A General Noninterference Policy for Polynomial Time

Emmanuel Hainry, Romain Péchoux

A General Noninterference Policy for Polynomial Time

EMMANUEL HAINRY and ROMAIN PÉCHOUX, Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

We introduce a new noninterference policy to capture the class of functions computable in polynomial time on an object-oriented programming language. This policy makes a clear separation between the standard noninterference techniques for the control flow and the layering properties required to ensure that each "security" level preserves polynomial time soundness, and is thus very powerful as for the class of programs it can capture. This new characterization is a proper extension of existing tractable characterizations of polynomial time based on safe recursion. Despite the fact that this noninterference policy is Π^0_1-complete, we show that it can be instantiated to some decidable and conservative instance using shape analysis techniques.

CCS Concepts: • **Theory of computation** → **Complexity theory and logic**; **Type theory**; **Logic and verification**.

Additional Key Words and Phrases: Polynomial time, Computational Complexity, Noninterference, Shape Analysis

ACM Reference Format:
Emmanuel Hainry and Romain Péchoux. 2023. A General Noninterference Policy for Polynomial Time. *Proc. ACM Program. Lang.* 7, POPL, Article 28 (January 2023), 27 pages. https://doi.org/10.1145/3571221

1 INTRODUCTION

Motivations. The use of information flow policies for protecting privacy and integrity of sensitive data has been deeply studied for decades (see [Sabelfeld and Myers 2003; Zdancewic 2004], for an overview of the topic). Noninterference is an example of such a policy [Goguen and Meseguer 1982]. Noninterference allows programs to manipulate and modify confidential data, so long as visible outputs of those programs do not improperly reveal information about the data and an attacker is assumed to be allowed to only access non-confidential information. For noninterference to hold, it is thus sufficient to demonstrate that the attacker cannot observe any difference between two program executions that differ only in their confidential data [Goguen and Meseguer 1984]. Several techniques for showing noninterference have been studied such as trace semantics [McLean 1992] or type systems [Heintze and Riecke 1998; Myers and Liskov 1997; Volpano et al. 1996; Volpano and Smith 1997]. Type system approaches have been very successful as they allow for some automatic security enforcement mechanism through static checking.

Type systems for noninterference have also had surprising and non-expected applications: in [Marion 2011], it is shown that noninterference can be used to guarantee polynomial time complexity properties of programs. In this context, confidential data corresponds to data that cannot control iteration (or recursion), while non-confidential data can. This new interpretation of security levels echoes the safe/normal separation of data (also known as safe recursion) in the work of [Bellantoni and Cook 1992], which provided the first tractable (*i.e.*, decidable in polynomial time)

Authors' address: Emmanuel Hainry, emmanuel.hainry@loria.fr; Romain Péchoux, romain.pechoux@loria.fr, Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France.

2475-1421/2023/1-ART28
https://doi.org/10.1145/3571221

characterization of the class of polynomial-time computable functions on a first-order programming language.

The notion of a type system has been a useful and efficient abstraction in the theory and practice of programming languages, since the very early days [Backus et al. 1963]. Types are assigned to program constructs and their purpose is to guarantee a kind of safety, *e.g.*, that well-typed programs cannot go wrong [Hughes et al. 1996] or terminate (in polynomial time) [Girard 1971; Reynolds 1974]. Noticeably, the converse property is very often wrong: there can be programs meeting the safety property which are not typable. This kind of incompleteness results is very likely to occur as the properties we want to ensure are most often non-trivial (undecidable) whereas the type systems aims at being automatable, thus decidable and preferably tractable. Hence, there is always a tension between expressiveness and tractability.

Concerning the class of functions computable in polynomial time, it is well known that the problem of checking whether a program computes a function of this class is Σ_2^0-complete in the arithmetical hierarchy, as shown in [Hájek 1979]. On the other hand, most of the type systems characterizing this class (*e.g.*, [Baillot and Mazza 2010; Baillot and Terui 2004; Marion 2011]) are tractable: type inference can be performed in polynomial time. There is thus a chasm between these two complexity results and the expressiveness of these type systems is very low in practice.

The main question we address in this paper is therefore whether it is possible to characterize polynomial time using noninterference without sacrificing the expressiveness.

Contributions. This paper's contributions can be summarized as follows:

- We introduce a new noninterference-based policy, named STR for "Stratified" (Definition 3.11), characterizing polynomial time computable functions. This policy uses countably many (security) levels $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$ and it clearly distinguishes for the first time the control flow policy NI from the stratification policy for complexity:
  - NI (Figure 5) is just a standard instance of noninterference-based type system à la Volpano [Volpano et al. 1996].
  - The stratification policy is implemented as follows: if a memory state reduces wrt the program semantics at level $\mathbf{n}$ then it cannot increase in size at memory addresses of level greater or equal to $\mathbf{n}$ (see Definition 3.11).
- Terminating programs in STR are sound (Theorem 3.13) and extensionally complete (Theorem 4.7) for polynomial time, *i.e.*, STR characterizes all functions of this complexity class.
- Moreover, we show that STR strictly encompasses the class of SAFE programs (Theorem 4.5): programs captured by safe recursion approaches ([Hainry and Péchoux 2018; Leivant and Marion 2013; Marion 2011]). This proper extension relies on the stratification property (Example 2.1) but also on the use of countably many levels by extending the many-level approach of [Hainry et al. 2022] to our setting (Example 3.15). We illustrate this proper extension by exhibiting several simple examples on inductive data, algorithms with destructive updating, and in-place algorithms.
- Verifying that a program is in STR is a hard problem, namely it is Π_1^0-complete (Theorem 4.8). This is proved by reducing the non-halting problem of Turing machines on the blank tape. Although not surprising, this is the first result of this kind in the literature. In particular, it implies that STR is not intensionally complete for polynomial time, *i.e.*, STR does not capture the full set of polynomial time algorithms known to be Σ_2^0-complete. However, this is not a major drawback because its complexity drops one level in the polynomial hierarchy: it is thus a better candidate for automation perspectives.
- Finally, we build on existing shape analysis techniques based on 3-valued logic [Reps et al. 2004] for inferring the shape of the memory and combine them into a new type system

named SA, for "Shape analysis" that implements both the noninterference and stratification policies. We show that SA is sound (Theorem 6.4) and (extensionally) complete (Theorem 6.6) for polynomial time, and that it also captures all the considered examples. We show that type inference for SA can be performed in exponential time (Theorem 6.7, the exponential worst case is due to the shape analysis), hence illustrating that decidable and expressive instances of STR can be designed.

Related Work. This work belongs to a family of techniques enforcing data stratification so that any doubling function cannot be programmed as an endomorphism on types. For example, for two levels, any program computing the double function on integers will have an input level **1** and an output level **0** (Hence it can be viewed as computing a function of type $\mathbf{1} \to \mathbf{0}$). This avoids any exponentiation by preventing such programs from being iterated. This data duality is at the root of many works studying the complexity of programming languages. It was introduced by cornerstone work on safe recursion [Bellantoni and Cook 1992] and ramified recurrence [Leivant and Marion 1993]. These function algebra can be viewed similarly to STR as noninterfering systems. Data duality can also be found in the linear logic based typing discipline for implicit complexity: soft linear logic [Lafont 2004], light linear logic [Girard 1998], and their variants [Baillot and Mazza 2010; Baillot and Terui 2004; Gaboardi et al. 2008], where the logical modalities play the role of levels. It is also very closely related to the "read-only/write-only" duality of [Jones 2001; Kop and Simonsen 2017] allowing to capture complexity hierarchies. The cons-free language of [Jones 2001] can easily be translated in STR using exclusively level **1** which would yield a characterization of polynomial time decision problems. The level **1** data are also reminiscent of the notion of non-size increasing [Hofmann 2002, 2003], a typing discipline ensuring that program computations do not increase more than linearly in size. Non-size-increasing technique [Hofmann 2003] is based on a non-predicative type system and, hence, cannot be compared to STR. In particular, contrarily to STR, this technique is neither sound nor complete for polynomial time. [Marion 2011] has shown for the first time that this data duality can be expressed in terms of a noninterference type-system. This analogy has been extended to polynomial space complexity [Hainry et al. 2013], graph languages [Leivant and Marion 2013], and object-oriented programs [Hainry and Péchoux 2018]. STR pursues this line of research by strictly extending the expressive power of these systems.

We are only aware of few works using shape analysis techniques or related formalisms for analyzing program complexity. [Manevich et al. 2016] proposes an algorithm for showing program termination in linear time and [Berdine et al. 2006] proposes an analysis for showing the termination of loops programs. The paper [Atkey 2010] is a related work extending amortized resource analysis to imperative programs using separation logics. However, it is concerned with inferring accurate complexity bounds and is hence not extensionally complete for polynomial time.

2 PRELIMINARIES: OBJECT ORIENTED PROGRAMS

This section introduces the syntax and semantics of a simple and generic typed Object Oriented programming language, which includes both reference and primitive types and which corresponds to a strict subset of Java programs.

2.1 Well-formed and Well-typed Programs

Let $\mathbb{V} \triangleq \{\mathsf{x}, \mathsf{y}, \mathsf{x}_1, \ldots\}$, $\mathbb{F} \triangleq \{\mathsf{a}, \mathsf{a}_1, \ldots\}$, $\mathbb{C} \triangleq \{\mathsf{C}, \mathsf{D}, \ldots\}$, and $\mathbb{O} \triangleq \{\mathsf{op}, \ldots\}$ be four disjoint and countable sets for variables, fields, class names, and operators (of fixed arity), respectively. The syntax of programs is provided by the grammar of Figure 1.

A *class* consists of a class name C, followed by a sequence of field declarations $(\tau\ \mathsf{a};)^*$.

$$
\begin{array}{lll}
\texttt{Identifiers} \ni \mathsf{s}, \ldots & ::= & \mathsf{x} \mid \mathsf{x.a} \\
\texttt{Expressions} \ni \mathsf{e}, \mathsf{e}_1, \ldots & ::= & \mathsf{s} \mid \mathsf{op}^{(\tau_1,\cdots,\tau_k)\to\pi}(\mathsf{e}_1,\cdots,\mathsf{e}_k) \mid \texttt{null} \mid \texttt{new C}(\mathsf{e}_1,\cdots,\mathsf{e}_n) \\
\texttt{Assignments} \ni \texttt{asg} & ::= & [\mathsf{s}^{\tau} := \mathsf{e}]; \\
\texttt{Statements} \ni \texttt{st} & ::= & \texttt{asg} \mid \texttt{st st} \mid \texttt{if(e)\{st\}else\{st\}} \mid \texttt{while(e)\{st\}} \\
\texttt{Classes} \ni \texttt{cl} & ::= & \texttt{C}\ (\tau\ \mathsf{a};)^* \\
\texttt{Programs} \ni \mathsf{p} & ::= & (\texttt{cl})^*\ (\tau\ \mathsf{x};)^*\ \mathsf{st}_\mathsf{p} \\
\texttt{Primitivetypes} \ni \pi & ::= & \texttt{int} \mid \texttt{bool} \\
\texttt{Referencetypes} \ni \rho & ::= & \texttt{C} \\
\texttt{Types} \ni \tau, \tau_1, \ldots & ::= & \pi \mid \rho
\end{array}
$$

Fig. 1. Programs and types

$$
\frac{(\tau\ \mathsf{x};)^*(\mathsf{x}_1) = \tau_1}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{x}_1 : \tau_1} \qquad \frac{(\tau\ \mathsf{x};)^*(\mathsf{x}_1) = \texttt{C} \qquad (\texttt{cl})^*(\texttt{C}, \mathsf{a}) = \tau_1}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{x}_1.\mathsf{a} : \tau_1}
$$

$$
\frac{}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \texttt{null} : \texttt{C}} \qquad \frac{\forall i \le n,\ (\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{e}_\mathsf{i} : (\texttt{cl})^*(\texttt{C}, \mathsf{a}_\mathsf{i})}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \texttt{new C}(\mathsf{e}_1, \ldots \mathsf{e}_\mathsf{n}) : \texttt{C}}
$$

$$
\frac{\forall i \le k,\ (\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{e}_\mathsf{i} : \tau_\mathsf{i}}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{op}^{(\tau_1,..,\tau_k)\to\pi}(\mathsf{e}_1,\cdots,\mathsf{e}_\mathsf{k}) : \pi} \qquad \frac{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{s} : \tau \qquad (\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{e} : \tau}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{s}^{\tau} := \mathsf{e};}
$$

$$
\frac{\forall i \le 2,\ (\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{st}_\mathsf{i}}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{st}_1\ \mathsf{st}_2} \qquad \frac{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{e} : \texttt{bool} \qquad \forall i \le 2,\ (\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{st}_\mathsf{i}}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \texttt{if(e)\{}\mathsf{st}_1\texttt{\}else\{}\mathsf{st}_2\texttt{\}}}
$$

$$
\frac{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \mathsf{e} : \texttt{bool} \qquad (\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \texttt{st}}{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \texttt{while(e)\{st\}}} \qquad \frac{(\tau\ \mathsf{x};)^*, (\texttt{cl})^* \vdash \texttt{st}}{\vdash (\texttt{cl})^*\ (\tau\ \mathsf{x};)^*\ \texttt{st}}
$$

Fig. 2. Well-typed programs

A *program* p consists of a sequence of classes $(\texttt{cl})^*$, a sequence of variable declarations $(\tau\ \mathsf{x};)^*$, and a statement st_p. Throughout the paper, variables b, b' may refer to any syntactical element. $[b]$ denotes an optional element. $b \in b'$ holds whenever b appears in b'. Given a set $\mathbb{K}$, let $\mathbb{K}(b)$ denote the set $\{b' \mid b' \in b\} \cap \mathbb{K}$. For example, $\mathbb{V}(\mathsf{p})$ is the set of variables of the program p and $\mathbb{F}(\texttt{cl})$ is the set of fields of the class cl. Each statement $\texttt{st} \in \mathsf{st}_\mathsf{p}$ is uniquely determined by a label l in the countable set $\mathbb{L}$. In a program, we write $l : \texttt{st}$ to mean that l is the label of st. Let also $\texttt{st}(l)$ denote the statement at label l in p. Labels will be omitted unless their use is explicitly required.

Throughout the paper, only *well-formed* programs will be considered. Well-formed programs follow some standard conditions ensuring syntactical correctness: class constructors and operators are always fully applied; all variables, fields, and classes of a program statement have to be declared in the right place; there are no name clashes. In order to avoid superfluous technicality, the considered programs do not include methods and inheritance, to which the presented results can be extended easily.

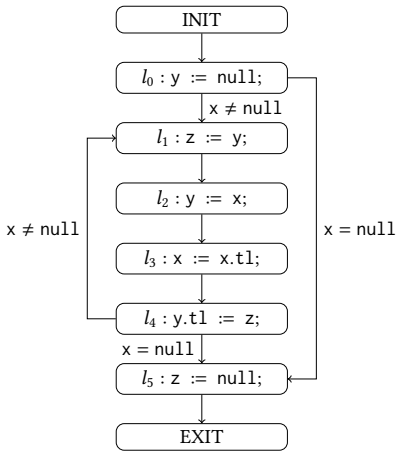
Types $\tau, \tau_1, \ldots \in \mathbb{T}$ are either reference types $\texttt{C}, \texttt{D}, \ldots$ (*i.e.*, class names) or primitive types $\pi \in \{\texttt{int}, \texttt{bool}\}$. The type bool is inhabited by the two constants 0 and 1 and int is the type of positive integers. The notation x^{τ} (resp. $\mathsf{x.a}^{\tau}$) will sometimes be used to explicitly mention the type τ of

variable x (resp. field selector x.a). Each operator $\mathsf{op}^{(\tau_1,\cdots,\tau_k)\to\pi}$ of arity k comes with some fixed type signature $(\tau_1,\cdots,\tau_k)\to\pi$, which may be omitted when clear from the context.

