Int. J. Inf. Secur. (2007) 6:85–106
DOI 10.1007/s10207-007-0015-0

SPECIAL ISSUE PAPER

Instruction-level security typing by abstract interpretation

Nicoletta De Francesco · Luca Martini

Published online: 13 February 2007

Abstract We present a method based on abstract interpretation to check secure information flow in programs with dynamic structures where input and output channels are associated with security levels. In the concrete operational semantics each value is annotated by a security level dynamically taking into account both the explicit and the implicit information flows. We define a collecting semantics which associates with each program point the set of concrete states of the machine when the point is reached. The abstract domains are obtained from the concrete ones by keeping the security levels and forgetting the actual values. Using this framework, we define an abstract semantics, called instruction-level security typing, that allows us to certify a larger set of programs with respect to the typing approaches to check secure information flow. An efficient implementation is shown, operating a fixpoint iteration similar to that of the Java bytecode verification.

Keywords Abstract interpretation · Information flow · Language based security

This work was partially supported by the Italian COFIN 2004 project "AIDA: Abstract Interpretation Design and Application".

N. De Francesco · L. Martini (✉)
Dipartimento di Ingegneria dell'Informazione,
Università di Pisa, Via Diotisalvi, 2, 52126 Pisa, Italy
e-mail: luca.martini@iet.unipi.it

N. De Francesco
e-mail: nico@iet.unipi.it

1 Introduction

Protecting confidential information has ever been an issue on computer systems. One of the most widespread methods to ensure confidentiality is using some Discretionary Access Control (DAC) mechanisms, both to prevent unauthorized access and to permit authorized access to information, for a given policy of authorization. As reported by [35], a DAC is defined as follows:

> "A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are 'discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject."

Unfortunately, the DAC mechanism can be in some cases inadequate. In fact, although it checks information release, it does not control its propagation. The *secure information flow* within programs in multilevel secure systems requires that information at a given security level does not flow to lower levels [18,19]. Analyzing secure information flow allows a finer inspection of confidentiality than that obtained by using DAC mechanisms. In fact, checking information flows makes it possible to control, once given an access right, whether the accessed information is properly used, according to some confidentiality policy. Assume that `x` and `y` are variables. Examples of information flows are the instructions `x:=y;` and `if (y=0) then x:=0; else x:=1;`. In the first case there is an *explicit* information flow from `y` to `x`, while, in the second case, there is an *implicit* information flow: in both cases, checking the final value of `x` reveals information on the value of `y`.

We consider sequential programs communicating with the external environment by means of input and output channels. The program defines also a security policy by assigning a security level to each channel. A program has secure information flow if the observation of a channel having some security level does not reveal any information about the values input from channels associated with higher security levels. The language includes dynamic structures and pointers.

We give a framework for analyzing secure information flow based on Abstract Interpretation (AI). AI [13–15] is a method for analyzing programs in order to collect approximate information about their run-time behavior. It is based on a non-standard semantics, that is a semantic definition in which a simpler (abstract) domain replaces the standard (concrete) one, and the operations are interpreted on the new domain. Using this approach different analyses can be systematically defined. Moreover, the proof of the correctness of the analysis can be done in a standard way.

In the paper we define a concrete instrumented operational semantics which handles, in addition to execution aspects, the level of the information flows of the program. The basis of the approach is that each value is annotated by a security level, representing the lub of the levels of the implicit and explicit flows on which the value depends. Also each channel is associated with a security level, representing the lub of the levels of the data present in the channel. The level of the input data is assumed to be that specified for the corresponding channel by the security policy. The level of data flowing through the variables and structures of the program is calculated dynamically taking into account the information flows. We define a collecting semantics which associates with each program point (instruction) the set of concrete states in which the machine can be when the point is reached.

We remark that the secure information flow property is defined in terms of independence of values produced for channels with a given security level from values taken from channels with higher security levels. A main result of the paper is the soundness of the concrete semantics. We prove that the program is secure if in all states of the collecting semantics the level of each channel is lower than or equal to that specified by the policy defined by the program. This theorem justifies the concrete semantics by relating it with the noninterference property. It shows that the concrete semantics correctly manipulates security annotations of data, that is data are associated with the security level on which they actually depend.

The abstract domains are obtained from the concrete ones by keeping the security levels and forgetting the actual values. A main point is the abstract domain of references. A reference in the concrete domain holds the creation point (that is the `new` instruction) of the structure it refers to. This enables the analysis to distinguish between structures created at different instructions. In the abstract semantics, each `new` instruction represents a different abstract object, and, to perform aliasing control, each abstract reference holds the set of the all possible abstract objects it can refer to.

