$$[\![\mathtt{prog}]\!](\overline{f}, \overline{w}) = w \text{ iff } (\emptyset, \mu_\emptyset[\overline{\mathtt{x}} \leftarrow \overline{w}, \overline{\mathtt{X}} \leftarrow \overline{f}], \mathtt{prog}) \rightarrow_{\mathtt{env}} w.$$

In the special case where $[\![\mathtt{prog}]\!]$ is a total function, the program prog is said to be terminating (strongly normalizing). We will denote by SN the set of terminating programs. For a given set of programs S, let $[\![S]\!]$ denote the set of functions computed by programs in S. Formally, $[\![S]\!] = \{[\![\mathtt{prog}]\!] \mid \mathtt{prog} \in S\}$.

Example 2. Consider the program ce provided in Example 1, where:

$$[\![\varepsilon]\!]() = \epsilon \in \mathbb{W}, \quad [\![!=]\!](w, v) = \begin{cases} 1 & \text{if } v = w \\ 0 & \text{otherwise,} \end{cases} \quad [\![\mathtt{pred}]\!](v) = \begin{cases} \epsilon & \text{if } v = \epsilon \\ u & \text{if } v = a.u \end{cases}$$

Program ce is in normal form and computes the second-order functional $F\colon (\mathbb{W} \to \mathbb{W}) \to \mathbb{W} \to \mathbb{W}$ defined by: $\forall f \in \mathbb{W} \to \mathbb{W}, \forall w \in \mathbb{W},\ F(f)(w) = F_{|w|}(f)$, where F_n is defined recursively as $F_0(f) = \epsilon$ and $F_{n+1}(f) = (f \circ f)([\![\restriction]\!](F_n(f), f(1))) = (f \circ f)(F_n(f)_{\restriction |f(1)|})$. That is a function that composes the input function $2|w|$ times f while restricting its input to a fixed size $|f(1)|$ every other iteration. Indeed, $[\![\varepsilon]\!]() = \epsilon$ and $[\![\restriction]\!](\epsilon, \epsilon) = \epsilon_{\restriction |\epsilon|} = 1$. Consequently, the oracle bound w in the oracle call $\mathtt{X}_2(\mathtt{z} \restriction \mathtt{w})$ is bound to value $f(1)$ in the store by the statement $\mathtt{w} := \mathtt{X}_1(\varepsilon \restriction \varepsilon)$.

Observe that the operators ε, != and pred are all neutral. An example of positive operator can be given by the successor operators defined by $[\![\mathtt{suc}_i]\!](v) = i.v$, for $i \in \{0, 1\}$. These operators are positive since $|[\![\mathtt{suc}_i]\!](v)| = |i.v| = |v|+1$.

3 Type system

Tiers and typing environments. Let W be the type of words in $\mathbb{W}$. *Simple types* over W are defined inductively by $\mathtt{T}, \mathtt{T}', \ldots ::= \mathtt{W} \mid \mathtt{T} \to \mathtt{T}$. Let $\mathcal{T}_{\mathtt{W}}$ be the set of simple types over W. The order of a simple type in $\mathcal{T}_{\mathtt{W}}$ is defined inductively by: $\mathtt{ord}(\mathtt{T}) = 0$, if $\mathtt{T} = \mathtt{W}$, and $\mathtt{ord}(\mathtt{T}) = \max(1 + \mathtt{ord}(\mathtt{T}_1), \mathtt{ord}(\mathtt{T}_2))$, if $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$.

Tiers 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, $\preceq$ is the standard 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 $\mathbf{k}, \mathbf{k}', \ldots, \mathbf{k}_1, \mathbf{k}_2, \ldots$ 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 *first-order tier* is of the shape $\mathbf{k}_1 \to \ldots \to \mathbf{k}_n \to \mathbf{k}'$, with $\mathbf{k}_i, \mathbf{k}' \in \mathbf{N}$.

A *simple typing environment* $\Gamma_{\mathtt{W}}$ is a finite partial map from $\mathbb{V}$ to $\mathcal{T}_{\mathtt{W}}$, which assigns simple types to variables.

$$\frac{}{(\sigma,\mu,\phi,\mathtt{x}) \to_{\mathtt{exp}} \mu(\mathtt{x})}\ (\mathrm{Var}) \qquad \frac{(\sigma,\mu,\phi,\overline{\mathtt{e}}) \to_{\mathtt{exp}} \overline{w}}{(\sigma,\mu,\phi,\mathtt{op}(\overline{\mathtt{e}})) \to_{\mathtt{exp}} [\![\mathtt{op}]\!](\overline{w})}\ (\mathrm{Op})$$

$$\frac{(\sigma,\mu,\phi,\mathtt{e}_1) \to_{\mathtt{exp}} v \qquad (\sigma,\mu,\phi,\mathtt{e}_2) \to_{\mathtt{exp}} u \qquad \phi(\mathtt{X}) = \{\mathtt{x} \to \mathtt{t}^0\} \qquad (\sigma,\mu[\mathtt{x} \leftarrow [\![\uparrow]\!](v,u)],\mathtt{t}^0) \to_{\mathtt{env}} w}{(\sigma,\mu,\phi,\mathtt{X}(\mathtt{e}_1 \uparrow \mathtt{e}_2)) \to_{\mathtt{exp}} w}\ (\mathrm{Orc})$$

(a) Expressions

$$\frac{}{(\sigma,\mu,\phi,\mathtt{skip}) \to_{\mathtt{st}} \mu}\ (\mathrm{Skip}) \qquad \frac{(\sigma,\mu,\phi,\mathtt{st}_1) \to_{\mathtt{st}} \mu' \qquad (\sigma,\mu',\phi,\mathtt{st}_2) \to_{\mathtt{st}} \mu''}{(\sigma,\mu,\phi,\mathtt{st}_1;\ \mathtt{st}_2) \to_{\mathtt{st}} \mu''}\ (\mathrm{Seq})$$

$$\frac{(\sigma,\mu,\phi,\mathtt{e}) \to_{\mathtt{exp}} w}{(\sigma,\mu,\phi,\mathtt{x} := \mathtt{e}) \to_{\mathtt{st}} \mu[\mathtt{x} \leftarrow w]}\ (\mathrm{Asg})$$