We will also restrict our analysis to *well-typed* programs, that are programs p for which the judgment $\Vdash \mathsf{p}$ can be derived using the rules of Figure 2. This type discipline ensures that there is no typing mismatch in program assignments and expressions of a well-typed program. In Figure 2, the notations $(\tau\ \mathsf{x};)^*(\mathsf{x}_1)$ and $(\mathtt{cl})^*(\mathsf{C},\mathsf{a}_1)$ are defined by $(\tau\ \mathsf{x};)^*(\mathsf{x}_1)\triangleq\tau_1$, whenever $\tau_1\ \mathsf{x}_1;\in(\tau\ \mathsf{x};)^*$ holds, and $(\mathtt{cl})^*(\mathsf{C},\mathsf{a}_1)\triangleq\tau_1$, whenever both $\mathsf{C}\ (\tau\ \mathsf{a};)^*\in(\mathtt{cl})^*$ and $\tau_1\ \mathsf{a}_1;\in(\tau\ \mathsf{a};)^*$ hold.

Example 2.1. The program rev below provides an illustration of a well-formed and well-typed program and will be our leading example throughout the paper. This program reverses a linked list of integers.

```
     List {int hd; List tl}
     List x; List y; List z;
l0 : y := null;
it : while (x ≠ null){
l1 :   z := y;
l2 :   y := x;
l3 :   x := x.tl;
l4 :   y.tl := z;
     }
l5 : z := null;
```



2.2 Memory Heap

For any type τ, let $\mathcal{V}_\tau$ be an infinite (countable) set of *memory addresses* such that $\tau\neq\tau'$ implies $\mathcal{V}_\tau\cap\mathcal{V}_{\tau'}=\{\bot\}$, with $\bot$ being the null memory address such that $\bot\notin\mathcal{V}_\pi$ and $\bot\in\mathcal{V}_\mathsf{C}$, for each $\mathsf{C}\in\mathbb{C}$. Let $\mathbb{W}$ be the set of words over a fixed alphabet Σ such that $\{\mathtt{0},\mathtt{1}\}\subseteq\mathbb{W}$. We fix an interpretation of types as follows $[\![\mathtt{int}]\!]\triangleq\mathbb{W}$, $[\![\mathtt{bool}]\!]\triangleq\{\mathtt{0},\mathtt{1}\}$, and $[\![\mathsf{C}]\!]\triangleq\mathcal{V}_\mathsf{C}$. We have chosen to encode integers as words in order to allow for several data representations (unary, binary, ...).
A *memory graph* is a labeled multidigraph $\mathcal{G}_\mathsf{p}=(\Sigma_V,\Sigma_A,V,A,s,t,l_V,l_A)$, s.t.:

- $V\subseteq\cup_{\tau\in\mathbb{T}(\mathsf{p})}\mathcal{V}_\tau$ is the set of vertices and $A\subseteq\mathcal{M}(V\times V)$ is the multiset of arrows;
- $\Sigma_V\triangleq\mathbb{W}\cup\mathbb{C}(\mathsf{p})\cup\{\bot\}$ is the set of vertex labels, $\Sigma_A\triangleq\mathbb{F}(\mathsf{p})$ is the set of arrow labels, and $l_X:X\to\Sigma_X, X\in\{A,V\}$, are labeling maps;
- and $s,t:A\to V$ give the source and target vertex of an arrow, respectively.

Some constraints are put on memory graphs so that they provide a structurally correct representation of the memory. For each $v\in V$:

- if $v\in\mathcal{V}_\mathtt{int}\cup\mathcal{V}_\mathtt{bool}\cup\{\bot\}$ then v cannot be the source of any arrow.
- if $v\in\mathcal{V}_\mathsf{C}-\{\bot\}$, for the class $\mathsf{C}\ \tau_1\ \mathsf{a}_1;\dots\tau_n\ \mathsf{a}_n;$, then there are exactly n arrows $a_1,\dots,a_n$ such that $s(a_i)=v$, $l_A(a_i)=\mathsf{a}_i$, and $t(a_i)\in\mathcal{V}_{\tau_i}$.
- If $v\in\mathcal{V}_\pi$ then $l_V(v)\in[\![\pi]\!]$. If $v\in\mathcal{V}_\mathsf{C}-\{\bot\}$ then $l_V(v)=\mathsf{C}$ and $l_V(\bot)=\bot$.

A memory graph satisfying the above requirements is said to be *well-shaped*. A memory graph is *deterministic* if $\forall v\in V,\forall\mathsf{a}\in\Sigma_A,\#\{a\in A\mid s(a)=v\wedge l_A(a)=\mathsf{a}\}\leq 1$.

A memory heap $\mathcal{H}_\mathsf{p}$ of a given program p is a pair $(\mathcal{G}_\mathsf{p},f_\mathsf{p})$ consisting of a memory graph $\mathcal{G}_\mathsf{p}$ and a partial map $f_\mathsf{p}:\mathbb{V}(\mathsf{p})\to V$ of domain $dom(f_\mathsf{p})$. In general, we will only consider memory

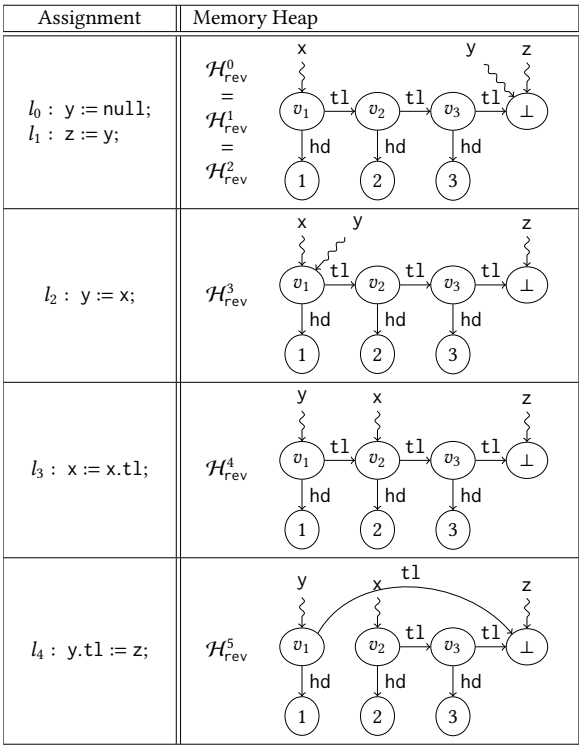


Fig. 3. Sequence of memory heaps of the program rev

heap whose memory graph is deterministic and well-shaped: let HEAP_p be the set of such memory heaps for the program p. We will use a graphical representation of memory heaps where vertices and arrows of the memory graph are annotated by their label. The partial map in $\mathbb{V}(\mathsf{p}) \to V$ is represented by snake arrows. The notion of memory heap and the corresponding graphical representation can be viewed as a generalization of the concept of (concrete) shape-graph introduced in [Sagiv et al. 1996].

Example 2.2. The memory heap $\mathcal{H}^0_{\text{rev}}$ whose graphical representation is provided in Figure 3 encodes a list of 3 integers $[1, 2, 3]$ corresponding to three memory addresses $v_1, v_2, v_3 \in \mathcal{V}_{\mathtt{List}}$ pointed to by variable x. Variables y and z point to the null memory address $\perp$. This memory heap is deterministic and well-shaped and, consequently, $\mathcal{H}^0_{\text{rev}} \in \mathsf{HEAP}_{\mathtt{rev}}$.

Let Err be a special symbol for faults. For any set S, $S^{\mathtt{Err}} \triangleq S \cup \{\mathtt{Err}\}$. For a given memory heap $\mathcal{H}_\mathsf{p} = (\mathcal{G}_\mathsf{p}, f_\mathsf{p})$, define $\mathcal{H}_\mathsf{p}(\mathsf{x}) \triangleq f_\mathsf{p}(\mathsf{x})$, if $\mathsf{x} \in dom(f_\mathsf{p})$, and $\mathcal{H}_\mathsf{p}(\mathsf{x}) \triangleq \mathtt{Err}$ otherwise. Define also $\mathcal{H}_\mathsf{p}(\mathsf{x.a}) \triangleq v$, if $\mathcal{H}_\mathsf{p}(\mathsf{x}) = v'$, $a \in \mathcal{G}_\mathsf{p}$, $s(a) = v'$, $t(a) = v$, and $l_A(a) = \mathsf{a}$. Otherwise $\mathcal{H}_\mathsf{p}(\mathsf{x.a}) \triangleq \mathtt{Err}$. Finally, for $v \in V$, $\mathcal{H}_\mathsf{p}(v) \triangleq l_V(v)$.

Let $\mathcal{H}_\mathsf{p}^{-\mathsf{x}}$ be the memory heap obtained from $\mathcal{H}_\mathsf{p}$ by removing x from $dom(f_\mathsf{p})$. Let also $\mathcal{H}_\mathsf{p}^{-\mathsf{x.a}}$ be the memory heap obtained from $\mathcal{H}_\mathsf{p}$ by removing the arrow in $\mathcal{G}_\mathsf{p}$ of source $f_\mathsf{p}(\mathsf{x})$ and label a. This operation can break the well-shapedness of memory heaps. $\mathcal{H}_\mathsf{p} \cup \{(v, \mathsf{a}, v')\}$ is the memory heap obtained from $\mathcal{H}_\mathsf{p}$ by adding the arrow (v, v') labeled by a in $\mathcal{G}_\mathsf{p}$. This operation can break the determinism of memory heaps. $\mathcal{H}_\mathsf{p} \cup \{(\mathsf{x}, v)\}$ is the memory heap obtained from $\mathcal{H}_\mathsf{p}$ by updating f_p such that $f_\mathsf{p}(\mathsf{x}) = v$ holds. Finally, $\mathcal{H}_\mathsf{p}[v := l]$ is the memory heap obtained from $\mathcal{H}_\mathsf{p}$ by updating the

label of memory address v to l in $\mathcal{G}_\mathsf{p}$. These notations are lifted to the fault in such a way that $\mathtt{Err}$ is always an absorbing element. *e.g.*, $\mathtt{Err}^{-\mathsf{x}} = \mathtt{Err}^{-\mathsf{x.a}} = \mathcal{H}_\mathsf{p} \cup \{[\mathsf{x}, \mathtt{Err}]\} = \mathcal{H}_\mathsf{p}[v := \mathtt{Err}] \triangleq \mathtt{Err}$.

2.3 Semantics of Assignments

Each operator $\mathsf{op}^{\tau_1 \times \cdots \times \tau_k \to \pi}$ computes a total function $[\![\mathsf{op}]\!] : [\![\tau_1]\!]^{\mathtt{Err}} \times \cdots \times [\![\tau_k]\!]^{\mathtt{Err}} \to [\![\pi]\!]^{\mathtt{Err}}$ fixed by the language implementation returning either a primitive value in $[\![\pi]\!]$ or a fault and such that $\mathtt{Err}$ is an absorbing element, *i.e.*, $[\![\mathsf{op}]\!](\ldots, \mathtt{Err}, \ldots) = \mathtt{Err}$. For example, the operator $\neq^{\mathtt{List} \times \mathtt{List} \to \mathtt{bool}}$ of Example 2.1 is associated with the function $[\![\neq]\!] \in \mathcal{V}^{\mathtt{Err}}_{\mathtt{List}} \times \mathcal{V}^{\mathtt{Err}}_{\mathtt{List}} \to \{0,1\}^{\mathtt{Err}}$ that computes the inequality of memory addresses. In the particular case where $k = 0$, the operator $\mathsf{op}^{() \to \pi}()$ represents a constant. Let $\langle \cdot, \cdot \rangle$ be a constructor for pairs. $\mathtt{fst}$ and $\mathtt{snd}$ will denote the first and second projectors on pairs. Consider the monad $(M, \mathtt{return}, \gg\!=)$, where:

- M is a monadic type associating the type $M(t) \triangleq \mathtt{HEAP}_\mathsf{p}^{\mathtt{Err}} \to t \times \mathtt{HEAP}_\mathsf{p}^{\mathtt{Err}}$ to the type t,
- $\mathtt{return}$ is a function of signature $t \to M(t)$ such that if a is of type t then $\mathtt{return}\ a \triangleq \lambda \mathcal{H}_\mathsf{p}.\langle a, \mathcal{H}_\mathsf{p} \rangle$ is of type $M(t)$,
- $\gg\!=$ is a function of signature $M(t) \to (t \to M(u)) \to M(u)$ defined by $(a \gg\!= b) \triangleq \lambda \mathcal{H}_\mathsf{p}.((b\ \mathtt{fst}(a\ \mathcal{H}_\mathsf{p}))\ \mathtt{snd}(a\ \mathcal{H}_\mathsf{p}))$.

Let $\mathtt{Unit}$ be a type inhabited by the single element $\mathtt{void}$. It can be considered as equivalent to the Java type $\mathtt{Void}$. The set of values is defined by $\mathtt{Values} \triangleq \mathbb{W} \cup (\cup_\rho \mathcal{V}_\rho)$. Hence each function $[\![\mathsf{op}]\!]$ can be viewed as a partial function in $(\mathtt{Values}^{\mathtt{Err}})^k \to \mathtt{Values}^{\mathtt{Err}}$. For a fixed memory heap $\mathcal{H}_\mathsf{p}$, we define the valuation function $val_{\mathcal{H}_\mathsf{p}} : \cup_\tau \mathcal{V}_\tau \to \mathtt{Values}$ by $val_{\mathcal{H}_\mathsf{p}}(v) \triangleq v$, if $v \in \mathcal{V}_\mathsf{C}$, and $val_{\mathcal{H}_\mathsf{p}}(v) \triangleq l_V(v)$, if $v \in \mathcal{V}_\pi$. We will simply write $val(v)$ when the memory heap $\mathcal{H}_\mathsf{p}$ is clear from the context. The concrete semantics maps every expression e to $[\![\mathsf{e}]\!] : M(\mathtt{Values}^{\mathtt{Err}})$, maps every assignment asg to $[\![\mathsf{asg}]\!] : M(\mathtt{Unit})$, and is described in Figure 4a, where $\mathtt{let}\ \langle x_1, \_ \rangle = a_1, \ldots, \langle x_n, \_ \rangle = a_n\ \mathtt{in}\ v$ is a shorthand notation for $a_1 \gg\!= \lambda x_1.(a_2 \gg\!= \lambda x_2.(\ldots a_n \gg\!= \lambda x_n.(v)\ldots)$. The semantics is well-defined as it preserves both the well-shapedness and determinism of memory graphs. For simplicity, we have just provided the case of a new instance with no operand $\mathtt{new\ C()}$. The general case can be derived by combining this rule together with rules for expressions.

As in Java, the semantics described in Figure 4a is pass-by-value on primitive types and pass-by-reference on reference types. The semantics can fail in a way similar to the semantics of [Bouajjani et al. 2011] on linked lists. In our setting, faults correspond to Java exceptions, including null pointer exceptions.

Example 2.3. Consider one iteration of the while statement in the program $\mathtt{rev}$ of Example 2.1, that is, consider the sequence of assignments $\mathtt{st}(l_1), \ldots, \mathtt{st}(l_4)$, wrt the memory heaps $\mathcal{H}^i_{\mathtt{rev}}$ described in Figure 3. For any i, $1 \leq i \leq 4$, we have $[\![\mathtt{st}(l_i)]\!](\mathcal{H}^i_{\mathtt{rev}}) = \langle \mathtt{void}, \mathcal{H}^{i+1}_{\mathtt{rev}} \rangle$.

2.4 Semantics of Programs

Let E denote a terminal symbol. The set of *memory configurations* of a program p is defined by $\mathtt{Conf} \triangleq (\mathtt{Statements} \cup \{E\}) \times \mathtt{HEAP}_\mathsf{p}^{\mathtt{Err}}$. For a given label $l \in \mathbb{L}(\mathsf{p})$, the semantics $\mapsto_l$ of a program p is a partial function in $\mathtt{Conf} \to \mathtt{Conf}$ defined in Figure 4b, where it is implicitly assumed that the equality $E\ \mathtt{st} = \mathtt{st}$ holds, for any statement $\mathtt{st}$.