Starting from the concrete semantics, different abstractions can be performed to statically analyze information flow. Higher is the abstraction, higher is the efficiency of the analysis, but lower is the level of accuracy. The most popular approach to static secure information flow is based on *security typing*: each variables has a security annotation which can be seen as a part of its type and secure information flow is checked by means of type-inference: see, for example [1,8,22,28,33,40,41], while a recent survey is [39]. The advantage of these approaches is efficiency, since algorithms based on type inference can be used. Security typing can be modeled in our framework as the highest abstraction of the concrete semantics. In the paper we show a different abstraction, which is more precise than standard security typing, since it produces a more refined information. In fact, it is able to certify a larger class of programs with respect to typing approaches to security. A state of the abstract semantics is a table having a row for each instruction. Each row is the abstraction of all concrete states in which the machine can be when executing the corresponding instruction. The program is safe if in all rows of the abstract semantics the level of each channel is lower than or equal to that specified by the policy defined by the program. A main difference between this approach and security typing is that we do not assign a security level to each variable, but to each pair variable-instruction. The security type computed by the abstract semantics for a variable x and instruction t is the least upper bound of the secrecy levels of the information flows on which x depends when t is executed in any possible computation. Hence, we allow a register to hold data with a different secrecy level for different instructions of the program, while in security typing a same security type is computed for each variable and must hold for the whole program. We can call our approach *instruction-level security typing*, in that it infers for each variable a security type for each instruction of the program. Instruction-level security typing is inspired by bytecode verification [31], applied to Java bytecode, which is the intermediate code produced by Java compilers. The bytecode verifier, in order to accomplish safety checks on the code, performs an instruction-level typing algorithm: for each instruction it infers a type for every register and stack element. The instruction-level security typing can be performed

by an efficient fixpoint iteration algorithm, similar to that used by bytecode verification. In the paper we also present a tool implementing the method. The instruction-level security typing was firstly sketched in [3,16].

Section 2 describes some scenarios related to information flow checking, Sect. 3 introduces the languages and the definition of secure information flow. Section 4 defines the concrete semantics. Section 5 describes the abstract domains, while Sect. 6 defines the abstract semantics. In Sect. 7a prototype tool that implements the proposed analysis is shown, together with some examples. Finally, Sect. 8 discusses related work and concludes.

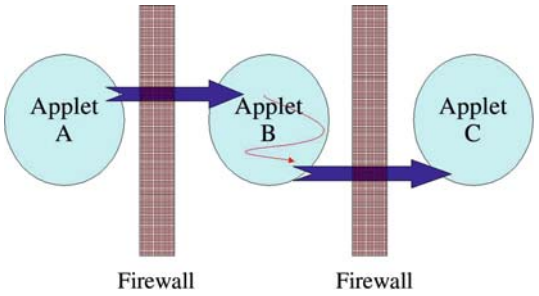

Fig. 1 A Java Card that hosts three applets. The applet `B` can disclose some `A`'s data to applet `B`, without violating the firewall policy

2 Motivations

In this section we describe some scenarios to better clarify how tracing information flows can be essential to preserve confidentiality.

Example 1 (Trojan horse) Most commercial operating systems implement DAC mechanisms to ensure confidentiality. Users are arranged in groups, processes runs on behalf of the users and can access the resources depending on a set of permissions. There can be different ways of accessing resources (for instance *read*, *write*, *execute*). In UNIX-like systems, the permissions for the files can be viewed through the `ls` command. Suppose for instance that in a directory there are two files, one called `secring.gpg` and owned by the administrator (`root` user) and the other called `myfile`, owned by the unprivileged user `john`. The former file can be read only by its owner and contains confidential information, while the latter can be read and written by every user.

```
root@myhost[mydir]# ls -l
total 8192
-rw-rw-rw-   2 john users      4096 2005-09-14 11:43 myfile
-rw-------   2 root wheel      4096 2005-11-25 12:01 secring.gpg
```

Now, the administrator can redirect the content of the secret file into the public file, for instance using the well-known command `cat`, thus allowing unprivileged users to access the confidential information:[1]

```
root@myhost[mydir]# cat secring.gpg >>
myfile
```

One may argue that this is a too naive example, because: (i) the administrator should be able to downgrade the privacy level of data (ii) the administrator executes the `cat` command of his own will and explicitly. Although there is sense in these objections, the problem is that the administrator could disclose the information without knowing it, because this action is "buried" into a program that he considers useful and harmless. This program could be either erroneous or malicious (in this latter case the term *trojan horse* is used). The administrator should be conscious that a program can downgrade information before executing it, to properly decide if this downgrading is acceptable or not.

Example 2 (Java Card Firewall) A Java Card [12] is a smart card running a Java Virtual Machine, the Java Card Virtual Machine (JCVM), and it is becoming a secure token in various fields, such as banking and public administration. The Java Card system was designed to speed up the development of applications (applets) and to increase portability. The JCVM is single-threaded, but more than one applet can coexist on the same card. Applets are normally isolated through the Java Card Firewall mechanism. This firewall allows an applet to access external objects only through an object sharing mechanism, called shareable interface. The firewall is based on an access control policy and therefore does not control information propagation.

Consider now a card that hosts three applets, say `A`, `B`, and `C`, each issued by a different commercial entity (see Fig. 1). Suppose that some partnership exists such that `A` must share some data with `B` and likewise `B` with `C` in order to be able to cooperate. Therefore the applets must be programmed to inform the firewall of this cooperation. Even if the applet `A` does not want that its confidential data would be propagated to `C`, it cannot rely on the firewall to control the behavior of `B`. Instead, before initiating cooperation with `B`, it must trust `B`'s code.

Example 3 (A tax calculation) This example is similar to the previous, but belongs to a different context. Imagine that an user would write out his/her tax return electronically, using a computer application. This application is

[1] The `cat` process is able to read the `secring.gpg` file because runs on behalf of the user `root`.

