Higher-Order Abstract Non-Interference

Damiano Zanardini

Dipartimento di Informatica, Università di Verona
Strada Le Grazie 15, I-37134 Verona, Italy
zanardini@sci.univr.it

Abstract. This work proposes a type system for checking Abstract Non-Interference in the setting of simply-typed lambda calculus with basic types and recursion. A lambda-expression satisfies Abstract Non-Interference relatively to a given semantic property if an attacker which can only see program data up to that property cannot infer, by observing a computation, private data from public ones. Attackers are abstract interpretations of program semantics. The type analysis infers, for an expression, a security type which approximates the secret kernel for the expression, i.e. the most powerful harmless attacker for which the expression is secure. The type system is proven to be correct, that is, private information is not revealed to an attacker which is unable to distinguish different values belonging to the inferred type.

1 Introduction

Problem. *Information flow* [12] is one of the relevant properties of programs, regarding the possibility that some pieces of secret information can be maliciously acquired by an attacker observing the execution of the program. Data are splitted into *private* and *public*: the requirement for a program to be secure is that no attacker should be able to guess the value of private data by observing only public data and the execution of the program. A program for which this information leakage is impossible is said to have the *Non-Interference* property [8]: two executions differing only on the value of the private input cannot be distinguished by merely observing the public output.

Non-Interference turns out to be too restrictive: some information flows are actually harmless and should be allowed in order not to reject many useful programs. This led to several efforts to weaken the notion of Non-Interference into a more useful and easy to obtain property; in particular, Giacobazzi and Mastroeni [6] proposed Abstract Non-Interference (ANI), an approximated property defined into the framework of Abstract Interpretation [3, 4]. In ANI an attacker can see public data only to a certain degree of precision, then it is not able to exploit some information flows which would be detected (and result in a rejecting of the program) in standard Non-Interference analysis.

Main contribution. Abstract Non-Interference was first defined for a simple imperative language with integers as the only data type. This paper defines

ANI and the relative analysis for a typed lambda calculus with recursion by developing a type system which infers for a lambda-expression a type describing the class of attackers for which the expression is secure.

Dealing with a functional language involves the definition of ANI on higher-order values: two functions can be distinguished by an observer iff, when applied, they give distinguishable results. Our type system is constructive in the sense that it tries to compute the *secret kernel* [6], i.e. the most powerful attacker for which a given program is secure, rather than checking if a program is secure for a given attacker. The result of the type inference is an upper approximation of the secret kernel: some expressions are rejected for a certain degree of precision even if they satisfy Abstract Non-Interference to that degree.

The type system is in general undecidable (even the basic domain of integers does not need to have finite abstract domains); decidability can only be reached if the abstract domains on values are divided into a finite number of classes and some rules are added *at hand* to help the analysis.

Related work. The type-based approach for security analyses was proposed by Volpano and Smith [14], then it was translated into Abstract Interpretation in [15]. Programs are given security types (*public* or *private*) which are propagated in the composition of bigger programs; the type of the main expression is public if there is not interference. Standard information flow was defined for functional languages in the SLam calculus [9] and in Flow Caml [13], which is a real implementation of control flow analysis. SLam calculus is proven to be a special case of the Dependency Core Calculus [1], a small extension of Moggi's computational lambda calculus which aims to unify into a single calculus several dependency concepts arising in security, partial evaluation, program slicing and call-tracking. In developing the type analysis we use a notion of *ternary relation* similar to [10]; this allows comparing properties displayed by pairs of computations instead of considering a single evaluation.

2 Preliminary notions

Information flow and Non-Interference. If a user wants to keep some data confidential, he or she could take as a policy requirement that confidential data cannot affect data which are visible to other (untrusted) users. This policy allows programs to manipulate private data as long as the visible outputs of a computation do not reveal information about the hidden data. The notion of *Non-Interference* was introduced by Goguen and Meseguer in [8]: a program has the property of Non-Interference if any two executions differing only in the *private* input data (and therefore indistinguishable by an untrusted user) cannot be distinguished in their output by observing only *public* data: there must be no *information flow* from private to public data. Information flow security techniques are more powerful than standard *access control* methods [11, 12], since they do not deal only with the ability to read information, but also with the possibility that information is dangerously propagated.

Abstract Interpretation. Abstract Interpretation, introduced by the Cousots in [3], is a framework to systematically derive non-standard approximated semantics. The main notion in defining approximation is that of *abstract domain*.

Abstract domains can be formulated either in terms of Galois connections or closure operators [4]. An *upper closure operator* (uco) on a poset $\langle C, \leq \rangle$ is a function $\rho : C \mapsto C$ monotone, idempotent and extensive ($\forall x.\ x \leq \rho(x)$). The set of all ucos on C is $uco(C)$. A closure operator is uniquely determined by the set of its fixpoints (called *abstract values*); this set is (isomorphic to) the abstract domain A approximating the concrete domain C. A set $X \subseteq C$ is the set of fixpoints of a uco iff X is a *Moore-family*, i.e. $X = \mathcal{M}(X) = \{\wedge S | S \subseteq X\}$.

The notion of approximation formalizes the idea that in the abstract domain there is less complexity (A is a subset of C) together with a loss of precision (some elements of C cannot be used when doing computations on A). A computation F_C on C can be approximated with an abstract computation F_A by providing the abstract versions of constants and functions. The abstraction is *sound* if the abstract result is always a correct approximation of the concrete one: $\forall x. F_C(x) \leq F_A(x)$ (where $\leq$ should be read as *more concrete* or *more precise*).

If C is a complete lattice (with $\bot$ and $\top$ as bottom and top elements), then $uco(C)$ ordered pointwise is also a complete lattice with $\lambda x.x$ and $\lambda x.\top$ as bottom and top elements. The first describes the identity abstraction, with $A = C$ and no loss of information; the second is the trivial abstraction that reduces the concrete domain into a one-element abstract domain. The *reduced product* $\overline{\wedge}$ [4] of a set of abstract domains $\{A_i\}$ is the most abstract among the domains more concrete (i.e. closer to C) than each A_i: $\overline{\wedge}_i A_i = \mathcal{M}(\bigcup_i A_i)$.

Abstract Non-Interference. The program property of Non-Interference can be weakened (abstracted) by modelling secrecy relatively to some observable property: a secure program is one that preserves secrecy only as regards of a particular amount of information, the one that the attacker can observe.

The concrete domain is the set of all properties of values (e.g. $\wp(\mathbb{N})$ for integers, where $P \subseteq \mathbb{N}$ is the set of values satisfying the property), representing which values can be possibly distinguished by the attacker (Sec. 4). The ability of an attacker to observe data is described by abstract domains: if an attacker has precision ρ it is unable to distinguish two values v_1 and v_2 such that $\rho(\{v_1\}) = \rho(\{v_2\})$ (i.e. values having the same property ρ). A program P is secure for a pair of domains η and ρ (written $[\eta]P(\rho)$) if no information flows are detected by an attacker which can see public input and output data only up to a level of precision characterized respectively by η and ρ:

$$\eta(l_1) = \eta(l_2) \Rightarrow \rho([\![P]\!](h_1, l_1)) = \rho([\![P]\!](h_2, l_2))$$

where l_i are public data and h_i can be any (private) values; if the input values l_1 and l_2 cannot be distinguished, then it is not possibile to guess the value of h_i by observing the (abstracted) output. For a given η, the *secret kernel* is the most concrete ρ_s such that $[\eta]P(\rho_s)$. ANI allows to describe several kinds of attackers by using the opportune abstract domains; standard Non-Interference is a special case, obtained by using the identity abstract domain for all data.