Define $\mapsto \triangleq \cup_{l \in \mathbb{L}(\mathsf{p})} \mapsto_l$. Let $\mapsto^*$ be the reflexive and transitive closure of $\mapsto$ and let $\mapsto^k$ be the k-fold composition of $\mapsto$. A program p computes the partial function $[\![\mathsf{p}]\!] : (\mathtt{HEAP}_\mathsf{p}^{\mathtt{Err}}) \to \mathtt{HEAP}_\mathsf{p}^{\mathtt{Err}}$ defined by $[\![\mathsf{p}]\!](\mathcal{H}_\mathsf{p}) = \mathcal{H}'_\mathsf{p}$ if $\langle \mathtt{st}_\mathsf{p}, \mathcal{H}_\mathsf{p} \rangle \mapsto^* \langle E, \mathcal{H}'_\mathsf{p} \rangle$. In the special case where $[\![\mathsf{p}]\!]$ is a total function in $\mathtt{HEAP}_\mathsf{p} \to \mathtt{HEAP}_\mathsf{p}$, p is called a terminating program (hence terminating programs are fault-free). Let $\mathtt{TERM}$ be the set of terminating programs. The orbit of a configuration c is the set of reachable configurations from c defined by $O(c) \triangleq \{c' \in \mathtt{Conf} \mid c \mapsto^* c'\}$. The state space of the program p is the set of all reachable configurations and is defined by $\mathcal{S}(\mathsf{p}) \triangleq \cup_{\mathcal{H}_\mathsf{p}} O(\langle \mathtt{st}_\mathsf{p}, \mathcal{H}_\mathsf{p} \rangle)$.

$$
\begin{aligned}
[\![\mathsf{x}]\!] &\triangleq \mathtt{return}\ \mathcal{H}_{\mathsf{p}}(\mathsf{x})\\
[\![\mathsf{x.a}]\!] &\triangleq \mathtt{return}\ \mathcal{H}_{\mathsf{p}}(\mathsf{x.a})\\
[\![\mathtt{null}]\!] &\triangleq \mathtt{return}\ \bot\\
[\![\mathtt{new\ C()}]\!] &\triangleq \lambda\mathcal{H}_{\mathsf{p}}.\langle v, \mathcal{H}_{\mathsf{p}} \cup \{(v,\mathsf{a}_i,\bot)\}_{i\in I} \cup \{(v,\mathsf{a}_j,v_j)\}_{j\in J}[v:=v, v_j:=\mathtt{0}]_{j\in J}\rangle,\\
&\qquad \text{for the class C } (\rho_i\ \mathsf{a}_i)_{i\in I}\ (\pi_j\ \mathsf{a}_j)_{j\in J} \text{ and fresh addresses } v\in\mathcal{V}_{\mathsf{C}},\ v_j\in\mathcal{V}_{\pi_j}.\\
[\![\mathsf{op}(\mathsf{e}_1,\dots,\mathsf{e}_k)]\!] &\triangleq \mathtt{let}\ \ \dots,\langle x_i,\_\rangle = [\![\mathsf{e}_i]\!],\dots\ \ \mathtt{in\ return}\ [\![\mathsf{op}]\!](val(x_1),\dots,val(x_k))\\
[\![\mathsf{x}^{\mathsf{C}}\!:=\mathsf{e};]\!] &\triangleq \mathtt{let}\ \langle y,\_\rangle = [\![\mathsf{e}]\!]\ \mathtt{in}\ \lambda\mathcal{H}_{\mathsf{p}}.\langle \mathtt{void}, \mathcal{H}_{\mathsf{p}}^{-\mathsf{x}} \cup \{(\mathsf{x},y)\}\rangle\\
[\![\mathsf{x.a}^{\mathsf{C}}\!:=\mathsf{e};]\!] &\triangleq \mathtt{let}\ \langle x,\_\rangle = [\![\mathsf{x}]\!], \langle y,\_\rangle = [\![\mathsf{e}]\!]\ \mathtt{in}\ \lambda\mathcal{H}_{\mathsf{p}}.\langle \mathtt{void}, \mathcal{H}_{\mathsf{p}}^{-\mathsf{x.a}} \cup \{(x,\mathsf{a},y)\}\rangle\\
[\![\mathsf{s}^{\pi}\!:=\mathsf{e};]\!] &\triangleq \mathtt{let}\ \langle x,\_\rangle = [\![\mathsf{s}]\!], \langle y,\_\rangle = [\![\mathsf{e}]\!]\ \mathtt{in}\ \lambda\mathcal{H}_{\mathsf{p}}.\langle \mathtt{void}, \mathcal{H}_{\mathsf{p}}[x:=val(y)]\rangle
\end{aligned}
$$

(a) Semantics of expressions and assignments

$$
\begin{aligned}
\langle l:\mathtt{asg}, \mathcal{H}_{\mathsf{p}}\rangle &\mapsto_l \langle E, \mathsf{snd}([\![\mathtt{asg}]\!](\mathcal{H}_{\mathsf{p}}))\rangle\\
\langle \mathtt{st}_1\ \mathtt{st}_2, \mathcal{H}_{\mathsf{p}}\rangle &\mapsto_l \langle \mathtt{st}_1'\ \mathtt{st}_2, \mathcal{H}_{\mathsf{p}}'\rangle \qquad \text{if } \langle \mathtt{st}_1, \mathcal{H}_{\mathsf{p}}\rangle \mapsto_l \langle \mathtt{st}_1', \mathcal{H}_{\mathsf{p}}'\rangle\\
\langle l:\mathtt{if(e)\{st_1\}else\{st_0\}}, \mathcal{H}_{\mathsf{p}}\rangle &\mapsto_l \langle \mathtt{st}_{\mathtt{fst}([\![\mathsf{e}]\!](\mathcal{H}_{\mathsf{p}}))}, \mathcal{H}_{\mathsf{p}}\rangle\\
\langle l:\mathtt{while(e)\{st\}}, \mathcal{H}_{\mathsf{p}}\rangle &\mapsto_l \langle E, \mathcal{H}_{\mathsf{p}}\rangle \qquad \text{if } \mathtt{fst}([\![\mathsf{e}]\!]) = \mathtt{0}\\
\langle l:\mathtt{while(e)\{st\}}, \mathcal{H}_{\mathsf{p}}\rangle &\mapsto_l \langle \mathtt{st\ while(e)\{st\}}, \mathcal{H}_{\mathsf{p}}\rangle \qquad \text{if } \mathtt{fst}([\![\mathsf{e}]\!]) = \mathtt{1}
\end{aligned}
$$

(b) Semantics of configurations

Fig. 4. Program semantics

Set $|\mathsf{C}| = |\bot| \triangleq 1$ and let $|w|$ be the length of a word $w \in \mathbb{W}$. The size of a memory heap is defined by $|\mathcal{H}_{\mathsf{p}}| \triangleq \sum_{v\in V} |l_V(v)|$. A program p is in POLY if there is a univariate polynomial $P \in \mathbb{N}[X]$ such that $\forall \mathcal{H}_{\mathsf{p}} \in \mathsf{HEAP}_{\mathsf{p}},\ \exists \mathcal{H}_{\mathsf{p}}' \in \mathsf{HEAP}_{\mathsf{p}},\ \exists k \in \mathbb{N},\ \langle \mathtt{st}_{\mathsf{p}}, \mathcal{H}_{\mathsf{p}}\rangle \mapsto^k \langle E, \mathcal{H}_{\mathsf{p}}'\rangle$ and $k \le P(|\mathcal{H}_{\mathsf{p}}|)$. By definition, it holds that POLY $\subsetneq$ TERM.

3 NONINTERFERENCE FOR COMPLEXITY ANALYSIS

In this section, we provide a new insight on the use of noninterference techniques as a proof method for verifying polynomial time complexity of programs.

3.1 Positive Operators

We start by restricting the class of considered operators depending on the total function they compute. Indeed, allowing for too powerful operators may lead programs to compute exponential functions (or even worse). We will only consider *positive operators*, polynomial time computable operators whose computation may make the memory size increase by at most some constant.

Definition 3.1. An operator $\mathsf{op}^{(\tau_1,\cdots,\tau_k)\to\pi}$ is *positive* if:

- the function $[\![\mathsf{op}]\!]$ is computable in polynomial time by a Turing machine,
- there is a constant $c_{\mathsf{op}} \in \mathbb{N}$ such that:
  $\forall (w_1,\dots,w_k) \in [\![\tau_1]\!] \times \dots \times [\![\tau_k]\!],\ |[\![\mathsf{op}]\!](w_1,\dots,w_k)| \le \max_{1\le i\le k} |w_i| + c_{\mathsf{op}}$.

The operator $\neq^{\mathtt{List\times List\to bool}}$ of Example 2.1 is positive as it outputs a value of type bool. Indeed, setting $c_{\neq} = 1$, it holds that $\forall w, w' \in \mathcal{V}_{\mathtt{List}},\ |[\![\mathsf{op}]\!](w,w')| \le \max(|\mathtt{0}|,|\mathtt{1}|) = c_{\neq}$. Throughout the paper, we will only consider positive operators.

$$\frac{\Gamma(\mathtt{s}) = \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{s} : \mathbf{n}}\ (\mathrm{Id}) \qquad \frac{\forall i \leq |\overline{\mathtt{e}}|,\ \Gamma \vdash_{\mathtt{NI}} \mathtt{e}_i : \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{op}(\overline{\mathtt{e}}) : \mathbf{n}}\ (\mathrm{Op}) \qquad \frac{}{\Gamma \vdash_{\mathtt{NI}} \mathtt{null} : \mathbf{n}}\ (\mathrm{Null})$$

$$\frac{\forall i \leq |\overline{\mathtt{e}}|,\ \Gamma \vdash_{\mathtt{NI}} \mathtt{e}_i : \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathbf{n}}\ (\mathrm{New}) \qquad \frac{\Gamma \vdash_{\mathtt{NI}} \mathtt{e} : \mathbf{n} \quad \mathbf{m} \preceq \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{e} : \mathbf{m}}\ (\mathrm{SubE})$$

$$\frac{}{\Gamma \vdash_{\mathtt{NI}} \mathtt{;} : \mathbf{0}}\ (\mathrm{Skip}) \qquad \frac{\Gamma(\mathtt{s}) = \mathbf{n} \quad \Gamma \vdash_{\mathtt{NI}} \mathtt{e} : \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{s := e;} : \mathbf{n}}\ (\mathrm{Asg})$$

$$\frac{\Gamma \vdash_{\mathtt{NI}} \mathtt{st}_1 : \mathbf{n} \quad \Gamma \vdash_{\mathtt{NI}} \mathtt{st}_2 : \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{st}_1\ \mathtt{st}_2 : \mathbf{n}}\ (\mathrm{Seq}) \qquad \frac{\Gamma \vdash_{\mathtt{NI}} \mathtt{e} : \mathbf{n} \quad \Gamma \vdash_{\mathtt{NI}} \mathtt{st}_1 : \mathbf{n} \quad \Gamma \vdash_{\mathtt{NI}} \mathtt{st}_2 : \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{if(e)\{st_1\}else\{st_2\}} : \mathbf{n}}\ (\mathrm{Cond})$$

$$\frac{\Gamma \vdash_{\mathtt{NI}} \mathtt{e} : \mathbf{n} \quad \Gamma \vdash_{\mathtt{NI}} \mathtt{st} : \mathbf{n} \quad \mathbf{1} \preceq \mathbf{n}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{while(e)\{st\}} : \mathbf{n}}\ (\mathrm{Iter}) \qquad \frac{\Gamma \vdash_{\mathtt{NI}} \mathtt{st} : \mathbf{n} \quad \mathbf{n} \preceq \mathbf{m}}{\Gamma \vdash_{\mathtt{NI}} \mathtt{st} : \mathbf{m}}\ (\mathrm{Sub})$$

Fig. 5. Type system for noninterfering programs (NI)

3.2 Noninterference-based Type System

Levels are elements of the totally ordered set $(\mathbf{N}, \preceq, \mathbf{0}, \sqcup, \sqcap)$ of positive integers; $\preceq$ ($\prec$) being the standard (strict) ordering on integers, and $\sqcup$ and $\sqcap$ being the max and min operators. We use the symbols $\mathbf{n}, \mathbf{m}, \mathbf{n}_1, \mathbf{n}_2, \ldots$ to denote level variables. For a finite set of levels $\{\mathbf{n}_1, \ldots, \mathbf{n}_k\}$, let $\sqcup_{i=1}^{k} \mathbf{n}_i$ ($\sqcap_{i=1}^{k} \mathbf{n}_i$, respectively) denote $\mathbf{n}_1 \sqcup \ldots \sqcup \mathbf{n}_k$ ($\mathbf{n}_1 \sqcap \ldots \sqcap \mathbf{n}_k$, respectively).

A *typing environment* Γ of program $\mathtt{p}$ is a total function in $\mathbb{V}(\mathtt{p}) \to \mathbf{N}$ associating a level $\Gamma(\mathtt{x})$ to each variable $\mathtt{x}$ of $\mathtt{p}$. Typing environment are extended to field selectors by $\Gamma(\mathtt{x.a}) \triangleq \Gamma(\mathtt{x})$. *Typing judgments* are of the form $\Gamma \vdash_{\mathtt{NI}} b : \mathbf{n}$, for some level $\mathbf{n}$ and some expression or statement b. $\mathbf{n}$ is called the *level of* b and we say that b is a level-$\mathbf{n}$ expression/statement. A *typing derivation* $\nabla_{\Gamma}^{b:\mathbf{n}}$ is a tree of root $\Gamma \vdash_{\mathtt{NI}} b : \mathbf{n}$, whose children nodes are obtained using the rules of Figure 5, and whose leaves are obtained using Rules (Id), (Null), and (Skip) (and exceptionally Rules (Op) or (New), when there are no operands). Notice that there are judgments which do not have any typing derivation. Moreover, there is in general no uniqueness for derivations. $\nabla_{\Gamma}^{b':\mathbf{n}'} \sqsubseteq \nabla_{\Gamma}^{b:\mathbf{n}}$ holds if $\nabla_{\Gamma}^{b':\mathbf{n}'}$ is a sub-tree of $\nabla_{\Gamma}^{b:\mathbf{n}}$.

Definition 3.2 (Noninterference). Given a typing environment $\Gamma : \mathbb{V}(\mathtt{p}) \to \mathbf{N}$, a program $\mathtt{p}$ is *noninterfering* wrt Γ if there is a level $\mathbf{n} \in \mathbf{N}$ such that $\nabla_{\Gamma}^{\mathtt{st_p}:\mathbf{n}}$ can be derived. Let $\mathtt{NI}_{\Gamma}$ be the set of noninterfering programs wrt Γ.

We say that $\mathtt{p} \in \mathtt{NI}_{\Gamma}$ wrt $\nabla_{\Gamma}^{\mathtt{st_p}:\mathbf{n}}$ when we want to make the typing derivation explicit.

Example 3.3. Consider the program `rev` of Example 2.1. `rev` is a noninterfering program in $\mathtt{NI}_{\Gamma}$ for the typing assignment $\Gamma : \mathbb{V}(\mathtt{rev}) \to \mathbf{N}$ defined by $\Gamma(\mathtt{x}) \triangleq \mathbf{1}$ and $\Gamma(\mathtt{y}) = \Gamma(\mathtt{z}) \triangleq \mathbf{0}$. $\Gamma \vdash_{\mathtt{NI}} \mathtt{st}_{\mathtt{rev}} : \mathbf{1}$ can be derived using rules of Figure 5.

Note that `rev` cannot be typed wrt a typing environment Γ such that $\Gamma(\mathtt{x}) = 0$ or $\Gamma(\mathtt{x}) \prec \Gamma(\mathtt{y})$.

We now give more intuition to the reader on the constraints enforced by the typing rules of Figure 5. Most of the rules are basic noninterference typing rules following Volpano *et al.* type discipline [Volpano et al. 1996]: levels in the rule premises (when there is one) are equal to the level in the rule conclusion so that there can be no information flow (in both directions) using these rules. Rules (Id), (Op), (Null), (New) and (SubE) ensure that the level is structurally decreasing for

expressions. Rule (Asg) relates the level of a variable to the level of the corresponding assignment, while enforcing a control flow policy: no expression can be assigned to a variable of strictly greater level. In this rule, the level of the identifier is directly taken from the typing environment in order to prevent expression subtyping (Rule (SubE)). Indeed, this would break the control flow policy. Rules (Skip), (Asg), (Seq), (Cond), (Iter), and (Sub) ensure that the level is structurally increasing for statements. Moreover, the inequality of Rule (Iter) implies that there cannot be any iteration guarded by a level-$\mathbf{0}$ expression. Hence level-$\mathbf{0}$ statements will correspond to constant time computations. It is good to interpret a level-$\mathbf{n}$ statement as a statement having strictly more computational power than a level-$\mathbf{m}$ statement, for $\mathbf{m} \prec \mathbf{n}$. At this very moment, however, type system NI does not ensure any complexity property.