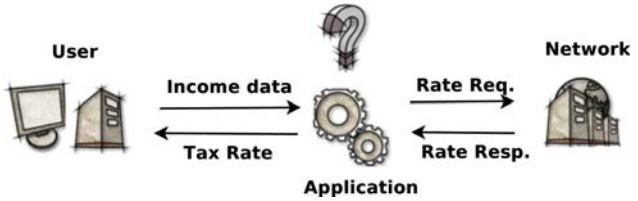

Fig. 2 An application that calculates electronically tax returns. How the user can be sure that his confidential data are not disclosed to third parties?

a server that holds users data, and, upon the requests of the user, provides them the tax rate they have to apply to their income. The application must contact the Internet to know the current income thresholds and the current tax rates, as illustrated in Fig. 2.

The question is: how can the user trust that the program do not leak any information about his/her income to the party that provides the thresholds and the rates? An access control mechanism could only either block the entire communication or allow the data to flow without any checking. Thus, a more fine-grained control is necessary.

3 The model

We consider the simple language illustrated in Fig. 3. Besides basic data, the language handles dynamic structures. We indicate with k a literal value and with s, f, x, a, respectively, generic structure, field, variable and channel name. E represents the expressions and C the commands. Each instruction is labeled by a label $t \in \mathcal{B} = \{0, 1, \ldots, n-1\}$, where n is the number of instructions in the program. We denote by `New` the subset of the `new` instructions in $\mathcal{B}$.

Every program P can retrieve data from a set of input channels and can send data to a set of output channels. If a is an input channel, the input command $a?x$ takes an item from a and assign it to variable x. The output command $a!e$ sends the value of expression e over the output channel a, provided that e is an expression returning a basic type (`int`). Given an input channel a, execution of an input command on a produces an *input action* (k, a) where k is the value taken from a. Analogously, the execution of an output command on an output channel a produces an *output action* (k, a) where k is the value inserted into a. In the following, we denote as $\texttt{Names}_I$ (respectively, $\texttt{Names}_O$) the set of input (output) channels used by a program; moreover $\texttt{Names}=\texttt{Names}_I \cup \texttt{Names}_O$ and $\texttt{Names}_I \cap \texttt{Names}_O = \emptyset$.

$$
\begin{array}{rl}
P & ::= \{D; C\} \\
D & ::= T\ x \mid \texttt{in}\ \sigma\ a \mid \texttt{out}\ \sigma\ a \mid D\ ;\ D \\
T & ::= \texttt{int} \mid S \\
S & ::= \texttt{struct}\ s\ \{D\} \\
C & ::= t : x = E \mid t : x.f = E \mid t : x = \texttt{new}\ s \mid t : a?x \mid t : a!E \mid t : \texttt{skip} \\
 & \quad\ t : \texttt{if}(E)\ C\ \texttt{else}\ C; \mid t : \texttt{while}(E)\ C; \mid C; C \\
E & ::= k \mid E\ Op\ E \mid x \mid x.f \\
Op & ::= + \mid - \mid \&\& \mid == \mid >
\end{array}
$$

Fig. 3 Language grammar

We assume that programs are type correct, that is:

- in any expression `x.f`, the variable `x` is a reference to structure with a field of name `f`,
- in any expression $E_1\ Op\ E_2$, E_1 and E_2 are of the same type,
- in any assignment `x=`E, the expression E is of the same type of the variable `x`,
- in any assignment `x.f=`E, the variable `x` is a reference to structure with a field of name `f`, and the expression E is of the same type of the field `f`,
- in any output statement `a!`E, the expression E is of type `int`,
- in any input statement `a?x`, the variable `x` is of type `int`.

These constraints can be easily checked by standard type checking algorithms.

The input and output channels represent the external environment in which the program is executed, that is all the interactions of the program occur by means of the channels. An external server is not able to inspect the internal state of the program, but can only send/receive messages to/from the program by means of the input/output channels. A *security policy* assigns to each input and output channel a security level, representing a fixed degree of secrecy. The security policy is expressed by the declaration of the channels. A channel a is declared by using the keyword `in` (`out`) to indicate that is an input (output) channel and by indicating also its security level. Security levels are defined as a finite lattice $(\mathcal{L}, \sqsubseteq_{\mathcal{L}})$, ranged over by $\sigma, \tau, \ldots$ and partially ordered by $\sqsubseteq_{\mathcal{L}}$. In the following we indicate by $\mathcal{S} : \texttt{Names} \rightarrow \mathcal{L}$ the security policy specified by the channels declarations.

Definition 1 (*secure information flow*) Let P be a program and $\mathcal{S}$ a security policy for P. Given $\sigma \in \mathcal{L}$, let us denote by $\texttt{Names}^{\sqsubseteq\sigma} = \{a \in \texttt{Names} \mid \mathcal{S}(a) \sqsubseteq \sigma\}$ the set of channels with security level lower than or equal to σ. Given a computation C of the program, let $\mathsf{actions}(C, \sigma)$ denote the sequence of input and output actions involving the channels in $\texttt{Names}^{\sqsubseteq\sigma}$ performed by C . We say that P has σ-secure information flow (is σ-secure) under $\mathcal{S}$ if, assuming that the external environment sends to the program the same sequence of values on each input channel in $\texttt{Names}^{\sqsubseteq\sigma}$, for any