3 Syntax and semantics of the language

Syntax. The language is the eager lambda calculus with arithmetical and boolean operations, conditional and recursion μ. $\mathcal{X}$ is the set of variables (denoted by x, y, $F\ldots$). The set $\mathcal{E}$ of expressions is defined by

$$e ::= n \mid b \mid x \mid e + e \mid \lambda x.e \mid e(e) \mid \mu F.e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e$$

where n and b are an integer and a boolean constant.

Semantics. The language semantics (see [2]) is described in Fig. 1 (here constants are not distinguished from their semantic interpretation). Every domain of values has a bottom element $\bot$; for functions the bottom element is $\lambda v.\bot$. The denotational semantics $[\![\_]\!] : \mathcal{E} \mapsto \mathcal{S}$ defines a call-by-value evaluation with run-time type checking (checking for type errors is left implicit).

Fig. 1. The semantics of simply-typed lambda calculus

$$n \in \mathbb{N}_\bot \triangleq \mathbb{N} \cup \{\bot\} \quad \text{integers} \qquad b \in \mathbb{B}_\bot \triangleq \{\texttt{true}, \texttt{false}, \bot\} \quad \text{booleans}$$

$$f \in \mathcal{U} \triangleq \mathbb{N}_\bot \cup \mathbb{B}_\bot \cup [\mathcal{U} \mapsto \mathcal{U}] \quad \text{values} \qquad \mathcal{W} \triangleq \{\omega\} \quad \text{wrong value}$$

$$\varepsilon \in \mathcal{S}_{env} \triangleq \mathcal{X} \mapsto \mathcal{U} \quad \text{environments} \qquad \phi \in \mathcal{S} \triangleq \mathcal{S}_{env} \mapsto \mathcal{U} \quad \text{semantic domain}$$

$$[\![n]\!]_\varepsilon \triangleq n \qquad [\![b]\!]_\varepsilon \triangleq b$$

$$[\![x]\!]_\varepsilon \triangleq \varepsilon(x) \qquad [\![e_1 + e_2]\!]_\varepsilon \triangleq [\![e_1]\!]_\varepsilon + [\![e_2]\!]_\varepsilon$$

$$[\![\lambda x.e]\!]_\varepsilon \triangleq \lambda v.[\![e]\!]_{\varepsilon[x \leftarrow v]} \qquad [\![e(e')]\!]_\varepsilon \triangleq [\![e]\!]_\varepsilon([\![e']\!]_\varepsilon)$$

$$[\![\mu F.e]\!]_\varepsilon \triangleq lfp(\lambda v.[\![e]\!]_{\varepsilon[F \leftarrow v]}) \qquad [\![e' = e'']\!]_\varepsilon \triangleq ([\![e']\!]_\varepsilon = [\![e'']\!]_\varepsilon)$$

$$[\![\texttt{if } e \texttt{ then } e' \texttt{ else } e'']\!] \triangleq \text{ if } [\![e]\!]_\varepsilon \text{ then } [\![e']\!]_\varepsilon \text{ else } [\![e'']\!]_\varepsilon$$

4 The type system

Non-Interference refers to the possibility that two computations can be distinguished by observing some public parts of data. By means of security annotations, this type analysis computes a pair of (abstract) values representing the possible simultaneous outcomes of two evaluations of e with indistinguishable input. e is secure if the attacker cannot distinguish the elements of the pair.

The type system has monomorphic types extended with security annotations. The set $\mathcal{T}$ of types contains the basic types `int` and `bool` and has $\rightarrow$ as the only constructor: $\tau, \tau' \in \mathcal{T} \Rightarrow (\tau \rightarrow \tau') \in \mathcal{T}$. Each type is a description of a set of semantic values: the function $\mathcal{V} : \mathcal{T} \mapsto \wp(\mathcal{U})$ is defined as

$$\mathcal{V}(\texttt{int}) = \mathbb{N}_\bot \qquad \mathcal{V}(\texttt{bool}) = \mathbb{B}_\bot \qquad \mathcal{V}((\tau \rightarrow \tau')) = [\mathcal{V}(\tau) \mapsto \mathcal{V}(\tau')]$$

The abstract domains. The power of an attacker in observing a computation is described by abstract domains; in general an attacker can see the n input data (free variables) and the output of the computation, then its observational power can be described by at most $n + 1$ abstract domains (on various types).

Let $C_\tau = \langle \wp(\mathcal{V}(\tau)), \subseteq \rangle$ be the concrete domain on the type τ, ordered by inclusion ($\bot \in X$ for each $X \subseteq C_\tau$). C_τ describes all the possible properties of values (a property p is described by the set of values satisfying it).

An abstract domain $\rho \in \mathcal{D}_\tau$ is the set of fixpoints of a closure operator $\rho \in uco(C_\tau)$ (we keep the same name for simplicity). The set of all abstract domains is $\mathcal{D} = \bigcup_\tau \mathcal{D}_\tau$.We write $\rho(v)$ for $\rho(\{v\})$. The ordering $\leq$ is $\subseteq$ (the described property is more precise); $\sqcup$ and $\sqcap$ are defined accordingly.

Two values cannot be distinguished by the domain iff they are mapped into the same (abstract) value: $v_1 \equiv_\rho v_2 \iff \rho(v_1) = \rho(v_2)$.

The security types. The type system uses normal types equipped with security annotations: this is described by a security relation which links the values of an expression after two possible computations.

Definition 1 (security relations and types). An expression $e : \tau$ is given a *security type* $(\!|\tau|\!)_R$, where $R \in \mathcal{R}$ is a *security relation* defined as follows:

$$\mathcal{R} = \wp(P(\tau) \times P(\tau))$$

where $P(\tau)$ is an abstract domain on τ (to be defined in Def. 8). $\bot$ is used as a shorthand for $\{(\bot, \bot)\}$; usually (s_1, s_2) or $s_1 s_2$ is written instead of $\{(s_1, s_2)\}$.

The predicate ST checks if two values v_1 and v_2 belong to the same type σ, i.e. if it is possible that, in two different computations (with possibly different input values) of an expression of type σ, the two outputs are, respectively, v_1 and v_2.