3.3 Properties of Noninterfering Programs

Programs in $\mathtt{NI}_\Gamma$ enjoy some standard noninterference properties. First, an expression can only access variables of higher level.

Lemma 3.4 (Simple security). *Given a program* $\mathsf{p} \in \mathtt{NI}_\Gamma$ *wrt the typing derivation* $\nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$*, for any* $\nabla_\Gamma^{\mathsf{e}:\mathbf{m}}$*, if* $\nabla_\Gamma^{\mathsf{e}:\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$ *then* $\mathbf{m} \preceq \sqcap_{\mathsf{y}\in\mathbb{V}(\mathsf{e})}\Gamma(\mathsf{y})$.

Proof. By structural induction on expressions using rules of Figure 5. □

Second, the level is structurally monotonic in the statements.

Lemma 3.5 (Monotonicity). *Given a program* $\mathsf{p} \in \mathtt{NI}_\Gamma$ *wrt the typing derivation* $\nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$*, for any* $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}}$ *and* $\nabla_\Gamma^{\mathsf{st}(l'):\mathbf{m}'}$ *such that* $\nabla_\Gamma^{\mathsf{st}(l'):\mathbf{m}'} \sqsubseteq \nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$*, if* $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathsf{st}(l'):\mathbf{m}'}$ *then* $\mathbf{m} \preceq \mathbf{m}'$.

Proof. By monotonicity of the level in the rules of Figure 5 for statements. □

Third, a $\mathtt{NI}_\Gamma$ program wrt the typing derivation $\nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$ has the property that any statement can only be controlled by data of level greater or equal.

Proposition 3.6 (Hierarchical control flow). *Given a program* $\mathsf{p} \in \mathtt{NI}_\Gamma$ *wrt the typing derivation* $\nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$*, for any* $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}}$*, if*

- *either there exists* $\nabla_\Gamma^{\mathtt{if(e)\{st_1\}else\{st_2\}}:\mathbf{m}'}$ *such that* $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathtt{if(e)\{st_1\}else\{st_2\}}:\mathbf{m}'} \sqsubseteq \nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$,
- *or there exists* $\nabla_\Gamma^{\mathtt{while(e)\{st\}}:\mathbf{m}''}$ *such that* $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathtt{while(e)\{st\}}:\mathbf{m}''} \sqsubseteq \nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$,

then $\mathbf{m} \preceq \sqcap_{\mathsf{y}\in\mathbb{V}(\mathsf{e})}\Gamma(\mathsf{y})$.

Proof. Assume that $\mathsf{p} \in \mathtt{NI}_\Gamma$, wrt the typing derivation $\nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$. In the case where $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathtt{if(e)\{st_1\}else\{st_2\}}:\mathbf{m}'} \sqsubseteq \nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$, as Rule (Cond) of Figure 5 is applied, there is $\mathbf{m}_1 \preceq \mathbf{m}'$ such that $\nabla_\Gamma^{\mathtt{if(e)\{st_1\}else\{st_2\}}:\mathbf{m}_1}$, $\nabla_\Gamma^{\mathsf{e}:\mathbf{m}_1}$, and $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathtt{if(e)\{st_1\}else\{st_2\}}:\mathbf{m}_1}$ hold. Hence:

$$\begin{aligned} \mathbf{m} &\preceq \mathbf{m}_1 && \text{By } \textit{Lemma } 3.5 \\ &\preceq \sqcap_{\mathsf{y}\in\mathbb{V}(\mathsf{e})}\Gamma(\mathsf{y}) && \text{By } \textit{Lemma } 3.4 \end{aligned}$$

The case of Rule (Iter) can be treated similarly. □

Let $\mathbb{A}(\mathtt{st})$ be the set of variables that are assigned to in statement $\mathtt{st}$, *e.g.*, $\mathbb{A}(\mathsf{y.a} := \mathsf{y};\ \mathsf{x} := \mathsf{z};) = \{\mathsf{x}, \mathsf{y}\}$. The confinement Lemma expresses the fact that statements cannot write into variables of strictly higher level.

Lemma 3.7 (Confinement). *Given a program* $\mathsf{p} \in \mathtt{NI}_\Gamma$ *wrt the typing derivation* $\nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$*, for any* $\nabla_\Gamma^{\mathsf{st}(l):\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathsf{st_p}:\mathbf{n}}$*, it holds that* $\sqcup_{\mathsf{x}\in\mathbb{A}(\mathtt{st}(l))}\Gamma(\mathsf{x}) \preceq \mathbf{m}$.

Proof. By structural induction on statements using Rules of Figure 5. □

To sum up, noninterfering programs have the following properties:

- statements are always guarded by variables of higher level (Proposition 3.6);
- statements can only write into variables of smaller level (Lemma 3.7).

On their own, these properties are not sufficient to guarantee polynomial time soundness as the corresponding space of memory configurations is still unbounded.

3.4 Stratification

This section is devoted to the introduction of a stratification property for polynomial time soundness, enforcing the space of reachable program configurations $\mathcal{S}(\mathsf{p})$ to have a polynomially bounded size (under termination assumption).

3.4.1 Memory Level. In this respect, we need to link and restrict the program dynamics to the statically inferred levels. For that purpose, we define the notion of *memory level* that links memory heaps and the levels provided by a typing assignment.

Definition 3.8 (Memory level). For a given memory heap $\mathcal{H}_\mathsf{p} = (\mathcal{G}_\mathsf{p}, f_\mathsf{p})$ and a typing environment $\Gamma : \mathbb{V}(\mathsf{p}) \to \mathbf{N}$, the *memory level* is a function $ml_\Gamma^{\mathcal{H}_\mathsf{p}} : (\cup_\tau \mathcal{V}_\tau) \to \mathbf{N}$, assigning a level to memory addresses, and is defined as the least function (using pointwise order) satisfying:

$$\forall (v, \mathsf{a}, v') \in \mathcal{H}_\mathsf{p},\ ml_\Gamma^{\mathcal{H}_\mathsf{p}}(v) \preceq ml_\Gamma^{\mathcal{H}_\mathsf{p}}(v')$$
$$\forall (\mathsf{x}, v) \in \mathcal{H}_\mathsf{p},\ \Gamma(\mathsf{x}) \preceq ml_\Gamma^{\mathcal{H}_\mathsf{p}}(v)$$

The memory level induces some *stratification* properties on a memory heap: a memory address cannot point to a memory address of strictly smaller level. Consequently, memory addresses belonging to a cycle of the memory graph are at the same level.

Given a memory heap $\mathcal{H}_\mathsf{p} = \langle \mathcal{G}_\mathsf{p}, f_\mathsf{p} \rangle$, let $\mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq}(\Gamma)$ be the graph obtained by removing any memory address of level strictly smaller than $\mathbf{n}$ in the memory graph $\mathcal{G}_\mathsf{p}$ (*i.e.*, by removing any address v such that $ml_\Gamma^{\mathcal{H}_\mathsf{p}}(v) \prec \mathbf{n}$). We will write $\mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq}$ when the typing environment Γ is clear from the context. It can happen that $\mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq}$ neither is a memory graph, nor has a structurally correct memory graph, due to address removal. Let $\trianglelefteq$ ($\triangleleft$) be the (strict) sub-word relation over $\mathbb{W}$, defined by $w' \trianglelefteq w$, if $\exists w_1, w_2 \in \mathbb{W},\ w = w_1.w'.w_2$, the symbol "." denoting the concatenation on words. For a given level $\mathbf{n}$, we define the preorder $\subseteq_\mathbf{n}$ on memory heaps by $\mathcal{H}_\mathsf{p} \subseteq_\mathbf{n} \mathcal{H}'_\mathsf{p}$ if $\mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq}$ is a subgraph of $\mathcal{H}'^{\mathbf{n}\preceq}_\mathsf{p}$ and $\forall v \in \mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq} \cap \mathcal{V}_\mathtt{int}, \mathcal{H}_\mathsf{p}(v) \trianglelefteq \mathcal{H}'_\mathsf{p}(v)$. $\mathcal{H}_\mathsf{p} \subsetneq_\mathbf{n} \mathcal{H}'_\mathsf{p}$ holds whenever $\mathcal{H}_\mathsf{p} \subseteq_\mathbf{n} \mathcal{H}'_\mathsf{p}$ holds and $\mathcal{H}'_\mathsf{p} \subseteq_\mathbf{n} \mathcal{H}_\mathsf{p}$ does not hold. Consequently, if $\mathcal{H}_\mathsf{p} \subsetneq_\mathbf{n} \mathcal{H}'_\mathsf{p}$ either $\mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq}$ is a proper subgraph of $\mathcal{H}'^{\mathbf{n}\preceq}_\mathsf{p}$ or there is some memory address $v \in \mathcal{H}_\mathsf{p}^{\mathbf{n}\preceq} \cap \mathcal{V}_\mathtt{int}$ such that $\mathcal{H}_\mathsf{p}(v) \triangleleft \mathcal{H}'_\mathsf{p}(v)$. Finally, define the equivalence relation $=_\mathbf{n}$ by $\mathcal{H}_\mathsf{p} =_\mathbf{n} \mathcal{H}'_\mathsf{p}$ if both $\mathcal{H}_\mathsf{p} \subseteq_\mathbf{n} \mathcal{H}'_\mathsf{p}$ and $\mathcal{H}'_\mathsf{p} \subseteq_\mathbf{n} \mathcal{H}_\mathsf{p}$ hold. This relation is extended to configurations by $\langle \mathtt{st}, \mathcal{H}_\mathsf{p} \rangle =_\mathbf{n} \langle \mathtt{st}', \mathcal{H}'_\mathsf{p} \rangle$ if $\mathtt{st} = \mathtt{st}'$ and $\mathcal{H}_\mathsf{p} =_\mathbf{n} \mathcal{H}'_\mathsf{p}$ both hold.

3.4.2 Level Function. We also need to relate the statement to be executed with its corresponding level in the typing derivation. This connection is performed through a notion of *level function*. Let $wh : \mathbb{L}(\mathsf{p}) \to \{0, 1\}$ be a simple predicate on labels that evaluates to true (1) or false (0) depending on whether the statement of label l is contained within a while loop or not, respectively.

Definition 3.9 (Level function). Consider a program p in $\mathtt{NI}_\Gamma$ wrt the typing derivation $\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}$, the function $lev_{\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}} \in \mathbb{L}(\mathsf{p}) \to \mathbf{N}$ is defined by:

$$lev_{\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}}(l) \triangleq \begin{cases} \sqcap\{\mathbf{m} \mid \nabla_\Gamma^{\mathtt{st}(l):\mathbf{n}'} \sqsubseteq \nabla_\Gamma^{\mathtt{while(e)\{st\}}:\mathbf{m}} \sqsubseteq \nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}\} & \text{if } wh(l), \\ \sqcup_{\mathsf{x}\in\mathbb{V}(\mathsf{p})}\Gamma(\mathsf{x}) + \mathbf{1} & \text{otherwise.} \end{cases}$$

Informally, $lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}(l)$ is the minimal level of a while loop containing the statement $\mathtt{st}(l)$, if there is at least one such level (*i.e.*, $wh(l) = 1$). Otherwise, if l does not correspond to statement within a while loop, $lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}(l)$ is equal to the maximal level of a variable plus one. The well-definedness of $lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}(l)$ is ensured by the fact that a typing derivation explores all statements (thus all labels) of a program. Its uniqueness is guaranteed as we consider a fixed typing derivation. Hence $lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}$ is a total function in $\mathbb{L}(\mathtt{p}) \to \mathbf{N}$.

Example 3.10. Consider the program `rev` of Example 2.1 together with the typing environment defined in Example 3.3. It holds that $lev_{\nabla_\Gamma^{\mathtt{st_{rev}}:\mathbf{1}}}(it) = lev_{\nabla_\Gamma^{\mathtt{st_{rev}}:\mathbf{1}}}(l_i) = \mathbf{1}, \forall i \notin \{0, 5\}$, by definition of the level, as these assignments are all contained within a level-**1** while loop. Moreover, $lev_{\nabla_\Gamma^{\mathtt{st_{rev}}:\mathbf{1}}}(l_0) = lev_{\nabla_\Gamma^{\mathtt{st_{rev}}:\mathbf{1}}}(l_5) = \sqcup_{\mathtt{x}\in\mathbb{V}(\mathtt{rev})}\Gamma(\mathtt{x}) + \mathbf{1} = \mathbf{2}$, as l_0 and l_5 are not contained within a while loop.

3.4.3 Stratification. We are now ready to state a stratification criterion for noninterfering programs. This property is expressed as a constraint on memory levels depending on the level function.

Definition 3.11 (Stratification). A program $\mathtt{p} \in \mathtt{NI}_\Gamma$ wrt the typing derivation $\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}$ is *stratified*, if for any $\langle \mathtt{st}, \mathcal{H}_\mathtt{p}\rangle \in \mathcal{S}(\mathtt{p})$, $\langle \mathtt{st}, \mathcal{H}_\mathtt{p}\rangle \mapsto_l \langle \mathtt{st}', \mathcal{H}'_\mathtt{p}\rangle$ and $\mathbf{0} \prec lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}(l)$ imply that $\mathcal{H}'_\mathtt{p} \subseteq_{lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}(l)} \mathcal{H}_\mathtt{p}$. Let `STR` be the set of stratified programs.

Example 3.12. Consider the program of Example 2.1. We have already demonstrated in Example 3.3 that $\mathtt{rev} \in \mathtt{NI}_\Gamma$ wrt the typing derivation $\nabla_\Gamma^{\mathtt{st_{rev}}:\mathbf{1}}$ and exhibited in Example 3.10 that $\forall l \in \{it, l_1, l_2, l_3, l_4\}$, $lev_{\nabla_\Gamma^{\mathtt{st_{rev}}:\mathbf{1}}}(l) = \mathbf{1}$. Consequently, for `rev` to be in `STR`, we have to show that for any reachable configuration in $\mathcal{S}(\mathtt{rev})$ of the shape $\langle \mathtt{st}(l)\ \mathtt{st}', \mathcal{H}_\mathtt{rev}\rangle$, with $l \in \{it, l_1, l_2, l_3, l_4\}$, if $\langle \mathtt{st}(l)\ \mathtt{st}', \mathcal{H}_\mathtt{rev}\rangle \mapsto_l \langle \mathtt{st}'', \mathcal{H}'_\mathtt{rev}\rangle$ then $\mathcal{H}'_\mathtt{rev} \subseteq_\mathbf{1} \mathcal{H}_\mathtt{rev}$. We perform a case analysis:

- if $l = it$ then $\mathtt{st}(l) = \mathtt{while}(\mathtt{x} \neq \mathtt{null})\{\ldots\}$ and $\mathcal{H}_\mathtt{rev} = \mathcal{H}'_\mathtt{rev}$ by reduction rules of Figure 4b for while loops;
- if $l = l_1$ or $l = l_4$ then only level-**0** variables are updated;
- if $l = l_2$ then $\mathtt{st}(l) = \mathtt{y} := \mathtt{x}$; and $\mathcal{H}'_\mathtt{rev} = \mathcal{H}_\mathtt{rev}$ by reduction rules of Figure 4b as the assignment of a variable for reference types does not alter the graph;
- if $l = l_3$ then $\mathtt{st}(l) = \mathtt{x} := \mathtt{x.tl}$ and $\mathcal{H}'_\mathtt{rev} \subseteq_\mathbf{1} \mathcal{H}_\mathtt{rev}$ by reduction rules of Figure 4b for assignments and since the graph reachable from the level-**1** variables (here `x`) has decreased by one memory address.

We conclude that $\mathtt{rev} \in \mathtt{STR}$.

3.5 Polynomial Time Soundness

Terminating programs that are noninterfering run in polynomial time (in the size of the input memory heap).