P1
```
1: a?x;
2: b!x;
```

P2
```
1: y=1;
2: a?x;
3: if (x==0)
4:   y=0;
   else
5:   skip;
6: b!y;
```

P3
```
1: a?x;
2: if (x==0)
3:   d?y;
   else
4:   skip;
```

P4
```
1: a?x;
2: while (x>0){
3:   b!1;
4:   x=x-1;
   }
```

P5
```
1: d?x;
2: while (x>0){
3:   b!1;
4:   a?x;
   }
```

P6
```
1: a?x;
2: while (x>0){
3:   x=x-1;
   }
4:   b!1;
```

P7
```
1: a?y;
2: y=0;
3: b!y;
```

P8
```
1: s1=new S;
2: s2=new S;
3: a?x;
4: if (x)
5:   s3=s1;
   else
6:   s3=s2;
7: s3.f=1;
```

Fig. 4 Some examples

two computations C_1 and C_2 of the program, it holds $\mathsf{actions}(C_1,\sigma) = \mathsf{actions}(C_2,\sigma)$.

We say that P has secure information flow (is secure) if it is σ-secure for each $\sigma \in \mathcal{L}$.

If a program is secure, an external attacker having secrecy level σ cannot infer information that is more secret than σ from a σ-secure program since he can inspect only input and output channels with levels that are lower than or equal to σ.

Similar properties are also referred in literature as *noninterference* properties.

The above definition considers computations that may or may not terminate, and therefore the property is *termination-sensitive* [40]. A weaker termination-insensitive property could consider only finite computations.

Note that, in general an attacker could gather secure information by observing the so-called *covert channels* [29], e.g. the time occurring between the input/output operations, or the power consumption. We do not consider this kind of noninterference.

Let us show some examples of programs. Consider the programs in Fig. 4 and suppose that a and d are input channels and b is an output channel. Moreover assume $\mathcal{S}(a) = H, \mathcal{S}(b) = \mathcal{S}(d) = L$, with $L \sqsubset H$. Since in this example there are only two security levels, we can say that channels b and d are public, while channel a is private.

Program P1 shows an explicit insecure information flow, since the value output on channel b depends on the value input from a: private information is made available to a public observer. Program P2 is insecure because it is possible to know if the private input is zero by observing the value present on the public output channel. In program P3 the private value affects the contents of input channel d, from which an item is taken only if the input is zero. Note that we consider observable both the input and the output channels. In program P4 the number of the values output on channel b depends on the input value. In program P5 the first iteration of the `while` is driven by a low value, while the following iterations depend on high level information. Also program P6 may have an illicit information flow, even though the value output on channel b is always the same: it is possible that, due to an infinite loop, no value is output on channel b. Program P7 is secure, since the output value, which is constant, does not depend on the input: even though `y` is written with a high value, afterward it is assigned a constant value, and this one is given as an output. Consider program P8 and suppose that `S` is an user-defined structure with two int fields `f` and `g`, and that `s1`, `s2`, `s3` are references of type `S`. Please notice that, depending on the value taken from the high level input channel `a`, instruction 7 updates field `f` of two different objects (created at the first two instructions). Now consider the two cases in which instruction 7 is followed by: (i) `8:b!s1.g;` (ii) `8:b!s1.f;`. In case (i) the program is secure because field `g` of object created at instruction 1 is the same in any computation. On the contrary, in case (ii), the value of the field `s1.f` depends on the input: by aliasing, the assignment in instruction 7 could have modified it.

4 Concrete semantics

In this section we define the concrete semantics of the language. To take into account the security level of data, we annotate each value v flowing through the variables and the structure fields with a security level, representing the least upper bound of the security levels of the explicit and implicit information flows on which v depends. A value is a pair (v^e, σ), where v^e is an execution value and σ a security level. The domains of the concrete semantics are shown in Fig. 5. An execution value may be an integer $k \in \mathbb{Z}$ or a reference to an user-defined structure. A reference is in turn a pair (ℓ, t), where $\ell \in \mathcal{A}^e$ is a heap address and $t \in \mathtt{New}$ is the label of the instruction which created the corresponding structure. This tag will be useful in the abstraction to coalesce into a same abstract structure all structures created at the same instruction. The memory is represented by means of two functions: one denoted by μ, that

$$
\begin{array}{lll}
v \in \mathcal{V} & = (\mathbb{Z} \cup \mathcal{A}) \times \mathcal{L} & \mathcal{A} = \mathcal{A}^e \times \mathtt{New} \\
\mu \in \mathcal{M} & = \mathit{Var} \to \mathcal{V} & \xi \in \Xi = \mathcal{A} \to \mathcal{M}_{\mathtt{struct}} \\
c \in \mathcal{C} & = \mathtt{Names} \to (\mathbb{Z}^\star \times \mathcal{L}) & q \in \mathcal{Q} = \mathcal{B} \times \mathit{Env} \times \mathcal{M} \times \Xi \times \mathcal{C}
\end{array}
$$