Definition 2 (same-type predicate). The predicate ST describes a ternary relation [10] between a security type and two values: the predicate is true (the triple is an element of the relation) if the values both belong to the security type.

$$\mathrm{ST}_{(\!|\tau|\!)_\bot}(v_1, v_2) \equiv v_1 = v_2 = \bot_\tau$$
$$\mathrm{ST}_{(\!|\tau|\!)_R}(v_1, v_2) \equiv \exists (s_1, s_2) \in R.\ (v_1 \in s_1 \wedge v_2 \in s_2)$$

- An expression of type $(\!|\tau|\!)_\bot$ must have value $\bot_\tau$ in every execution (then its semantics is the constant $\bot_\tau$-function).
- For $e : (\!|\tau|\!)_{(s_1,s_2)}$ any two computations must yield values belonging resp. to the classes s_1 and s_2. By definition of $\bot$, $\mathrm{ST}_{(\!|\tau|\!)_R}(\bot_\tau, \bot_\tau)$ for every R.
- For functional values the definition amounts to
$$\mathrm{ST}_{(\!|(\tau \to \tau')|\!)_R}(f_1, f_2) \equiv \forall (s_1, s_2) \in R.\ \forall v_1, v_2.$$
$$\mathrm{ST}_{(\!|\tau|\!)_{(t_1,t_2)}}(v_1, v_2) \Rightarrow \mathrm{ST}_{(\!|\tau'|\!)_{(s_1(t_1), s_2(t_2))}}(f_1(v_1), f_2(v_2))$$

ST induces an ordering on security types: $(\!|\tau|\!)_R \leq (\!|\tau'|\!)_{R'} \equiv \tau = \tau' \wedge R \leq R'$ and

$$R \leq R' \equiv \forall v_1, v_2.\ \mathrm{ST}_{(\!|\tau|\!)_R}(v_1, v_2) \Rightarrow \mathrm{ST}_{(\!|\tau|\!)_{R'}}(v_1, v_2)$$

On security relations least upper bound and greatest lower buond are defined:

$$\sqcup = \cup \qquad R' \sqcap R'' = \{(s_1 \sqcap t_1, s_2 \sqcap t_2) \mid (s_1, s_2) \in R' \wedge (t_1, t_2) \in R''\}$$

Remark 3. Given a security relation R with $S \times S \subseteq R$ and $\sqcup S = t$, it can be normalized into an equivalent relation R' with $S \times S$ replaced by (t, t), and viceversa. R' is equivalent to R but is structurally (and computationally) smaller: when applying the typing rules we do not need to consider all the pairs in $S \times S$, but only (t, t) (see also the second part of Section 9).

Lemma 4. $\forall v_1 \leq v_2.\ ST_\sigma(v_2, u) \Rightarrow ST_\sigma(v_1, u)$ *(on both arguments).*

Proof. – Basic types: $\mathbb{N}_\perp$ and $\mathbb{B}_\perp$ are flat domains with bottom; the partial ordering is $v_1 \leq v_2 \Leftrightarrow v_1 = v_2 \vee v_1 = \perp$; lemma holds by def. of $\perp$ and ST.
– Higher-order types: Ordering on functions is pointwise, then, $v_1 \leq v_2$ implies $v_1(x) \leq v_2(x)$ for all x. Result follows by induction and Def. 2 on functions.

Lemma 5. *Let ST^τ be the restriction of ST on the type τ, i.e. $ST^\tau_\sigma(v_1, v_2)$ is defined if $\sigma = (\!|\tau|\!)_R$ and $v_1, v_2 \in \mathcal{V}(\tau)$. Then ST^τ is continuous on its arguments, i.e. for directed sets S, X and Y*

$$ST^\tau_{\sqcup S}(\sqcup X, \sqcup Y) = \sqcup_{\sigma \in S} \sqcap_{x \in X} \sqcap_{y \in Y} ST^\tau_\sigma(x, y)$$

Proof. ST^τ is a function mapping $\mathcal{T}^\tau_{sec} \times \mathcal{V}(\tau) \times \mathcal{V}(\tau)$ to $\mathbb{B}$ (with $\texttt{false} < \texttt{true}$, $\sqcup \equiv \vee$ and $\sqcap \equiv \wedge$). For basic types $v \sqcup v = v$, $v \sqcup \perp = \perp \sqcup v = v$ and $\nexists (v \sqcup u)$ if $\perp \neq v \neq u \neq \perp$; for functional types $(f \sqcup g)(x) = f(x) \sqcup g(x)$. The continuity of ST^τ can be proven separately on the three arguments:

– Continuity on the first argument: $\mathrm{ST}^\tau_{\sqcup S}(x, y) = \sqcup_{\sigma \in S} \mathrm{ST}^\tau_\sigma(x, y)$ for every directed set S. This is easily proven by seeing that on booleans $\sqcup \equiv \vee$ and $\mathrm{ST}^\tau_{\sqcup S}(x, y) = \bigvee_{\sigma \in S} \mathrm{ST}^\tau_\sigma(x, y)$.
– Continuity on the second argument (third is similar): for every directed set X, $\mathrm{ST}^\tau_\sigma(\sqcup X, y) = \sqcap_{x \in X} \mathrm{ST}^\tau_\sigma(x, y)$.
  • Basic types: a directed set on integers or booleans must be either $\{\perp\}$, $\{v\}$ or $\{\perp, v\}$ for a certain v. In the first two cases the result is trivial, in the third it follows from Lemma 4.
  • Functional types: for higher order types X is directed iff for each v the set $X_v = \{f(v) \mid f \in X\}$ is directed. Then (by def. of app, see Fig. 2):

$$\begin{aligned}
&\mathrm{ST}^\tau_{(\!|(\tau' \to \tau'')|\!)_R}(\sqcup X, g) \equiv \\
\forall v_1, v_2.\ \mathrm{ST}^\tau_{(\!|\tau'|\!)_{R'}}(v_1, v_2) \Rightarrow \mathrm{ST}^\tau_{(\!|\tau''|\!)_{\mathrm{app}(R,R')}}((\sqcup X)(v_1), g(v_2)) &\equiv [hyp.] \\
\forall v_1, v_2.\ \mathrm{ST}^\tau_{(\!|\tau'|\!)_{R'}}(v_1, v_2) \Rightarrow \sqcap_{f \in X} \mathrm{ST}^\tau_{(\!|\tau''|\!)_{\mathrm{app}(R,R')}}(f(v_1), g(v_2)) &\equiv \\
\forall v_1, v_2.\ \sqcap_{f \in X} (\mathrm{ST}^\tau_{(\!|\tau'|\!)_{R'}}(v_1, v_2) \Rightarrow \mathrm{ST}^\tau_{(\!|\tau''|\!)_{\mathrm{app}(R,R')}}(f(v_1), g(v_2))) &\equiv \\
\sqcap_{f \in X} \forall v_1, v_2.\ (\mathrm{ST}^\tau_{(\!|\tau'|\!)_{R'}}(v_1, v_2) \Rightarrow \mathrm{ST}^\tau_{(\!|\tau''|\!)_{\mathrm{app}(R,R')}}(f(v_1), g(v_2))) &\equiv \\
&\sqcap_{f \in X} \mathrm{ST}^\tau_{(\!|(\tau' \to \tau'')|\!)_R}(f, g)
\end{aligned}$$

In Non-Interference data are splitted into a public and a private part: an attacker can read information from the public but not from the private. In our language input data are the free variables of an expression, and the security type of a variable characterizes it as public or private.

Definition 6 (public and private variables). $E(x)$ is the security type of x in the type environment $E \in \mathcal{T}_{env} = [\mathcal{X} \mapsto \mathcal{T}_{sec}]$.

$$\forall_1(\rho) \stackrel{\Delta}{=} \{(\rho(x), \rho(x)) \mid x \in \mathcal{V}(\tau)\} \qquad \forall_2(\rho) \stackrel{\Delta}{=} \{(\rho(x), \rho(y)) \mid x, y \in \mathcal{V}(\tau)\}$$
$$\mathit{public}(x : \tau) \Rightarrow E(x) = (\!|\tau|\!)_{\forall_1(P(\tau))} \qquad \mathit{private}(x : \tau) \Rightarrow E(x) = (\!|\tau|\!)_{\forall_2(P(\tau))}$$

where $P(\tau)$ is an abstract domain on τ, to be defined in def. 8.

