Declassification Policy for Program Complexity Analysis

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

Declassification Policy for Program Complexity Analysis

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

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

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

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

ABSTRACT

In automated complexity analysis, noninterference-based type systems statically guarantee, via *soundness*, the property that well-typed programs compute functions of a given complexity class, e.g., the class FP of functions computable in polynomial time. These characterizations are also extensionally *complete* – they capture all functions – but are not intensionally complete as some polytime algorithms are rejected. This impact on expressive power is an unavoidable cost of achieving a tractable characterization. To circumvent this issue, an avenue arising from security applications is to find a relaxation of noninterference based on a *declassification* mechanism that allows critical data to be released in a safe and controlled manner. Following this path, we present a new and intuitive declassification policy preserving FP-soundness and capturing strictly more programs than existing noninterference-based systems. We show the versatility of the approach: it also provides a new characterization of the class BFF of second-order polynomial time computable functions in a second-order imperative language, with first-order procedure calls. Type inference is tractable: it can be done in polynomial time.

CCS CONCEPTS

• **Theory of computation → Complexity theory and logic**; **Type theory**.

KEYWORDS

Complexity analysis, Noninterference, Declassification, Polynomial time, Basic Feasible Functionals

ACM Reference Format:
Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. 2024. Declassification Policy for Program Complexity Analysis. In *39th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '24), July 8–11, 2024, Tallinn, Estonia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3661814.3662100

LICS '24, July 8–11, 2024, Tallinn, Estonia

ACM ISBN 979-8-4007-0660-8/24/07
https://doi.org/10.1145/3661814.3662100

1 INTRODUCTION

1.1 Motivations

Noninterference-based type disciplines have been introduced in static analysis to ensure security properties of programs [Goguen and Meseguer 1982, 1984; Volpano et al. 1996]. They have also been applied to study the complexity properties of programs (e.g., bounded runtime, bounded memory), noting that a stratification between iterable and not iterable data is key to preventing exponential behavior [Marion 2011]. The main intuition is as follows: data on which it is safe to iterate cannot increase and are annotated with level 1 and data that can increase with level 0. A variable of level 0 is not safe for iteration, i.e., it cannot be used to control loops. Data can flow from 1 to 0, under the hypothesis that it is passed-by-value. Flows in the opposite direction are strictly prohibited. Under a termination hypothesis, safety implies polynomial time soundness by ensuring that the space of level 1 data is polynomially bounded in the size of the program input.

This approach depends on the cornerstone works of [Bellantoni and Cook 1992; Leivant 1991; Leivant and Marion 1993] that provided the first implicit characterizations of (first-order) polynomial time FP. These methodologies are known as *safe recursion*, *ramified recursion*, *cons-free programs*, or *tiering* and were adapted not only to characterize polytime [Bhaskar et al. 2023; de Carvalho and Simonsen 2014; Jones 2001; Marion 2011] but also other well-known complexity classes, such as (first-order) polynomial space FPSPACE [Hainry et al. 2013; Jones 2001], second-order polynomial time BFF [Hainry et al. 2020], or probabilistic polynomial time PP [Dal Lago et al. 2021]. The main advantage of the underlying type systems is the tractability of their type inference, which can be done in polynomial time. Consequently, it has led to practice-oriented extensions such as the development of type systems for Java programs to statically ensure that their heap and their stack have a polynomially bounded size or that their runtime is polynomially bounded [Hainry and Péchoux 2018, 2023]. The software ComplexityParser introduced in [Hainry et al. 2021] ensures this kind of property.

This main advantage also implies their main weakness: while these characterizations are *sound* and *extensionally complete*, i.e., all functions of the target complexity class are captured, they are not *intensionally complete*. This means that there exist false negatives, e.g., algorithms that run in polynomial time but are wrongly rejected. This incompleteness is no great surprise for the theoretician who knows, for example, that the set of programs running in

```
//x,y,z: unary
y⁰ := 1⁰;
while (x¹>0){
  z¹ := declass (y⁰)¹;
  while (z¹>0){
      y⁰ := y+1⁰;
      z¹ := z-1¹
  };
  x¹ := x-1¹
}
```

Listing 1: exponential 1

```
//y: binary, x: bool
x¹ := true¹;
while (x¹){
  y⁰ := y-1⁰;
  x¹ := declass (y⁰>0)¹
}
```

Listing 2: exponential 2

level 0: increasing/noniterable

level 1: nonincreasing/iterable

polynomial time is not decidable (and in fact has been proven to be Σ_2^0-complete in [Hájek 1979]) and that a tractable characterization of FP must therefore reject some of these polytime programs.

Since then, a number of studies have sought to increase the expressive power of these characterizations. For example, [Hainry and Péchoux 2023] extended levels to any positive integer and has program flows to pass-by-reference data by combining noninterference-based typing with shape-analysis. Despite the extensions made to date, the restrictions encountered with the complexity framework can be too rigid, similar to the problems encountered with traditional information-flow security and privacy applications of noninterference [Backes and Pfitzmann 2003; Volpano et al. 1996]. The security community has tried to solve this issue by introducing relaxed versions, as in [Myers 1999; Volpano and Smith 2000] which, in order to provide more expressive power allow some forms of information flow that violate the noninterference principle, by making it possible to release some information in a controlled way. Such control takes the form of *declassification* or downgrading policies allowing for instance the publishing of some part of the secret data. A classic example of declassification is hashing: a password is strictly private but its hash must be shared for password verification to work. Allowing intentional information release while avoiding attack vectors is a challenge that has been studied under various frameworks. In this respect, [Sabelfeld and Sands 2009] presents a list of principles that declassification should preserve and a taxonomy of those frameworks on *what* can be declassified, *who* may receive the declassified data, *where* declassification is allowed, and *when*, in the sense of how often, or under which complexity constraints.

A natural question, then, is whether similar declassification policies can be defined and implemented in the context of complexity analysis. At present, the only declassification options under consideration are far too limited. [Hainry and Péchoux 2018] considered declassification outside loops. Hence only a constant number of declassifications can be performed at runtime. In [Marion 2011], declassification is *a priori* restricted to a given type level, *à la* [Myers and Liskov 2000]. In general, declassifying any predetermined finite number of times will be acceptable as it preserves polynomial soundness; but declassifying data in a loop can lead to exponential program behavior. Indeed, declassifying one single bit of data is not a big deal but repeating this process breaks polytime soundness.

To illustrate the danger linked with declassification when trying to control polynomial complexity, let us introduce two code samples in Listings 1 and 2, where the syntactic elements are annotated by their level as a superscript. Declassification is used as a primitive construct to raise data from 0 to 1. The program of Listing 1 typechecks with respect to a control-flow based policy with 2 levels but computes the exponential $y = 2^x$ in unary, hence in exponential time. This illustrates why the addition of declassification should not be treated superficially at the risk of breaking polytime soundness. Naively, we might think that it is enough to limit information leaks to a restricted number of bits to avoid such examples. This hope is dashed by the example of Listing 2, one single loop manipulating a binary number y where the declassification only leaks a boolean value: this loop explores all integers smaller than y, hence is exponential in the size of y.

There is therefore a need not only to develop declassification techniques that preserve the soundness of noninterference policies for complexity analysis, but also to demonstrate that they can be used to drastically increase the expressive power of these methods.

1.2 Contributions

We introduce a new declassification policy for complexity analysis. This policy preserves polynomial time soundness and answers a question already posed in [Marion 2011] for having a more general form of declassification. This policy consists of two ingredients; a `declass` operator and an *aperiodicity* condition. Similarly to [Myers 1999], `declass` takes 2 arguments: the expression to declassify and an expression whose level will bound the level up to which the value will be declassified. In effect, the declassified value loses some information as the output is a unary representation of the length of the first argument and is bounded in size by the length of the second argument. The aperiodicity condition checks that loops do not encounter a configuration more than once, hence do not enter a periodic state. This is necessary to prevent abuses such as in Listing 2 above. We can say that the declassification policy acts on the *where* dimension in the taxonomy of [Sabelfeld and Sands 2009]. We illustrate the use of this declassifying policy and its tamed power in two programming languages: i) a simple imperative language with a while loop construct, ii) a programming language with second-order imperative procedures. The declassification policy is expressed through a type system, which defines a *safety* property. A safe program is a program that can be typed. Additional properties are required to ensure polynomial time soundness: *aperiodicity* and, in the second-order case, *termination*. A program is aperiodic if in the guard of each while loop, no two equivalent states can be reached in (the depth of) the derivation tree; a program terminates if it halts on all inputs. Aperiodicity implies termination on safe first-order programs and this explains why termination is only required for second-order programs.

The main contributions of this paper are:

- a new declassification policy (Figure 3), called safety, for program complexity analysis, that corresponds to a partial release of information: only the length of data can be declassified. This policy extends the expressive power of previous work [Marion 2011] by allowing to consider programs with restricted interference.

- a new characterization of the class of polytime computable functions FP that captures strictly more programs (like Example 2.6) than the ones implementing a strict noninterference policy. Functions in FP are exactly those computed by programs that are both safe and aperiodic (soundness: Theorem 2.11, completeness: Theorem 2.10).
- a proof that safety is *tractable*: it can be decided in polynomial time (Theorem 2.13).
- a proof that aperiodicity is Π_1^0-complete (Theorem 2.14). Hence, aperiodicity is not more difficult than termination, known to be Π_2^0-complete [Endrullis et al. 2011], used as a condition in previous work [Marion 2011]. Moreover, we show in Theorem 2.15 that a decidable and completeness-preserving criterion trivially ensuring aperiodicity can be designed at the price of reduced expressive power. Consequently, our methodology is automatable.
- an application of the declassification policy: a new characterization of the class of second-order polytime computable functions BFF implementing the *Strongly PolyTime* (SPT) criterion of [Kapron and Steinberg 2018]. In this setting, safety remains tractable (Theorem 3.11) and aperiodicity is Π_1^0-hard (Corollary 3.4).

1.3 Application of the Declassification Policy

One of the latest characterizations of BFF, introduced in [Kapron and Steinberg 2018], cannot be obtained without declassification. SPT is defined as the class of second-order functionals computable by an oracle Turing machine with i) *Polynomial Step Count*: the machine works in time polynomial in the size of the input and the maximum size of an oracle answer, ii) *Finite Length Revision*: there exists a natural number n s.t. for any oracle and any input, in the run of the machine, it happens at most n times that an oracle answer has size that exceeds the size of all previous answers. The class of functionals with polynomial step counts is named OPT (for *Oracle Polynomial Time*, see [Cook 1992]), and the class of functionals with finite length revision is named FLR. By definition, SPT = OPT∩FLR. Let $\lambda(X)_2$ be the restriction to second-order functions of the simply-typed lambda-closure of terms with constants in the set X. The class BFF can be recovered from SPT as follows:

Theorem 1.1 ([Kapron and Steinberg 2018]). $\lambda(\mathsf{SPT})_2 = \mathsf{BFF}$.

As the above characterization relies on a purely semantic criterion (FLR) and only deals with machines, it was an open issue to know to what extent a similar and tractable characterization of BFF could be obtained for a programming language using noninterference-based analysis. However this issue cannot be resolved in a strictly noninterfering system for the simple reason that SPT needs interferences. Indeed, in the definition of FLR the control flow needs to be guarded by the oracle answers (or at least a counter that keeps information on the number n of times an answer size has exceeded the size of all previous answer). Hence the loop can be controlled by increasing data (oracle answers are not known in advance and cannot be trusted), which breaks noninterference. In other words, since in SPT, it is the number of increases for the outputs of oracles that needs to be observed, those outputs need to be declassified in order to manage the control flow. Therefore, an application of interest is whether declassification techniques could help in capturing the SPT characterization. We give a positive answer to this question in the paper.