Fig. 5 Domains of the concrete semantics

$$\textbf{Const}\ \frac{}{\langle k,\mu,\xi\rangle \xrightarrow{E} (k,\bot_{\mathcal{L}})} \qquad \textbf{Op}\ \frac{\langle E_1,\mu,\xi\rangle \xrightarrow{E} (v_1^e,\sigma_1),\ \langle E_2,\mu,\xi\rangle \xrightarrow{E} (v_2^e,\sigma_2)}{\langle E_1\ op\ E_2,\mu,\xi\rangle \xrightarrow{E} (v_1^e\ \underline{op}\ v_2^e,\sigma_1\sqcup_{\mathcal{L}}\sigma_2)}$$

$$\textbf{Value}_{\texttt{x}}\ \frac{}{\langle x,\mu,\xi\rangle \xrightarrow{E} \mu(x)} \qquad \textbf{Value}_{\texttt{x.f}}\ \frac{\mu(x)=((\ell,t),\sigma_1),\ \xi(\ell,t)(f)=(v^e,\sigma_2)}{\langle x.f,\mu,\xi\rangle \xrightarrow{E} (v^e,\sigma_1\sqcup_{\mathcal{L}}\sigma_2)}$$

Fig. 6 Concrete semantics of expressions

associates every variable with its value, and the other, denoted by ξ, that associates the addresses (references) with the respective structure instances. Every structure in the heap can be represented by a memory whose variables are the fields. Valid fields names are in the domain $\mathcal{F}$. We denote by $\mathcal{M}_S$ the domain of memories having the fields of structure S as variables, and by $\mathcal{M}_{\texttt{struct}}$ the set: $\mathcal{M}_{\texttt{struct}} = \bigcup\{\mathcal{M}_S | S \text{ used in } P\}$. The state of input and output channels $c \in \mathcal{C}$ is a mapping from the names of the channels to pairs (s,σ), where $s \in \mathbb{Z}^\star$ is a finite sequence of values and σ a security level. We use $\cdot$ to denote sequence concatenation, and λ to denote the empty sequence. Initially, the security level of each input channel a is set to $\mathcal{S}(a)$, that is the security level defined for a by the security specification. As a consequence, each value taken from an input channel a is annotated with $\mathcal{S}(a)$. The security level of the output channels is initially set to the minimum level $\bot_{\mathcal{L}}$. The security level of the channels can be modified by the computation, when the channel is accessed.

The concrete semantics is defined by means of a set of rules: the rules for expressions are shown in Fig. 6 and the rules for instructions in Fig. 7. Let us consider the rules for expressions, defining a relation $\xrightarrow{E} \subseteq (expr \times \mathcal{M} \times \Xi) \times \mathcal{V}$. Rule **Const** assigns the bottom security level to any constant value. Rule **Op** calculates the security level of the result of an operation as the least upper bound (from now on, lub) of the security levels of the operands. Rule $\textbf{Value}_x$ returns the value of the variable in the memory. Rule $\textbf{Value}_{x.f}$ annotates the resulting value with the lub of the security levels of the reference and of the value stored in the field.

The rules for instructions (Fig. 7) define a relation $\longrightarrow \subseteq \mathcal{Q} \times \mathcal{Q}$ between the states of the computation. The set of concrete states is $\mathcal{Q} = \mathcal{B} \times Env \times \mathcal{M} \times \Xi \times \mathcal{C}$, where $Env = \mathcal{B} \to \mathcal{L}$. Each state $q \in \mathcal{Q}$ is a tuple $\langle t,\rho,\mu,\xi,c\rangle$ describing the configuration of the machine when executing the command t: μ and ξ define the values of variables and structures fields, while c represent the status of the channels. We also keep in each state a *security environment* $\rho \in Env$. The environment ρ assigns to every program point a security level representing the level of the implicit flow under which the corresponding command is executed. In the following, given an instruction label t and a set Q of states, we use the notation $Q(t)$ to denote the set of states in Q corresponding to instruction t. A value (v^e,τ) evaluated, assigned or tested while the execution is under a security environment σ, changes its security level into $\sigma \sqcup \tau$. The

Fig. 7 Concrete semantics of commands

$$\textbf{Skip}_{\texttt{skip}}\ \frac{}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \langle succ(t),\rho,\mu,\xi,c\rangle}$$

$$\textbf{Assign}_{\texttt{t:x=E}}\ \frac{\langle E,\mu,\xi\rangle \xrightarrow{E} (v^e,\sigma)}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \langle succ(t),\rho,\mu\left[x \leftarrow (v^e,\rho(t)\sqcup_{\mathcal{L}}\sigma)\right],\xi,c\rangle}$$

$$\textbf{Assign}_{\texttt{t:x.f=E}}\ \frac{\langle E,\mu,\xi\rangle \xrightarrow{E} (v^e,\sigma_1),\ \mu(x)=((\ell,t_1),\sigma_2),\ \sigma_3=\sigma_1\sqcup_{\mathcal{L}}\sigma_2\sqcup_{\mathcal{L}}\rho(t)}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \langle succ(t),\rho,\mu,\xi\left[(\ell,t_1).f \leftarrow (v^e,\sigma_3)\right],c\rangle}$$

$$\textbf{New}_{\texttt{t:x=new S}}\ \frac{\text{fresh}(\xi)=\ell}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \langle succ(t),\rho,\mu\left[x \leftarrow ((\ell,t),\rho(t))\right],\xi\left[(\ell,t) \leftarrow \mu_{S\bot}\right],c\rangle}$$

$$\textbf{Input}_{\texttt{t:a?x}}\ \frac{c(a)=(k\cdot s,\sigma),a\in \texttt{Names}_I}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \langle succ(t),\rho,\mu\left[x \leftarrow (k,\rho(t)\sqcup_{\mathcal{L}}\sigma)\right],\xi,c\left[a \leftarrow (s,\rho(t)\sqcup_{\mathcal{L}}\sigma)\right]\rangle}$$