Theorem 3.13 (Polynomial time soundness). $\mathtt{STR} \cap \mathtt{TERM} \subsetneq \mathtt{POLY}$.

Proof. Consider a program $\mathtt{p} \in \mathtt{STR} \cap \mathtt{TERM}$ wrt the typing derivation $\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}$. For showing that $\mathtt{p} \in \mathtt{POLY}$ it suffices to show that there exists a polynomial $P \in \mathbb{N}[X]$ such that for any memory heap the size of the orbit $O(\langle \mathtt{st_p}, \mathcal{H}_\mathtt{p}\rangle)$ is bounded by $P(|\mathcal{H}_\mathtt{p}|)$.

Since `p` is terminating and since the control flow is hierarchical, by Proposition 3.6, for any memory graph $\mathcal{H}_\mathtt{p}$ there cannot be two configurations $c, c' \in O(\langle \mathtt{st_p}, \mathcal{H}_\mathtt{p}\rangle)$ such that $c \mapsto_l \circ \mapsto^* c'$, and $c =_{lev_{\nabla_\Gamma^{\mathtt{st_p}:\mathbf{n}}}(l)} c'$ hold. Otherwise this reduction can be iterated infinitely many and it contradicts the termination assumption. Consequently bounding the size of $O(\langle \mathtt{st_p}, \mathcal{H}_\mathtt{p}\rangle)$ amounts to bounding the number of equivalence classes for $=_\mathbf{n}$ in the orbit.

Now assume that the highest level of a statement in the program is $\mathbf{n}$. Level-$\mathbf{m}$ variables, with $\mathbf{n} \prec \mathbf{m}$, cannot be modified, by Lemma 3.7. The number of distinct values taken by a level-$\mathbf{n}$ variable in the orbit $O(\langle \mathtt{st}_\mathsf{p}, \mathcal{H}_\mathsf{p} \rangle)$ is bounded by $|\mathcal{H}_\mathsf{p}|$ as such a variable cannot increase by stratification (Definition 3.11). Consequently, if there are $k_\mathbf{n}$ level-$\mathbf{n}$ variables, then there are $O(|\mathcal{H}_\mathsf{p}|^{k_\mathbf{n}})$ distinct equivalence classes for $=_\mathbf{n}$. Moreover, each level-$\mathbf{n}$ assignment can make the memory increase by at most a constant at a strictly lower level (by adding a constant number of new memory addresses and by applying a constant number of positive operators). Hence the memory increase performed by these level-$\mathbf{n}$ assignments is also in $O(|\mathcal{H}_\mathsf{p}|^{k_\mathbf{n}})$. By iterating this reasoning at lower levels, we obtain that the number of equivalence classes at each level strictly greater than $\mathbf{0}$ is bounded by a finite composition of polynomials. Moreover, level-$\mathbf{0}$ statements are constant time statements, as level-$\mathbf{0}$ variables cannot guard while loops. Hence, for any memory graph $\mathcal{H}_\mathsf{p}$ the total number of equivalence classes for $=_\mathbf{n}$ is bounded by $P(|\mathcal{H}_\mathsf{p}|)$, for some polynomial P that only depends on the typing derivation. We conclude by noting that it implies that $\mathsf{p} \in \mathtt{POLY}$. □

The inclusion of Theorem 3.13 is strict since there exist programs that terminate in a polynomial number of steps and that are not stratified. As a trivial counterexample, consider the statement $\mathtt{x} := 0;\ \mathtt{while}(\mathtt{x} < 28)\{\mathtt{x} := \mathtt{x} + 1;\}$. This statement runs in polynomial time and is noninterfering by setting, for example, $\Gamma(\mathtt{x}) = \mathbf{1}$. However it is not stratified as, in this case, the reduction of assignment $\mathtt{x} := \mathtt{x} + 1;$ make the level-$\mathbf{1}$ value increase. At first sight, such a counter-example could be seen as a severe drawback. However this is not the case as one has to keep in mind that the set of programs running in polynomial time is known to be Σ_2^0-complete in the arithmetical hierarchy ([Hájek 1979]). Consequently, there is no hope to have a complete and tractable method for characterizing POLY and finding tractable but incomplete methods for certifying a program to be in POLY is then of relevant interest. In practice, some transformation techniques can be used on such programs to obtain equivalent stratified programs.

3.6 Examples

Example 3.14 (Bounded concatenation). The following program bconcat concatenates two lists given as input up to some fixed bound.

```
    List {int hd; List tl}
    List x; List y; int z; int w;
l0 : y := x;
it : while (x ≠ null && w > 0){
l1 :     z := x.hd;
l2 :     y := new List(z,y);
l3 :     w := w-1;
l4 :     x := x.tl;
    }
```

It can be shown to be in $\mathtt{STR} \cap \mathtt{TERM}$. Indeed, the judgment $\Gamma \vdash_{\mathtt{NI}} \mathtt{bconcat} : \mathbf{1}$ can be derived for the typing environment Γ defined by $\Gamma(\mathtt{x}) = \Gamma(\mathtt{w}) \triangleq \mathbf{1}$ and $\Gamma(\mathtt{y}) = \Gamma(\mathtt{z}) \triangleq \mathbf{0}$. Hence $\mathtt{bconcat} \in \mathtt{NI}_\Gamma$ Moreover, the level-$\mathbf{1}$ statements correspond to labels it, l_1, l_2, l_3, and l_4. Each of these statements either does not alter the memory graph (*e.g.*, it and l_0), or changes the memory on strictly smaller levels (*e.g.*, l_1 and l_2) or is non-increasing on level-$\mathbf{1}$ data (*e.g.*, l_3 and l_4). Consequently, $\mathtt{bconcat} \in \mathtt{STR} \cap \mathtt{TERM}$, as it is terminating, and hence in POLY, by Theorem 3.13.

Example 3.15 (Nested loops). The program nested adds a number n of new elements in a list, where n is equal to the sum of the integers of the initial list.

```
     List {int hd; List tl}
     List x; List y; int z;
l0 : y := x;
it1 : while (x ≠ null){
l1 :   z := x.hd;
it2 :  while (z > 0){
l2 :     y := new List(1,y);
l3 :     z := z-1;
       }
l4 :   x := x.tl;
     }
```

The judgment $\Gamma \vdash_{\mathsf{NI}} \texttt{nested} : \mathbf{2}$ can be derived for the typing environment Γ defined by $\Gamma(\texttt{x}) \triangleq \mathbf{2}$, $\Gamma(\texttt{z}) \triangleq \mathbf{1}$, and $\Gamma(\texttt{y}) = \mathbf{0}$. Hence $\texttt{nested} \in \mathsf{NI}_\Gamma$. Moreover, the level-**2** statements correspond to labels it_1, l_1, and l_4 and either do not alter the memory graph (*e.g.*, it_1), or make the memory data increase on strictly smaller levels (*e.g.*, l_1 on level **1**), or is non-increasing on level-**2** data (*e.g.*, l_4). The level-**1** statements correspond to labels it_2, l_2, and l_3 and either does not alter the memory graph (*e.g.*, it_2), or makes the memory increase on strictly smaller levels (*e.g.*, l_2 on level **0**), or is non-increasing on data of level greater than **1** (*e.g.*, l_3). Consequently, $\texttt{nested} \in \mathsf{STR} \cap \mathsf{TERM}$, as it is terminating, and hence in POLY, by Theorem 3.13. Notice that this program can also be typed in $\mathsf{NI}_{\Gamma'}$, for some typing environment Γ' with only two levels (*i.e.*, $\Gamma' \vdash_{\mathsf{NI}} \texttt{nested} : \mathbf{1}$ can be derived). However, for such a typing derivation, we will fail to show that the program is stratified. In this setting, z will still be enforced to be of level **1** as z appears in a while loop guard. However, the assignment z := x.hd; is of level **1** and can make the memory increase. Consequently, while two levels are sufficient for (extensional) completeness (Theorem 4.7), the use of more than two levels increases the number of captured programs.

Example 3.16. The program rev of Example 2.1 is terminating (hence in TERM) and we have shown in Example 3.12 that it is in STR. Consequently, by Theorem 3.13, we can conclude that it is polynomial time sound.

4 A PROPER EXTENSION FOR POLY-TIME SOUNDNESS

In this section, we show that the criterion for poly-time soundness (Theorem 3.13) is a proper extension of previous tier-based type systems for complexity analysis. For that purpose, we introduce a generic type system, named TS (*cf.*, Figure 6), that implements the standard type discipline of [Hainry and Péchoux 2018; Marion 2011].

4.1 A Reminder on Safe Programs

The type discipline of [Hainry and Péchoux 2018; Marion 2011] is restricted to levels in $\{\mathbf{0}, \mathbf{1}\}$ (where they are called *tiers*). In this setting, typing environments Γ will be total functions in $\mathbb{V}(\mathsf{p}) \to \{\mathbf{0}, \mathbf{1}\}$. To avoid confusion with previous section, levels in $\{\mathbf{0}, \mathbf{1}\}$ will be denoted by $\alpha, \alpha_1, \ldots$ A *type-1 level* is of the shape $\alpha_1 \times \cdots \times \alpha_k \to \alpha$. Let $\mathbb{T}_1^n \triangleq \{\alpha_1 \times \cdots \times \alpha_n \to \alpha \mid \alpha_1, \ldots, \alpha_n, \alpha \in \{\mathbf{0}, \mathbf{1}\}\}$. An *operator typing environment* Δ associates a set $\Delta(\mathsf{op})$ of type-1 levels in $\mathbb{T}_1^k$ to each operator $\mathsf{op}^{\tau_1 \times \ldots \times \tau_k \to \pi}$ of arity k in $\mathbb{O}(\mathsf{p})$.

Definition 4.1. A positive operator $\mathsf{op}^{(\tau_1, \cdots, \tau_k) \to \pi}$ is *neutral* if:

- $[\![\mathsf{op}]\!]$ is constant, *i.e.*, $k = 0$;
- or $[\![\mathsf{op}]\!]$ is a predicate, *i.e.*, $\pi = \mathsf{bool}$;

$$\frac{\Gamma(\mathsf{s}) = \alpha}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{s} : \alpha}\ (\mathrm{Id}_{\mathsf{TS}}) \qquad \frac{\alpha_1 \times \cdots \times \alpha_k \to \alpha \in \Delta(\mathsf{op}) \qquad \forall i,\ \Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e}_i : \alpha_i}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{op}(\mathsf{e}_1, \cdots, \mathsf{e}_n) : \alpha}\ (\mathrm{Op}_{\mathsf{TS}})$$

$$\frac{}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{null} : \alpha}\ (\mathrm{Null}_{\mathsf{TS}}) \qquad \frac{\forall i,\ \Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e}_i : \mathbf{0}}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{new\ C}(\mathsf{e}_1, \ldots, \mathsf{e}_n) : \mathbf{0}}\ (\mathrm{New}_{\mathsf{TS}})$$

$$\frac{}{\Gamma, \Delta \vdash_{\mathsf{TS}}; : \mathbf{0}}\ (\mathrm{Skip}_{\mathsf{TS}}) \qquad \frac{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{s} : \alpha \qquad \Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e} : \beta \qquad \alpha \preceq \beta}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{s}^{\pi} := \mathsf{e}; : \alpha}\ (\mathrm{Asg}^{\pi}_{\mathsf{TS}})$$

$$\frac{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{x.a} : \mathbf{0} \qquad \Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e} : \mathbf{0}}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{x.a} := \mathsf{e}; : \mathbf{0}}\ (\mathrm{A}^{\mathrm{fs}}_{\mathsf{TS}}) \qquad \frac{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{x} : \alpha \qquad \Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e} : \alpha}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{x}^{\mathsf{C}} := \mathsf{e}; : \alpha}\ (\mathrm{A}^{\rho}_{\mathsf{TS}})$$

$$\frac{\forall i,\ \Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st}_i : \alpha}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st}_1\ \mathtt{st}_2 : \alpha}\ (\mathrm{Seq}_{\mathsf{TS}}) \qquad \frac{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e} : \alpha \qquad \forall i,\ \Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st}_i : \alpha}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{if(e)\{st_1\}else\{st_2\}} : \alpha}\ (\mathrm{Cond}_{\mathsf{TS}})$$

$$\frac{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathsf{e} : \mathbf{1} \qquad \Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st} : \alpha}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{while(e)\{st\}} : \mathbf{1}}\ (\mathrm{Iter}_{\mathsf{TS}}) \qquad \frac{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st} : \mathbf{0}}{\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st} : \mathbf{1}}\ (\mathrm{Sub}_{\mathsf{TS}})$$

Fig. 6. level-based type systems TS

- or $[\![\mathsf{op}]\!]$ computes a subword of its inputs:
  $\forall (w_1, \ldots, w_k) \in [\![\tau_1]\!] \times \ldots \times [\![\tau_k]\!],\ \exists j,\ [\![\mathsf{op}]\!](w_1, \ldots, w_k) \trianglelefteq w_j$.

Let $\mathbb{N}(\mathsf{p})$ be the set of neutral operators in $\mathbb{O}(\mathsf{p})$.

Each operator is associated to a set of admissible type-1 levels that depends on its computational power. For that purpose, we introduce the notion of *safe* operator typing environment that allows a distinct treatment of operators depending on their kind (neutral or positive only).

Definition 4.2. An operator typing environment Δ of a program p is *safe* if for each op in $\mathbb{O}(\mathsf{p})$, for each $\alpha_1 \times \cdots \times \alpha_k \to \alpha \in \Delta(\mathsf{op})$, the following hold:

- $\alpha \preceq \sqcap^k_{i=1} \alpha_i$.
- if $\mathsf{op} \in \mathbb{O}(\mathsf{p}) - \mathbb{N}(\mathsf{p})$ (*i.e.*, op is positive but not neutral) then $\alpha = \mathbf{0}$.

An operator of arity $k = 0$ can be typed, by abuse of notation, by any level in $\{\mathbf{0}, \mathbf{1}\}$, that is, we set $\Delta(\mathsf{op}^{\pi}) \triangleq \{\mathbf{0}, \mathbf{1}\}$, by identifying $\{\mathbf{0}, \mathbf{1}\}$ and $\mathbb{T}^0_1$ as such operators compute constants.

Typing judgments are of the shape $\Gamma, \Delta \vdash_{\mathsf{TS}} b : \alpha$, for some typing environment Γ, some operator typing environment Δ, some expression or statement b, and some level $\alpha \in \{\mathbf{0}, \mathbf{1}\}$. The corresponding type system, named TS, is presented in Figure 6. TS corresponds exactly to the type system of [Hainry and Péchoux 2018] without method calls and method declarations and to the type system of [Marion 2011] extended with reference types. In this latter case, it suffices to remove rules $(\mathrm{A}^{\rho}_{\mathsf{TS}})$ and $(\mathrm{A}^{\mathrm{fs}}_{\mathsf{TS}})$ from TS, and to specify that $\mathsf{s} \in \mathbb{V}(\mathsf{p})$ in rule $(\mathrm{Id}_{\mathsf{TS}})$ to recover the type system of [Marion 2011].

Definition 4.3. A program p is *safe* if there exist a variable typing environment Γ, a safe operator typing environment Δ, and a level $\alpha \in \{\mathbf{0}, \mathbf{1}\}$ such that $\Gamma, \Delta \vdash_{\mathsf{TS}} \mathtt{st}_{\mathsf{p}} : \alpha$ holds. Let SAFE be the set of safe programs.

4.2 STR **vs** SAFE

First, we show that the programs of Example 2.1, Example 3.14, and Example 3.15 are not safe.

Lemma 4.4. rev, bconcat, nested $\notin$ SAFE

Proof. rev is not in SAFE. Indeed, the type discipline of TS enforces variable x to be at level **1** (by Rule (Iter_{TS}), as x appears in the while loop guard) and variable y to be at level **0** (by Rule ($\text{A}^{\text{fs}}_{\text{TS}}$) as y modifies the memory graph structure in the assignment at label l_4: y.tl := z;). Hence the assignment at label l_2 cannot be typed (y := x). Indeed, Rule ($\text{A}^{\rho}_{\text{TS}}$) can be used only when the levels match.

bconcat cannot be typed in TS as x has to be a level-**1** variable (it appears in the guard of the while loop), y has to be a level-**0** variable (because it makes the memory increase in y := new List(z, y);), and, hence, y := x; cannot be typed.

nested cannot be typed in TS because the type system is restricted to 2 levels. □

Next we show that any safe program is stratified.