$$\frac{(\sigma,\mu,\phi,\mathtt{e}) \to_{\mathtt{exp}} w \qquad (\sigma,\mu,\phi,\mathtt{st}_w) \to_{\mathtt{st}} \mu' \qquad w \in \{0,1\}}{(\sigma,\mu,\phi,\mathtt{if(e)\{st_1\}\ else\ \{st_0\}}) \to_{\mathtt{st}} \mu'}\ (\mathrm{Cond})$$

$$\frac{(\sigma,\mu,\phi,\mathtt{e}) \to_{\mathtt{exp}} 0}{(\sigma,\mu,\phi,\mathtt{while(e)\{st\}}) \to_{\mathtt{st}} \mu}\ (\mathrm{Wh}_0)$$

$$\frac{(\sigma,\mu,\phi,\mathtt{e}) \to_{\mathtt{exp}} 1 \qquad (\sigma,\mu,\phi,\mathtt{st};\ \mathtt{while(e)\{st\}}) \to_{\mathtt{st}} \mu'}{(\sigma,\mu,\phi,\mathtt{while(e)\{st\}}) \to_{\mathtt{st}} \mu'}\ (\mathrm{Wh}_1)$$

(b) Statements

$$\frac{\mathtt{t}^0 \downarrow \mathtt{x}}{(\sigma,\mu,\mathtt{t}^0) \to_{\mathtt{env}} \mu(\mathtt{x})}\ (\mathrm{TVar}) \qquad \frac{\mathtt{t}^0 \downarrow \mathtt{X@t}_1^0 \qquad (\sigma,\mu,\mathtt{t}_1^0) \to_{\mathtt{env}} w}{(\sigma,\mu,\mathtt{t}^0) \to_{\mathtt{env}} \mu(\mathtt{X})(w)}\ (\mathrm{OA})$$

$$\frac{\mathtt{t}^0 \downarrow \mathtt{call\ P}(\overline{\mathtt{c}},\overline{\mathtt{t}^0}) \qquad (\sigma,\mu,\overline{\mathtt{t}^0}) \to_{\mathtt{env}} \overline{w} \qquad (\sigma,\mu[\overline{\mathtt{x}} \leftarrow \overline{w},\overline{\mathtt{y}} \leftarrow \overline{\epsilon}],\overline{\mathtt{X}} \mapsto \overline{\mathtt{c}},\mathtt{st}) \to_{\mathtt{st}} \mu'}{(\sigma \cup \{\mathtt{P}(\overline{\mathtt{X}},\overline{\mathtt{x}})\{\mathtt{var}\ \overline{\mathtt{y}};\ \mathtt{st\ return\ z}\}\},\mu,\overline{\mathtt{t}^0}) \to_{\mathtt{env}} \mu'(\mathtt{z})}\ (\mathrm{Call})$$

(c) Type-0 terms

$$\frac{(\sigma \cup \{\mathtt{p}\},\mu,\mathtt{prog}) \to_{\mathtt{env}} w}{(\sigma,\mu,\mathtt{declare\ p\ in\ prog}) \to_{\mathtt{env}} w}\ (\mathrm{Dec}) \qquad \frac{(\sigma,\mu,\mathtt{prog}) \to_{\mathtt{env}} w \qquad \mathtt{a} \in \mathit{dom}(\mu)}{(\sigma,\mu,\mathtt{box\ [a]\ in\ prog}) \to_{\mathtt{env}} w}\ (\mathrm{Box})$$

(d) Programs

Fig. 2: Big step operational semantics

A *variable typing environment* Γ is a finite partial map from $\mathbb{V}_0$ to $\mathbf{N}$, which assigns single tiers to type-0 variables.

An *operator typing environment* Δ is a mapping that associates to some operator $\mathtt{op}$ and some tier $\mathbf{k} \in \mathbf{N}$ a set of admissible first-order tiers $\Delta(\mathtt{op})(\mathbf{k})$ of the shape $\mathbf{k}_1 \to \ldots \to \mathbf{k}_{ar(\mathtt{op})} \to \mathbf{k}'$.

A *procedure typing environment* Ω is a mapping that associates to each procedure name $\mathtt{P}$ a pair $\langle \Gamma, \overline{\mathbf{k}} \rangle$ consisting of a variable typing environment Γ and a triplet of tiers $\overline{\mathbf{k}}$. Let $\Omega_i \triangleq \pi_i(\Omega)$, $i \in \{1, 2\}$.

Let $dom(\Gamma)$, $dom(\Gamma_{\mathtt{W}})$, $dom(\Delta)$, and $dom(\Omega)$ denote the sets of variables typed by Γ and $\Gamma_{\mathtt{W}}$, the set of operators typed by Δ, and the set of procedures typed by Ω, respectively.

For a procedure typing environment Ω, it will be assumed that for every $\mathtt{P} \in dom(\Omega)$, $\mathtt{param}(\mathtt{P}) \cup \mathtt{local}(\mathtt{P}) \subseteq dom(\Omega_1(\mathtt{P}))$.

While operator and procedure typing environments are global, *i.e.*, defined for the whole program, variable typing environments are local, *i.e.*, relative to the procedure under analysis. In a program typing judgment, the simple typing environment can be viewed as the typing environment for the main program.

Typing judgments and type system. The typing discipline includes two distinct kinds of typing judgments: *Procedure typing judgments* $\Gamma, \Delta \vdash o : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$ and *Term typing judgments* $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{prog} : \mathtt{T}$, with $\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out} \in \mathbf{N}$, $o \in \mathtt{Expressions} \cup \mathtt{Statements}$, and $\mathtt{T} \in \mathcal{T}_{\mathtt{W}}$.

The meaning of the procedure typing judgment is that the *expression tier* (or *statement 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 statement is located. The meaning of term typing judgments is that the program $\mathtt{prog}$ is of simple type $\mathtt{T}$ under the operator typing environment Δ, the procedure typing environment Ω and the simple typing environment $\Gamma_{\mathtt{W}}$.

A program $\mathtt{prog}$ (or term $\mathtt{t}$) is of *type-i*, if $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{prog} : \mathtt{T}$ ($\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{t} : \mathtt{T}$) can be derived for some typing environments and type $\mathtt{T}$ s.t. $\mathtt{ord}(\mathtt{T}) = i$.

The type system for the considered programming language is provided in Figure 3. A *well-typed program* is a program that can be given the type $(\overline{\mathtt{W} \to \mathtt{W}}) \to \overline{\mathtt{W}} \to \mathtt{W}$, *i.e.*, the judgment $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{prog} : (\overline{\mathtt{W} \to \mathtt{W}}) \to \overline{\mathtt{W}} \to \mathtt{W}$ can be derived for the environments $\Gamma_{\mathtt{W}}, \Omega, \Delta$. Consequently, a well-typed program is a type-i program, for some $i \leq 2$, computing a functional.