$$\textbf{Output}_{\texttt{t:b!E}}\ \frac{\langle E,\mu,\xi\rangle \xrightarrow{E} (k,\sigma_1),\ c(b)=(s,\sigma_2),\ b\in \texttt{Names}_O}{\langle (t,\rho,\mu,\xi,c\rangle \longrightarrow \langle succ(t),\rho,\mu,\xi,c\left[b \leftarrow (s\cdot k,\rho(t)\sqcup_{\mathcal{L}}\sigma_1\sqcup_{\mathcal{L}}\sigma_2)\right]\rangle}$$

$$\textbf{If}_{\substack{\texttt{t:if (E) C}\\ \texttt{else C, (true)}}}\ \frac{\langle E,\mu,\xi\rangle \xrightarrow{E} (\underline{\text{true}},\sigma)}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \left\langle succ_{true}(t),\rho\left[t' \leftarrow \rho(t')\sqcup_{\mathcal{L}}\sigma\right]_{\forall t'\in scope(t)},\mu,\xi,c\right\rangle}$$

$$\textbf{While}_{\substack{\texttt{t:while (E) C}\\ \texttt{(true)}}}\ \frac{\langle E,\mu,\xi\rangle \xrightarrow{E} (\underline{\text{true}},\sigma)}{\langle t,\rho,\mu,\xi,c\rangle \longrightarrow \left\langle succ_{true}(t),\rho\left[t' \leftarrow \rho(t')\sqcup_{\mathcal{L}}\sigma\right]_{\forall t'\in scope(t)},\mu,\xi,c\right\rangle}$$

environment, initially set to $\bot_{\mathcal{L}}$ for all commands, can be updated by the conditional and repetitive commands. With $succ(t)$ we indicate the successive instruction to be executed. All commands have only one successor, except the conditional and repetitive commands that have two successors, depending on the value of the guard; they are denoted by $succ_{true}(t)$ and $succ_{false}(t)$. We assume that the first instruction of the program has label t_0 and that for the last instruction is $succ(t) = \mathsf{end}$.

the instructions belonging to only one branch starting from the `if`. For the `while` command, $scope(t)$ includes the instructions belonging to the loop (the *true* part of the `while`) and also all instructions after the loop until the end of the program (the *false* part). The inclusion of these instructions takes into account the possibility of an infinite loop: in this case, the commands following the loop will never be executed.

The function $scope\colon \mathcal{B} \to \wp(\mathcal{B})$ is defined as follows:

$$scope(t) = \begin{cases} \bigcup \{scope(t') | t' \in C_1 \cup C_2\} & \text{if } t\texttt{:if(}E\texttt{)\{}C_1\texttt{\} else \{}C_2\texttt{\}} \\ \bigcup \{scope(t') | t' \in C_1 \cup C_2\} & \text{if } t\texttt{:while(}E\texttt{)\{}C_1\texttt{\}}C_2 \\ \{t\} & \text{otherwise} \end{cases}$$

Rule $\mathbf{Assign}_{t:x=e}$ annotates the security level of the value to be assigned with the lub of the security level resulted by the evaluation of the expression and the environment of the instruction t. The notation $\mu\left[x \leftarrow (v^e, \sigma)\right]$ stands for the memory obtained by μ by updating the contents for the variable x with the value (v^e, σ). Rule $\mathbf{Assign}_{t:x.f=e}$ annotates the value to be assigned with the lub of (1) the security level resulted by the evaluation of the expression, (2) the security level of the reference, and (3) the environment of t. In the rule, the notation $\xi\left[(\ell, t).f \leftarrow v\right]$ indicates the heap ξ' obtained from ξ by updating the field f of the structure located at address ℓ (and created at instruction t) with the value v.

Rule **New** contains the notation $\xi\,[(\ell, t) \leftarrow \mu_{S\bot}]$, meaning that, during the execution of instruction t, in the heap ξ a new structure of type S is created at address ℓ, its fields containing the default value. We assume the default value is the pair $(0, \bot_{\mathcal{L}})$. In the premise of the rule the function fresh : $\Xi \to \mathcal{A}^e$ is used to find a free location in the heap to store the new structure.

Rule **Input** takes a value from the specified input channel and assigns it to the destination variable, annotated with the lub of the level σ of the channel and the environment of t. Also the level of the channel is updated in the same way. As a consequence, if $\rho(t)$ is higher than σ, the level of the channel is upgraded, to record the fact that the manipulation of the channel depends on an information flow with level $\sigma \sqsubset_{\mathcal{L}} \rho(t)$. Analogously, in the **Output** rule, the level of the specified output channel can be upgraded taking into account the level of the value and that of the environment of the instruction.

The **If** and **While** rules, whatever branch is chosen, affect the environment of all the instructions belonging to the scope of the command, taking into account the level of the condition. The set $scope(t)$ contains all the instructions that can be executed or not depending on the condition. In the **If** case, $scope(t)$ includes all where C_1 and C_2 are sequences of instructions. Note that we adopt a termination-sensitive noninterference property. A weaker termination-insensitive property, which is defined only for terminating executions, can be analyzed by using a different definition of $scope(t)$, obtained by modifying the second clause of the above definition. The clause becomes $\bigcup \{scope(t') | t' \in C\}$if t`:while(`E`){`C`}`. In this way we consider as scope of a `while` command only the body of the command and not the continuation of the program. Hence, the level of data manipulated after the `while` is not affected by the level of the condition.