Theorem 4.5. SAFE $\subsetneq$ STR

Proof. We start to show that SAFE programs are noninterfering. Consider a program $\text{p} \in \text{SAFE}$. It means that $\Gamma, \Delta \vdash_{\text{TS}} \text{st}_\text{p} : \alpha$ can be derived. We show that $\Gamma \vdash_{\text{NI}} \text{st}_\text{p} : \alpha$ holds, by structural induction on the typing derivation and by performing a case analysis on rules:

- Rule (Op_{TS}): Assume, by induction hypothesis, that $\Gamma \vdash_{\text{NI}} \text{e}_i : \alpha_i$ holds, then we can derive:
$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\text{NI}} \text{e}_i : \alpha_i}{\Gamma \vdash_{\text{NI}} \text{e}_i : \sqcap_{i=1}^{|\overline{\text{e}}|}\alpha_i}\ (\text{SubE})}{\Gamma \vdash_{\text{NI}} \text{op}(\overline{\text{e}}) : \sqcap_{i=1}^{|\overline{\text{e}}|}\alpha_i}\ (\text{Op})}{\Gamma \vdash_{\text{NI}} \text{op}(\overline{\text{e}}) : \alpha}\ (\text{SubE})$$
using several instances of rule (subE), since that $\alpha \preceq \sqcap_{i=1}^{|\overline{\text{e}}|}\alpha_i$, by definition of safe operator typing environments.
- Rule (Iter_{TS}): Assume, by induction hypothesis, that $\Gamma \vdash \text{e} : \mathbf{1}$ and $\Gamma \vdash \text{st} : \alpha$ can be derived.
  - If $\alpha = \mathbf{1}$ then the result straightforwardly holds as we can derive:
$$\dfrac{\Gamma \vdash \text{e} : \mathbf{1} \quad \Gamma \vdash \text{st} : \mathbf{1}}{\Gamma \vdash \texttt{while(e)\{st\}} : \mathbf{1}}\ (\text{Iter})$$
  - If $\alpha = \mathbf{0}$ then we can derive:
$$\dfrac{\Gamma \vdash \text{e} : \mathbf{1} \quad \dfrac{\Gamma \vdash \text{st} : \mathbf{0}}{\Gamma \vdash \text{st} : \mathbf{1}}\ (\text{Sub})}{\Gamma \vdash \texttt{while(e)\{st\}} : \mathbf{1}}\ (\text{Iter})$$
- For all the remaining rules of TS, we just straightforwardly apply the corresponding rule in NI together with the induction hypothesis.

Now we show that SAFE $\subseteq$ STR. It is shown in [Hainry and Péchoux 2018] that if a level-**1** variable is assigned to then the assignment is of the shape (i) x^ρ := e; (*i.e.*, assignments of a field selector are prohibited for level-**1** variables of reference type because of Rule ($\text{A}^{\rho}_{\text{TS}}$)) or of the shape (ii) s^π := e;. Moreover, e only involves neutral operators (as positive operator are enforced to have output **0** by definition of safe programs and a new instance as output level **0**, by Rule (New_{TS})).

- The evaluation of assignment (i) does not alter the memory graph structure.
- The evaluation of assignment (ii) only updates labels corresponding to primitive memory addresses. However, the restriction to neutral operators implies that values are non-increasing.

Consequently, $\mathsf{p} \in \mathsf{STR}$. The strictness of the inclusion is a consequence of Lemma 4.4 together with the fact that $\mathtt{rev} \in \mathsf{STR}$, as shown in Example 3.12. □

Since `rev` is a terminating program, we obtain the following corollary.

Corollary 4.6. $\mathsf{SAFE} \cap \mathsf{TERM} \subsetneq \mathsf{STR} \cap \mathsf{TERM}$

4.3 Polynomial Time Completeness

A consequence of Corollary 4.6 is that polynomial time completeness is achieved for free: any polynomial time computable function is computed by at least one program that types in $\mathsf{STR} \cap \mathsf{TERM}$. For $n \in \mathbb{N}$, let $\underline{n}$ denote a word in $\mathbb{W}$ encoding the integer n. A program p computes the function $f : \mathbb{N} \to \mathbb{N}$ if $\exists \mathsf{x}^{\mathsf{int}} \in \mathbb{V}(\mathsf{p})$ s.t. $\forall n \in \mathbb{N}$ and $\forall \mathcal{H}_\mathsf{p}$ such that $l_V(\mathcal{H}_\mathsf{p}(\mathsf{x})) = \underline{n}$, $[\![\mathsf{p}]\!](\mathcal{H}_\mathsf{p}) = \mathcal{H}'_\mathsf{p}$, and $l_V(\mathcal{H}'_\mathsf{p}(\mathsf{x})) = \underline{f(n)}$.

Theorem 4.7 (Completeness). *For any function in $f : \mathbb{N} \to \mathbb{N}$ computable in polynomial time, there exists a program $\mathsf{p} \in \mathsf{STR} \cap \mathsf{TERM}$ computing f.*

Proof. Completeness of $\mathsf{STR} \cap \mathsf{TERM}$ is obtained as a direct consequence of the completeness proof of Theorem 4 in [Hainry and Péchoux 2018]. This proof simulates a polynomial time Turing Machine by only using Boolean, integer, and list datatypes and is preserved by $\mathsf{SAFE} \cap \mathsf{TERM}$. Hence completeness of $\mathsf{STR} \cap \mathsf{TERM}$ is a consequence of the completeness of $\mathsf{SAFE} \cap \mathsf{TERM}$ and of Corollary 4.6. □

As a consequence of Corollary 4.6, STR is the most expressive noninterference-based type system that is both poly-time sound and poly-time complete. STR includes natural algorithms such as `rev` that are not in SAFE. This gain in terms of expressive power is due to the fact that this characterization clearly distinguishes the stratification properties (STR) from the noninterference policy (NI) for ensuring complexity properties.

4.4 STR in the Arithmetical Hierarchy

Once we have proven that STR captures strictly more programs than SAFE, one natural issue is to study the hardness of showing that a given program is in STR. It turns out that this problem is unsurprisingly not decidable.

Theorem 4.8. *STR is Π_1^0-complete.*

Proof. To show the Π_1^0-hardness of this set, we consider a reduction to the blank-tape non-halting problem that consists in knowing whether a given Turing machine (TM) does not halt when it starts on the empty tape ([Endrullis et al. 2011]).

We encode deterministic one-tape TMs on the alphabet $\{0, 1\}$ by imperative programs as follows: the state and the left and right parts of the tape are encoded by integer variables `s`, `left`, and `right`, respectively. The symbol scanned by the head is assumed to be the first symbol of the right portion of the tape. Contrarily to `right`, `left` encodes a portion of the tape in reverse order (*i.e.*, the tape symbols are represented from right to left). Now, each transition of the machine can be encoded by a code of the shape:

```
if(s = 5 && head(right) = 0){
  s := 6;
  left := cons(1,left);
  right := tail(right);
}
```

where tail, head, and cons are positive operators that compute the tail, the head, and head insertion on strings, respectively. For example, the above statement simulates that if the state is '5' and the read symbol is '0' then the next state is '6', the symbol '1' is written in place of '0', and the head performs a move to the right.

Any transition function δ_M of a TM M can be encoded by a statement consisting in a finite sequence of such codes. Let $\mathtt{st}_{\delta_M}(\mathtt{s}, \mathtt{left}, \mathtt{right})$ be such an encoding. The integers 0 and 1 will be used for encoding the initial state and final state, respectively. Let also true and ϵ be operators for the boolean constant 1 and the empty string, respectively. Consider the program p_M below:

```
   s := 0;
   left := ϵ;
   right := ϵ;
   while(true){
     st_{δ_M}(s, left, right)
     if(s = 1){
       while(s > 0){
l :      s := s+1;
       }
     }
   }
```

First, notice that this program can be typed in NI by setting $\Gamma(\mathtt{s}) = \Gamma(\mathtt{left}) = \Gamma(\mathtt{right}) \triangleq \mathbf{1}$ and checking that $\Gamma \vdash_{\mathtt{NI}} \mathtt{true} : \mathbf{2}$ can be derived. Level $\mathbf{0}$ is not admissible for these variables as s appears in a while loop guard and is controlled by right. The subderivation $\nabla_\Gamma^{\mathtt{true}:\mathbf{1}}$ is not admissible for the program to be in STR as, otherwise, any increase of left or right would break the noninterference. The case of programs where either left or right does not increase can be treated easily. It holds that p_M is in STR iff M does not halt on the empty tape. Indeed, if M halts on the empty state then p_M will reach label l after the final state has been reached (*i.e.*, $\mathtt{s} = 1$). Consequently, p_M is not in STR, as the statement $\mathtt{st}(l)$ breaks the noninterference (s increases in a loop guarded by itself). Conversely, if M does not halt on the empty state then label l is never reached (as it always holds that $\mathtt{s} \neq 1$). Consequently, the program runs the statement $\mathtt{st}_{\delta_M}(\mathtt{s}, \mathtt{left}, \mathtt{right})$ infinitely many without reaching label l. By construction, this program is noninterfering (as there is no variable of level $\mathbf{2}$).

It remains to show that STR is in Π_1^0. Checking that a program p is in STR is performed by first showing that $\mathsf{p} \in \mathtt{NI}_\Gamma$ wrt some Γ. This can be done in time polynomial in the size of the program as a consequence of the proof of complexity of type inference for many-level SAFE-like systems [Hainry et al. 2022]. Indeed, any typing derivation can be reduced to an instance of a 2-SAT problem.

Now it remains to study the complexity of showing that the program satisfies that $\forall \langle \mathtt{st}, \mathcal{H}_\mathsf{p} \rangle \in \mathcal{S}(\mathsf{p})$, $\langle \mathtt{st}, \mathcal{H}_\mathsf{p} \rangle \mapsto_l \langle \mathtt{st}', \mathcal{H}'_\mathsf{p} \rangle$ and $\mathbf{0} \prec lev_{\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}}(l)$ implies that $\mathcal{H}'_\mathsf{p} \subseteq_{lev_{\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}}(l)} \mathcal{H}_\mathsf{p}$.

First, notice this property can be rephrased as $\forall \mathcal{H}^0_\mathsf{p}, \mathcal{H}^1_\mathsf{p}, \mathcal{H}^2_\mathsf{p}, \mathtt{st}_1, \mathtt{st}_2$,

$$(\langle \mathtt{st}_\mathsf{p}, \mathcal{H}^0_\mathsf{p} \rangle \mapsto^* \langle \mathtt{st}_1, \mathcal{H}^1_\mathsf{p} \rangle \mapsto_l \langle \mathtt{st}_2, \mathcal{H}^2_\mathsf{p} \rangle \wedge \mathbf{0} \prec lev_{\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}}(l)) \Rightarrow \mathcal{H}^2_\mathsf{p} \subseteq_{lev_{\nabla_\Gamma^{\mathtt{st}_\mathsf{p}:\mathbf{n}}}(l)} \mathcal{H}^1_\mathsf{p}.$$

To conclude, this problem is in Π_1^0 by a standard encoding of graphs, statements, reductions, and graph inclusion. □

Hence, by Theorem 4.8 the set of STR programs is co-recursively enumerable. This undecidability result could be seen as a negative result. However we will show in the next section that STR can be specialized to some decidable instance that strictly encompasses SAFE.

5 SHAPE ANALYSIS

In this section, we provide a decidable type system ensuring that a program is stratified. This type system will extend the NI type system of Figure 5 by making use of shape analysis techniques to enforce stratification.

5.1 Uniform Collecting Semantics

As we want to ensure complexity properties on programs, we need to analyze the programs for any "reasonable" input. Whereas standard shape analysis methods study the program for an arbitrary but fixed input. To overcome this issue, we restrict the admissible inputs to a notion of *separable* inputs and define a collecting semantics for programs on such inputs.

Each program p is represented in a standard manner by a Control-Flow-Graph (CFG) $G_\mathsf{p} \triangleq (\mathbb{L}(\mathsf{p}), T_\mathsf{p}, lbl)$, where $T_\mathsf{p} \subseteq \mathbb{L}(\mathsf{p}) \times \mathbb{L}(\mathsf{p})$ is a set of arrows (transitions). We assume that G_p has a unique initial vertex $\mathrm{INIT} \in \mathbb{L}(\mathsf{p})$ with no predecessor and a unique final vertex $\mathrm{EXIT} \in \mathbb{L}(\mathsf{p})$ with no successor. The label function lbl is a partial function which maps arrows to boolean expressions. In the graphical representation of CFGs, arrows are annotated by their label, whenever it is defined. To illustrate this notion, the CFG of program rev is provided in Example 2.1.

Given a memory heap $\mathcal{H}_\mathsf{p} = (\mathcal{G}_\mathsf{p}, f_\mathsf{p})$ and $\mathsf{x} \in \mathbb{V}(\mathsf{p})$, let $\mathrm{Reach}_{\mathcal{H}_\mathsf{p}}(\mathsf{x})$ be the set of memory addresses distinct from $\bot$ which are reachable from x, *i.e.*, $v \in \mathrm{Reach}_{\mathcal{H}_\mathsf{p}}(\mathsf{x})$, $v \neq \bot$, if there exists a path from $f_\mathsf{p}(\mathsf{x})$ to v in the memory graph $\mathcal{G}_\mathsf{p}$.

Definition 5.1. A memory heap $\mathcal{H}_\mathsf{p}$ is *separable* if $\forall \mathsf{x} \neq \mathsf{y} \in \mathbb{V}(\mathsf{p}),\ \mathrm{Reach}_{\mathcal{H}_\mathsf{p}}(\mathsf{x}) \cap \mathrm{Reach}_{\mathcal{H}_\mathsf{p}}(\mathsf{y}) = \emptyset$. Let SEP_p be the set of separable memory heaps of HEAP_p.

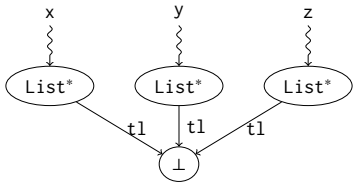
Example 5.2. The memory heap $\mathcal{H}^0_{\mathsf{rev}}$ of program rev described in Figure 3 is separable as $\mathrm{Reach}_{\mathcal{H}^0_{\mathsf{rev}}}(\mathsf{x}) = \{v_1, v_2, v_3, v'_1, v'_2, v'_3\}$, with $\forall i \leq 3$, $\mathcal{H}_\mathsf{p}(v'_i) = i$, and $\mathrm{Reach}_{\mathcal{H}^0_{\mathsf{rev}}}(\mathsf{y}) = \mathrm{Reach}_{\mathcal{H}^0_{\mathsf{rev}}}(\mathsf{z}) = \emptyset$. Separability is not preserved by the semantics of assignments as illustrated by the memory heap $\mathcal{H}^3_{\mathsf{rev}}$ of Figure 3 which is obtained from $\mathcal{H}^0_{\mathsf{rev}}$ and not separable. The restriction to separable inputs ensures that the stratification property will not be defeated by a specially crafted input. However, stratification is finer than separability and the non-preservation of separability has no impact on our type discipline, that guarantees that loops do not iterate on data modified at the same level.

Definition 5.3. The *uniform collecting semantics* UCS of a program p is a function from $\mathbb{L}(\mathsf{p}) \to 2^{\mathsf{HEAP}^{\mathsf{Err}}_\mathsf{p}}$ defined as the least fixed point under set inclusion of the following equations:

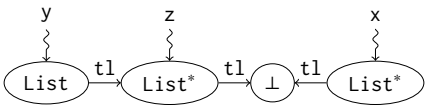
$$\mathsf{UCS}_l = \begin{cases} \mathsf{SEP}_\mathsf{p} & \text{if } l = \mathit{INIT} \\ \{\mathsf{snd}([\![\mathsf{st}(l')]\!](\mathcal{H}'_\mathsf{p})) \mid \langle l', l\rangle \in T_\mathsf{p},\ \mathcal{H}'_\mathsf{p} \in \mathsf{UCS}_{l'}\} & \text{otherwise.} \end{cases}$$

Separability holds if distinct variables point to disjoint memory locations. Consequently, the uniform collecting semantics associates the set of all reachable memory heaps of a given program, starting with any separable memory heaps, to each line label of a program.