Expr	e	:=	x | op($\bar{e}$) | declass(e, e)
Stmt	s	:=	skip | x := e | s; s | if(e){s} else{s} | while(e) {s} | break(e)
Prg	P	:=	prog($\bar{x}$){s return x}

Figure 1: Syntax

1.4 Related Work

Our results build upon work on declassification, noninterference type-based approaches in implicit computational complexity, and BFF characterizations. The type system we define has roots in [Volpano et al. 1996] and the declassification policy was inspired from works on downgrading, endorsement, and declassification [Li and Zdancewic 2005; Sabelfeld and Sands 2009; Volpano and Smith 2000; Zdancewic and Myers 2001]. In contrast, our aim is to characterize tractable programs rather than prove security or confidentiality properties, which accounts for the additional restrictions of aperiodicity and termination. On the other side, our framework provides the possibility of declassifying in a loop which is challenging when only partial release of data is allowed. The implicit complexity line of work draws from noninterference ideas applied to data ramification/safe recursion [Bellantoni and Cook 1992; Leivant and Marion 1993] to characterize complexity classes such as FP or FPSPACE, e.g., [Jones 2001; Marion 2011].

In contrast to [Cook 1992; Cook and Urquhart 1993; Kapron and Cook 1996; Kapron and Steinberg 2018], that study BFF on machines, the current work characterizes BFF in the setting of programming languages. Hence it follows the approach of [Danner and Royer 2006; Irwin et al. 2001] but differs from the latter in the sense that the bounds do not need to be explicitly provided in the language. This work is also linked to higher-order complexity analysis based on costs and potentials of [Danner et al. 2013], although the theoretical and practical objectives differ. [Hainry et al. 2020, 2022] provide characterizations of BFF based on another noninterfering criterion (called MPT in [Kapron and Steinberg 2018]) and, hence, do not capture algorithmic schemes based on SPT. Several work [Baillot et al. 2024; Hainry and Péchoux 2020] have designed alternative elegant characterizations of BFF using higher-order polynomial interpretations. However, contrarily to safety, these interpretation methods are not tractable.

2 DECLASSIFYING POLICY FOR COMPLEXITY

In this section, we show how declassification for complexity analysis can be adapted to a simple imperative programming language.

2.1 Imperative Language with Declassification

2.1.1 Syntax. The syntax of the considered imperative language is provided in Figure 1, where x is a variable in a fixed set of variables $\mathbb{V}$ and where $\bar{e}$ (resp: $\bar{x}$) represents a finite sequence of expressions (resp: variables). Let $\ell(\bar{x})$ be the length of the sequence $\bar{x}$.

Operators op are basic operations of fixed arity $ar(\mathsf{op})$ in the set $\mathbb{O}$. We assume that $\mathbb{O}$ includes some basic boolean and arithmetic

$$\frac{}{(\mu, \mathsf{x}) \to_{\mathsf{ex}} \mu(\mathsf{x})} \qquad \frac{(\mu, \overline{\mathsf{e}}) \to_{\mathsf{ex}} \overline{w}}{(\mu, \mathsf{op}(\overline{\mathsf{e}})) \to_{\mathsf{ex}} [\![\mathsf{op}]\!](\overline{w})} \qquad \frac{(\mu, \mathsf{e}_1) \to_{\mathsf{ex}} w_1 \quad (\mu, \mathsf{e}_2) \to_{\mathsf{ex}} w_2}{(\mu, \mathtt{declass}(\mathsf{e}_1, \mathsf{e}_2)) \to_{\mathsf{ex}} 1^{\min(|w_1|, |w_2|)}}$$

$$\frac{}{(\mu, \mathtt{skip}) \to_{\mathsf{st}} (\top, \mu)} \qquad \frac{(\mu, \mathsf{e}) \to_{\mathsf{ex}} w}{(\mu, \mathsf{x} := \mathsf{e}) \to_{\mathsf{st}} (\top, \mu[\mathsf{x} \leftarrow w])} \qquad \frac{(\mu, \mathsf{s}_1) \to_{\mathsf{st}} (\top, \mu') \quad (\mu', \mathsf{s}_2) \to_{\mathsf{st}} (\square, \mu'')}{(\mu, \mathsf{s}_1; \mathsf{s}_2) \to_{\mathsf{st}} (\square, \mu'')} \ (\square \in \{\top, \bot\})$$

$$\frac{(\mu, \mathsf{s}_1) \to_{\mathsf{st}} (\bot, \mu')}{(\mu, \mathsf{s}_1; \mathsf{s}_2) \to_{\mathsf{st}} (\bot, \mu')} \qquad \frac{(\mu, \mathsf{e}) \to_{\mathsf{ex}} w \quad (\mu, \mathsf{s}_w) \to_{\mathsf{st}} (\square, \mu')}{\mathtt{if}(\mathsf{e})\{\mathsf{s}_1\}\mathtt{else}\{\mathsf{s}_{\underline{0}}\} \to_{\mathsf{st}} (\square, \mu')} \ (\square \in \{\top, \bot\}, w \in \{\underline{0}, 1\}) \qquad \frac{(\mu, \mathsf{e}) \to_{\mathsf{ex}} \underline{0}}{(\mu, \mathtt{while}(\mathsf{e})\ \{\mathsf{s}\}) \to_{\mathsf{st}} (\top, \mu)}$$

$$\frac{(\mu, \mathsf{e}) \to_{\mathsf{ex}} 1 \quad (\mu, \mathsf{s}; \mathtt{while}(\mathsf{e})\ \{\mathsf{s}\}) \to_{\mathsf{st}} (\square, \mu')}{(\mu, \mathtt{while}(\mathsf{e})\ \{\mathsf{s}\}) \to_{\mathsf{st}} (\top, \mu')} \ (\square \in \{\top, \bot\}) \qquad \frac{(\mu, \mathsf{e}) \to_{\mathsf{ex}} \underline{0}}{(\mu, \mathtt{break}(\mathsf{e})) \to_{\mathsf{st}} (\top, \mu)} \qquad \frac{(\mu, \mathsf{e}) \to_{\mathsf{ex}} 1}{(\mu, \mathtt{break}(\mathsf{e})) \to_{\mathsf{st}} (\bot, \mu)}$$

Figure 2: Semantics

operators such as $\{=, <, \leq, 0, +1, -1, \mathsf{not}, \mathsf{and}, \mathsf{or}\}$. Any constant can be viewed as an operator of arity 0.

An *expression* e can be a variable x, an operator application $\mathsf{op}(\bar{\mathsf{e}})$ or the result of a declassification $\mathtt{declass}(\mathsf{e}_1, \mathsf{e}_2)$. Operators will be used in infix, postfix, or prefix notations and they will always be fully applied, i.e., in $\mathsf{op}(\bar{\mathsf{e}})$, it always holds that $\ell(\bar{\mathsf{e}}) = ar(\mathsf{op})$. We will say that an expression is *declassified*, if it appears as a subexpression of some e_1 in $\mathtt{declass}(\mathsf{e}_1, \mathsf{e}_2)$.

Statements in *Stmt* are standard imperative constructs with assignments, conditionals, loops, and breaks.

Finally, a *program* takes a list of parameter variables $\bar{\mathsf{x}}$, executes a (body) statement $\mathsf{body}(\mathsf{P}) \triangleq \mathsf{s}$, and returns one output variable x.

Example 2.1. In Listing 3, we have implemented the bubble sort of a string with some type annotations as superscript. Bubble uses the following arity-2 operators: + to append a character to a string, < to compare two characters, hd and tl to get respectively the first character and the rest of the string. An explicit call to declass is performed on line 14 inside a loop and we will see shortly (see Example 2.6) that the program cannot type without this feature. Such a program cannot be typed through the discipline of [Hainry and Péchoux 2018; Marion 2011].

2.1.2 Semantics. Let $\mathbb{W} \triangleq \Sigma^*$ be the set of words, also called *values*, over a fixed and finite alphabet Σ such that $\{0, 1\} \subseteq \Sigma$. ϵ denotes the empty word; $v \cdot w$ denotes the concatenation of words v and w; for $n \in \mathbb{N}$, v^n is defined inductively as $v^0 \triangleq \epsilon$ and $v^{n+1} \triangleq v \cdot v^n$. Let $\trianglelefteq$ be the sub-word relation defined by $v \trianglelefteq w$ if there exist words u and u' such that $w = u \cdot v \cdot u'$. The size of word w is denoted by $|w|$.

The semantics assigns a total function $[\![op]\!] : \mathbb{W}^{ar(\mathsf{op})} \to \mathbb{W}$ to each operator $\mathsf{op} \in \mathbb{O}$. For example, +1 and −1 are operators on unary numbers, computing increment $[\![+1]\!] : \mathbb{W} \to \mathbb{W}$ and decrement $[\![-1]\!] : \mathbb{W} \to \mathbb{W}$, respectively, and outputting ϵ in case of failure (if the input word is not a unary number). I.e., for $n \in \mathbb{N}$, $[\![+1]\!](1^n) \triangleq 1^{n+1}$ and $[\![+1]\!](w) \triangleq \epsilon$, if $w \neq 1^n$. The output of Boolean operators is either false (0) or true (1). By extension, any value that is not 1, including ϵ, will be denoted by $\underline{0}$ and considered as false. The semantics assigned to comparison operators is the standard shortlex order.

A *memory store* $\mu \in \mathtt{Stores}$ is a total map from variables in $\mathbb{V}$ to values in $\mathbb{W}$. Let $dom(\mu)$ be the domain of the store μ. Given $\mathsf{x} \in \mathbb{V}$, and a value $w \in \mathbb{W}$, let $\mu[\mathsf{x} \leftarrow w]$ denote the store obtained from μ by updating the value of x to w. This notation is extended naturally to sequences $\mu[\bar{\mathsf{x}} \leftarrow \bar{w}]$, whenever $\ell(\bar{\mathsf{x}}) = \ell(\bar{w})$. Finally, let $\mu_{\emptyset}$ denote the empty store, mapping each variable to ϵ.

```
1  bubble(list¹){
2    list1¹ := list¹;
3    len⁰ := 0⁰;
4    while (list1¹ ≠ ϵ){
5      list1¹ := tl(list1)¹;
6      len⁰ := (len+1)⁰
7    };
8    list2⁰ := list¹;
9    len1¹ := len⁰;
10   while (len1¹ > 0){
11     r⁰ := ε⁰;
12     x⁰ := hd(list2)⁰;
13     list2⁰ := tl(list2)⁰;
14     len2¹ := declass (len⁰,list¹);
15     while (len2¹>1){
16       y⁰ := hd(list2)⁰;
17       list2⁰ := tl(list2)⁰;
18       if (x < y)⁰{
19         r⁰ := (r+x)⁰;
20         x⁰ := y⁰
21       }else{
22         r⁰ := (r+y)⁰
23       };
24       len2⁰ := (len2-1)⁰
25     }
26     r⁰ := (r+x)⁰;
27     list2⁰ := r⁰;
28     len1¹ := (len1-1)¹
29   }
30   return r⁰
31 }
```