It is now possible, with ST, to understand the meaning of public and private:

(i) In two different computations the values of a public variable must belong to the same element of the domain; if s describes a singleton, then $v_1 = v_2$ as in the definition of standard Non-Interference. In Abstract Non-Interference we

allow public data to be different as long as they cannot be distinguished by an observer. It should be noted that no restrictions are posed on the value of a public variable in itself: the constraints apply to values in pairs of computations. For example, in the parity abstract domain $\{\bot, even, odd, \top\}$ a public variable would have $\{(even, even), (odd, odd)\}$ as its security relation.

(ii) Nothing can be said about the values of a private variable: every two values of type τ are possible values in two different computations.

Typing rules. The type analysis assigns a security type to basic expressions (constants, variables) and infers the types for composed expressions following a set of typing rules. It is an approximate (abstract) analysis because abstract values (sets of values) rather than concrete are taken into account in computing properties: the abstract values are elements of abstract domains.

The observational power of an attacker can be described by the $n+1$-tuple IO of the abstract domains describing the ability to see input and output values (i.e. the n domains ρ_x on the n input variables and ρ_{out} on the result of the computation). When assigning a type to a (public or private) variable the natural choice is to use, in Def. 6, the domain ρ_x as $P(\tau)$. This has some limitations, as shown in the example:

Example 7. Let $e \equiv x$ and IO $= \{\rho_x, \rho_{out}\}$ where $\rho_x = \{\bot, pos, neg, \top\}$ is the domain on x (public) and $\rho_{out} = \{\bot, even, odd, \top\}$ is the one on the output. Then an attacker can observe only the sign of the input and the parity of the output. If x is given a security type based on ρ_x (i.e. $(\!|\texttt{int}|\!)_{\{(pos,pos),(neg,neg)\}}$), then, since two numbers with the same sign can have different parities, the expression is classified as insecure because the attacker can distinguish two computations with the same input sign by watching the output parity.

In the above example ρ_x is a too abstract domain to capture information about parity. Then, in general, we need to choose the right domains to assign types to basic expressions, in order to avoid some unexpected losses of information and to deal with constants. The idea is to build two sets $Z_{\texttt{int}}$ and $Z_{\texttt{bool}}$ as follows: while IO $\neq \emptyset$ pick a ρ from IO:

- If $\rho \in \mathcal{D}_{\texttt{int}}$, then ρ is inserted into $Z_{\texttt{int}}$.
- If $\rho \in \mathcal{D}_{\texttt{bool}}$, then ρ is inserted into $Z_{\texttt{bool}}$.
- If $\rho \in \mathcal{D}_{(\tau' \to \tau'')}$, then there exist ρ' and ρ'' such that the elements of ρ map elements of ρ' to elements of ρ''; ρ' and ρ'' are inserted into IO.

When IO is empty the two sets $Z_{\texttt{int}}$ and $Z_{\texttt{bool}}$ cointain all the relevant information about basic types; collecting the information into a single domain is done by computing the reduced product (Sec. 2) of all the elements of the sets, thus obtaining ρ_i and ρ_b: they are the most abstract domains capturing the relevant properties of the analysis: $\rho_i = \overline{\wedge} Z_{\texttt{int}}$ and $\rho_b = \overline{\wedge} Z_{\texttt{bool}}$. It is easy to see that, in Example 7, using $\rho_i = \overline{\wedge}\{\rho_x, \rho_{out}\}$ instead of ρ_x avoids the loss of information on the variable value: the expression is now considered as secure.

Definition 8 (abstract domains on a given type). The function $P : \mathcal{T} \mapsto \mathcal{D}$ extends the construction of ρ_i and ρ_b to function types:

$$P(\texttt{int}) = \rho_i \qquad P(\texttt{bool}) = \rho_b \qquad P((\tau \to \tau')) = P(\tau')^{P(\tau)}$$

where $\rho^{\rho'}$ is the *Reduced Cardinal Power* [4] of two domains, building the domain of the monotone functions from ρ' to ρ [5].

The syntax of the typing rules is standard: the basic judgement is $E \vdash e : \sigma$ and holds if an expression e is given a security type σ when the computation is performed in a type environment $E \in \mathcal{T}_{env}$ (input variables x have type $E(x)$).

The typing rules are described in Figure 2 and explained below (the function $\mathcal{F}$ is a rewriting of the typing rules: $\mathcal{F}(e, E) = \sigma \iff E \vdash e : \sigma$):

- `Iconst`, `Bconst`: The type for constants expresses the condition that, in any two computations, the value is indistinguishable by the abstract domain.
- `eq`: $\mathrm{eq}(R', R'')$ specifies when it is possible to have equal values for e' and e''. The (dis)equalities in the formula are treated as boolean values.
- `add`: The security type of the sum is such to contain all the values that can be obtained by adding two values of the addendi. Rules `sub`, `mul` and `div` for integers are similar, as well as rules `and`, `or`, `not` for boolean connectives.
- `lam`, `app`: The type of a function is inferred by collecting the types obtained, for all possible types of the abstraction variable, by evaluating the function body in the updated environment. Application rule is dual.
- `rec`: To compute the security type of a recursive function we need to find the fixpoint of a functional $\mathbf{G}_{E,e} = \lambda\sigma.\mathcal{F}(e, E[F \leftarrow \sigma])$. Since $\mathbf{G}$ is continuous, this can be done by starting from $\{\tau\}_\perp$ and iterating the application of $\mathbf{G}$, thus obtaining the least fixed point.
- `if`: In defining if' we have in mind these requirements:
  (1) If $x = y$ we have $\mathrm{ST}_{\{\tau\}_{\mathrm{if'(t\,t},R',R'')}}(v_1, v_2) \Leftrightarrow \mathrm{ST}_{\{\tau\}_{R'}}(v_1, v_2)$. That is, if in two computations the boolean guard is true (false), then the values of the expression are the same as the first (second) branch; therefore they belong to its security type.
  (2) If $x \neq y$, then $\mathrm{ST}_{\{\tau\}_{\mathrm{if'(t\,f},R',R'')}}(v_1, v_2) \Leftrightarrow \mathrm{ST}_{\{\tau\}_{R'}}(v_1, u_2) \wedge \mathrm{ST}_{\{\tau\}_{R''}}(u_1, v_2)$ for each u_1, u_2. That is, if in the two computations the boolean guards evaluate respectively to `true` and `false` (`false` and `true`), then the expression evaluates, in the first one, to the value of the *then* (*else*) branch, and in the second one to the value of the *else* (*then*) branch.
  The derivation of if'$(\mathrm{t\,f}, v_1, v_2)$ is illustrated in the following equalities:

$$\exists u_1, u_2.\ \mathrm{ST}_{\{\tau\}_{R'}}(v_1, u_2) \wedge \mathrm{ST}_{\{\tau\}_{R''}}(u_1, v_2) \Leftrightarrow$$
$$\exists u_1, u_2, (s_1', s_2') \in R', (s_1'', s_2'') \in R''. v_1 \in s_1' \wedge v_2 \in s_2'' \wedge u_1 \in s_1'' \wedge u_2 \in s_2' \Leftrightarrow$$
$$\mathrm{ST}_{\{\tau\}_{\mathrm{if'(t\,f},R',R'')}}(v_1, v_2)$$