For a given typing judgment j, a *typing derivation* $\pi \rhd j$ is a tree whose root is the (procedure or term) typing judgment j and whose children are obtained by applications of the typing rules of Figure 3. The name π will be used alone whenever mentioning the root of a typing derivation is not explicitly needed. A typing sub-derivation of a typing derivation π is a subtree of π.

Intuitions. We now give some brief intuition to the reader on the type discipline in the particular case where exactly two tiers, **0** and **1**, are involved. The type system splits program variables, expressions, and statements between the two disjoint *tiers*:

$$\frac{\Gamma(\mathtt{x}) = \mathbf{k}}{\Gamma, \Delta \vdash \mathtt{x} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (E-VAR)}$$

$$\frac{\mathbf{k}_1 \to \cdots \to \mathbf{k}_{|\overline{\mathtt{e}}|} \to \mathbf{k} \in \Delta(\mathtt{op})(\mathbf{k}_{in}) \qquad \forall i \leq |\overline{\mathtt{e}}|,\ \Gamma, \Delta \vdash \mathtt{e}_i : (\mathbf{k}_i, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathtt{op}(\overline{\mathtt{e}}) : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (E-OP)}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e}_1 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathtt{e}_2 : (\mathbf{k}_{out}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \mathbf{k} \prec \mathbf{k}_{in} \wedge \mathbf{k} \preceq \mathbf{k}_{out}}{\Gamma, \Delta \vdash \mathtt{X}(\mathtt{e}_1 \restriction \mathtt{e}_2) : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (E-OR)}$$

$$\frac{}{\Gamma, \Delta \vdash \mathtt{skip} \ : (\mathbf{0}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (S-SK)} \qquad \frac{\Gamma, \Delta \vdash \mathtt{st} \ : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathtt{st} \ : (\mathbf{k}{+}\mathbf{1}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (S-SUB)}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{st}_1 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathtt{st}_2 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathtt{st}_1; \ \mathtt{st}_2 \ : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (S-SEQ)}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{x} : (\mathbf{k}_1, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathtt{e} : (\mathbf{k}_2, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \mathbf{k}_1 \preceq \mathbf{k}_2}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{e} \ : (\mathbf{k}_1, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (S-ASG)}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathtt{st}_1 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathtt{st}_0 : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})}{\Gamma, \Delta \vdash \mathtt{if(e)\{st_1\}\ else\ \{st_0\}} \ : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (S-CND)}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}) \qquad \Gamma, \Delta \vdash \mathtt{st} : (\mathbf{k}, \mathbf{k}, \mathbf{k}) \qquad \mathbf{1} \preceq \mathbf{k}}{\Gamma, \Delta \vdash \mathtt{while(e)\{st\}} \ : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{0})} \text{ (S-WINIT)}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{e} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Gamma, \Delta \vdash \mathtt{st} : (\mathbf{k}, \mathbf{k}, \mathbf{k}_{out}) \qquad \mathbf{1} \preceq \mathbf{k} \preceq \mathbf{k}_{out}}{\Gamma, \Delta \vdash \mathtt{while(e)\{st\}} \ : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{ (S-WH)}$$

(a) Tier-based typing rules for expressions and statements

$$\frac{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \overline{\mathtt{X}} : \overline{\mathtt{W} \to \mathtt{W}} \qquad \Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \overline{\mathtt{x}}, \overline{\mathtt{y}}, \mathtt{x} : \overline{\mathtt{W}}}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})\{[\mathtt{var}\ \overline{\mathtt{y}};]\ \mathtt{st\ return\ x}\} : (\overline{\mathtt{W} \to \mathtt{W}}) \to \overline{\mathtt{W}} \to \mathtt{W}} \text{ (PR-DEC)}$$

$$\frac{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})\{\ldots\} : (\overline{\mathtt{W} \to \mathtt{W}}) \to \overline{\mathtt{W}} \to \mathtt{W} \qquad \Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \overline{\mathtt{c}} : \overline{\mathtt{W} \to \mathtt{W}} \qquad \Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \overline{\mathtt{t}} : \overline{\mathtt{W}}}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{call\ P}(\overline{\mathtt{c}}, \overline{\mathtt{t}}) : \mathtt{W}} \text{ (P-CALL)}$$

$$\frac{\Gamma_{\mathtt{W}}(\mathtt{a}) = \mathtt{T}}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{a} : \mathtt{T}} \text{ (P-VAR)} \qquad \frac{\Gamma_{\mathtt{W}} \uplus \{\mathtt{a} : \mathtt{T}\}, \Omega, \Delta \vdash \mathtt{t} : \mathtt{T}'}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \lambda\mathtt{a.t} : \mathtt{T} \to \mathtt{T}'} \text{ (P-ABS)}$$

$$\frac{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{t}_1 : \mathtt{T} \to \mathtt{T}' \qquad \Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{t}_1 @ \mathtt{t}_2 : \mathtt{T}'} \text{ (P-APP)}$$

$$\frac{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{prog} : \mathtt{T} \qquad \Gamma, \Delta \vdash \mathtt{body(n(p))} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \qquad \Omega(\mathtt{n(p)}) = \langle \Gamma, (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \rangle}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{declare\ p\ in\ prog} : \mathtt{T}} \text{ (P-DEC)}$$

$$\frac{\Gamma_{\mathtt{W}} \uplus \{\mathtt{x} : \mathtt{W}\}, \Omega, \Delta \vdash \mathtt{t} : \mathtt{W}}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \{\mathtt{x} \to \mathtt{t}\} : \mathtt{W} \to \mathtt{W}} \text{ (P-CLOS)}$$

$$\frac{\Gamma_{\mathtt{W}} \uplus \{\mathtt{a} : \mathtt{T}\}, \Omega, \Delta \vdash \mathtt{prog} : \mathtt{T}'}{\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{box\ [a]\ in\ prog} : \mathtt{T} \to \mathtt{T}'} \text{ (P-BOX)}$$

(b) Simple typing rules for procedures, terms, closures and programs

Fig. 3: Tier-based type system

– **0** corresponds to a program component whose execution may result in a memory increase (in size) and that cannot control the program flow.
– **1** corresponds to a program component whose execution cannot result in a memory increase and that may control the program flow.