Example 5.4. Consider the program rev of Example 2.1. The set of separable memory heaps $\mathsf{SEP}_{\mathsf{rev}}$ contains an (infinite) number of memory graphs that all follow the same pattern (we omit the head nodes for simplicity):



where each node of label "$\texttt{List}^*$" denotes a sequence of `List` memory addresses and `tl` arrows of finite length, possibly 0. In the case of length 0, the corresponding variable points directly to $\bot$. Now consider the label l_4 of Example 2.1, its uniform collecting semantics UCS_{l_4} is included in $2^{\mathsf{HEAP}_{\mathsf{p}}^{\mathsf{Err}}}$. Moreover, any memory graph in UCS_{l_4} has the following shape:



It means that y points to a memory address distinct from $\bot$, whereas x and z may point to $\bot$.

5.2 3-valued Logic and Abstractions

The use of Kleene's 3-valued logic [Kleene 1952] to infer shape analysis algorithms automatically from a program specification has been deeply studied in the literature [Reps et al. 2004; Sagiv et al. 2002]. In this setting, predicate-logic formulae are interpreted over a 2-valued world in order to represent concrete stores (memory states) and over a 3-valued world in order to represent abstract stores, where the third truth value $1/2$ is a value for "unknown".

As an illustrative example, consider the three memory addresses $v_i \in \mathcal{V}_{\texttt{List}}$, $1 \leq i \leq 3$, of the memory heap $\mathcal{H}^0_{\mathsf{rev}}$ in Example 2.3 described in Figure 3. Some partial information on the memory graph and the partial map of $\mathcal{H}^0_{\mathsf{rev}}$ can be encoded using the binary predicate tl and the unary predicates x, y, z on memory addresses described below.

tl	v_1	v_2	v_3
v_1	0	1	0
v_2	0	0	1
v_3	0	0	0

	x	y	z
v_1	1	0	0
v_2	0	0	0
v_3	0	0	0

Suppose that one wants to keep information about the memory addresses, distinct from null, directly pointed to by a variable. This can be represented by the abstract predicates $\mathrm{tl}^\#$, $\mathrm{x}^\#$, $\mathrm{y}^\#$, $\mathrm{z}^\#$ below, provided that the abstract memory address v represents v_1 and that the abstract memory address v' represents both v_2 and v_3, *i.e.*, non pointed memory addresses. In this setting, $\mathrm{tl}^\#(v', v') = 1/2$ as it is unknown whether v' points to itself or not. Indeed v_2 points to v_3 but v_3 points neither to v_2 nor to itself. $\mathrm{sm}^\#$ indicates whether an abstract memory address represents strictly more than 1 concrete memory address.

The abstract 3-valued predicates may be represented by the graphical notion of *abstract heap* $\mathcal{A}_{\mathsf{p}}$ of a program p, a labeled multidigraph standing for the abstract counterpart of memory heaps. However, abstract heaps differ from memory heaps on the following two points:

- They are neither assumed to be deterministic, nor to be well-shaped, as, for example, a variable may point to several abstract memory addresses. Moreover, their vertices are only labeled by their type, as the primitive values are of no use.
- Each of their arrows encodes a partial information on a given predicate $p^\#$ and is, consequently, labeled by $p^\#$. This arrow may be plain or dashed, depending on whether the corresponding

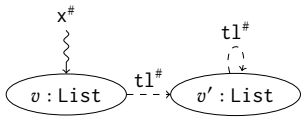
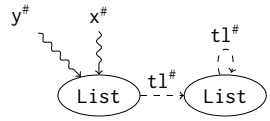
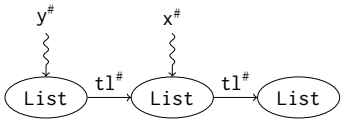
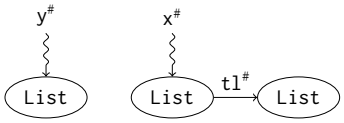
Assignment	Memory Heap		Abstract heap
$l_0:$ y := null; $l_1:$ z := y;	$\mathcal{H}^0_{\mathsf{rev}}$ = $\mathcal{H}^1_{\mathsf{rev}}$ = $\mathcal{H}^2_{\mathsf{rev}}$	$\mathcal{A}^0_{\mathsf{rev}}$ = $\mathcal{A}^1_{\mathsf{rev}}$ = $\mathcal{A}^2_{\mathsf{rev}}$	
$l_2:$ y := x;	$\mathcal{H}^3_{\mathsf{rev}}$	$\mathcal{A}^3_{\mathsf{rev}}$	
$l_3:$ x := x.tl;	$\mathcal{H}^4_{\mathsf{rev}}$	$\mathcal{A}^4_{\mathsf{rev}}$	
$l_4:$ y.tl := z;	$\mathcal{H}^5_{\mathsf{rev}}$	$\mathcal{A}^5_{\mathsf{rev}}$	

Fig. 7. Sequence of abstract heaps of the program rev

predicate evaluates to 1 or 1/2, respectively. The truth value 0 is encoded by the absence of an arrow.

Rather than providing a formal definition, we illustrate the concept of abstract heaps with the examples $\mathcal{A}^i_{\mathsf{rev}}$, for $0 \le i \le 5$, of Figure 7. Each $\mathcal{A}^i_{\mathsf{rev}}$ abstracts the memory heap $\mathcal{H}^i_{\mathsf{rev}}$ of Figure 3.

The truth tables for the predicates $\mathsf{tl}^\#$, $\mathsf{x}^\#$, $\mathsf{y}^\#$, and $\mathsf{z}^\#$, provided below, abstract the predicates tl, x, y, and z, respectively, and are represented graphically in the abstract heaps $\mathcal{A}^i_{\mathsf{rev}}$.

$\mathsf{tl}^\#$	v	v'
v	0	1/2
v'	0	1/2

	$\mathsf{x}^\#$	$\mathsf{y}^\#$	$\mathsf{z}^\#$	$\mathsf{sm}^\#$
v	1	0	0	0
v'	0	0	0	1

Let $\mathsf{AHEAP}_\mathsf{p}$ be the set of abstract heaps of the program p and let $\mathsf{AHEAP}^{\mathsf{Err}}_\mathsf{p} = \mathsf{AHEAP}_\mathsf{p} \cup \{\mathtt{Err}\}$. For $x, x' \in \{0, 1, 1/2\}$, the *information order* $\preccurlyeq$ on truth values is defined by $x \preccurlyeq x'$, if $x = x'$ or $x' = 1/2$. Let $\cup$ denote the least upper bound operation with respect to $\preccurlyeq$. We extend in the natural way $\preccurlyeq$ and $\cup$ to abstract heaps and error. $\mathcal{A}^1_\mathsf{p} \preccurlyeq \mathcal{A}^2_\mathsf{p}$ holds either if any arrow of $\mathcal{A}^1_\mathsf{p}$ is an arrow of $\mathcal{A}^2_\mathsf{p}$ and their corresponding truth values x_1 and x_2 satisfy $x_1 \preccurlyeq x_2$, or if $\mathcal{A}^2_\mathsf{p} = \mathtt{Err}$. For $\mathcal{A}^1_\mathsf{p}, \mathcal{A}^2_\mathsf{p} \neq \mathtt{Err}$, $\mathcal{A}^1_\mathsf{p} \cup \mathcal{A}^2_\mathsf{p}$ is the union of abstract heaps $\mathcal{A}^1_\mathsf{p}$ and $\mathcal{A}^2_\mathsf{p}$ obtained by using the least upper bound operation on predicates. $\mathcal{A}_\mathsf{p} \cup \mathtt{Err} = \mathtt{Err} \cup \mathcal{A}_\mathsf{p} = \mathtt{Err}$.

In order to prevent the "unknown" value from becoming pervasive in the abstract interpretation analysis, the predicates are always complemented by *instrumentation predicates* (abstract 3-valued predicates) [Reps et al. 2010], recording information in a logical structure, providing a mechanism for the user to fine-tune an abstraction and, hence, to control the amount of information lost. In our setting, we are interested in the following standard instrumentation predicates:

p	meaning
$\mathsf{a}^\#(v_1, v_2)$	Is v_2 reachable from v_1 along one or more a-fields?
$\mathsf{x}^\#(v)$	Is v reachable from variable x?

An abstraction $\alpha : \mathtt{HEAP}_\mathsf{p}^\mathtt{Err} \to \mathtt{AHEAP}_\mathsf{p}^\mathtt{Err}$ encoding the above predicates and which is fault-preserving, *i.e.*, $\alpha(\mathtt{Err}) = \mathtt{Err}$, can be generated. An example of such an abstraction is illustrated in Figure 3, where it holds that $\alpha(\mathcal{H}^i_{\mathtt{rev}}) = \mathcal{A}^i_{\mathtt{rev}}$, $0 \leq i \leq 5$ (we have omitted integer nodes and hd arrows for readability). Let $\overline{\alpha} : 2^{\mathtt{HEAP}_\mathsf{p}^\mathtt{Err}} \to \mathtt{AHEAP}_\mathsf{p}^\mathtt{Err}$ be the extension of α defined by $\overline{\alpha}(S) = \cup_{\mathcal{H}_\mathsf{p} \in S} \alpha(\mathcal{H}_\mathsf{p})$.

5.3 Abstract Semantics and its Properties

For a given fault-preserving abstraction function $\alpha : \mathtt{HEAP}_\mathsf{p}^\mathtt{Err} \to \mathtt{AHEAP}_\mathsf{p}^\mathtt{Err}$, we can use the refined approach of [Sagiv et al. 2002] (with focus and coerce operations) to generate a precise abstract semantics $[\![\mathtt{asg}]\!]^\# : \mathtt{AHEAP}_\mathsf{p}^\mathtt{Err} \to \mathtt{AHEAP}_\mathsf{p}^\mathtt{Err}$ of assignments. This semantics could also have been taken to be the best abstract transformer [Cousot and Cousot 2002] but it would not have been computable. In our setting, we assume this abstract semantics to be given as input but we put some standard semantics requirements. The shape analysis algorithm can then be described as an iterative procedure computing the least fixpoint under the following system of equations:

$$\mathcal{A}^\#_{l,\mathsf{p}} = \begin{cases} \overline{\alpha}(\mathtt{SEP}_\mathsf{p}) & \text{if } l = \mathrm{START} \\ \cup_{\langle l', l\rangle \in T_\mathsf{p}} [\![\mathtt{st}(l')]\!]^\#(\mathcal{A}^\#_{l',\mathsf{p}}) & \text{otherwise.} \end{cases}$$

The abstract semantics must satisfy the following standard properties:

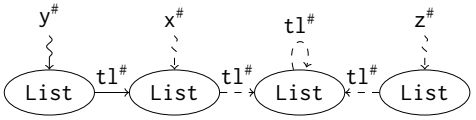
- $\forall \mathtt{asg} \in \mathsf{p},\ \forall \mathcal{A}^1_\mathsf{p}, \mathcal{A}^2_\mathsf{p} \in \mathtt{AHEAP}_\mathsf{p}^\mathtt{Err}$, if $\mathcal{A}^1_\mathsf{p} \lessdot \mathcal{A}^2_\mathsf{p}$ then $[\![\mathtt{asg}]\!]^\#(\mathcal{A}^1_\mathsf{p}) \lessdot [\![\mathtt{asg}]\!]^\#(\mathcal{A}^2_\mathsf{p})$
- Local safety: $\forall \mathtt{asg} \in \mathsf{p},\ \forall \mathcal{H}_\mathsf{p} \in \mathtt{HEAP}_\mathsf{p}^\mathtt{Err}$, $\alpha([\![\mathtt{asg}]\!](\mathcal{H}_\mathsf{p})) \lessdot [\![\mathtt{asg}]\!]^\#(\alpha(\mathcal{H}_\mathsf{p}))$
- Global safety: $\forall l \in L_\mathsf{p}$, we have $\overline{\alpha}(\mathtt{UCS}_l) \lessdot \mathcal{A}^\#_{l,\mathsf{p}}$

The analysis of [Sagiv et al. 1996], based on the notion of *shape-graphs*, can be seen as a particular case of the above analysis, restricted to one fixed input and to program on linked lists. In particular, it satisfies the monotonicity ans safety properties described above.

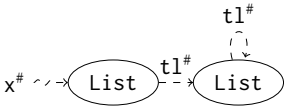
In the next section (Theorem 6.7), the algorithm for computing $\mathcal{A}^\#_{l,\mathsf{p}}$ will be shown to have an exponential time worst-case complexity.

Example 5.5. The abstract heap $\mathcal{A}^\#_{l_4,\mathtt{rev}}$ of program rev and of vertex $l_4 \in \mathbb{L}(\mathtt{rev})$ (Example 2.1) is provided below:

$\mathcal{A}^\#_{l_4,\mathtt{rev}} =$



It is worth noticing that although the input domain of memory heaps is not finite, the set of abstractions of separable memory heaps $\mathtt{SEP}_\mathsf{p}$ is finite. Hence the above algorithm is terminating on separable inputs. For example, $\alpha(\mathtt{SEP}_\mathsf{p})$ is equal to the following abstract heap:



6 NONINTERFERENCE TYPE SYSTEM WITH SHAPE ANALYSIS

In this section, we introduce a new type system named SA (Figure 6) ensuring that a typable program is in STR. SA is a new level-based type system that uses shapes.

The type system SA aims at restricting the programming discipline in such a way that typable programs are programs whose abstract semantics preserves some stratification property on loops at level-$\mathbf{n}$.

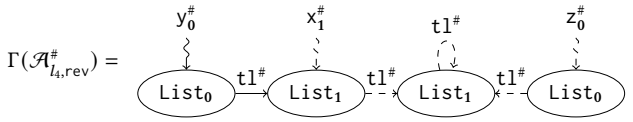
6.1 Shape Annotations

For a given program p, a given abstract heap $\mathcal{A}^{\#}_{l,\mathsf{p}}$, with $l \in L_{\mathsf{p}}$, and a given variable typing environment Γ such that $\mathbb{V}(\mathsf{p}) \subseteq dom(\Gamma)$, let $\Gamma(\mathcal{A}^{\#}_{l,\mathsf{p}})$ be the abstract heap obtained by annotating each node with a level sub-script such that the two following constraints are satisfied:

- Each node corresponding to a variable abstract predicate $\mathsf{x}^{\#}$ is annotated by the level of this variable in Γ as follows: $\mathsf{x}^{\#}_{\Gamma(\mathsf{x})}$,
- Each node of type τ is annotated by the least upper bound of level annotations of the nodes pointing to it in the abstract heap through dashed and plain arrows.

By construction, an annotated abstract heap is stratified, in the sense that there is no arrow from a node annotated by level $\mathbf{m}$ to a node annotated by level $\mathbf{n}$, for $\mathbf{n} \prec \mathbf{m}$. Hence it can be viewed as the adaptation of the memory level function to abstract heaps. For $\mathsf{x} \in \mathbb{V}(\mathsf{p})$, $l \in \mathbb{L}(\mathsf{p})$, and $\mathbf{n} \in \mathbf{N}$, $\mathsf{x} \in \mathbf{n}_{l,\mathsf{p},\Gamma}$ holds if any arrow of source $\mathsf{x}^{\#}$ in $\Gamma(\mathcal{A}^{\#}_{l,\mathsf{p}})$ points to a target annotated by $\mathbf{m}$, for $\mathbf{m} \preceq \mathbf{n}$.

Example 6.1. Consider a typing environment Γ such that $\Gamma(\mathsf{x}) = \mathbf{1}$ and $\Gamma(\mathsf{y}) = \Gamma(\mathsf{z}) = \mathbf{0}$. The annotated graph $\Gamma(\mathcal{A}^{\#}_{l_4,\mathtt{rev}})$ corresponding to the abstract heap of Example 5.5 is provided below. It holds that $\mathsf{y}, \mathsf{z} \in \mathbf{0}_{l_4,\mathtt{rev},\Gamma}$ and $\mathsf{x} \notin \mathbf{0}_{l_4,\mathtt{rev},\Gamma}$.