5 Correctness of type inference

We are interested in a soundness result for the type inference algorithm: the inferred type is correct if it is not possible to have computations with values not

Fig. 2. The typing rules

$$\frac{}{E \vdash n : (\!|\texttt{int}|\!)_{(\rho_i(n),\rho_i(n))}}\,\texttt{[Iconst]} \qquad \frac{}{E \vdash b : (\!|\texttt{bool}|\!)_{(\rho_b(b),\rho_b(b))}}\,\texttt{[Bconst]}$$

$$\frac{}{E \vdash x : E(x)}\,\texttt{[var]} \qquad \frac{E \vdash e' : (\!|\texttt{int}|\!)_{R'} \quad E \vdash e'' : (\!|\texttt{int}|\!)_{R''}}{E \vdash e' = e'' : (\!|\texttt{bool}|\!)_{\mathrm{eq}(R',R'')}}\,\texttt{[eq]}$$

$$\frac{E \vdash e' : (\!|\texttt{int}|\!)_{R'} \qquad E \vdash e'' : (\!|\texttt{int}|\!)_{R''}}{E \vdash e' + e'' : (\!|\texttt{int}|\!)_{\mathrm{add}(R',R'')}}\,\texttt{[add]}$$

$$\frac{\forall s_1, s_2 \in P(\tau).\ E[x \leftarrow (\!|\tau|\!)_{(s_1,s_2)}] \vdash e : (\!|\tau'|\!)_{R_{s_1,s_2}}}{E \vdash \lambda x : \tau.\ e : (\!|(\tau \to \tau')|\!)_{\mathrm{lam}(P(\tau),\lambda x,y.R_{x,y})}}\,\texttt{[lam]}$$

$$\frac{E \vdash e : (\!|(\tau' \to \tau)|\!)_R \qquad E \vdash e' : (\!|\tau'|\!)_{R'}}{E \vdash e(e') : (\!|\tau|\!)_{\mathrm{app}(R,R')}}\,\texttt{[app]} \qquad \frac{E[F \leftarrow \sigma] \vdash e : \sigma \quad \sigma \text{ minimal}}{E \vdash \mu F.e : \sigma}\,\texttt{[rec]}$$

$$\frac{E \vdash b : (\!|\texttt{bool}|\!)_{R_b} \qquad E \vdash e' : (\!|\tau|\!)_{R'} \qquad E \vdash e'' : (\!|\tau|\!)_{R''}}{E \vdash \texttt{if } b \texttt{ then } e' \texttt{ else } e'' : (\!|\tau|\!)_{\mathrm{if}(R_b,R',R'')}}\,\texttt{[if]}$$

$$\frac{E \vdash e : (\!|\tau|\!)_R \qquad R \le R'}{E \vdash e : (\!|\tau|\!)_{R'}}\,\texttt{[subt]}$$

$$\begin{aligned}
\mathrm{eq}(R', R'') &= \bigcup\{\{\rho_b(s_1 \sqcap t_1 \neq \bot), \rho_b(s_1 = t_1 = \{v\})\} \times \\
&\qquad \{\rho_b(s_2 \sqcap t_2 \neq \bot), \rho_b(s_2 = t_2 = \{u\})\} | (s_1, s_2) \in R', (t_1, t_2) \in R''\} \\
s + t &= \rho_i(\{v + u \mid v \in s \wedge u \in t\}) \\
\mathrm{add}(R', R'') &= \{(s_1 + t_1, s_2 + t_2) \mid (s_1, s_2) \in R', (t_1, t_2) \in R''\} \\
\mathrm{lam}(X, F) &= \{(t_1, t_2) \mid \forall s_1, s_2 \in X.(t_1(s_1), t_2(s_2)) \in F(s_1, s_2)\} \\
\mathrm{app}(R', R'') &= \{(t_1(s_1), t_2(s_2)) \mid (t_1, t_2) \in R' \wedge (s_1, s_2) \in R''\} \\
\mathrm{if}(R_b, R', R'') &= \bigcup\{\mathrm{if'}(xy, R', R'') \mid (X, Y) \in R_b \wedge x \in X \wedge y \in Y\} \\
\mathrm{if'}(\mathrm{t}\,\mathrm{t}, R', R'') &= R' \qquad (\mathrm{t} \equiv \texttt{true}, \mathrm{f} \equiv \texttt{false}) \\
\mathrm{if'}(\mathrm{f}\,\mathrm{f}, R', R'') &= R'' \\
\mathrm{if'}(\mathrm{t}\,\mathrm{f}, R', R'') &= \{(s', t'') \mid (s', t') \in R' \wedge (s'', t'') \in R''\} \\
\mathrm{if'}(\mathrm{f}\,\mathrm{t}, R', R'') &= \{(s'', t') \mid (s', t') \in R' \wedge (s'', t'') \in R''\}
\end{aligned}$$

belonging to the type. The function $\mathrm{ENV}_E(\varepsilon_1, \varepsilon_2) = \forall x.\ \mathrm{ST}_{E(x)}(\varepsilon_1(x), \varepsilon_2(x))$ extends the ST predicate to environments.

The correctness theorem is proven by induction on the structure of a derivation: this is the same as induction on expressions (every typing rule *builds* an expression out of zero, one or more subexpressions), except for subtyping (in this rule the expression is not composed). Subtyping rule can be applied virtually everywhere and many times without affecting the validity of the proof. In reasoning about the application of rules, we can infer (backwards) the type of the subexpressions by observing the main type: for example, if `add` rule is applied, then from the type $(\!|\texttt{int}|\!)_R$ of an expression $e' + e''$ it is possible to say that e_1 and e_2 have types $(\!|\texttt{int}|\!)_{R'}$ and $(\!|\texttt{int}|\!)_{R''}$ with $R = \mathrm{add}(R', R'')$ (otherwise the rule would not have been applicable).

Theorem 9 (Correctness). $E \vdash e : \sigma \ \wedge \ ENV_E(\varepsilon_1, \varepsilon_2) \Rightarrow ST_\sigma([\![e]\!]_{\varepsilon_1}, [\![e]\!]_{\varepsilon_2})$

Proof. Induction on the last rule applied in the derivation (some cases omitted).