The type system of Figure 3 is composed of two sub-systems. The typing rules provided in Figure 3b enforce that terms follow a standard simply-typed discipline. The typing rules of Figure 3a will implement a standard non-interference type discipline à la Volpano et al. [30] on the expression (and statement) tier, preventing data flows from tier **0** to tier **1**. The transition between the two sub-type-systems is performed in the rule (P-DEC) of Figure 3b that checks that the procedure body follows the tier-based type discipline once and for all in a procedure declaration.

In Figure 3a, as tier **1** data cannot grow (but can decrease) and are the only data driving the program flow, the number of distinct memory configurations on such data for a terminating procedure is polynomial in the size of the program input (*i.e.*, number of symbols). Hence a typable and terminating procedure has a *polynomial step count* (in the sense of [11]), *i.e.*, on any input, the execution time of a procedure is bounded by a first-order polynomial in the size of their input and the maximal size of any answer returned by an oracle call.

The innermost tier is used to implement a declassification mechanism on operators improving the type-system's expressive power: an operator may be typed differently depending on its calling context (the statement where it is applied). This is the reason why more than 2 tiers can be used in general.

The outermost tier is used to ensure that oracles are only called on inputs of bounded size. This latter restriction on oracle calls enforces a semantic restriction, called *finite lookahead revision*, introduced in [22,20] and requiring that, during each computation, the number of calls performed by the oracle on an input of increasing size is bounded by a constant.

Let MPT be the class of second-order functionals computable by an oracle Turing machine with a polynomial step count and a finite lookahead revision. [20] shows that $\texttt{BFF} = \lambda(\text{MPT})_2$. The type system of Figure 3 ensures that each terminating procedure of a well-typed program computes a function in MPT.

Safe programs. In this section, we restrict the set of admissible operators to prevent programs admitting exponential growth from being typable. A program satisfying such a restriction will be called *safe*.

An operator typing environment Δ is *safe* if for each $\texttt{op} \in dom(\Delta)$ such that $ar(\texttt{op}) > 0$, $\texttt{op}$ is neutral or positive, $[\![\texttt{op}]\!]$ is a polynomial time computable function, and for each $\mathbf{k} \in \mathbf{N}$, and for each $\mathbf{k}_1 \to \ldots \mathbf{k}_{ar(\texttt{op})} \to \mathbf{k}' \in \Delta(\texttt{op})(\mathbf{k})$, the two conditions below hold:

1. $\mathbf{k}' \preceq \wedge_{i=1}^{ar(\texttt{op})} \mathbf{k}_i \preceq \vee_{i=1}^{ar(\texttt{op})} \mathbf{k}_i \preceq \mathbf{k}$,
2. if $\texttt{op}$ is a positive operator then $\mathbf{k}' \prec \mathbf{k}$.

Example 3. Consider the operators $!=$, $\texttt{pred}$, and $\texttt{suc}_i$ discussed in Example 1 and an operator typing environment Δ that is safe and such that $!=$, $\texttt{pred}$, $\texttt{suc}_i$

$\in dom(\Delta)$. We can set $\Delta(!=)(\mathbf{1}) \triangleq \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}\} \cup \{\mathbf{k} \to \mathbf{k}' \to \mathbf{0} \mid \mathbf{k}, \mathbf{k}' \preceq \mathbf{1}\}$, as $!=$ is neutral. However $\mathbf{1} \to \mathbf{0} \to \mathbf{1} \notin \Delta(!=)(\mathbf{1})$ as it breaks Condition 1) above (*i.e.*, $\mathbf{1} \not\preceq \mathbf{1} \wedge \mathbf{0}$).

We can also set $\Delta(\mathtt{pred})(\mathbf{2}) \triangleq \{\mathbf{2} \to \mathbf{k} \mid \mathbf{k} \preceq \mathbf{2}\} \cup \{\mathbf{1} \to \mathbf{k} \mid \mathbf{k} \preceq \mathbf{1}\} \cup \{\mathbf{0} \to \mathbf{0}\}$. We also 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 $\mathtt{suc}_i$ is a positive operator and, due to Condition 2) above, the operator output tier has to be strictly smaller than $\mathbf{1}$.

A program `prog` is a *safe program* if there exist a simple typing environment $\Gamma_{\mathtt{W}}$, a procedure typing environment Ω, and a safe operator typing environment Δ, such that it is well-typed for these environments, *i.e.*, $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{prog} : (\overline{\mathtt{W} \to \mathtt{W}}) \to \overline{\mathtt{W}} \to \mathtt{W}$ can be derived. Let SAFE be the set of safe programs.

Example 4. We consider the program `ce` of Example 1. We define the operator typing environment Δ by$\Delta(!=)(\mathbf{2}) \triangleq \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}\}$, $\Delta(\mathtt{pred})(\mathbf{1}) \triangleq \{\mathbf{1} \to \mathbf{1}\}$, and $\Delta(\varepsilon)(\mathbf{2}) \triangleq \{\mathbf{0}, \mathbf{1}\}$. As the three operators $!=$, `pred`, and ε are neutral, the environment Δ is safe. We define the simple typing environment $\Gamma_{\mathtt{W}}$ by $\Gamma_{\mathtt{W}}(\mathtt{w}) \triangleq \mathtt{W}, \Delta(\mathtt{v}) \triangleq \mathtt{W}, \Delta(\mathtt{z}) \triangleq \mathtt{W}, \Gamma_{\mathtt{W}}(\mathtt{X}_1) \triangleq \mathtt{W} \to \mathtt{W}$, and $\Gamma_{\mathtt{W}}(\mathtt{X}_2) \triangleq \mathtt{W} \to \mathtt{W}$. We define the variable typing environment Γ by $\Gamma(\mathtt{w}) \triangleq \mathbf{1}, \Delta(\mathtt{v}) \triangleq \mathbf{1}, \Delta(\mathtt{z}) \triangleq \mathbf{0}$. Finally, define the procedure typing environment Ω by $\Omega(\mathtt{KS}) \triangleq \langle \Gamma, (\mathbf{1}, \mathbf{2}, \mathbf{1}) \rangle$. Using the rules of Figure 3, the following typing judgement can be derived $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{ce} : (\mathtt{W} \to \mathtt{W}) \to \mathtt{W} \to \mathtt{W}$. Hence $\mathtt{ce} \in$ SAFE.