Listing 3: Bubble sort

The operational semantics is described in Figure 2 as a standard big-step semantics with control flow breaks. The semantics of expressions and statements are defined by the maps

$$\to_{\mathsf{ex}} : (\mathtt{Stores} \times \mathtt{Expr}) \to \mathbb{W}$$
$$\to_{\mathsf{st}} : (\mathtt{Stores} \times \mathtt{Stmt}) \to (\{\top, \bot\} \times \mathtt{Stores}).$$

$$\frac{\Gamma(\mathsf{x}) = \tau}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{x} : \tau}\ (\text{VAR}) \qquad \frac{\tau_1 \to \cdots \to \tau_{ar(\mathsf{op})+1} \in \Delta(\mathsf{op})(\tau_{in}, \tau_{out}) \qquad \forall i \le ar(\mathsf{op}),\ \Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{e}_i : \tau_i}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{op}(\bar{\mathsf{e}}) : \tau_{ar(\mathsf{op})+1}}\ (\text{OP})$$

$$\frac{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{e}_1 : \tau_1 \qquad \Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{e}_2 : \tau_{out} \qquad \tau_1 \le \tau \le \tau_{out}}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{declass}(\mathsf{e}_1, \mathsf{e}_2) : \tau}\ (\text{DCL}) \qquad \frac{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s} : \tau_1 \qquad \tau_1 \le \tau_2}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s} : \tau_2}\ (\text{SUB}) \qquad \frac{}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{skip} : 0}\ (\text{SKP})$$

$$\frac{\Gamma(\mathsf{x}) = \tau_1 \qquad \Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{e} : \tau_2 \qquad (\tau_{out} = 0) \vee (\tau_1 \le \tau_2)}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{x} := \mathsf{e} : \tau_1}\ (\text{ASG}) \qquad \frac{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s}_1 : \tau \qquad \Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s}_2 : \tau}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s}_1; \mathsf{s}_2 : \tau}\ (\text{SEQ})$$

$$\frac{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{e} : \tau \qquad \Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s}_1 : \tau \qquad \Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{s}_0 : \tau}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{if}(\mathsf{e})\{\mathsf{s}_1\}\,\mathtt{else}\{\mathsf{s}_0\} : \tau}\ (\text{CND}) \qquad \frac{\Gamma, \Delta \vdash^{\tau}_{\tau_{out}} \mathsf{e} : \tau \qquad \Gamma, \Delta \vdash^{\tau}_{\tau_{out}} \mathsf{s} : \tau \qquad 1 \le \tau \le \tau_{out}}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{while}(\mathsf{e})\{\mathsf{s}\} : \tau}\ (\text{WH})$$

$$\frac{\Gamma, \Delta \vdash^{\tau}_{\tau} \mathsf{e} : \tau \qquad \Gamma, \Delta \vdash^{\tau}_{\tau} \mathsf{s} : \tau \qquad 1 \le \tau}{\Gamma, \Delta \vdash^{0}_{0} \mathtt{while}(\mathsf{e})\{\mathsf{s}\} : \tau}\ (\text{WI}) \qquad \frac{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathsf{e} : \tau \qquad \tau_{in} \le \tau}{\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{break}(\mathsf{e}) : \tau_{in}}\ (\text{BRK})$$

Figure 3: Level-based typing rules

Evaluating a $\mathtt{declass}(\mathsf{e}_1, \mathsf{e}_2)$ yields the value $1^{\min(|w_1|, |w_2|)}$, provided that e_i evaluates to the value w_i, for $i \in \{1, 2\}$. Though this evaluation treats both operands symmetrically, as we will see shortly, the first operand e_1 is the one to declassify and the second operand e_2 bounds the length that is transmitted. When evaluating a guard expression e, assumed to evaluate to a boolean, we write $(\mu, \mathsf{e}) \to_{\mathsf{ex}} \underline{0}$ to denote that $(\mu, \mathsf{e}) \to_{\mathsf{ex}} w$ with $w \neq 1$.

The first component in the codomain of $\to_{\mathtt{st}}$ is a flag encoding whether the flow has to be broken ($\bot$) or not ($\top$). In rules describing the semantics of a sequence $\mathsf{s}_1; \mathsf{s}_2$, it is implicitly assumed that $\mathsf{s}_1 \neq \mathsf{s}_1'; \mathsf{s}_1''$.

For a *configuration* $c \in (\mathtt{Stores} \times \mathtt{Stmt}) \cup (\mathtt{Stores} \times \mathtt{Expr})$, such that there exists $x \in (\{\top, \bot\} \times \mathtt{Stores})$ with $c \to_{\mathtt{st}} x$, let π_c be the *(evaluation) tree* with root $c \to_{\mathtt{st}} x$ obtained by evaluating c using the rules of Figure 2. For conciseness, we will call c the *root* of the tree. π_c can be infinite if the program does not terminate. We write $\pi_c \sqsubseteq \pi_{c'}$ (resp. $\pi_c \sqsubset \pi_{c'}$) when π_c is a (strict) subtree of $\pi_{c'}$.

2.1.3 Restriction on Operators. Following [Marion 2011], we define three classes of operators called neutral, positive, and polynomial depending on the total function they compute. This classification of operators will be used by the type system as the admissible types for operators will depend on their category.

Definition 2.2. An operator $\mathsf{op} \in \mathbb{O}$ of arity k is:
– *neutral* if: either $[\![\mathsf{op}]\!] \in \mathbb{W}^k \to \{0, 1\}$,
or $\exists i \le k,\ \forall \bar{w} \in \mathbb{W}^k,\ [\![\mathsf{op}]\!](\bar{w}) \trianglelefteq w_i$;
– *positive* if $\exists c_{\mathsf{op}} \in \mathbb{N},\ \forall \bar{w} \in \mathbb{W}^k,\ |[\![\mathsf{op}]\!](\bar{w})| \le \max_i |w_i| + c_{\mathsf{op}}$;
– *polynomial* if $\exists P \in \mathbb{N}[X],\ \forall \bar{w} \in \mathbb{W}^k,\ |[\![\mathsf{op}]\!](\bar{w})| \le P(\max_i |w_i|)$.

The basic intuition behind this classification is that neutral operators are iterable; positive operators are polynomially iterable; and polynomial operators are not iterable. Each operator that does not fall in these categories will be rejected by the type system. A neutral operator is always positive and a positive operator is always polynomial but the converse properties are not true. In the sequel, we reserve the name positive (resp. polynomial) for those operators that are positive but not neutral (resp. polynomial but not positive).

Example 2.3. In Listing 3, operators $\neq$, $>$, $<$ compute boolean predicates, hence they are *neutral*. hd, tl, and -1 (unary) are subword operations, as such, they are neutral. $+1$ adds one character, it is hence a positive operator (with c_{+1} equal to 1). The $+$ operator (line 26) takes a word $w \in \mathbb{W}$ and a character $a \in \Sigma$ and outputs the word $w \cdot a$, it is hence a positive operator as $|w \cdot a| = |w| + 1$.

2.2 Type System and Safe Programs

We now describe the type discipline implementing the noninterference policy with declassification.

2.2.1 Levels and Typing Environments. *Levels* are types that belong to the set of non-negative integers $\mathbb{N}$. Let $\le$ (resp. $<$) be the standard (strict) order on $\mathbb{N}$ and let $\sqcup$ and $\sqcap$ denote the max and min of a set of integers. Variables $\tau, \tau', \tau_1, \ldots$ will be used to denote levels. Variables τ_{in} and τ_{out} will be used to denote the innermost and outermost levels. The innermost (resp. outermost) level is the level of the innermost (resp. outermost) while loop under consideration. Outside of loops, τ_{in} and τ_{out} are set to 0. In essence, the outermost level τ_{out} will be used to restrict declassification, whereas the innermost level τ_{in} will restrict the type of the admissible operators in the corresponding statement.

An operator op will be associated to *functional levels* of the shape $\tau_1 \to \ldots \to \tau_{ar(\mathsf{op})+1}$. They can be viewed as a tuple in $\mathbb{N}^{ar(\mathsf{op})+1}$.

The type system uses two kinds of typing environments:

- a *variable typing environment* Γ is a finite map in $\mathbb{V} \to \mathbb{N}$;
- an *operator typing environment* Δ maps each operator $\mathsf{op} \in \mathbb{O}$ to a finite map $\Delta(\mathsf{op}) \in \mathbb{N}^2 \to \mathcal{P}(\mathbb{N}^{ar(\mathsf{op})+1})$.

Intuitively, given an operator op applied in a context of innermost level $\tau_{in} \in \mathbb{N}$ and an outermost level $\tau_{out} \in \mathbb{N}$, $\Delta(\mathsf{op})(\tau_{in}, \tau_{out})$ is the set of functional levels of the shape $\tau_1 \to \ldots \to \tau_{ar(\mathsf{op})+1}$ that are admissible in the caller context. See Example 2.6 for an illustrating use of typing environments. Given a typing environment Γ (resp. Δ), $dom(\Gamma)$ (resp. $dom(\Delta)$) denotes the domain of Γ (resp. Δ).

2.2.2 Typing Judgments and Typing Rules. Expression/Statement typing judgments are of the shape $\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} b : \tau$, with $b \in \mathtt{Expr} \cup$

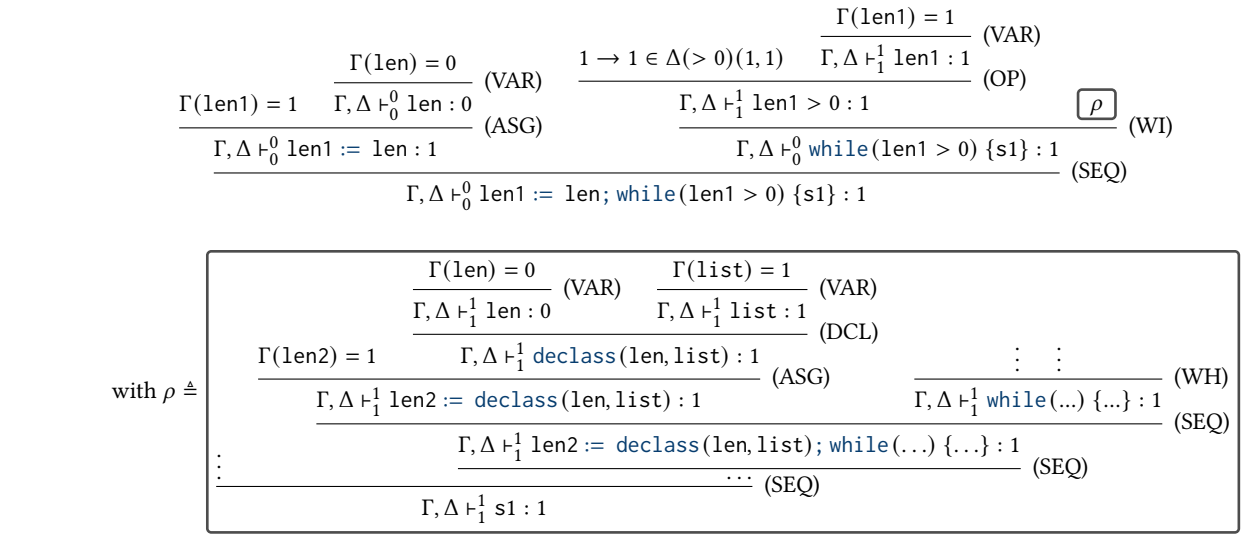

Figure 4: Typing derivation of program bubble (Listing 3)

`Stmt`. The meaning is that under the *innermost level* τ_{in} and the *outermost level* τ_{out}, the level of b is τ.

The typing rules are detailed in Figure 3. Rule (OP) restricts the functional levels of the operator depending on the current innermost and outermost levels. Rule (DCL) gives the possibility of declassifying the value of an expression of level τ_1 to any level less or equal to τ_{out}. As we have seen in the semantics, the value e_1 is not declassified as is, only its length (in unary) is transmitted and the τ_{out} compatibility is ensured by bounding this value by the second argument e_2. Those constraints will ensure that the set of possible values for variables of level τ_{out} will not be exponential in their size. In effect, it prevents the first dangerous code sample of Listing 1 in the introduction. The other rules enforce the noninterference discipline, preventing data to flow from a lower level to a strictly higher level inside loops. Notice that it is possible to subtype statements, that is consider them to be of higher level. On the other hand, inside a loop (i.e., when $\tau_{out} \neq 0$), it is possible to put values of higher level into variables of lower level in Rule (ASG). This is the reason why we do not explicitly add subtyping for expressions. Notice that there are no constraints on flows outside loops (when $\tau_{out} = 0$) as there are no complexity concerns. Rule (WI) sets the innermost and outermost levels τ_{in} and τ_{out} of an outermost while loop, rule (WH) propagates the outermost level τ_{out} and sets the new innermost level τ_{in} for nested loops. Rule (BRK) requires a break statement to be controlled by an expression of level at least τ_{in} to enforce the noninterference policy.

2.2.3 Safe Programs. To ensure soundness, we put restrictions on admissible operator types, depending on their computational power.