- $e \equiv n$: We have $[\![e]\!]_{\varepsilon_1} = [\![e]\!]_{\varepsilon_2} = n$ and $E \vdash n : (\mathtt{int})_{(\rho_i(n),\rho_i(n))}$; then $\mathrm{ST}_{(\mathtt{int})_{(\rho_i(n),\rho_i(n))}}(n, n)$ since, by definition of closure operators, $n \in \rho_i(n)$.
- $e \equiv x : E \vdash x : E(x)$ and $[\![x]\!]_{\varepsilon_i} = \varepsilon_i(x)$ imply (def. ENV) $\mathrm{ST}_{E(x)}(\varepsilon_1(x), \varepsilon_2(x))$.
- $e \equiv e' + e''$:
$$\begin{array}{r} E \vdash e' + e'' : (\mathtt{int})_{\mathrm{add}(R',R'')} \Longrightarrow \\ E \vdash e' : (\mathtt{int})_{R'} \ \wedge \ E \vdash e'' : (\mathtt{int})_{R''} \Longrightarrow [hyp.] \\ \mathrm{ST}_{(\mathtt{int})_{R'}}([\![e']\!]_{\varepsilon_1}, [\![e']\!]_{\varepsilon_2}) \ \wedge \ \mathrm{ST}_{(\mathtt{int})_{R''}}([\![e'']\!]_{\varepsilon_1}, [\![e'']\!]_{\varepsilon_2}) \Longrightarrow [def. \text{ add}] \\ \mathrm{ST}_{(\mathtt{int})_{\mathrm{add}(R',R'')}}([\![e' + e'']\!]_{\varepsilon_1}, [\![e' + e'']\!]_{\varepsilon_2}) \end{array}$$
- $e \equiv \lambda x : \tau.\ e_0$: let us suppose to have, for every pair $(s_1, s_2) \in P(\tau) \times P(\tau)$, two values $v_1^{s_1,s_2}$ and $v_2^{s_1,s_2}$ such that $\mathrm{ST}_{(\tau)_{(s_1,s_2)}}(v_1^{s_1,s_2}, v_2^{s_1,s_2})$.
Then, by def. of ENV, $\mathrm{ENV}_{E[x \leftarrow (\tau)_{(s_1,s_2)}]}(\varepsilon_1[x \leftarrow v_1^{s_1,s_2}], \varepsilon_2[x \leftarrow v_2^{s_1,s_2}])$.
Let $\mathrm{lam}(P(\tau), \lambda x, y.R_{x,y})$ be defined as in the rule for abstraction. Then
$$\begin{array}{r} E \vdash \lambda x : \tau.\ e_0 : (\!(\tau \to \tau')\!)_{\mathrm{lam}(P(\tau),\lambda x,y.R_{x,y})} \Longrightarrow \\ \forall s_1, s_2 \in P(\tau).\ E[x \leftarrow (\tau)_{(s_1,s_2)}] \vdash e_0 : (\tau')_{R_{s_1,s_2}} \Longrightarrow [hyp.] \\ \forall s_1, s_2 \in P(\tau).\ \mathrm{ST}_{(\tau')_{R_{s_1,s_2}}}([\![e_0]\!]_{\varepsilon_1[x \leftarrow v_1^{s_1,s_2}]}, [\![e_0]\!]_{\varepsilon_2[x \leftarrow v_2^{s_1,s_2}]}) \Longrightarrow \\ \mathrm{ST}_{\mathrm{lam}(P(\tau),\lambda x,y.R_{x,y})}([\![e]\!]_{\varepsilon_1}, [\![e]\!]_{\varepsilon_2}) \end{array}$$
The last step follows from the definition of $[\![\bullet]\!]$, ST and lam.
- $e \equiv e'(e'')$:
$$\begin{array}{r} E \vdash e'(e'') : (\tau)_{\mathrm{app}(R',R'')} \Longrightarrow \\ E \vdash e' : (\!(\tau'' \to \tau)\!)_{R'} \ \wedge \ E \vdash e'' : (\tau'')_{R''} \Longrightarrow [hyp.] \\ \mathrm{ST}_{(\!(\tau'' \to \tau)\!)_{R'}}([\![e']\!]_{\varepsilon_1}, [\![e']\!]_{\varepsilon_2}) \ \wedge \ \mathrm{ST}_{(\tau'')_{R''}}([\![e'']\!]_{\varepsilon_1}, [\![e'']\!]_{\varepsilon_2}) \Longrightarrow [def. \text{ app}] \\ \mathrm{ST}_{(\tau)_{\mathrm{app}(R',R'')}}([\![e'(e'')]\!]_{\varepsilon_1}, [\![e'(e'')]\!]_{\varepsilon_2}) \end{array}$$
- $e \equiv \mu F.e_0$: We take $\sigma = (\tau)_R$; the non-security part τ is the type inferred for e by a standard type inference algorithm (we have no problems with non-termination since the type system is monomorphic). Then $lfp(\mathbf{G}_{E,e_0}) = (\tau)_R$ where $\mathbf{G}$ is defined above (`rec` rule).
The following result (for each f_1, f_2, σ_0) is true by inductive hypothesis ($\mathbf{H}_{\varepsilon,e} = \lambda v.\ [\![e]\!]_{\varepsilon[F \leftarrow v]}$ is the semantic function for recursion):
$$\mathrm{ST}_{\sigma_0}(f_1, f_2) \Rightarrow \mathrm{ST}_{\mathbf{G}_{E,e_0}(\sigma_0)}(\mathbf{H}_{\varepsilon_1,e_0}(f_1), \mathbf{H}_{\varepsilon_2,e_0}(f_2))$$
So we have the chain of implications
$$\begin{array}{c} \mathtt{true} = \mathrm{ST}_{(\tau)_\bot}(\bot, \bot) \Rightarrow \\ \mathrm{ST}_{\mathbf{G}_{E,e_0}((\tau)_\bot)}(\mathbf{H}_{\varepsilon_1,e_0}(\bot), \mathbf{H}_{\varepsilon_2,e_0}(\bot)) \Rightarrow \ldots \Rightarrow \\ \mathrm{ST}_{\mathbf{G}^n_{E,e_0}((\tau)_\top)}(\mathbf{H}^n_{\varepsilon_1,e_0}(\bot), \mathbf{H}^n_{\varepsilon_2,e_0}(\bot)) \end{array}$$
Let $G_n = \mathbf{G}^n_{E,e_0}((\tau)_\bot)$, $S = \{G_n\}$, $H^1_n = \mathbf{H}^n_{\varepsilon_1,e_0}(\bot)$, $X = \{H^1_n\}$, $H^2_n = \mathbf{H}^n_{\varepsilon_2,e_0}(\bot)$ and $Y = \{H^2_n\}$. Then, by Lemma 5,
$$\mathrm{ST}_{(\tau)_R}([\![e]\!]_{\varepsilon_1}, [\![e]\!]_{\varepsilon_2}) \equiv \mathrm{ST}_{\sqcup S}(\sqcup X, \sqcup Y) \equiv \sqcup_{\sigma \in S} \sqcap_{x \in X} \sqcap_{y \in Y} \mathrm{ST}_\sigma(x, y)$$
and, for every $H^1_n \in X$, $H^2_m \in Y$, we can take $p = max(m, n)$ to have $\mathrm{ST}_\sigma(H^1_p, H^2_p) \Rightarrow \mathrm{ST}_\sigma(H^1_n, H^2_m)$ (by Lemma 4) and $\mathrm{ST}_{G_p}(H^1_p, H^2_p)$ (by the chain of implications above).
Therefore it is always possible to find $\sigma \in S$ such that $\mathrm{ST}_\sigma(x, y)$ is true; consequently $\mathrm{ST}_{(\tau)_R}([\![e]\!]_{\varepsilon_1}, [\![e]\!]_{\varepsilon_2})$ is also true.

6 Computations and attackers

As shown above, there is a close correspondence between abstract domains and security types: types identify properties which domains cannot distinguish. This relation between $\mathcal{T}_{sec}$ and $\mathcal{D}$ can be formalized as $\rho \rightsquigarrow \sigma = \forall v_1, v_2.\ \mathrm{ST}_\sigma(v_1, v_2) \Rightarrow v_1 \equiv_\rho v_2$, meaning that ρ cannot distinguish σ-related values. In this case we say that ρ is *corresponding* to σ.