4 Characterizations of the class of Basic Feasible Functionals

Safe and terminating programs. In this section, we show that typable (safe) and terminating programs capture exactly the class of basic feasible functionals.

For a given set of functionals $\mathcal{S}$, let $\mathcal{S}_2$ be the restriction of $\mathcal{S}$ to second-order functionals and let $\lambda(\mathcal{S})$ be the set of functions computed by closed simply-typed lambda terms using functions in $\mathcal{S}$ as constants. Formally, let $\lambda(\mathcal{S})$ be the set of functions denoted by 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 $F \in \mathcal{S}$ of type τ, F^τ is a term,
- for any term $t^{\tau'}$ and variable x^τ, $\lambda x^\tau.t^{\tau'}$ is a term of type $\tau \to \tau'$,
- for any terms $t^{\tau \to \tau'}$ and s^τ, $t^{\tau \to \tau'} s^\tau$ is a term of type τ'.

Each lambda term of type τ represents a function of type τ and terms are considered up to β and η equivalences. $\lambda(\mathcal{S})_2$ is called the *second-order simply-typed lambda closure* of $\mathcal{S}$.

A procedure $\mathtt{p} \triangleq \mathtt{P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})\{[\mathtt{var}\ \overline{\mathtt{y}};]\ \mathtt{st}\ \mathtt{return}\ \mathtt{x}\}$ is safe if there exist a simple typing environment $\Gamma_{\mathtt{W}}$, a safe operator typing environment Δ, and a triplet of tiers $(\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$, such that `p` is well-typed for these environments, i.e $\Gamma_{\mathtt{W}}, \Delta \vdash \mathtt{st} : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})$ can be derived using the rules of Figure 3. `P` computes a

second-order partial functional $[\![\mathtt{P}]\!] \in (\mathbb{W} \to \mathbb{W})^{|\overline{\mathtt{X}}|} \to \mathbb{W}^{|\overline{\mathtt{x}}|} \to \mathbb{W}$, defined by $[\![\mathtt{P}]\!](\overline{f}, \overline{w}) = w$ iff $(\{\mathtt{p}\}, \mu_\emptyset[\overline{\mathtt{x}} \leftarrow \overline{w}, \overline{\mathtt{X}} \leftarrow \overline{f}], \mathtt{call\ P}(\overline{\mathtt{X}}, \overline{\mathtt{x}})) \to_{\mathtt{env}} w$ (see Figure 2). If $[\![\mathtt{P}]\!]$ is a total function, then the procedure terminates. Let ST be the set of safe and terminating procedures.

The characterization of BFF in terms of safe and terminating procedures discussed in the introduction can be stated as follows.

Theorem 1 ([16]). $\lambda([\![\mathrm{ST}]\!])_2 = \mathtt{BFF}$.

We are now ready to state a first characterization of BFF in terms of safe (SAFE) and terminating (SN) programs, showing that the external simply-typed lambda-closure of Theorem 1 can be removed.

Theorem 2. $[\![\mathrm{SN} \cap \mathrm{SAFE}]\!]_2 = \mathtt{BFF}$.

We want to highlight that the characterization of Theorem 2 is not just "moving" the simply-typed lambda-closure inside the programming language by adding a construct for lambda-abstraction. Indeed, the soundness of this result crucially depends on some choices on the language design that we have enforced: the restricted ability to compose oracles using closures, and the read-only mode of oracles inside a procedure call, implemented through continuations.

Safe and terminating rank-r programs. More importantly, we also show that this characterization is still valid in the absence of lambda-abstraction.

A safe program prog w.r.t. to a typing derivation π is a *rank-r* program, if for any typing sub-derivation $\pi' \rhd \ \Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \lambda \mathtt{a.t} : \mathtt{T}$ of π, it holds that $\mathtt{ord}(\mathtt{T}) \leq r$. In other words, all lambda-abstractions are at most type-k terms, for $k \leq r$. In particular, a rank-$(r+1)$ program, for $r \geq 1$, has variables that are at most type-r variables. Rank-0 and rank-1 programs may have both type-0 and type-1 variables as these variables can still be captured by closures, procedure declarations, or boxes.

For a given set S of well-typed programs, let S_r be the subset of rank-r programs in S, *i.e.*, $S_r \triangleq \{\mathtt{prog} \in S \mid \mathtt{prog}$ is a rank-r program$\}$. For example, SAFE_r denotes the set of safe rank-r programs. It trivially holds that $\mathrm{SAFE} = \cup_{r \in \mathbb{N}} \mathrm{SAFE}_r$. The rank is clearly not uniquely determined for a given program. In particular, any rank-r program is also a rank-$(r+1)$ program. Consequently, for any set S of well-typed programs and any $i \leq j$, it trivially holds that $S_i \subseteq S_j$.

Example 5. Program ce of Example 1 is in SAFE_0. Indeed, $\mathtt{ce} \in \mathrm{SAFE}$, *cf.* Example 4, and ce is a rank-0 program, as it does not use any lambda-abstraction.

Now we revisit the syntax and semantics of safe rank-0 programs in SAFE_0. The programs are generated by the syntax of Figure 1, where the terms are all of type-0 and redefined by:

$$\mathtt{Terms} \qquad \mathtt{t}^0, \mathtt{t}^0_1, \mathtt{t}^0_2, \ldots \ ::= \mathtt{x} \mid \mathtt{X@t}^0 \mid \mathtt{call\ P}(\overline{\mathtt{c}}, \overline{\mathtt{t}^0})$$

Moreover, there is no longer a need for call-by-name reduction in the big step operational semantics. As a consequence, the rules (TVar), (OA), and (Call) of

Figure 2c can be replaced by the following simplified rules:

$$\frac{}{(\sigma,\mu,\mathtt{x}) \to_{\mathtt{env}} \mu(\mathtt{x})}\ (\mathrm{TVar}^0) \qquad \frac{(\sigma,\mu,\mathtt{t}_1^0) \to_{\mathtt{env}} w}{(\sigma,\mu,\mathtt{X@t}_1^0) \to_{\mathtt{env}} \mu(\mathtt{X})(w)}\ (\mathrm{OA}^0)$$

$$\frac{(\sigma,\mu,\overline{\mathtt{t}^0}) \to_{\mathtt{env}} \overline{w} \quad (\sigma,\mu[\overline{\mathtt{x}} \leftarrow \overline{w}, \overline{\mathtt{y}} \leftarrow \overline{\epsilon}], \overline{\mathtt{X}} \mapsto \overline{\mathtt{c}}, \mathtt{st}) \to_{\mathtt{st}} \mu'}{(\sigma \cup \{\mathtt{P}(\overline{\mathtt{X}},\overline{\mathtt{x}})\{\mathtt{var}\ \overline{\mathtt{y}};\ \mathtt{st\ return\ z}\}\}, \mu, \mathtt{call\ P}(\overline{\mathtt{c}},\overline{\mathtt{t}^0})) \to_{\mathtt{env}} \mu'(\mathtt{z})}\ (\mathrm{Call}^0)$$

We are now ready to characterize BFF in terms of safe and terminating rank-0 programs.