Definition 2.4. An operator typing environment Δ is *safe* if for each $\mathsf{op} \in dom(\Delta)$, op is either neutral, positive, or polynomial, $[\![\mathsf{op}]\!]$ is a polynomial time computable function, and for each $(\tau_{in}, \tau_{out}) \in \mathbb{N}^2$, and each $\tau_1 \to \ldots \to \tau_{ar(\mathsf{op})+1} \in \Delta(\mathsf{op})(\tau_{in}, \tau_{out})$, it holds that:

(1) if op is a neutral operator then $\tau_{ar(\mathsf{op})+1} \leq \sqcap_{i=1}^{ar(\mathsf{op})} \tau_i$;
(2) if op is a positive operator then $\tau_{ar(\mathsf{op})+1} \leq \sqcap_{i=1}^{ar(\mathsf{op})} \tau_i$ and either $\tau_{ar(\mathsf{op})+1} < \tau_{in}$ or $\tau_{ar(\mathsf{op})+1} = 0$;
(3) if op is a polynomial operator then $\tau_{out} = 0$.

The intuition is as follows. An operator typing environment is safe if the flow cannot go in the wrong direction for neutral and positive operators ($\tau_{ar(\mathsf{op})+1} \leq \sqcap_{i=1}^{ar(\mathsf{op})} \tau_i$); for positive operators, the computed output cannot be reused in the calling loop ($\tau_{ar(\mathsf{op})+1} < \tau_{in}$ or $\tau_{ar(\mathsf{op})+1} = 0$); for polynomial operators, the outermost level is enforced to be 0. Hence, it is not called in a loop.

Definition 2.5 (Safety). Given a safe operator typing environment Δ, a program P is Δ*safe* if there are a level τ, a variable typing environment Γ such that $\Gamma, \Delta \vdash_0^0 \mathsf{body}(\mathsf{P}) : \tau$ holds. Let ΔSAFE be the set of Δsafe programs.

A program P is *safe* if there exists a safe operator typing environment Δ such that P is Δsafe. Let SAFE be the set of safe programs.

Example 2.6. The `bubble` program from Listing 3 can be typed with Γ, Δ such that $\Gamma(\mathtt{list}) = \Gamma(\mathtt{list1}) = \Gamma(\mathtt{len1}) = \Gamma(\mathtt{len2}) = 1$ and $\Gamma(\mathtt{list2}) = \Gamma(\mathtt{len}) = \Gamma(\mathtt{x}) = \Gamma(\mathtt{y}) = \Gamma(\mathtt{r}) = 0$, and

$$\begin{aligned} \{1 \to 1 \to 1\} &\subset \Delta(=)(1,1) \cap \Delta(>)(1,1) \cap \Delta(\neq)(1,1), \\ \{1 \to 1\} &\subset \Delta(\mathtt{tl})(1,1) \cap \Delta(-1)(1,1), \\ \{0 \to 0 \to 0\} &\subset \Delta(+)(1,1), \\ \{0 \to 0\} &\subset \Delta(+1)(1,1) \cap \Delta(\mathtt{hd})(1,1) \cap \Delta(\mathtt{tl})(1,1). \end{aligned}$$

`bubble` is safe: its typing derivation tree is sampled in Figure 4, presenting the typing of the main loop preceded by the assignment of its guard variable `len1`. In Figure 4, the statement s_1 is the body of the loop (line 11-28 in Listing 3). In Listing 3, we have also provided the typing of each expression as superscript notations.

2.3 Aperiodicity

Safety is not enough on its own to ensure polytime soundness since the declassification of one bit of information in a loop can lead to an exponential behavior as illustrated by the example of Listing 2. In order to avoid this phenomenon, we complement safety with a fairly simple and natural notion of aperiodicity.

Given an expression e, we define the set of *undeclassified variables* of e, $U(\mathsf{e})$, by structural induction on expressions

$$U(\mathsf{x}) \triangleq \{\mathsf{x}\},$$
$$U(\mathsf{op}(\bar{\mathsf{e}})) \triangleq \bigcup_{i=1}^{\ell(\bar{\mathsf{e}})} U(\mathsf{e}_i),$$
$$U(\mathtt{declass}(\mathsf{e}_1, \mathsf{e}_2)) \triangleq U(\mathsf{e}_2).$$

Two stores μ and μ' are e-*equivalent*, noted $\mu \equiv_{\mathsf{e}} \mu'$, if $\forall \mathsf{x} \in U(\mathsf{e})$, $\mu(\mathsf{x}) = \mu'(\mathsf{x})$.

Definition 2.7 (Aperiodicity). A program P is *aperiodic* if for each store μ there are no $\pi_{(\mu',\mathsf{s}')}, \pi_{(\mu'',\mathsf{s}')} \sqsubseteq \pi_{(\mu,\mathsf{body}(\mathsf{P}))}$, such that

$$\mathsf{s}' = \mathtt{while}(\mathsf{e})\ \{\mathsf{s}''\}, \quad \pi_{(\mu'',\mathsf{s}')} \sqsubset \pi_{(\mu',\mathsf{s}')}, \quad \text{and} \quad \mu' \equiv_{\mathsf{e}} \mu''.$$

Let AP be the set of aperiodic programs.

In other words, for any input store of an aperiodic program, there cannot be a subtree of root $(\mu', \mathtt{while}(\mathsf{e})\ \{\mathsf{s}''\})$ with a strict subtree of root $(\mu'', \mathtt{while}(\mathsf{e})\ \{\mathsf{s}''\})$ such that the stores μ' and μ'' match on undeclassified variables. Hence, this notion of aperiodicity is very natural as it means that a given while loop is never evaluated twice under (e-)equivalent stores. Declassified variables in e are treated differently as the domain of their value is bounded by the declassification semantics. Hence there is no need to require aperiodicity on them for the result to hold. This can be done, but it would reduce the expressive power of the analysis.

Example 2.8. Consider the dangerous code sample from Listing 2 in the introduction. This program is safe by taking the variable typing environment Γ defined by $\Gamma(\mathsf{x}) \triangleq 1$ and $\Gamma(\mathsf{y}) \triangleq 0$. However, it is not aperiodic as the undeclassified x will have value 1 more than once in the guard of line 3.

Example 2.9. The program bubble of Listing 3 is aperiodic.

2.4 Properties of Safe and Aperiodic Programs

2.4.1 Polytime Soundness and Completeness. A program P of the shape $\mathtt{prog}(\bar{\mathsf{x}})\{\mathsf{s}\ \mathtt{return}\ \mathsf{y}\}$ computes a partial function

$$[\![\mathsf{P}]\!] : \mathbb{W}^{\ell(\bar{\mathsf{x}})} \to \mathbb{W}$$

$$[\![\mathsf{P}]\!](\bar{w}) = v \text{ iff } (\mu[\bar{x} \leftarrow \bar{w}], \mathsf{s}) \to_{\mathsf{ex}} (\top, \mu') \text{ and } \mu'(\mathsf{y}) = v.$$

If $[\![\mathsf{P}]\!]$ is a total function, we say that the program is *terminating*. Let TERM be the set of terminating programs. Given a set of programs S, we define $[\![S]\!]$ by $[\![S]\!] \triangleq \{[\![\mathsf{P}]\!] \mid \mathsf{P} \in S\}$.

We now show a completeness result stating that for each function f in FP, the class of functions computable in polynomial time by a Turing machine, there exists a safe and aperiodic program P such that $f = [\![\mathsf{P}]\!]$. This completeness result relies on simulating any Turing machine that halts in polynomial time with a safe and aperiodic terminating program.

Theorem 2.10 (Completeness). $\mathsf{FP} \subseteq [\![\mathrm{SAFE} \cap \mathrm{AP}]\!]$.

For soundness, the intuition is as follows. For a fixed loop of level τ, the safety property implies a polynomial bound on the set of potentially computed values for variables of level greater than or equal to τ. Aperiodicity then makes it impossible to reach twice the same configuration of variables of level at least τ. Hence the number of iterations is polynomially bounded. Moreover, variables of strictly lower level can only increase polynomially by a constant (using positive operators). Repeating the reasoning for any level, the total runtime is polynomially bounded in the program input. This bound is obtained by a constant composition of polynomials.

Theorem 2.11 (Soundness). $[\![\mathrm{SAFE} \cap \mathrm{AP}]\!] \subseteq \mathsf{FP}$.

As a corollary, this implies termination. Indeed, on any input store, the space of reachable values (and hence reachable stores) corresponding to the program derivation tree is polynomially bounded.

Corollary 2.12 (Termination). $\mathrm{SAFE} \cap \mathrm{AP} \subsetneq \mathrm{TERM}$.

Safe and aperiodic programs contain polytime programs with loops guarded by data that may increase. For example, the program bubble of Listing 2.1 has an inner loop controlled by len2, which is reinitialized at the size of len at each iteration of the outer loop. Such programs cannot be captured by existing criteria, e.g., [Marion 2011]. Consequently, this result improves greatly on the expressive power of noninterference-based type systems for complexity.

2.4.2 Complexity of Type Inference and Aperiodicity. We now show that type inference can be decided in polynomial time in the *size of a program* $|\mathsf{P}|$ (i.e., number of symbols) , which implies that type inference can be efficiently implemented.

Theorem 2.13 (Type inference). *Given a safe operator typing environment* Δ*, deciding whether* $\mathsf{P} \in \Delta\mathrm{SAFE}$ *can be done in time* $O(|\mathsf{P}|^3)$.

Now we show that aperiodicity is a Π_1^0-complete problem in the arithmetical hierarchy.

Theorem 2.14 (Aperiodicity). *Deciding whether a program* P *is aperiodic (i.e.,* $\mathsf{P} \in \mathrm{AP}$*) is* Π_1^0*-complete.*

This undecidability result is not a negative result as the set of programs in the class FP is known to be Σ_2^0-complete [Hájek 1979]. Hence aperiodicity is a strictly simpler problem. Moreover, the termignation hypothesis in previous work (e.g., [Hainry and Péchoux 2018; Marion 2011]) is Π_0^2-complete ([Endrullis et al. 2011]), thus strictly harder than aperiodicity as a consequence of Post's Theorem.

2.4.3 Sound and Complete Criterion with Decidable Aperiodicity. Last but not least, it is possible to specify a decidable sufficient condition for aperiodicity that preserves the completeness of Theorem 2.10. We define a for loop statement $\mathtt{for}\,\mathsf{x} = \mathsf{e}\ \mathtt{to}\ \mathsf{d}\ \{\mathsf{s}\}$ as syntactic sugar for the statement $\mathsf{x} := \mathsf{d}; \mathtt{while}(\mathsf{x} \geq \mathsf{e})\{\mathsf{s}; \mathsf{x} := \mathsf{x} - 1\}$, under the proviso that x does not occur in s. Let *for-programs* be programs that only contains for loop (and no other while loop). For-programs are trivially aperiodic, by definition. Let FOR be the set of for-programs. We can show that safe for-programs are sound and complete for FP.

Theorem 2.15 (Decidable criterion). $[\![\mathrm{SAFE} \cap \mathrm{FOR}]\!] = \mathsf{FP}$

Expr	e	≜	x | op(ē) | declass(e, e) | X(ē)
Stmt	s	≜	skip | x := e | s; s | if(e){s} else{s} | while(e){s} | break(e)
Proc	p	≜	p(X̄, x̄){[var ȳ;] s return x}
Terms	t	≜	x | call p(c̄, t̄)
Closures	c	≜	X | λx̄.t
Prg	P	≜	t | declare p in P | box [a] in P

Figure 5: Syntax of second-order programs

Soundness and tractability of type inference are obtained as a direct corollary of Theorems 2.11 and 2.13 as for-programs are aperiodic programs. Completeness is also obtained easily as the programs used in the proof of Theorem 2.10 are all for-programs.

3 EXAMPLE: CHARACTERIZING BFF

We now enrich the language introduced in Section 2.1 with *procedures*, *oracle calls*, and *closures* with the aim of characterizing the second-order complexity class of Basic Feasible Functionals, BFF. The language additions are minimal and aim at extending the language to second-order – a prerequisite to characterize BFF – and are in no way linked to noninterference or declassification issues.