6.2 Shape-based Type System

Typing judgments will be of the shape $\Gamma \vdash^{\mathbf{m}}_{\mathsf{SA}} b : \mathbf{n}$, with $\mathbf{n}, \mathbf{m} \in \mathbf{N}$. $\mathbf{m}$ will be called the *context level*. The context level has to be understood as the level fixed by the surrounding environment. The output level $\mathbf{n}$ has to be understood as the level assigned to element b with respect to a fixed context level $\mathbf{m}$. Hence the output level is constrained by the context level.

Definition 6.2 (Shape analysis). A program p is SA for the typing environment $\Gamma : \mathbb{V}(\mathsf{p}) \to \mathbf{N}$ and the levels $\mathbf{n}, \mathbf{m} \in \mathbf{N}$ if the judgment $\Gamma \vdash^{\mathbf{m}}_{\mathsf{SA}} \mathtt{st}_{\mathsf{p}} : \mathbf{n}$ can be derived using the rules of Figure 8.

We provide the main intuitions on the design of the SA type system. First, the context level just takes account of the innermost while loop level and is, consequently, only updated in Rule ($\mathrm{Iter}_{\mathsf{SA}}$) for the loop body. This context level is only used in three distinct rules:

(1) Rule ($\mathrm{Pos}_{\mathsf{SA}}$) ensures that a non-neutral operator (in $\mathbb{O}(\mathsf{p}) - \mathbb{N}(\mathsf{p})$) cannot make a primitive value increase at a level greater or equal to the context level,
(2) Rule ($\mathrm{New}_{\mathsf{SA}}$) ensures that nodes cannot be created at a level greater or equal to the context level,
(3) Rule ($\mathrm{Asg}^{\rho}_{\mathsf{SA}}$) ensures that the memory graph cannot be modified at a level greater or equal to the context level. Moreover, it can be modified at smaller level under the assumption that these levels cannot be accessed by higher level variables (*i.e.*, $\mathsf{x} \in \mathbf{n}_{l,\mathsf{p},\Gamma}$).

The remaining rules are similar to the rules of NI.

$$\frac{\Gamma(s) = \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} s : \mathbf{n}}\ (Id_{SA}) \qquad \frac{\forall i \leq |\overline{e}|,\ \Gamma \vdash^{\mathbf{m}}_{SA} e_i : \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} op(\overline{e}) : \mathbf{n}}\ (Ntr_{SA}) \qquad \frac{\forall i \leq |\overline{e}|,\ \Gamma \vdash^{\mathbf{m}}_{SA} e_i : \mathbf{n} \qquad \mathbf{n} \prec \mathbf{m}}{\Gamma \vdash^{\mathbf{m}}_{SA} op(\overline{e}) : \mathbf{n}}\ (Pos_{SA})$$

$$\frac{}{\Gamma \vdash^{\mathbf{m}}_{SA} \texttt{null} : \mathbf{n}}\ (Null_{SA}) \qquad \frac{\forall i \leq |\overline{e}|,\ \Gamma \vdash^{\mathbf{m}}_{SA} e_i : \mathbf{n} \qquad \mathbf{n} \prec \mathbf{m}}{\Gamma \vdash^{\mathbf{m}}_{SA} \texttt{new C}(\overline{e}) : \mathbf{n}}\ (New_{SA})$$

$$\frac{\Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{n} \qquad \mathbf{m} \preceq \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{m}}\ (SubE_{SA}) \qquad \frac{}{\Gamma \vdash^{\mathbf{m}}_{SA} ; : \mathbf{0}}\ (Skip_{SA}) \qquad \frac{\Gamma(x) = \mathbf{n} \qquad \Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} x := e; : \mathbf{n}}\ (Asg_{SA})$$

$$\frac{\Gamma(x.a) = \mathbf{n} \qquad \Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} x.a^{\pi} := e; : \mathbf{n}}\ (Asg^{\pi}_{SA}) \qquad \frac{\Gamma(x.a) = \mathbf{n} \qquad \Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{n} \qquad x \in \mathbf{n}_{l,p,\Gamma} \qquad \mathbf{n} \prec \mathbf{m}}{\Gamma \vdash^{\mathbf{m}}_{SA} x.a^{\rho} := e; : \mathbf{n}}\ (Asg^{\rho}_{SA})$$

$$\frac{\Gamma \vdash^{\mathbf{m}}_{SA} st_1 : \mathbf{n} \qquad \Gamma \vdash^{\mathbf{m}}_{SA} st_2 : \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} st_1\ st_2 : \mathbf{n}}\ (Seq_{SA}) \qquad \frac{\Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{n} \qquad \Gamma \vdash^{\mathbf{m}}_{SA} st_1 : \mathbf{n} \qquad \Gamma \vdash^{\mathbf{m}}_{SA} st_2 : \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} \texttt{if(e)\{st}_1\texttt{\}else\{st}_2\texttt{\}} : \mathbf{n}}\ (Cond_{SA})$$

$$\frac{\Gamma \vdash^{\mathbf{m}}_{SA} e : \mathbf{n} \qquad \Gamma \vdash^{\mathbf{n}}_{SA} st : \mathbf{n} \qquad \mathbf{1} \preceq \mathbf{n}}{\Gamma \vdash^{\mathbf{m}}_{SA} \texttt{while(e)\{st\}} : \mathbf{n}}\ (Iter_{SA}) \qquad \frac{\Gamma \vdash^{\mathbf{m}}_{SA} st : \mathbf{n} \qquad \mathbf{n} \preceq \mathbf{m}}{\Gamma \vdash^{\mathbf{m}}_{SA} st : \mathbf{m}}\ (Sub_{SA})$$

Fig. 8. Type system for noninterfering programs

6.3 Examples

Example 6.3. The program `rev` of Example 2.1 can be typed in SA w.r.t. the variable typing environment Γ defined by $\Gamma(y) = \Gamma(z) = \mathbf{0}$ and $\Gamma(x) = \mathbf{1}$. We have shown in Example 6.1 that $y \in \mathbf{0}_{l_4,\texttt{rev},\Gamma}$ holds. We just give the two subderivations of interest, corresponding to the statements of label l_2 and l_4 in $\mathbb{L}(\texttt{rev})$:

$$\frac{\Gamma(y) = \mathbf{0} \qquad \dfrac{\Gamma, \Delta \vdash^{\mathbf{1}}_{SA} x : \mathbf{1}}{\Gamma, \Delta \vdash^{\mathbf{1}}_{SA} x : \mathbf{0}}\ (SubE_{SA})}{\Gamma, \Delta \vdash^{\mathbf{1}}_{SA} l_2 : y^{\texttt{List}} := x; : \mathbf{0}}\ (Asg_{SA})$$

The assignment corresponding to vertex l_4 can be typed as follows:

$$\frac{\Gamma(y.tl) = \mathbf{0} \qquad \Gamma, \Delta \vdash^{\mathbf{1}}_{SA} z : \mathbf{0} \qquad y \in \mathbf{0}_{l_4,\texttt{rev},\Gamma} \qquad \mathbf{0} \prec \mathbf{1}}{\Gamma, \Delta \vdash^{\mathbf{1}}_{SA} l_4 : y.tl := z; : \mathbf{0}}\ (A^{\rho}_{SA})$$

The two programs of Example 3.14 and Example 3.15 can also be shown to belong to SA.

6.4 Properties of SA

6.4.1 Polynomial Time Soundness. The set of typable programs is also sound for polynomial time under termination assumption.

Theorem 6.4. $\mathsf{SAFE} \subsetneq \mathsf{SA} \subsetneq \mathsf{STR}$

Proof. The inclusions hold as any proof derivation in one system can be derived in the other and, finally, by noticing that SA enforces the stratification policy of STR. The left inclusion is strict as `rev` $\in$ SA. The right inclusion is also strict as a consequence of the decidability of SA (Theorem 6.7) vs the undecidability of STR (Theorem 4.8). □

Corollary 6.5. $\mathsf{SAFE} \cap \mathsf{TERM} \subsetneq \mathsf{SA} \cap \mathsf{TERM} \subsetneq \mathsf{STR} \cap \mathsf{TERM} \subsetneq \mathsf{POLY}$

6.4.2 Polynomial Time Completeness. As SAFE ∩ TERM is already known to be complete for polynomial time computable functions [Hainry and Péchoux 2018], we can deduce completeness from Corollary 6.5.

Theorem 6.6 (Completeness). *For any function in $f : \mathbb{N} \to \mathbb{N}$ computable in polynomial time (by a TM), there exists a program* $\mathsf{p} \in \mathsf{SA} \cap \mathsf{TERM}$ *computing* f.

6.4.3 Type Inference. We now show that type inference is decidable in SA. Let $|\mathsf{p}|$ be the size of the program p, *i.e.*, number of symbols. The shape analysis algorithm presented in Section 5.3 is terminating and, as the size of abstract heaps is exponential in $|\mathsf{p}|$ in the worst-case, we obtain the following bounds on the complexity of type inference.

Theorem 6.7 (Type inference). *For a given program* p*, the problem of deciding whether* $\mathsf{p} \in \mathsf{SA}$ *can be done in time* $2^{O(|\mathsf{p}|)}$.

Proof. SA is a subsystem of the type system of [Hainry et al. 2022], whose type inference is known to be in $O(|\mathsf{p}|^3)$ using a reduction to a 2-SAT instance.

Type inference in SA has to take into account the complexity of the shape analysis algorithm used in rule $(\mathrm{Asg}^{\rho}_{\mathsf{SA}})$. The number of nodes of any abstract heap, implementing the instrumentation predicates $\mathsf{x}^{\#}(v)$ and $\mathsf{a}^{\#}(v_1, v_2)$ is bounded exponentially in the size of the program $|\mathsf{p}|$. Indeed, let n and m be the number of variables and the total number of fields in the program p, respectively. There are at most $2^{|\mathsf{p}|}$ distinct nodes, as $n + m \leq |\mathsf{p}|$. Abstract heaps being multi-graphs whose arrows are labeled by fields, the maximal size of an abstract heap is $|\mathsf{p}| \times 2^{|\mathsf{p}|} \times 2^{|\mathsf{p}|}$. By monotonicity of the abstract semantics, for any $l \in \mathbb{L}(\mathsf{p})$, computing $\mathcal{A}^{\#}_{l,\mathsf{p}}$ can be done in time at most $|\mathsf{p}|2^{2|\mathsf{p}|}$.

The number of applications of rule $(\mathrm{Asg}^{\rho}_{\mathsf{SA}})$ is bounded by the number of field selector assignments (*i.e.*, assignments of the shape x.a := e;), hence by $|\mathsf{p}|$. Provided that all required $\mathcal{A}^{\#}_{l,\mathsf{p}}$ have been precomputed, the type inference is linear in $|\mathsf{p}|$, as in the case of TS. Consequently, the time complexity of the type inference in SA is $|\mathsf{p}|^3 + |\mathsf{p}|^2 2^{2|\mathsf{p}|} = 2^{O(|\mathsf{p}|)}$. □

7 CONCLUSION

We have exhibited a new noninterference-based policy characterizing the class of polynomial time computable functions. This policy is Π^0_1-complete and strictly generalizes previous noninterference-based type systems for complexity analysis. We have also exhibited some decidable instances such as SAFE or SA. Whereas SAFE has a weak expressiveness, SA uses shape analysis techniques to overcome this issue. In SA, types can be inferred fully automatically as no programmer annotation is needed. Moreover, the type inference algorithm has an exponential worst-case execution time for some pathological programs but is linear in practice as it depends on the number of possible aliasing configurations. Our experience to date suggests that this is unlikely to arise, as the number of possible aliasing configurations is small for most programs in practice.

ACKNOWLEDGMENTS

We are grateful to the POPL reviewers for their careful feedback and many suggestions for improving this paper. The authors have been supported by Inria associate team TC(Pro)^3.

REFERENCES

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *ESOP 2010*. 85–103. https://doi.org/10.1007/978-3-642-11957-6_6

John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Peter Naur, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. 1963. Revised report on the algorithmic language Algol 60. *Comput. J.* 5, 4 (1963), 349–367.

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

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

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

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2006. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *CAV 2006*. Springer, 386–400. https://doi.org/10.1007/11817963_35

Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. 2011. Programs with lists are counter automata. *Formal Methods Syst. Des.* 38, 2 (2011), 158–192. https://doi.org/10.1007/s10703-011-0111-7

Patrick Cousot and Radhia Cousot. 2002. Systematic design of program transformation frameworks by abstract interpretation. In *POPL 2002*. ACM, 178–190. https://doi.org/10.1145/503272.503290

Jörg Endrullis, Herman Geuvers, Jakob Grue Simonsen, and Hans Zantema. 2011. Levels of undecidability in rewriting. *Inf. Comput.* 209, 2 (2011), 227–245.

Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. 2008. A logical account of pspace. In *POPL 2008*. ACM, 121–131. https://doi.org/10.1145/1328438.1328456

Jean-Yves Girard. 1971. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*. Elsevier, 63–92. https://doi.org/10.1016/s0049-237x(08)70843-7

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

Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE. 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. https://doi.org/10.1109/sp.1984.10019

Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. 2022. A tier-based typed programming language characterizing Feasible Functionals. *Logical Methods in Computer Science* Volume 18, Issue 1 (feb 2022). https://doi.org/10.46298/lmcs-18(1:33)2022

Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. 2013. Type-based complexity analysis for fork processes. In *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, 1 (2018), 78–115. https://doi.org/10.1016/j.ic.2018.05.006

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

Nevin Heintze and Jon G Riecke. 1998. The SLam calculus: programming with secrecy and integrity. In *POPL 1998*. ACM, 365–377. https://doi.org/10.1145/268946.268976

Martin Hofmann. 2002. The strength of non-size increasing computation. In *POPL 2002*. ACM, 260–269. https://doi.org/10.1145/503272.503297

Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Information and Computation* 183, 1 (may 2003), 57–85. https://doi.org/10.1016/s0890-5401(03)00009-9

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 410–423.

Neil D. Jones. 2001. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.* 11, 1 (2001), 5–94.

Stephen Cole Kleene. 1952. *Introduction to metamathematics*. North Holland.

Cynthia Kop and Jakob Grue Simonsen. 2017. Complexity Hierarchies and Higher-order Cons-free Term Rewriting. *Log. Methods Comput. Sci.* 13, 3 (2017). https://doi.org/10.23638/LMCS-13(3:8)2017

Yves Lafont. 2004. Soft linear logic and polynomial time. *Theor. Comput. Sci.* 318, 1-2 (2004), 163–180. https://doi.org/10.1016/j.tcs.2003.10.018

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

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

Roman Manevich, Boris Dogadov, and Noam Rinetzky. 2016. From Shape Analysis to Termination Analysis in Linear Time. In *CAV 2016, Part I*. 426–446. https://doi.org/10.1007/978-3-319-41528-4_23

Jean-Yves Marion. 2011. A Type System for Complexity Flow Analysis. In *LICS 2011*. 123–132. https://doi.org/10.1109/LICS.2011.41

John McLean. 1992. Proving noninterference and functional correctness using traces. *J. Comput. Secur.* 1, 1 (1992), 37–57.

Andrew C Myers and Barbara Liskov. 1997. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review* 31, 5 (1997), 129–142.

Thomas W. Reps, Mooly Sagiv, and Alexey Loginov. 2010. Finite differencing of logical formulas for static analysis. *ACM Trans. Program. Lang. Syst.* 32, 6 (2010), 24:1–24:55. https://doi.org/10.1145/1749608.1749613

Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. 2004. Static Program Analysis via 3-Valued Logic. In *CAV 2004*. 15–30. https://doi.org/10.1007/978-3-540-27813-9_2

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*. 408–425. https://doi.org/10.1007/3-540-06859-7_148

Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19.

Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 1996. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *POPL 1996*. 16–31. https://doi.org/10.1145/237721.237725

Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. https://doi.org/10.1145/514188.514190

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. 1997. A type-based approach to program security. In *Colloquium on Trees in Algebra and Programming 1997*. 607–621.

Steve Zdancewic. 2004. Challenges for information-flow security. In *PLID'04*, Vol. 6.

Received 2022-07-07; accepted 2022-11-07