Theorem 3. $[\![\mathrm{SN} \cap \mathrm{SAFE}_0]\!] = \mathtt{BFF}$.

Hence the characterization of Theorem 2 is just a conservative extension of Theorem 3: lambda-abstractions, viewed as a construct of the programming language, allow for more expressive power in the programming discipline but do not capture more functions. As lambda-abstraction is fully removed from the programming language, this also shows that the simply-typed lambda closure of Theorem 1 can be simulated through restricted oracle compositions in our programming language (using closures and continuations). Moreover, the full hierarchy of safe and terminating rank-r programs collapses.

Corollary 1. $\forall r \in \mathbb{N},\ [\![\mathrm{SN} \cap \mathrm{SAFE}_r]\!] = \mathtt{BFF}$.

Tractable type inference. Let the size $|\mathtt{prog}|$ of the program prog be the total number of symbols in prog. Type inference is tractable for safe programs.

Theorem 4. *Given a program* prog *and a safe operator typing environment* Δ*,*

- *deciding whether* $\mathtt{prog} \in \mathrm{SAFE}$ *holds is a* P*-complete problem.*
- *deciding whether* $\mathtt{prog} \in \mathrm{SAFE}_0$ *holds can be done in time* $\mathcal{O}(|\mathtt{prog}|^3)$.

Tractability of type inference is a nice property of the type system. Showing $\mathtt{prog} \in \mathrm{SN}$ is at least as hard as showing the termination of a first-order program, hence Π_2^0-hard in the arithmetical hierarchy. Therefore, the characterizations of Theorems 1, 2, and 3 are unlikely to be decidable, let alone tractable.

5 A completeness-preserving termination criterion

In this section, we show that the undecidable termination assumption (SN) can be replaced with a criterion, called SCP_S, adapted from the Size-Change Termination (SCT) techniques of [23], that is decidable in polynomial time and that preserves the completeness of the characterizations. We first show that studying safe program termination can be reduced to the study of procedure termination.

Lemma 1. *For a given* $\mathtt{prog} \in \mathrm{SAFE}$*, if there exists* $\mathtt{P} \in \mathtt{Proc(prog)}$ *that terminates, then* prog *is terminating.*

Hence, ensuring the termination of any procedure of a given safe program is a sufficient condition for the program to terminate. The converse trivially does not hold as, for example, a procedure with an infinite loop may be declared and not be called within a given safe program.

Size-Change Termination. SCT relies on the fact that if all infinite executions imply an infinite descent in a well-founded order, then no infinite execution exists. To apply this fact for proving termination, [23] defines Size-Change Graphs (SCGs) that exhibit decreases in the parameters of function calls and then studies the infinite paths in all possible infinite sequences of calls. If all those infinite sequences have at least one strictly decreasing path, then the program must terminate for all inputs. While SCT is PSpace-complete, [7] develops a more effective technique, called SCP, that is in P. The SCP technique is strong enough for our use case. In the literature, SCT and SCP are applied to pure functional languages. As we shall enforce termination of procedures, we will follow the approach of [1] adapting SCT to imperative programs.

First, we distinguish two kinds of operators that will enforce some (strict) decrease. An operator op is *(strictly) decreasing in* i, for $i \leq ar(\texttt{op})$, if $\forall \overline{w} \in \overline{\mathbb{W}}$, $\overline{w} \neq \bar{\epsilon}$, $|[\![\texttt{op}]\!](\overline{w})| \leq |w_i|$ ($|[\![\texttt{op}]\!](\overline{w})| < |w_i|$, respectively) and $[\![\texttt{op}]\!](\bar{\epsilon}) = \epsilon$. For operators of arity greater than 2, i may not be unique but will be fixed for each operator in what follows.

For simplicity, we will assume that assignments of the considered programs are *flattened*, that is for any assignment $\texttt{x} := \texttt{e}$, either $\texttt{e} = \texttt{y} \in \mathbb{V}_0$, or $\texttt{e} = \texttt{op}(\overline{\texttt{x}})$, with $\overline{\texttt{x}} \in \overline{\mathbb{V}_0}$, or $\texttt{e} = \texttt{X}(\texttt{y} \restriction \texttt{z})$, with $\texttt{y}, \texttt{z} \in \mathbb{V}_0$ and $\texttt{X} \in \mathbb{V}_1$. Notice that, by using extra type-0 variables, any program can be easily transformed into a program with flattened assignments, while preserving semantics and safety properties.

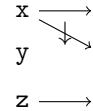
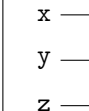
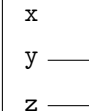
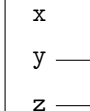
For each assignment of a procedure P, we design a bipartite graph, called a SCG, whose nodes are type-0 variables in $(\texttt{local}(\texttt{P}) \cup \texttt{param}(\texttt{P})) \cap \mathbb{V}_0$ and arrows indicates decreases or stagnation from the old variable to the new. If a variable may increase, then the new variable will not have an in-arrow.

The bipartite graph is generated for any flattened assignment $\texttt{x} := \texttt{e}$ by:

- for each y, $\texttt{y} \neq \texttt{x}$, we draw arrows from left y to right y.
- If $\texttt{e} = \texttt{y}$, we draw an arrow from left y to right x.
- If $\texttt{e} = \texttt{op}(\overline{\texttt{x}})$, with op a:
  - decreasing operator in i, we draw an arrow from $\texttt{x}_i$ to x.
  - strictly decreasing operator in i, we draw a "down-arrow" from $\texttt{x}_i$ to x.

In all other cases (neutral and non-decreasing operators, positive operators, oracle calls), we do not draw arrows. We will name this SCG graph $\mathrm{G}(\texttt{x} := \texttt{e})$. Finally, for a set V of variables, G^V will denote the SCG obtained as a subgraph of G restricted to the variables of V.