3.1 Type-2 Programming Language

3.1.1 Syntax. *Programs* are defined by the grammar of Figure 5 and can be either a term, a *procedure declaration* within a program, or the declaration of a *boxed variable*, called *box*, followed by a program. Boxed variables represent program inputs and can be either order-1 upper-case variables $\mathtt{X} \in \mathbb{V}_1$ or order-0 lower-case variables $\mathtt{x} \in \mathbb{V}_0$. In a box, the variable $\mathtt{a} \in \mathbb{V} \triangleq \mathbb{V}_0 \uplus \mathbb{V}_1$ can be of arbitrary order.

Terms t are order-0 constructs and can be either a variable x or a *procedure call* call p(c̄, t̄) corresponding to the application of procedure of name p to order-1 inputs c̄ and order-0 inputs t̄.

A *procedure declaration* p(X̄, x̄){[var ȳ;] s return x} is the corresponding order-2 abstraction. It maps order-1 and order-0 *parameters* param(p) ≜ {X̄, x̄} to a order-0 output x and consists of a body statement body(p) ≜ s with optional declaration of *local variables* local(p) ≜ {ȳ}. For convenience, we identify a procedure declaration with its procedure name. Moreover, we assume that each order-1 procedure parameter comes with a fixed arity $ar(\mathtt{X})$.

Statements are the same as in Figure 1 and thus can still only assign to order-0 variables. Similarly, *expressions* include those of Figure 1 plus a new construct X(ē), named *oracle call*, consisting in the application of a order-1 variable X to some sequence of expressions ē, called the *input data*. We assume that they are always fully applied, i.e., that $\ell(\bar{\mathtt{e}}) = ar(\mathtt{X})$.

Closures are order-1 abstractions corresponding to oracle calls and can be written either as an order-1 variable X or as λx̄.t, where the order-0 term t may contain free variables. Notice that procedure calls can only take closures as order-1 parameters and that procedure declarations with no oracle call are exactly the first-order programs of Section 2.1.

A variable is free if it is neither boxed nor bound by one of the two possible abstractions. A program is *closed* if it has no free variable. We define the following syntactic sugar:

$$\mathtt{declare}\ \bar{\mathtt{p}}\ \mathtt{in}\ \mathtt{P} \triangleq \mathtt{declare}\ \mathtt{p}_1\ \mathtt{in}\ \ldots \mathtt{declare}\ \mathtt{p}_n\ \mathtt{in}\ \mathtt{P}$$
$$\mathtt{box}\ [\bar{\mathtt{a}}]\ \mathtt{in}\ \mathtt{P} \triangleq \mathtt{box}\ [\bar{\mathtt{a}}_1]\ \mathtt{in}\ \ldots \mathtt{box}\ [\bar{\mathtt{a}}_m]\ \mathtt{in}\ \mathtt{P}$$

with $\bar{\mathtt{p}} \triangleq \mathtt{p}_1, \ldots, \mathtt{p}_n$ and $\bar{\mathtt{a}} \triangleq \bar{\mathtt{a}}_1, \ldots, \bar{\mathtt{a}}_m$.

Throughout the rest of the paper, we will restrict attention to closed programs in *normal form*. These consist of programs that can be written as follows box [X̄, x̄] in declare p̄ in t, for some term t such that the following *well-formedness conditions* hold:

i) *no name clash*: the set of the term free variables, each set of procedure parameters, and each set of procedure local variables are all pairwise distinct,
ii) *closed procedures*: each variable of a procedure body is either a procedure parameter or a local variable,
iii) *determinism*: each procedure name called in t is declared exactly once in p.

A program in normal form computes a second-order functional.

Example 3.1. Program I in Listing 6 is closed and in normal form and computes a variant of Cook-Urquhart's second-order bounded iterator [Cook and Urquhart 1993]

$$\mathcal{I} : (\mathbb{W} \to \mathbb{W}) \times \mathbb{W}^3 \to \mathbb{W}$$
$$\mathcal{I}(f, u, v, w) \triangleq (\lambda x.\restriction(f(x), v))^{|w|}(u)$$

where f^n is the n-iterate of f and $\restriction : \mathbb{W} \times \mathbb{W} \to \mathbb{W}$, defined by

$$\restriction(v, w) \triangleq \begin{cases} v & \text{if } |v| \le |w| \\ v' \text{ s.t. } |v'| = |w| \wedge v = v' \cdot v'' & \text{otherwise} \end{cases}$$

is a function truncating its first operand with respect to the size of its second operand.

In this example, words can encode lists to represent pairs and triples. To account for this, the alphabet Σ on which words are defined includes a constructor symbol # in addition to 1 and 0. On boolean and arithmetic operators, the function ⟦op⟧ behaves as expected on suitable data and returns the empty word ϵ in case of failure, e.g., ⟦| · |⟧(100#11) = 110, ⟦≤⟧(110, 111) = 1, and ⟦+1⟧(1#0) = ϵ. Constructor and destructor semantics can be defined similarly:

$$[\![\mathtt{cons}]\!](u, v) \triangleq \begin{cases} u \cdot \# \cdot v & \text{if } \# \notin u \\ \epsilon & \text{otherwise} \end{cases}$$
$$[\![\mathtt{hd}]\!]([\![\mathtt{cons}(u, v)]\!]) \triangleq u$$
$$[\![\mathtt{tl}]\!]([\![\mathtt{cons}(u, v)]\!]) \triangleq v$$

The operator pad is a padding operator defined by:

$$[\![\mathtt{pad}]\!](u, v) \triangleq \begin{cases} [\![\mathtt{cons}(v, 0^n)]\!] & \text{if } \exists n \ge 0,\ |v| + n + 1 = |u| \\ \epsilon & \text{otherwise} \end{cases}$$

We use this operator in procedure K to produce outputs of non-decreasing size (line 27). Finally, the operator truncate is the syntactic counterpart of the function ↾ defined by

$$[\![\mathtt{truncate}]\!](v, w) \triangleq \restriction(v, w)$$

and is used in the two procedures to bound the oracle outputs.

```
2  declare J(Y,l,m,n){
3    var b; var t; var z; var l0; var n0;
4    l0 := l;
5    n0 := n;
6    z := ε;
7    b := n+m+1;
8    while (n>0){
9      break (|Y(l,n)|>|Y(l0,n0)|);
10     t := truncate(Y(l,n),b);
11     l := hd(tl(t));
12     n := declass (tl(tl(t)),b)
13   };
14   if (n=0){
15     z := l
16   }
17   return z
18 }
```

Listing 4: Procedure J

```
19 declare K(X,p,q,r,s){
20   var i; var j;
21   i := r;
22   while (s>0){
23     break (|X(i)|>|X(r)|);
24     i := truncate(X(i),p);
25     s := s-1
26   };
27   j := pad(p+q+1,cons(i,s))
28   return j
29 }
```

Listing 5: Procedure K

```
1  box [F,u,v,w] in
2-18   declare J(Y,l,m,n){...} in
19-29    declare K(X,p,q,r,s){...} in
30         call J(λx,y.call K(F,v,w,x,y),u,v,w)
```

Listing 6: Program I

Program I makes use of two intermediate procedures J and K defined in Listings 4 and 5. K iterates at most s times an oracle call X(i) (hence computes $(\lambda x.\lceil(f(x), v))^{|s|}(u)$), for some s up to which f outputs does not exceed the size of the first call to f. This check is performed using a break statement. Hence K computes a functional in FLR. J iterates $|w|$ times K, using a closure.

Procedure J uses the declass construct. This will be required for the program to type as, in the loop, the output of an oracle call is assigned to the variable n that guards the loop. In general, such behaviors will be prohibited as there may be exponentially many oracle calls (even for a fixed size) unless an explicit declassification is performed. Here the declassification has the effect of restricting the output of the oracle call into a polynomially bounded domain.

3.1.2 Operational Semantics. In what follows, let $f, g, \ldots$ denote total functions in $\mathbb{W} \to \mathbb{W}$.

Stores μ and *environments* ϕ are defined by:

$$\text{(Stores)} \qquad \mu \in (\mathbb{V}_0 \to \mathbb{W}) \uplus \mathbb{V}_1 \to (\mathbb{W} \to \mathbb{W})$$

$$\text{(Env)} \qquad \phi \in \mathbb{V}_1 \to \mathtt{Closures}$$

Stores are total functions assigning words or functions on words to a variable, depending on its order. Environments assign a closure to an order-1 variable X. Again, $\mu_\emptyset$ will denote the empty store, mapping each order-0 variable x to ε and each order-1 variable X to the constant function $\lambda w.\varepsilon$. In the same vein, $\mu[\bar{x} \leftarrow \bar{w}, \bar{X} \leftarrow \bar{f}]$ denotes a store update. We now introduce the following judgments

$$\to_{\mathtt{exp}} \in (\mathcal{P}(\mathtt{Proc}) \times \mathtt{Stores} \times \mathtt{Env} \times \mathtt{Expr}) \to \mathbb{W}$$

$$\to_{\mathtt{stm}} \in (\mathcal{P}(\mathtt{Proc}) \times \mathtt{Stores} \times \mathtt{Env} \times \mathtt{Stmt}) \to (\{\top, \bot\} \times \mathtt{Stores})$$

$$\to_{\mathtt{prg}} \in (\mathcal{P}(\mathtt{Proc}) \times \mathtt{Stores} \times \mathtt{Prg}) \to \mathbb{W}$$

The judgment $(\sigma, \mu, \phi, \mathtt{e}) \to_{\mathtt{exp}} w$ means that the expression e evaluates to the word $w \in \mathbb{W}$ with respect to the set of procedure declarations σ, the store μ, and the environment ϕ. The judgment $(\sigma, \mu, \phi, \mathtt{s}) \to_{\mathtt{stm}} (\square, \mu')$, with $\square \in \{\top, \bot\}$, expresses that, under the set of procedure declarations σ, the store μ, and the environment ϕ, the statement s terminates and outputs the store μ'. As in Section 2, the symbol $\bot$ indicates that a break instruction has been executed.

The judgment $(\sigma, \mu, \mathtt{P}) \to_{\mathtt{prg}} w$ deterministically maps a set σ of procedure declarations, a store μ, and a program P in normal form to a word $w \in \mathbb{W}$.

The operational semantics is defined in Figure 6. Most rules are very similar to that of Figure 2. The new rules, corresponding to the new syntactic constructs, are highlighted with boxes. We provide an intuition for each of them. Rule (Orc) deals with an oracle calls. The rules evaluates the order-0 term t of the closure $\lambda\bar{x}.\mathtt{t}$ corresponding to the order-1 variable X in the environment ϕ with respect to the values $\bar{v}$ of the operands $\bar{e}$. Rule (Var) outputs the word $\mu(\mathtt{x})$. In Rule (Call), $\bar{X} \mapsto \bar{c}$, with $\ell(\bar{X}) = \ell(\bar{c})$, is the environment mapping each $X_i \in \mathbb{V}_1$ to the closure c_i. It is important to stress that Rule (Call) is the only rule that updates the environment to the closures passed as arguments: hence, oracle calls are fixed for each procedure call. Rule (Dec) just adds the procedure declaration to the set of procedures in the judgments and Rule (Box) is just a no-op rule.

The notation π_c is overloaded to denote the evaluation tree of root c using the rules of Figure 6, with c being an element in the definition domain of $\to_{\mathtt{exp}}$, $\to_{\mathtt{stm}}$, or $\to_{\mathtt{prg}}$ and the strict subtree relation $\sqsubset$ is updated accordingly.