Updating the environment is necessary to trace implicit flow: the value of the condition (with its security level) drives the execution of the instructions in $scope(t)$. The table shows only the rule to be applied when the condition is true. The rule to be applied when the condition is false (not shown) is equal except that has $succ_{false}$ instead of $succ_{true}$.

Definition 2 (*initial state*) Given an initial configuration i_0 : $\texttt{Names}_I \to \mathbb{Z}^\star$ of the input channels, the initial state is defined as $q(i_0) = \langle t_0, \rho_\bot, \mu_0, \xi_\lambda, c_0 \rangle$, where $\forall t \in \mathcal{B}, \rho_\bot(t) = \bot_{\mathcal{L}}$, and $\forall x \in X, \mu_0(x) = (0, \bot_{\mathcal{L}})$, and ξ_λ is the heap with empty domain (i.e. the everywhere undefined function). The state c_0 is such that for all $a \in \texttt{Names}_I, c_0(a) = (i_0(a), \mathcal{S}(a))$ and for all $a \in \texttt{Names}_O, c_0(a) = (\lambda, \bot_{\mathcal{L}})$.

We now define a collecting semantics, which associates with each instruction the set of states in which the instruction can be executed in any computation.

First we define an alignment operation $\mathsf{align}(Q)$ which, given a set of states Q, aligns all the states corresponding to the same instruction. $\mathsf{align}(Q)$ increments Q with some extra states for each instruction t: for each state $q \in Q(t)$, a state q' is added to Q having the same execution values occurring in q, but where the security levels of the environment, memory variables, fields of

Fig. 8 Auxiliary functions for merging

$$
\begin{array}{lll}
\mathsf{max}_E & : \mathcal{Q} \times \mathcal{B} \to \mathcal{L} & \mathsf{max}_E(Q,t) = \bigsqcup_{\mathcal{L}} \{\rho(t) |\, \langle t, \rho, \mu, \xi, c\rangle \in Q\} \\
\mathsf{max}_{\mathcal{M}} & : \mathcal{Q} \times Var \to \mathcal{L} & \mathsf{max}_{\mathcal{M}}(Q,x) = \bigsqcup_{\mathcal{L}} \{\sigma |\, \langle t, \rho, \mu, \xi, c\rangle \in Q, \mu(x) = (v^e, \sigma)\} \\
\mathsf{max}_{\Xi} & : \mathcal{Q} \times \mathcal{A} \times \mathcal{F} \to \mathcal{L} & \mathsf{max}_{\Xi}(Q,(\ell,t),f) = \bigsqcup_{\mathcal{L}} \{\sigma |\, \langle t, \rho, \mu, \xi, c\rangle \in Q, \xi(\ell,t)(f) = (v^e, \sigma)\} \\
\mathsf{max}_{\mathcal{C}} & : \mathcal{Q} \times \mathtt{Names} \to \mathcal{L} & \mathsf{max}_{\mathcal{C}}(Q,a) = \bigsqcup_{\mathcal{L}} \{\sigma |\, \langle t, \rho, \mu, \xi, c\rangle \in Q, c(a) = (s, \sigma)\}
\end{array}
$$

Fig. 9 The align function

$$
\begin{array}{l}
\mathsf{align}(\langle t, \rho, \mu, \xi, c\rangle, Q) = \langle t, \rho', \mu', \xi', c'\rangle \\
\text{with:} \\
\quad \forall t' \in \mathcal{B} : \quad \rho'(t') = \mathsf{max}_E(Q(t), t') \\
\quad \forall x \in Var : \quad \mu'(x) = \mathsf{up}(\mu(x), \mathsf{max}_{\mathcal{M}}(Q(t), x)) \\
\quad \forall (\ell,t') \in dom(\xi), f \in dom(\xi(\ell,t')) : \xi'(\ell,t')(f) = \mathsf{up}(\xi(\ell,t')(f), \mathsf{max}_{\Xi}(Q(t), \ell, t', f)) \\
\quad \forall a \in \mathtt{Names} : \quad c'(a) = \mathsf{up}(c(a), \mathsf{max}_{\mathcal{C}}(Q,a))
\end{array}
$$

$$\mathsf{align}(Q) = (\textstyle\bigcup_{q \in Q} \mathsf{align}(q, Q)) \cup Q$$

structures and channels are upgraded to the lub in $\mathcal{L}$ of the levels occurring in the states in $Q(t)$ for the same items.

In Fig. 8 some auxiliary functions used in the alignment process are shown. Let Q be a set of states: then $\mathsf{max}_{\mathcal{M}}(Q,x)$ is the lub of the security levels of x in the memories occurring in the states of Q. For each $t \in \mathcal{B}$, $\mathsf{max}_E(Q,t)$ is the lub of the values of $\rho(t)$ in the environment occurring in the states of Q. For each field f of each structure created at instruction $(\ell,t) \in \mathcal{A}$, $\mathsf{max}_{\Xi}(Q,\ell,t,f)$ is the lub of the values held by the field f in the heap occurring in the states of Q. For each channel $a \in \mathtt{Names}$, $\mathsf{max}_{\mathcal{C}}(Q,a)$ is the lub of the security levels held by the channel a in the states of Q.