Example 6. Here are the SCGs associated to simple assignments of a procedure with three type-0 variables x, y, z using a strictly decreasing operator in 1 (pred), a decreasing operator (min) in 1, a positive operator (+1), and an oracle call.

y := pred(x)	y := min(x, y)	x := x + 1	x := X(y ↾ z)
			

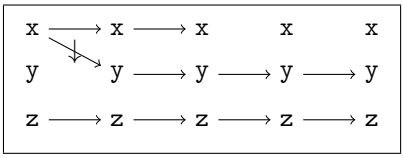
The language $\mathcal{L}(\mathtt{st})$ of (potentially infinite) *sequences of SCG* associated with the statement $\mathtt{st}$ is defined inductively as an ∞-regular expression.

$$\mathcal{L}(\mathtt{x := e}) \triangleq G(\mathtt{x := e}) \qquad \mathcal{L}(\mathtt{if(e)\{st_1\}else\{st_2\}}) \triangleq \mathcal{L}(\mathtt{st_1}) + \mathcal{L}(\mathtt{st_2})$$
$$\mathcal{L}(\mathtt{st_1; st_2}) \triangleq \mathcal{L}(\mathtt{st_1}).\mathcal{L}(\mathtt{st_2}) \qquad \mathcal{L}(\mathtt{while(e)\{st_1\}}) \triangleq \mathcal{L}(\mathtt{st_1})^{\infty}$$

where, following the standard terminology for automata [28], $\mathcal{L}(\mathtt{st})^{\infty}$ is defined by $\mathcal{L}(\mathtt{st})^{\infty} \triangleq \mathcal{L}(\mathtt{st})^* + \mathcal{L}(\mathtt{st})^{\omega}$. In the composition of SCGs, we are interested in paths that advance through the whole concatenated graph. Such a path implies that the final value of the destination variable is of size at most equal to the initial value of the source variable. If the path contains a down-arrow, then the size of the corresponding words decreases strictly.

Following the terminology of [7], a (potentially infinite) sequence of SCGs has a *down-thread* if the associated concatenated graph contains a path spanning every SCG in the sequence and this path includes a down-arrow.

Example 7. Consider the statement $\mathtt{st} \triangleq \mathtt{y := pred(x)}; \mathtt{y := min(x, y)}; \mathtt{x := x+1}; \mathtt{x := X(y \restriction z)}$, whose SCGs are described in Example 6. The concatenated graph obtained from the (unique and finite) sequence of SCGs in $\mathcal{L}(\mathtt{st})$ is provided below. It contains a down-thread (the path from $\mathtt{x}$ to $\mathtt{y}$).

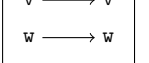
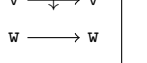
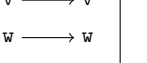


A (potentially infinite) sequence of SCGs is *fan-in free* if the in-degree of nodes is at most 1. By construction, all the considered SCGs are fan-in free.

Safety and Polynomial Size-Change. Unfortunately, programs with down-threads can loop infinitely in the ϵ state. To prevent this, we restrict the analysis to cases where while loops explicitly break out when the decreasing variable reaches ϵ, that is procedures with while loops of the shape $\mathtt{while(x != \varepsilon)\{st\}}$.

For a given set V of variables, we will say that $\mathtt{st}$ *satisfies the simple graph property for* V if for any while loop $\mathtt{while(x != \varepsilon)\{st'\}}$ in $\mathtt{st}$ all sequences of SCGs $G_1^V G_2^V \ldots$ such that $G_1 G_2 \ldots \in \mathcal{L}(\mathtt{st'})$ are fan-in free and contain a down-thread from $\mathtt{x}$ to $\mathtt{x}$. A procedure is in $\mathrm{SCP_S}$ if its statement satisfies the simple graph property for the set of variables in while guards. A program is in $\mathrm{SCP_S}$ if all its procedures are in $\mathrm{SCP_S}$.

Example 8. The program $\mathtt{ce}$ of Example 1 is in $\mathrm{SCP_S}$. The language $\mathcal{L}(\mathtt{body(KS)})$ corresponding to the body of procedure $\mathtt{KS}$ is equal to $G_1.G_2.(G_3.G_4)^{\infty}$, where the SCGs G_i are defined as follows:

G_1	G_2	G_3	G_4
$\mathtt{w} := \mathtt{X}_1(\varepsilon \restriction \varepsilon)$	$\mathtt{z} := \varepsilon$	$\mathtt{v} := \mathtt{pred}(\mathtt{v})$	$\mathtt{z} := \mathtt{X}_2(\mathtt{z} \restriction \mathtt{w})$
			

First, the procedure body satisfies the syntactic restrictions on programs (flattened expressions and restricted while guards). Moreover, the procedure body satisfies the simple graph property for $\{\mathtt{v}\}$ as there is always a down-thread on the path from $\mathtt{v}$ to $\mathtt{v}$ in $(G_3.G_4)^\infty$ and any corresponding sequence is fan-in free. Consequently, the program $\mathtt{ce}$ is in $\mathrm{SCP_S} \cap \mathrm{SAFE}_0$, by Example 5.

$\mathrm{SCP_S}$ preserves completeness on safe programs for $\mathtt{BFF}$.

Theorem 5. $[\![\mathrm{SCP_S} \cap \mathrm{SAFE}_0]\!] = [\![\mathrm{SCP_S} \cap \mathrm{SAFE}]\!] = \mathtt{BFF}$.

While in general deciding if a program satisfies the size-change principle is $\mathtt{PSpace}$-complete, $\mathrm{SCP_S}$ can be checked in quadratic time and, consequently, we obtain the following results.

Theorem 6. *Given a program* $\mathtt{prog}$ *and a safe operator typing environment,*

- *deciding whether* $\mathtt{prog} \in \mathrm{SCP_S} \cap \mathrm{SAFE}$ *is a* $\mathtt{P}$*-complete problem.*
- *deciding whether* $\mathtt{prog} \in \mathrm{SCP_S} \cap \mathrm{SAFE}_0$ *can be done in time* $\mathcal{O}(|\mathtt{prog}|^3)$.