As we will see shortly, the type discipline will ensure that a program $\mathtt{P} = \mathtt{box}\ [\bar{X}, \bar{x}]\ \mathtt{in}\ \mathtt{declare}\ \bar{p}\ \mathtt{in}\ \mathtt{t}$ computes the second-order partial functional $[\![\mathtt{P}]\!]$ in $(\mathbb{W} \to \mathbb{W})^{\ell(\bar{X})} \to \mathbb{W}^{\ell(\bar{x})} \to \mathbb{W}$, defined by:

$$[\![\mathtt{P}]\!](\bar{f}, \bar{w}) = w \text{ iff } (\emptyset, \mu_\emptyset[\bar{x} \leftarrow \bar{w}, \bar{X} \leftarrow \bar{f}], \mathtt{P}) \to_{\mathtt{prg}} w.$$

The store $\mu_\emptyset[\bar{x} \leftarrow \bar{w}, \bar{X} \leftarrow \bar{f}]$ is called an *input store*. If $[\![\mathtt{P}]\!]$ is a total function, the program P is said to be *terminating*. Let TERM_2 be the set of terminating programs.

3.2 FLR Restrictions and Aperiodicity

3.2.1 Syntactic Restrictions. For showing soundness and completeness with respect to BFF, we need to put some restrictions and hypothesis on the programming language. First, we assume that the set of operators $\mathbb{O}$ includes $\{=, <, \leq, 0, +1, -1, \mathtt{not}, \mathtt{and}, \mathtt{or}\}$, that are used in Section 2, and all the operators defined in Example 3.1.

$$\frac{}{(\sigma,\mu,\phi,\mathsf{x}) \to_{\mathsf{exp}} \mu(\mathsf{x})} \qquad \frac{(\sigma,\mu,\phi,\bar{\mathsf{e}}) \to_{\mathsf{exp}} \bar{w}}{(\sigma,\mu,\phi,\mathsf{op}(\bar{\mathsf{e}})) \to_{\mathsf{exp}} [\![\mathsf{op}]\!](\bar{w})} \qquad \frac{\forall i \le 2,\ (\sigma,\mu,\phi,\mathsf{e}_i) \to_{\mathsf{exp}} w_i}{(\sigma,\mu,\phi,\mathsf{declass}(\mathsf{e}_1,\mathsf{e}_2)) \to_{\mathsf{exp}} 1^{\min(|w_1|,|w_2|)}}$$

$$\frac{(\sigma,\mu,\phi,\bar{\mathsf{e}}) \to_{\mathsf{exp}} \bar{v} \quad \phi(\mathsf{X}) = \lambda\bar{\mathsf{x}}.\mathsf{t} \quad (\sigma,\mu[\bar{\mathsf{x}} \leftarrow \bar{v}],\mathsf{t}) \to_{\mathsf{prg}} w}{(\sigma,\mu,\phi,\mathsf{X}(\bar{\mathsf{e}})) \to_{\mathsf{exp}} w}\ (\mathrm{Orc})$$

$$\frac{}{(\sigma,\mu,\phi,\mathtt{skip}) \to_{\mathtt{stm}} (\top,\mu)} \qquad \frac{(\sigma,\mu,\phi,\mathsf{e}) \to_{\mathsf{exp}} w}{(\sigma,\mu,\phi,\mathsf{x} := \mathsf{e}) \to_{\mathtt{stm}} (\top,\mu[\mathsf{x} \leftarrow w])} \qquad \frac{(\sigma,\mu,\phi,\mathsf{s}_1) \to_{\mathtt{stm}} (\bot,\mu')}{(\sigma,\mu,\phi,\mathsf{s}_1\,;\,\mathsf{s}_2) \to_{\mathtt{stm}} (\bot,\mu')}$$

$$\frac{(\sigma,\mu,\phi,\mathsf{s}_1) \to_{\mathtt{stm}} (\top,\mu') \quad (\sigma,\mu',\phi,\mathsf{s}_2) \to_{\mathtt{stm}} (\square,\mu'')}{(\sigma,\mu,\phi,\mathsf{s}_1\,;\,\mathsf{s}_2) \to_{\mathtt{stm}} (\square,\mu'')}\ (\square \in \{\top,\bot\}) \qquad \frac{(\sigma,\mu,\phi,\mathsf{e}) \to_{\mathsf{exp}} w \quad (\sigma,\mu,\phi,\mathsf{s}_w) \to_{\mathtt{stm}} (\square,\mu')}{(\sigma,\mu,\phi,\mathtt{if}(\mathsf{e})\{\mathsf{s}_1\}\,\mathtt{else}\,\{\mathsf{s}_{\underline{0}}\}) \to_{\mathtt{stm}} (\square,\mu')}\ (\square \in \{\top,\bot\}, w \in \{\underline{0},1\})$$

$$\frac{(\sigma,\mu,\phi,\mathsf{e}) \to_{\mathsf{exp}} \underline{0}}{(\sigma,\mu,\phi,\mathtt{while}(\mathsf{e})\{\mathsf{s}\}) \to_{\mathtt{stm}} (\top,\mu)} \qquad \frac{(\sigma,\mu,\phi,\mathsf{e}) \to_{\mathsf{exp}} 1 \quad (\sigma,\mu,\phi,\mathsf{s}\,;\,\mathtt{while}(\mathsf{e})\{\mathsf{s}\}) \to_{\mathtt{stm}} (\square,\mu')}{(\sigma,\mu,\phi,\mathtt{while}(\mathsf{e})\{\mathsf{s}\}) \to_{\mathtt{stm}} (\top,\mu')}\ (\square \in \{\top,\bot\})$$

$$\frac{(\sigma,\mu,\phi,\mathsf{e}) \to_{\mathsf{exp}} \underline{0}}{(\sigma,\mu,\phi,\mathtt{break}(\mathsf{e})) \to_{\mathtt{stm}} (\top,\mu)} \qquad \frac{(\sigma,\mu,\phi,\mathsf{e}) \to_{\mathsf{exp}} 1}{(\sigma,\mu,\phi,\mathtt{break}(\mathsf{e})) \to_{\mathtt{stm}} (\bot,\mu)} \qquad \frac{}{(\sigma,\mu,\mathsf{x}) \to_{\mathtt{prg}} \mu(\mathsf{x})}\ (\mathrm{Var})$$

$$\frac{\mathsf{p}(\bar{\mathsf{X}},\bar{\mathsf{x}})\{\mathsf{s}\ \mathtt{return}\ \mathsf{z}\} \in \sigma \quad (\sigma,\mu,\bar{\mathsf{t}}) \to_{\mathtt{prg}} \bar{w} \quad (\sigma,\mu[\bar{\mathsf{x}} \leftarrow \bar{w},\bar{\mathsf{y}} \leftarrow \bar{\epsilon}],\bar{\mathsf{X}} \mapsto \bar{\mathsf{c}},\mathsf{s}) \to_{\mathtt{stm}} (\square,\mu')}{(\sigma,\mu,\mathtt{call}\ \mathsf{p}(\bar{\mathsf{c}},\bar{\mathsf{t}})) \to_{\mathtt{prg}} \mu'(\mathsf{z})}\ (\square \in \{\top,\bot\})\ (\mathrm{Call})$$

$$\frac{(\sigma \cup \{\mathsf{p}\},\mu,\mathsf{P}) \to_{\mathtt{prg}} w}{(\sigma,\mu,\mathtt{declare}\ \mathsf{p}\ \mathtt{in}\ \mathsf{P}) \to_{\mathtt{prg}} w}\ (\mathrm{Dec}) \qquad \frac{(\sigma,\mu,\mathsf{P}) \to_{\mathtt{prg}} w}{(\sigma,\mu,\mathtt{box}\ [\mathsf{a}]\ \mathtt{in}\ \mathsf{P}) \to_{\mathtt{prg}} w}\ (\mathrm{Box})$$

Figure 6: Semantics of second-order programs

Indeed, this example is used in the proof of completeness and, hence, it is important to include all its operators for completeness to hold. Now we syntactically restrict the study to *guarded programs*.

Definition 3.2 (Guarded program). A program is *guarded* if

(1) oracle calls cannot be nested and appear either in assignments or in break statements $\mathtt{break}(|\mathsf{X}(\bar{\mathsf{e}})| > |\mathsf{X}(\bar{\mathsf{x}})|)$,
(2) in a loop, each assignment containing an oracle call $\mathsf{X}(\bar{\mathsf{e}})$ is preceded by a break statement $\mathtt{break}(|\mathsf{X}(\bar{\mathsf{e}})| > |\mathsf{X}(\bar{\mathsf{x}})|)$.

Guardedness just put simple syntactic restrictions on how oracle calls can be used in a program. The intuition behind is to ensure the FLR property, i.e., that size of oracle calls cannot increase more than a constant number of times during the program execution. The aim of item (2) is to ensure FLR by the increase of oracle outputs with a break statement before assigning them. The restriction of oracle calls to break statements and assignments (1) avoids uncontrolled increases in other expressions of the program (guard of while, of conditionals, ...). However, this syntactic restriction is not sufficient on its own to ensure that programs compute a functional in FLR (a fortiori in SPT). This property will be achieved by complementing the restriction by the typing discipline. In what follows, we will restrict our study to guarded programs.

Example 3.3. Program I of Listing 6 is guarded. Indeed, the oracle calls line 10 in procedure J and at line 24 in procedure K are immediately preceded by a break statement of the good shape (2). Moreover, there are no other oracle calls in the program (1).

3.2.2 Aperiodicity Revisited. We lift the notion of aperiodicity introduced in Definition 2.7 to this second-order programming language. The set of *undeclassified variables* of e, $U(\mathsf{e})$, is extended by

$$U(\mathsf{X}(\bar{\mathsf{e}})) \triangleq \bigcup_{i=1}^{\ell(\bar{\mathsf{e}})} U(\mathsf{e}_i),$$

and the definition of $\equiv_{\mathsf{e}}$ is updated accordingly.

Definition 3.4 (II-Aperiodicity). A program P is *II-aperiodic* if, for each store μ, there are no $\pi_{(\sigma',\mu',\phi',\mathsf{s})}$ $\pi_{(\sigma'',\mu'',\phi'',\mathsf{s})} \sqsubseteq \pi_{(\emptyset,\mu,\mathsf{P})}$ s.t. $\mathsf{s} = \mathtt{while}(\mathsf{e})\,\{\mathsf{s}'\}$, $\pi_{(\sigma',\mu',\phi',\mathsf{s})} \sqsubset \pi_{(\sigma'',\mu'',\phi'',\mathsf{s})}$, and $\mu' \equiv_{\mathsf{e}} \mu''$. Let AP_2 be the set of II-aperiodic programs.

II-aperiodicity is very similar to the notion of aperiodicity for first-order programs in Definition 2.7: it still enforces that there cannot be two consecutive calls to a while loop of a given procedure under equivalent stores.

For a given set of programs S, let $[\![S]\!] \triangleq \{[\![\mathsf{P}]\!] \mid \mathsf{P} \in S\}$ denote the set of functions computed by programs in S. E.g., $[\![\mathrm{AP}_2 \cap \mathrm{TERM}_2]\!]$ is the set of functions computed by aperiodic and terminating programs. In aperiodic and terminating programs, termination does not depend on the oracle output.

Example 3.5. The program I of Listing 6 is in TERM_2. Indeed, procedure K trivially terminates on any input. Moreover, when called on the appropriate closure (line 30 of Listing 6), procedure J assigns to n a value that has decreased by at least 1 or that is equal to 0 (line 8 of Listing 4). Notice however that it does not entail that the procedures are terminating. Indeed, procedure J only terminates on specific closures (including $\lambda\mathsf{x},\mathsf{y}.\mathtt{call}\ \mathsf{K}(\mathsf{F},\mathsf{v},\mathsf{w},\mathsf{x},\mathsf{y})$). Program I is also aperiodic: on any input store, in procedure K, the value of variable s used in the guard of the loop strictly decreases and, in procedure J, the value of variable n used in the guard of the

loop strictly decreases, as this procedure is called on the closure λx, y.call K(F, v, w, x, y). We conclude that $\mathtt{I} \in \mathrm{AP}_2 \cap \mathrm{TERM}_2$.

As the proof of Theorem 2.14 is only based on procedure bodies, II-aperiodicity is also Π^1_0-hard and in Π^1_1.

Corollary 3.6. *Deciding whether a program* P *is II-aperiodic (i.e.,* $\mathtt{P} \in \mathrm{AP}_2$*) is* Π^0_1*-hard.*

3.3 Type System

Now we adapt the declassification policy presented in Section 2 to the second-order programming language.