For a given σ there always exists such a ρ (the top abstract domain, characterizing a blind observer, is corresponding to every type). The functions $\alpha : \mathcal{T}^\tau_{sec} \mapsto \mathcal{D}_\tau = \lambda\sigma.\ \overline{\wedge}\,\{\rho \mid \rho \rightsquigarrow \sigma\}$ and $\gamma : \mathcal{D}_\tau \mapsto \mathcal{T}^\tau_{sec} = \lambda\rho.\{\!|\tau|\!\}_{\{(s_1,s_2)|\forall v_1 \in s_1, v_2 \in s_2.\, v_1 \equiv_\rho v_2\}}$ clearly identify a Galois connection [3] between $\mathcal{T}^\tau_{sec}$ and $\mathcal{D}_\tau$.

We say that a type environment E defines a policy consistent with IO (Sec. 4) if, for every variable x, $E(x)$ is defined (Defs. 6 and 8) using the domains ρ_i and ρ_b induced by IO and the information about public and private data. The meaning of this notion is that the policy, together with the classification of the variables into private and public, is calibrated on the data we want to protect and on the attacker we want to be protected from.

The Abstract Non-Interference condition for an expression e, an attacker IO and a consistent policy described by E can be written as:

$$\mathrm{ANI}_E(e, \mathrm{IO}) \stackrel{\Delta}{=} \mathrm{ENV}_E(\varepsilon_1, \varepsilon_2) \Rightarrow [\![e]\!]_{\varepsilon_1} \equiv_{\rho_{out}} [\![e]\!]_{\varepsilon_2}$$

Thus, if an attacker cannot distinguish inputs, neither can he distinguish outputs. It is easy to see that this is a translation of the original definition of ANI into our functional framework.

A direct corollary of Theorem 9 is (for E consistent with IO and $\rho_{out} \in \mathrm{IO}$):

Theorem 10. $\rho_{out} \rightsquigarrow \sigma \wedge E \vdash e : \sigma \Rightarrow ANI_E(e, IO)$

The inferred type is an upper approximation of the secret kernel (Sec. 2): some non-interfering expressions are rejected since the abstraction induced by the abstract domains leads to a loss of information. For example, let ρ_i be the parity domain; then $e \equiv x + x$, where x is a private input variable, would be typed with $\{\!|\texttt{int}|\!\}_{\top\top}$, even if it clearly is always even. In terms of abstract interpretation, this is an incompleteness situation [7]: viewing the type of an object as an abstraction of its meaning, it turns out that abstracting (typing) the final value of the concrete computation (yielding, in this case, $(even, even)$) is not equal to performing the abstract computation (the type inference process) starting from the abstract values (types) of the input. To avoid this kind of problems there should be a set of axioms and rules (in this case a rule stating that $x + x = 2 * x$ is needed) giving informations about special cases; however, in general it is not possible to have a complete rule system to handle such situations.

7 An example

In this example it is possible to see how an expression showing (standard) dangerous information flows can be accepted by this type system if an attacker cannot see anything but the parity of integer numbers.

The evaluation of the expression yields a function from integers to integers, then the observational power of the attacker on the output must be an abstract domain on ($\texttt{int} \to \texttt{int}$). The only input datum is the secret free variable y; since y is referred in the body of the function an algorithm for standard Non-Interference would find a forbidden information flow from y to the result of the computation (in facts, the result of the function applied to a value v depends on both v and y). However, our type system is able to accept the expression as secure since no information about the secret data can be revealed by observing the parity of numbers (for every value of y, $[\![e]\!](v)$ is an even value).

The expression to analyze is

$$e \;\equiv\; \mu F.\ \lambda x.\ \texttt{if}\ x = 0\ \texttt{then}\ 2\ \texttt{else}\ 2 * y * F(x-1)$$

and the observational power of a generic attacker is

$$\rho_i = \{\bot, e, o, \top\} \qquad \rho_b = \mathbb{B}_\bot \times \mathbb{B}_\bot$$
$$\rho_{out} = \{\langle e \mapsto e; o \mapsto e\rangle, \langle e \mapsto e; o \mapsto o\rangle, \langle e \mapsto o; o \mapsto e\rangle, \langle e \mapsto o; o \mapsto o\rangle\}$$

- the only visible information on integers is parity (e is even, o is odd);
- no abstraction on boolean values (the attacker can see the truth value of boolean data). In this case we are not following the definition of ρ_b ($Z_{\texttt{bool}} = \emptyset$ should imply $\rho_b = \overline{\wedge}(\emptyset) = \top$): the analysis will be more precise.
- functions from integers to integers are divided by the abstract domain into five classes (the notation should be clear): functions (i) mapping all numbers to even numbers; (ii) keeping the parity value; (iii) inverting the parity value; (iv) mapping all numbers to odd numbers; (v) all the other functions (these functions are mapped to $\top$). ρ_{out} is an abstraction of $P((\texttt{int} \to \texttt{int}))$.

We evaluate e in the type environment E (with $\mathrm{ENV}_E(\varepsilon_1, \varepsilon_2)$); the fact that y is private is written as $E(y) = (\top, \top) = \{ee, eo, oe, oo\}$ (ab stands for (a, b)).

The evaluation begins with the first iteration of the fixpoint construction; let E' be $E[F \leftarrow (\!|(\texttt{int} \to \texttt{int})|\!)_{\bot\bot}]$, and E'_{ab} be $E'[x \leftarrow ab]$.

(0) $E'_{ee} \vdash 0 : (\!|\texttt{int}|\!)_{ee}$ [Iconst]
(1) $E'_{ee} \vdash 1 : (\!|\texttt{int}|\!)_{oo}$ [Iconst]
(2) $E'_{ee} \vdash 2 : (\!|\texttt{int}|\!)_{ee}$ [Iconst]
(3) $E'_{ee} \vdash y : (\!|\texttt{int}|\!)_{ee,eo,oe,oo}$ [var]
(4) $E'_{ee} \vdash x : (\!|\texttt{int}|\!)_{ee}$ [var]
(5) $E'_{ee} \vdash x - 1 : (\!|\texttt{int}|\!)_{oo}$ $[(1,4), \texttt{min}]$
(6) $E'_{ee} \vdash F : (\!|(\texttt{int} \to \texttt{int})|\!)_{\bot\bot}$ [var]
(7) $E'_{ee} \vdash F(x-1) : (\!|\texttt{int}|\!)_{\bot\bot}$ $[(5,6), \texttt{app}]$
(8) $E'_{ee} \vdash y * F(x-1) : (\!|\texttt{int}|\!)_{\bot\bot}$ $[(3,7), \texttt{mul}]$
(9) $E'_{ee} \vdash 2 * y * F(x-1) : (\!|\texttt{int}|\!)_{\bot\bot}$ $[(2,8), \texttt{mul}]$
(10) $E'_{ee} \vdash x = 0 : \{\mathrm{t\,t}, \mathrm{t\,f}, \mathrm{f\,t}, \mathrm{f\,f}\}$ $[(0,4), \texttt{eq}]$
(11) $E'_{ee} \vdash \texttt{if}\ x = 0\ \texttt{then}\ 2\ \texttt{else}\ 2 * y * F(x-1) : (\!|\texttt{int}|\!)_{R_{e\bot}}$ $[(2,9,10), \texttt{if}]$

$$R_{e\bot} = \{ee, \bot\bot, e\bot, \bot e\} = ee$$

To get the value of the lambda-expression the evaluation must be done also in the other E'_{ab}, where $a, b \in \{\bot, e, o, \top\}$; the final security relation is

$$R_0 = \{(f, g) \mid f(e) \in \{e, \bot\} \wedge f(o) = \bot \wedge g(e) \in \{e, \bot\} \wedge g(o) = \bot\}$$

giving the typing judgement $E' \vdash e : (\!|(\texttt{int} \to \texttt{int})|\!)_{R_0}$.