6 Conclusion and future work

We have presented a typing discipline and a termination criterion for a programming language that is sound and complete for the class of second-order polytime computable functionals, $\mathtt{BFF}$. This characterization has three main advantages: 1) it is based on a natural higher-order programming language with imperative procedures; 2) it is pure as it does not rely on an extra semantic requirements (such as taking the lambda closure); 3) belonging to the set $\mathrm{SCP_S} \cap \mathrm{SAFE}$ can be decided in polynomial time. The benefits of tractability is that our method can be automated. However the expressive power of the captured programs is restricted. This drawback is the price to pay for tractability and we claim that the full SCT method, known to be $\mathtt{PSpace}$-complete, could be adapted in a more general way to our programming language in order to capture more programs at the price of a worse complexity. Moreover, any termination criterion based on the absence of infinite data flows with respect to some well-founded order could work and preserve completeness of our characterizations. Another issue of interest is to study whether the presented approach could be extended to characterize $\mathtt{BFF}$ in a purely functional language. We leave these open issues as future work.

Acknowledgements. The authors would like to thank the anonymous reviewers for their suggestions and comments. Bruce M. Kapron's work was supported in part by NSERC RGPIN-2021-02481.

References

1. Avery, J.: Size-change termination and bound analysis. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. Lecture Notes in Computer Science, vol. 3945, pp. 192–207. Springer (2006). https://doi.org/10.1007/11737414_14
2. Baillot, P., Dal Lago, U.: Higher-order interpretations and program complexity. Inf. Comput. **248**, 56–81 (2016). https://doi.org/10.1016/j.ic.2015.12.008
3. Baillot, P., Mazza, D.: Linear logic by levels and bounded time complexity. Theor. Comput. Sci. **411**(2), 470–503 (2010). https://doi.org/10.1016/j.tcs.2009.09.015
4. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: Logic in Computer Science, LICS 2004. pp. 266–275. IEEE (2004). https://doi.org/10.1109/LICS.2004.1319621
5. Beame, P., Cook, S.A., Edmonds, J., Impagliazzo, R., Pitassi, T.: The relative complexity of NP search problems. J. Comput. Syst. Sci. **57**(1), 3–19 (1998). https://doi.org/10.1006/jcss.1998.1575
6. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2**, 97–110 (1992). https://doi.org/10.1007/BF01201998
7. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. ACM Trans. Program. Lang. Syst. **29**(1), 5:1–5:37 (2007). https://doi.org/10.1145/1180475.1180480
8. Bonfante, G., Marion, J., Moyen, J.: Quasi-interpretations a way to control resources. Theor. Comput. Sci. **412**(25), 2776–2796 (2011). https://doi.org/10.1016/j.tcs.2011.02.007
9. Cobham, A.: The intrinsic computational difficulty of functions. In: Bar-Hillel, Y. (ed.) Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, pp. 24–30. North-Holland, Amsterdam (1965)
10. Constable, R.L.: Type two computational complexity. In: Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA. pp. 108–121. ACM (1973). https://doi.org/10.1145/800125.804041
11. Cook, S.A.: Computability and complexity of higher type functions. In: Logic from Computer Science. pp. 51–72. Springer (1992). https://doi.org/10.1007/978-1-4612-2822-6_3
12. Cook, S.A., Kapron, B.M.: Characterizations of the basic feasible functionals of finite type. In: 30th Annual Symposium on Foundations of Computer Science (FOCS 1989). pp. 154–159. IEEE (1989). https://doi.org/10.1109/SFCS.1989.63471
13. Férée, H., Hainry, E., Hoyrup, M., Péchoux, R.: Characterizing polynomial time complexity of stream programs using interpretations. Theor. Comput. Sci. **585**, 41–54 (2015). https://doi.org/10.1016/j.tcs.2015.03.008
14. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7364, pp. 286–300. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_23
15. Girard, J.Y.: Light linear logic. Inf. Comput. **143**(2), 175–204 (1998). https://doi.org/10.1006/inco.1998.2700
16. Hainry, E., Kapron, B.M., Marion, J., Péchoux, R.: A tier-based typed programming language characterizing feasible functionals. In: LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020. pp. 535–549 (2020). https://doi.org/10.1145/3373718.3394768

17. Hainry, E., Péchoux, R.: Theory of higher order interpretations and application to basic feasible functions. Log. Methods Comput. Sci. **16**(4) (2020), https://lmcs.episciences.org/6973
18. Irwin, R.J., Royer, J.S., Kapron, B.M.: On characterizations of the basic feasible functionals (part I). J. Funct. Program. **11**(1), 117–153 (2001). https://doi.org/10.1017/S0956796800003841
19. Kapron, B.M., Cook, S.A.: 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. pp. 342–347. IEEE (1991). https://doi.org/10.1109/SFCS.1991.185389
20. Kapron, B.M., Steinberg, F.: Type-two polynomial-time and restricted lookahead. In: Logic in Computer Science, LICS 2018. pp. 579–588. ACM (2018). https://doi.org/10.1145/3209108.3209124
21. Kawamura, A., Cook, S.A.: Complexity theory for operators in analysis. ACM Trans. Comput. Theory **4**(2), 5:1–5:24 (2012). https://doi.org/10.1145/2189778.2189780
22. Kawamura, A., Steinberg, F.: Polynomial running times for polynomial-time oracle machines. In: 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017. pp. 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.23
23. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Hankin, C., Schmidt, D. (eds.) POPL 2001. pp. 81–92. ACM (2001). https://doi.org/10.1145/360204.360210
24. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Fundam. Inform. **19**(1/2), 167–184 (1993)
25. Mairson, H.G.: Linear lambda calculus and ptime-completeness. J. Funct. Program. **14**(6), 623–633 (2004). https://doi.org/10.1017/S0956796804005131
26. Marion, J.: A type system for complexity flow analysis. In: Logic in Computer Science, LICS 2011. pp. 123–132. IEEE Computer Society (2011). https://doi.org/10.1109/LICS.2011.41
27. Mehlhorn, K.: Polynomial and abstract subrecursive classes. J. Comp. Sys. Sci. **12**(2), 147–178 (1976). https://doi.org/10.1016/S0022-0000(76)80035-9
28. Nivat, M., Perrin, D.: Automata on infinite words, vol. 192. Springer Science & Business Media (1985)
29. Townsend, M.: Complexity for type-2 relations. Notre Dame J. Formal Log. **31**(2), 241–262 (1990). https://doi.org/10.1305/ndjfl/1093635419
30. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of computer security **4**(2-3), 167–187 (1996). https://doi.org/10.3233/JCS-1996-42-304