3.3.1 Simple Types and Levels. Let W be the type of words in $\mathbb{W}$. The set $\mathcal{T}_{\mathtt{W}}$ contains all *simple types* over W that are defined inductively by the following grammar $\mathtt{T} ::= \mathtt{W} \mid \mathtt{T} \to \mathtt{T}$.

The order of a simple type is defined inductively by

$$\mathtt{ord}(\mathtt{T}) \triangleq \begin{cases} \max(1 + \mathtt{ord}(\mathtt{T}_1), \mathtt{ord}(\mathtt{T}_2)) & \text{if } \mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2 \\ 0 & \text{if } \mathtt{T} = \mathtt{W} \end{cases}$$

To account for oracle outputs, the considered set of levels is now $\mathbb{N}_\infty \triangleq \mathbb{N} \cup \{\infty\}$. The level ∞ behaves as expected: for each $\tau \in \mathbb{N}_\infty$, $\tau \leq \infty$, $\tau + \infty = \infty$, $\tau \sqcup \infty = \infty$, and $\tau \sqcap \infty = \tau$ hold. Similarly to simple types, levels can be extended to functional levels by $\mathtt{L} ::= \tau \mid \mathtt{L} \to \mathtt{L}$, with $\tau \in \mathbb{N}_\infty$ and their order can be defined in the same way.

As usual, an operator op will have functional levels of the shape $\tau_1 \to \ldots \to \tau_{ar(\mathtt{op})+1}$ of order 1 that can thus be viewed as a tuple in $\mathbb{N}_\infty^{ar(\mathtt{op})+1}$.

3.3.2 Typing Environments. We define four kinds of typing environments:

- *variable typing environments* Γ are finite maps in $\mathbb{V}_0 \to \mathbb{N}_\infty$,
- *operator typing environments* Δ map each operator $\mathtt{op} \in \mathbb{O}$ to a finite map $\Delta(\mathtt{op}) \in \mathbb{N}^2 \to \mathcal{P}(\mathbb{N}_\infty^{ar(\mathtt{op})+1})$,
- *simple typing environments* $\Gamma_{\mathtt{W}}$ are finite maps in $\mathbb{V} \to \mathcal{T}_{\mathtt{W}}$,
- *procedure typing environments* Ω map each procedure name to a pair $\langle \Gamma, \bar{\tau} \rangle$ consisting of a variable typing environment Γ and a triplet of levels $\bar{\tau} \in \mathbb{N}_\infty \times \mathbb{N} \times \mathbb{N}$.

Variable and operator typing environment are defined as in Section 2.2, with the distinction that ∞ can appear in the functional level of an operator. Simple typing environments just assign the type W to variables in $\mathbb{V}_0$ and the type $\mathtt{W} \to \mathtt{W}$ to variables in $\mathbb{V}_1$. A procedure typing environment assigns a typing context to each procedure. Operator, procedure, and simple typing environments are global, i.e. defined for the whole program. Variable typing environments are local, i.e. relative to the procedure under analysis.

For a procedure typing environment Ω, it will be assumed that for every $\mathtt{p} \in \mathit{dom}(\Omega)$, if $\Omega(\mathtt{p}) = \langle \Gamma, \bar{\tau} \rangle$ then $(\mathtt{param}(\mathtt{p}) \cup \mathtt{local}(\mathtt{p})) \cap \mathbb{V}_0 \subseteq \mathit{dom}(\Gamma)$. I.e., Ω is defined for the local variables and order-0 parameters of the procedure p.

3.3.3 Typing Judgments. The type system uses two kinds of judgments:

(1) *Statement (resp. expression) typing judgments* $\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{s} : \tau$ (resp. $\Gamma, \Delta \vdash^{\tau_{in}}_{\tau_{out}} \mathtt{e} : \tau$), with $\tau_{in}, \tau_{out} \in \mathbb{N}$ and $\tau \in \mathbb{N}_\infty$. The meaning of this judgment is that the *statement level* (resp. *expression level*) is $\tau \in \mathbb{N}_\infty$, under the prerequisite that the *innermost level* is τ_{in}, and the *outermost level* is τ_{out}. These levels are defined similarly to Section 2.2 and are still finite (i.e., $\neq \infty$) as our type discipline will prevent a loop from being guarded by level ∞ data.

(2) *Program (resp. term and closure) typing judgments* $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{P} : \mathtt{T}$ ($\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{t} : \mathtt{W}$ and $\Gamma_{\mathtt{W}}, \Omega, \Delta \vdash \mathtt{c} : \bar{\mathtt{W}} \to \mathtt{W}$, resp.). The meaning of this judgment is that P (t and c, resp.) is of simple type T (W and $\bar{\mathtt{W}} \to \mathtt{W}$, resp.) under the considered typing environment. Moreover, if $\mathtt{ord}(\mathtt{T}) = i$ then P is said to be of *order-i*. On the other hand, a term (resp. closure) is necessarily of order-0 (resp. 1), i.e., of type W (resp. $\bar{\mathtt{W}} \to \mathtt{W}$).

3.3.4 Typing Rules. The type system is provided in Figure 7. *Well-typed programs* are normal form and guarded order-2 programs that can be given the type $(\overline{\mathtt{W} \to \mathtt{W}}) \to \bar{\mathtt{W}} \to \mathtt{W}$, i.e., second-order programs computing a functional. The type system is composed of two sub-systems. The typing rules provided at the lower part of Figure 7 enforce that terms follow a standard simply-typed discipline. The typing rules presented in the upper part of Figure 7 ensure that procedure bodies follow a level-based type discipline, adapting smoothly the policy of Section 2 to the second-order setting. The transition between the two subsystems is performed in the Rule (DEC) of Figure 7 that checks that the procedure body follows the level-based type discipline once and for all in a procedure declaration. In Figure 7, the main differences with the type system of Figure 3 are highlighted in boxes. For simplicity, we have not specified the sequence lengths in the typing rules. However, it is clear that programs must check the following trivial syntactic constraints: in Rule (CALL), the type arity of each closure must match the arity $ar(\mathtt{X})$ of the corresponding procedure parameter; in Rule (ORC) , the length of $\bar{\mathtt{e}}$ is equal to $ar(\mathtt{X})$.

3.3.5 Intuition. We now provide some intuitions on the typing discipline.

The typing rules of Figure 7 that are not in boxes implement a standard noninterference policy with declassification, as already presented in Figure 3. In a loop of level τ, data of level τ or higher can only be copied or decreased using neutral operators (or declassifications). As neutral operators compute subwords or predicates, the corresponding space is polynomially bounded in the size of the program input. The restrictions put on declassified data preserve this property. Moreover, data of strictly smaller level can increase by a constant in each assignment. Hence under suitable assumptions: aperiodicity and safety (see next section), the loop terminates in a polynomial number of steps and the lower level data increase at most polynomially. The same kind of reasoning can be performed on loops of strictly smaller level. Hence the program cannot run for time greater than a composition of a constant (in the size of the program) number of polynomials in the input size and the maximal output of an oracle call. This ensure that the program computes a functional in Oracle PolyTime (OPT), as discussed in the introduction.

We now discuss the modification on the level-based discipline. Rule (ORC) takes oracle outputs to level ∞. On this other hand, by the side condition $\tau \neq \infty$, Rules (WH) and (WI) ensure that loops cannot be guarded by data of level ∞. Consequently, by the side condition $\tau_2 \neq \infty$ of Rule (ASG), oracle outputs cannot be directly

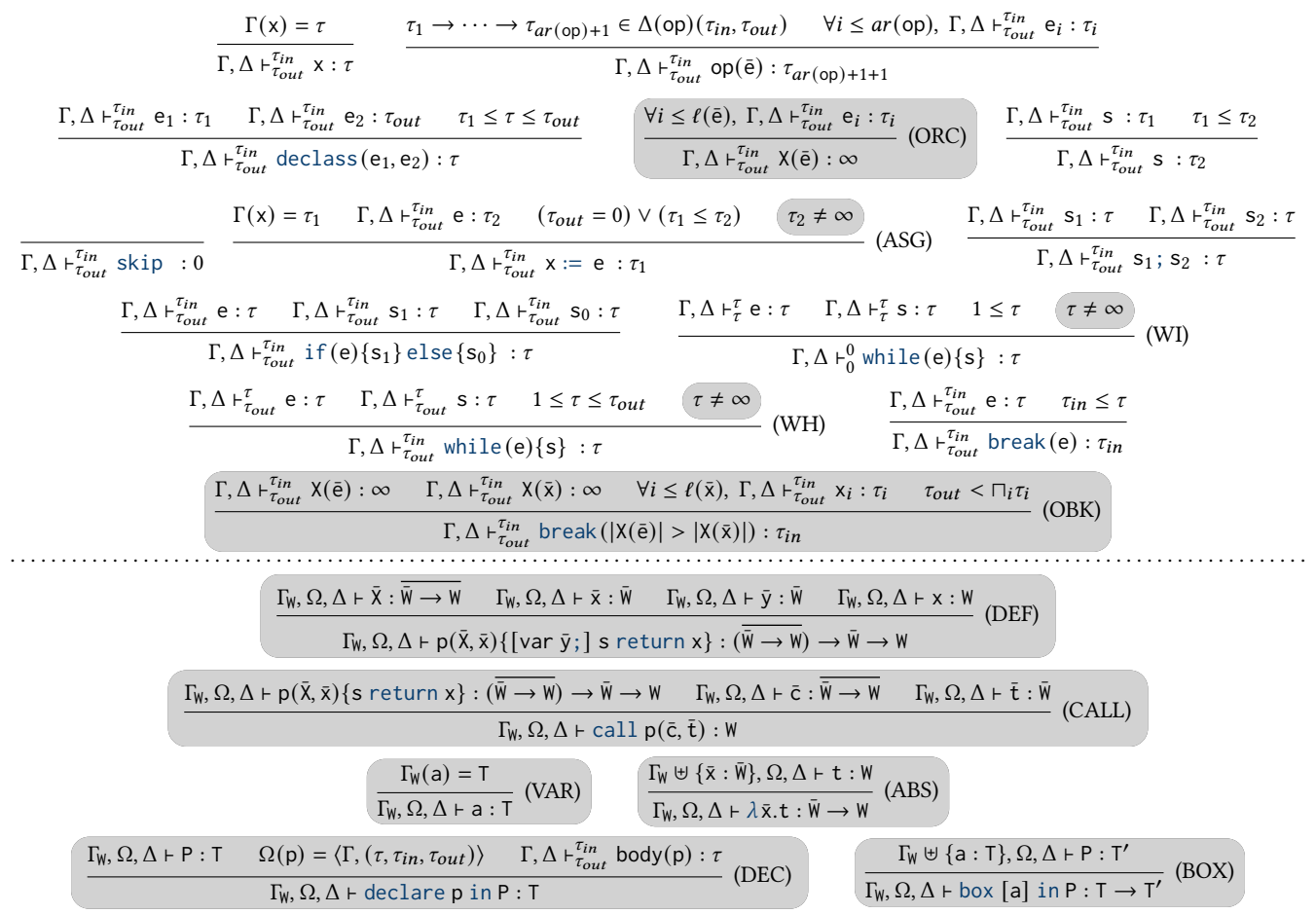

Figure 7: Level-based typing rules for second-order programs

assigned to in a loop. Hence the only option for an oracle output to be assigned is to be declassified, i.e., truncated and converted to unary (see the reduction rule for `declass` in Figure 6). We will relax a bit this strong constraint by also allowing the (non unary) truncation of oracle output in the next section. As a consequence of the restrictions we will put later on admissible operator types and as programs are guarded, only a constant number of unbounded oracle calls can be performed in a typed programs, most of which occur outside while loops (only 1 will be admitted per outermost while loop). Rule (OBK) enforces that break statements with oracle calls are controlled by expressions of the shape $|\mathsf{X}(\bar{e})| > |\mathsf{X}(\bar{x})|$: the while loop breaks if the oracle call has size greater than the size of the evaluation of $\mathsf{X}(\bar{x})$. However expressions in $\bar{x}$ have level τ_i such that $\tau_{out} < \tau_i$. Hence, they cannot be modified inside the (outermost) while loop. The while loop breaks if the size of the oracle call $\mathsf{X}(\bar{e})$ exceeds the size of the (constant in the loop) call $\mathsf{X}(\bar{x})$. Notice that the result of the call does not need to be fully explored for the test to be checked (i.e., if value v is the result of evaluating $\mathsf{X}(\bar{x})$ the test can be performed in at most $|v| + 1$ steps). This means that while we remain in the loop, the oracle calls are all of size bounded by the size of the result of $\mathsf{X}(\bar{x})$, i.e., the computed functional remains in Finite Length Revision (FLR).