So far the first step of the fixpoint computation; the second step is performed in the type environment $E'' = E[F \leftarrow (\!|(\texttt{int} \to \texttt{int})|\!)_{R_0}]$.

The rest of the computation is omitted; at the second step the fixpoint is reached, giving a functional security type σ_{fix} for which the result of the application is $(\!|\texttt{int}|\!)_{ee}$ for every parity value of the input:

$$\sigma_{fix} = (\!|(\texttt{int} \to \texttt{int})|\!)_{\{(\lambda s.e, \lambda s.e)\}}$$

It is then easy to see that $\rho_{out} \rightsquigarrow \sigma_{fix}$, i.e. if two functions f_1 and f_2 satisfy $\mathrm{ST}_{\sigma_{fix}}(f_1, f_2)$, then they both belong to the first equivalence class described by the abstract domain ρ_{out}: the expression e is secure.

8 Making it more practical

Type approximation. Remark 3 shows how a security relation R can be transformed into an equivalent yet simpler one R' by replacing some elements of an abstract domain with their least upper bound; namely, given a set S of elements, the set of pairs $S \times S \subseteq R$ can be replaced by $\{(\sqcup S, \sqcup S)\}$.

This can be done even if some elements of $S \times S$ are missing in R, thus introducing an upper approximation R'' of R: if $X \subseteq S \times S$ and $X \subseteq R$, R'' is obtained by replacing X with $\{(\sqcup S, \sqcup S)\}$, and would be equivalent to $R \cup (S \times S)$.

This approximation could be performed via subtyping after the application of each typing rule. However, it is not clear when such an operation can be applied without losing too much precision; in particular, we have to choose the right S and decide whether X contains *enough* elements of $S \times S$.

A possible approach is to consider S and X good candidates if X contains all the elements of $\{(s, s) | s \in S\}$ plus some pair (s, t) with $s \neq t$, thus leading to the following modified subtyping rule:

$$\frac{E \vdash e : (\!|\tau|\!)_R \quad X \subseteq R \quad X \subseteq S \times S \quad X \supset \{(s, s) | s \in S\}}{E \vdash e : (\!|\tau|\!)_{(R \setminus X) \cup \{(\sqcup S, \sqcup S)\}}}\ \texttt{[subt2]}$$

The bigger S is, the more the loss of information; then, in order not to lose too much precision, this rule should be applied to a small enough set S. This approximation acts on non-public types (containing at least one pair (s_1, s_2)) and transforms them into approximated (less precise) types with less elements, thus improving complexity results.

Lazy lambda abstraction. In the rule `lam` some computations are necessary for every pair in $P(\tau) \times P(\tau)$; however, in many cases most of them are not used in the rest of type inference (e.g. when the function is applied to a constant value). In order to avoid useless computations, the type of a function shall be computed only as long as it is used in the following derivations; for example, if F is applied to a constant with type $(\!|\texttt{int}|\!)_{ee}$ it is useless to infer its type for oo or $\perp e$. Such a strategy can reduce considerably the complexity of type inference.

Abstract operators. The application of arithmethic rules, such as `add`, involves performing integer operations on possibly infinite sets (such as even numbers). This is clearly impractical unless some computation rules are provided; for

example, rules like *even plus odd equals odd* would solve the problem of adding infinite sets in the parity abstract domain. Such a set of rules cannot, in general, be complete nor automatically generated.

9 Conclusions

This type system is an attempt to compute the secret kernel for a given expression; the inferred security type is an upper approximation of the secret kernel (Section 6), i.e. some harmless attackers are erroneously considered as dangerous.

Computability. This type system is, in general, undecidable: it is not surprising, since there are infinite semantic domains (e.g. integers and functions).

When a concrete domain is infinite, either the abstract domain is infinite or some elements of the abstract domain represent an infinite set of concrete values. In the first case some rules, like `lam`, can diverge, because of the universal quantification on the infinite set $P(\tau)$. In the second some operators on security types must deal with infinite sets (e.g. the sum of the sets of evens and odds).

Then, to have decidability the abstract domains must be finite and some set of rules must be provided to help computations (see third part of Section 8).

Complexity. Provided decidability conditions are met, some complexity results can be obtained. Let N be the (finite) cardinality of ρ_i and p be the highest number of arrow constructors occurring in the type of subexpressions. The cardinality of a domain on functions is, in the worst case, superexponential on N (bounded by $\lambda n.2^n$ applied p times to N); since, in `lam` rule, every pair of elements of the domain must be checked, the complexity of the algorithm is exponential on N.

Section 8 (first part) shows how a security relation can be transformed into a simpler one by introducing some loss of information. This can have significant benefits on complexity, since it decreases the numbers of elements to check in a security relation. Some rule better than `subt2` could be possibly found to improve the ratio *complexity benefits / precision loss*.

Again, Section 8 (second part) outlines lazy type inference as a method to avoid useless computations in lambda abstractions. This is particularly useful in presence of big domains and many applications to constant values.

Future work. Some features can be added to the language, such as product types or polymorphism. A real-world language could be considered, as in [13].

The complexity of the algorythm could be improved by finding simpler ways to manipulate security types; in particular, the set of operation rules (Sec. 8, third part) should be designed in order to be efficient and partially mechanizable (i.e. automatically generated given an abstract domain).

The approximation of security relations can significantly improve the complexity; however, a proper rule to do abstractions should be found in order to reach the best tradeoff between complexity and precision.

One of the main features of the type system is that it builds the secret kernel rather than checking Non-Interference for a given attacker; this constructive method could be applied to other classes of languages, such as imperative and object-oriented.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160. ACM Press, Jan. 1999.
2. P. Cousot. Types as abstract interpretations, invited paper. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 316–331. ACM Press, Jan. 1997.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.
5. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. International Conf. on Computer Languages*, pages 95–112. IEEE Computer Society Press, May 1994.
6. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197. ACM Press, Jan. 2004.
7. R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In *Proc. International Conf. on Algebraic Methodology and Software Technology*, volume 1349 of *LNCS*, pages 231–245. Springer-Verlag, 1997.
8. J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
9. N. Heintze and J. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*. ACM Press, Jan. 1998.
10. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Dept. of Computing, Imperial College of Science Technology and Medicine, 1991.
11. B. Lampson. Protection. In *Proc. Princeton Symp. on Information Sciences and Systems*, pages 437–443, Princeton University, Mar. 1971. Reprinted in *Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.
12. A. Myers and A. Sabelfeld. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
13. F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.
14. D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, Apr. 1997.
15. M. Zanotti. Security typings by abstract interpretation. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 360–375. Springer-Verlag, Sept. 2002.