To conclude, the functionals computed by each procedure call are in SPT = OPT ∩ FLR and the $\lambda(-)_2$ closure of Theorem 1.1 can just be simulated by our notions of closure and procedure call.

3.4 Safe Programs and their Properties

We now adapt the notion of safety to the second-order setting.

Definition 3.7. An operator typing environment Δ is *safe* if

$$\Delta(\mathtt{truncate})(\tau_{in}, \tau_{out}) \triangleq \{\infty \to \tau \to \tau' \mid \tau' < \tau_{in} \text{ and } \tau_{out} \leq \tau\}$$

and for each $\mathsf{op} \in dom(\Delta)$, $\mathsf{op} \neq \mathtt{truncate}$, is either neutral, positive, or polynomial, $[\![\mathsf{op}]\!] \in \mathsf{FP}$, and for each $\tau_{in}, \tau_{out} \in \mathbb{N}$, and for each $\tau_1 \to \ldots \to \tau_{ar(\mathsf{op})+1} \in \Delta(\mathsf{op})(\tau_{in}, \tau_{out})$,

(1) if op is a neutral operator then $\tau_{ar(\mathsf{op})+1} \leq \sqcap_{i=1}^{ar(\mathsf{op})} \tau_i \leq \sqcup_{i=1}^{ar(\mathsf{op})} \tau_i \neq \infty$;
(2) if op is a positive operator then $\tau_{ar(\mathsf{op})+1} \leq \sqcap_{i=1}^{ar(\mathsf{op})} \tau_i \leq \sqcup_{i=1}^{ar(\mathsf{op})} \tau_i \neq \infty$, and either $\tau_{ar(\mathsf{op})+1} < \tau_{in}$ or $\tau_{ar(\mathsf{op})+1} = 0$;
(3) if op is a polynomial operator then $\tau_{out} = 0$.

We briefly explain the intuition that lies behind safe operator typing environments. Operator `truncate` is treated apart to allow non-unary flows from level ∞ to a finite level τ'. However the price to pay for that permissiveness is that the output of a truncate can never be used to guard the corresponding loops (as $\tau' < \tau_{in}$). The other constraints are similar to the ones of Definition 2.4, as there cannot be an operator applied to an oracle output (level ∞ data), a declassification or a truncation have to occur first.

Definition 3.8 (Safety). Given a procedure typing environment Ω and a safe operator typing environment Δ, P is a *safe program* if it is a well-typed program with respect to Ω and Δ, that is, $\emptyset, \Omega, \Delta \vdash \mathsf{P} : (\overline{\mathsf{W} \to \mathsf{W}}) \to \bar{\mathsf{W}} \to \mathsf{W}$ can be derived.

Let SAFE_2 be the set of safe programs and $\Delta\mathrm{SAFE}_2$ be the set of safe programs with respect to Δ.

Example 3.9. Let us study how procedure K from Listing 5 can be part of a safe program. The `while` loop can be typed as follows:

```
22 while (s¹>0)¹{
23   break (|X(i⁰)^∞|>|X(r²)^∞|): 1;
24   i⁰ := truncate(X(i⁰)^∞,p¹)⁰:0;
25   s¹ := (s¹-1)¹ : 1
26 }
```

The level of `s` at line 22 is enforced to be at least 1 (and cannot be ∞), by Rule (WI). Hence in the while loop body the innermost and outermost levels are equal to 1. The level of `r` at line 23 is enforced to be 2, as it needs to be strictly greater than the outermost level, by Rule (OBK). The level of `p` at line 24 is enforced to be equal to the outermost level 1, by definition of safety. The level of the whole expression `truncate(X(i),p)` is 0. Indeed, by definition of a safe operator typing environment, it has to be smaller than the level of its first operand ∞ and strictly smaller than the innermost level 1. Hence at line 24, the level of `i` is enforced to be 0 by Rule (ASG) and the statement can be given the level 0. Finally, the full loop body can be typed using the rules for sequence and subtyping (SEQ) and (SUB). Notice that the remaining lines of Listing 5 type as there are no constraints on assignments outside loops.

First, we can show our main result, stating that the set of functionals computed by safe, terminating, and aperiodic programs is exactly BFF. Hence, we have succeeded to capture this class using the SPT scheme of Theorem 1.1.

Theorem 3.10. $[\![\mathrm{SAFE}_2 \cap \mathrm{AP}_2 \cap \mathrm{TERM}_2]\!] = \mathsf{BFF}$.

And we can show that safety is tractable. Again, let $|\mathsf{P}|$ be the number of symbols in P.

Theorem 3.11. *Given a safe operator typing environment Δ, deciding whether $\mathsf{P} \in \Delta\mathrm{SAFE}_2$ can be done in time polynomial in $|\mathsf{P}|$.*

4 CONCLUSION AND FUTURE WORK

We have provided a new declassification policy for program complexity analysis and shown that it can be applied to provide an expressive characterization of FP and the first characterization of the class BFF based on SPT. The declassification policy uses a declassification construct that can be used inside while loop and is, hence, quite general. The characterization is achieved by putting extra and intuitive restrictions: aperiodicity and termination in the second-order case. We also show that type inference for safety is tractable and that aperiodicity is a Π^0_1-complete problem. We have also exhibited a tractable and completeness-preserving criterion for aperiodicity. A future line of research is the search for tractable and more expressive criteria for aperiodicity.

ACKNOWLEDGMENTS

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. The work of Emmanuel Hainry and Romain Péchoux was supported by Inria associate team TC(Pro)^3.

REFERENCES

Michael Backes and Birgit Pfitzmann. 2003. Intransitive non-interference for cryptographic purposes. In *SSP 2003*. 140–152. https://doi.org/10.1109/SECPRI.2003.1199333

Patrick Baillot, Ugo Dal Lago, Cynthia Kop, and Deivid Vale. 2024. On Basic Feasible Functionals and the Interpretation Method. In *Foundations of Software Science and Computation Structures, FOSSACS 2024 (Lecture Notes in Computer Science, Vol. 14575)*. Springer, 70–91. https://doi.org/10.1007/978-3-031-57231-9_4

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

Siddharth Bhaskar, Cynthia Kop, and Jakob Grue Simonsen. 2023. Subclasses of Ptime Interpreted by Programming Languages. *Theory Comput. Syst.* 67, 3 (2023), 437–472. https://doi.org/10.1007/S00224-022-10074-Z

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 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

Ugo Dal Lago, Reinhard Kahle, and Isabel Oitavem. 2021. A Recursion-Theoretic Characterization of the Probabilistic Class PP. In *Mathematical Foundations of Computer Science (MFCS 2021)*, Vol. 202. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 35:1–35:12. https://doi.org/10.4230/LIPICS.MFCS.2021.35

Norman Danner, Jennifer Paykin, and James S. Royer. 2013. A static cost analysis for a higher-order language. In *Programming languages meets program verification, PLPV 2013*. ACM, 25–34. https://doi.org/10.1145/2428116.2428123

Norman Danner and James S. Royer. 2006. Adventures in time and space. In *Principles of Programming Languages, POPL 2006*. ACM, 168–179. https://doi.org/10.1145/1111037.1111053

Daniel de Carvalho and Jakob Grue Simonsen. 2014. An Implicit Characterization of the Polynomial-Time Decidable Sets by Cons-Free Rewriting. In *Rewriting and Typed Lambda Calculi (RTA-TLCA 2014) (Lecture Notes in Computer Science, Vol. 8560)*. Springer, 179–193. https://doi.org/10.1007/978-3-319-08918-8_13

Jörg Endrullis, Herman Geuvers, Jakob Grue Simonsen, and Hans Zantema. 2011. Levels of undecidability in rewriting. *Information and Computation* 209, 2 (2011), 227–245. https://doi.org/10.1016/j.ic.2010.09.003

Joseph A. Goguen and José Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–11. https://doi.org/10.1109/SP.1982.10014

Joseph A. Goguen and José Meseguer. 1984. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy*. IEEE, 75–75. https://doi.org/10.1109/SP.1984.10019

Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. 2021. ComplexityParser: An Automatic Tool for Certifying Poly-Time Complexity of Java Programs. In *Theoretical Aspects of Computing - ICTAC 2021*. Springer, 357–365. https://doi.org/10.1007/978-3-030-85315-0_20

Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. 2020. A tier-based typed programming language characterizing Feasible Functionals. In *Logic in Computer Science, LICS 2020*. 535–549. https://doi.org/10.1145/3373718.3394768

Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. 2022. Complete and tractable machine-independent characterizations of second-order polytime. In *Foundations of Software Science and Computation Structures, FOSSACS 2022*. 368–388. https://doi.org/10.1007/978-3-030-99253-8_19

Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. 2013. Type-based complexity analysis for fork processes. In *Foundations of Software Science and Computation*

Structures, FOSSACS 2013. Springer, 305–320. https://doi.org/10.1007/978-3-642-37075-5_20

Emmanuel Hainry and Romain Péchoux. 2018. A type-based complexity analysis of Object Oriented programs. *Inf. Comput.* 261 (2018), 78–115. https://doi.org/10.1016/j.ic.2018.05.006

Emmanuel Hainry and Romain Péchoux. 2020. Theory of higher order interpretations and application to Basic Feasible Functions. *Log. Methods Comput. Sci.* 16, 4 (2020). https://doi.org/10.23638/LMCS-16(4:14)2020

Emmanuel Hainry and Romain Péchoux. 2023. A General Noninterference Policy for Polynomial Time. In *Principles of programming languages, POPL 2023*. ACM, 806–832. https://doi.org/10.1145/3571221

Petr Hájek. 1979. Arithmetical Hierarchy and Complexity of Computation. *Theor. Comput. Sci.* 8 (1979), 227–237. https://doi.org/10.1016/0304-3975(79)90046-X

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. 2001. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.* 11, 1 (2001), 5–94. https://doi.org/10.1017/S0956796800003889

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

Daniel Leivant. 1991. A foundational delineation of computational feasibility. In *Logic in Computer Science (LICS'91)*. IEEE, 2–3. https://doi.org/10.1109/LICS.1991.151625

Daniel Leivant and Jean-Yves Marion. 1993. Lambda Calculus Characterizations of Poly-Time. *Fundam. Inform.* 19, 1/2 (1993), 167–184. https://doi.org/10.3233/FI-1993-191-207

Peng Li and Steve Zdancewic. 2005. Downgrading policies and relaxed noninterference. In *Principles of programming languages, POPL 2005*. ACM, 158–170. https://doi.org/10.1145/1040305.1040319

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

Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *Principles of programming languages, POPL 1999*. ACM, 228–241. https://doi.org/10.1145/292540.292561

Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.* 9, 4 (2000), 410–442. https://doi.org/10.1145/363516.363526

Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *J. Comput. Secur.* 17, 5 (oct 2009), 517–548. https://doi.org/10.3233/jcs-2009-0352

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

Dennis Volpano and Geoffrey Smith. 2000. Verifying secrets and relative secrecy. In *Principles of programming languages, POPL 2000*. ACM, 268–276. https://doi.org/10.1145/325694.325729

Steve Zdancewic and Andrew C. Myers. 2001. Robust declassification. In *Computer Security Foundations Workshop, CSFW 2001*. IEEE, 15–23. https://doi.org/10.1109/csfw.2001.930133