The definition of align is shown in Fig. 9. Given a value $v = (v^e, \tau)$, with $v^e \in (\mathbb{Z} \cup \mathcal{A})$, $\mathsf{up}(v,\sigma) = (v^e, \tau \sqcup_{\mathcal{L}} \sigma)$ is the value obtained by maintaining the execution part of the value and upgrading the annotation of v.

For example, consider to have a memory μ with only one variable x and only a channel a and consider the states:

$$q_1 = \langle 2, \rho, \mu(x) = (2, L), \xi, c(a) = (1 \cdot 2, H)\rangle$$
$$q_2 = \langle 2, \rho, \mu(x) = (3, H), \xi, c(a) = (2 \cdot 3, L)\rangle$$
$$q_3 = \langle 3, \rho, \mu(x) = (4, L), \xi, c(a) = (3, L)\rangle$$

We have that:

$$\mathsf{align}(\{q_1, q_2, q_3\}) = \{q_1, q_2, q_3, q_4, q_5\}$$

where:

$$q_4 = \langle 2, \rho, \mu(x) = (2, H), \xi, c(a) = (1 \cdot 2, H)\rangle \text{ and}$$
$$q_5 = \langle 2, \rho, \mu(x) = (3, H), \xi, c(a) = (2 \cdot 3, H)\rangle$$

State q_4 is equal to q_1 with respect to numeric values, but variable x contains an high value, which is the maximum value of x in the set of states with program counter 2. Analogously, state q_5 has the same values of q_2 but aligns the level of channel a to H. State q_3 is the only one with program counter 3 and thus no new state needs to be added.

Definition 3 (*concrete next operator* next) Given a set of concrete states $Q \subseteq \mathcal{Q}$, the application of the next operator yields the aligned set of states that are either in Q, or reached in one step of computation starting from a state in Q.

$$\mathsf{next}(Q) = \mathsf{align}(Q \cup \{q' | \exists q \in Q : q \longrightarrow q'\})$$

Proposition 1 (monotonicity of next) next *is monotone in* $(\wp(\mathcal{Q}), \subseteq)$.

The concrete collecting semantics $sem \in \wp(\mathcal{Q})$ is the set of all aligned concrete states belonging to all executions.

Definition 4 (*collecting semantics*) The concrete collecting semantics $sem \in \wp(\mathcal{Q})$ is the lub of the following increasing chain, defined for all $n \in \mathbb{N}$:

$$sem_0 = \{q(i_0) |\, \forall i_0 \in (\mathtt{Names}_I \to \mathbb{Z}^\star)\}$$
$$sem_{n+1} = \mathsf{next}(sem_n)$$

Performing align at each step of sem_n aligns the security annotations of the states corresponding to the join point of different branches of a conditional instruction, in order to properly manage implicit flows. This function is necessary for the soundness of the concrete semantics, proved by Theorem 1. Consider, for example, program P2 of Fig. 4. If we consider an execution in which the input value is 0, the branch *true* of the if command is executed, and at instruction 6 the state is $q = \langle 6, \rho, \mu, \xi_\lambda, c\rangle$, with $\rho(4) = \rho(5) = H$, $\mu(y) = (0, H)$ and where the annotation H of 0 records the implicit flow of level H under which the assignment to y has been performed. On the other hand, if the input value is different from 0, variable x is not affected in the conditional command and the state $q' = \langle 6, \rho, \mu', \xi_\lambda, c'\rangle$ is reached, where $\mu'(y) = (1, \bot_{\mathcal{L}})$. This state does not represent the implicit flow, since the level of the value held by y is low. Instead, the contents of y has been affected also in this case by the implicit flow of level H. Since the alignment operation is applied to the chain sem_n, there exists at least one j such that sem_j contains a state $\langle 6, \rho, \mu'', \xi_\lambda, c'\rangle$ where $\mu''(y) = (1, H)$. This state derives from the alignment of

q to the levels occurring in q' and represents the effect of the implicit flow on y in the case in which the *false* branch has been chosen.

The security annotations used by the concrete semantics can be tied to the secure information flow property via a soundness theorem. To prove such theorem we need to define an equivalence relation between states.

Definition 5 (σ-*equivalence*) Let $\sigma \in \mathcal{L}$.

- Given two heaps $\xi_1, \xi_2 \in \Xi$, we define a relation $=_\sigma^{\xi_1,\xi_2} \subseteq \mathcal{V} \times \mathcal{V}$ as the greatest relation satisfying:
  $v_1 =_\sigma^{\xi_1,\xi_2} v_2$ if and only if one of the following cases holds:
  $\sigma_1 \not\sqsubseteq_\mathcal{L} \sigma$ and $\sigma_2 \not\sqsubseteq_\mathcal{L} \sigma$
  $\sigma_1 \sqsubseteq_\mathcal{L} \sigma, \sigma_2 \sqsubseteq_\mathcal{L} \sigma, v_1 = (k, \sigma_1), v_2 = (k, \sigma_2)$
  $\sigma_1 \sqsubseteq_\mathcal{L} \sigma, \sigma_2 \sqsubseteq_\mathcal{L} \sigma, v_1 = (l_1, t_1, \sigma_1), v_2 = (l_2, t_2\sigma_2)$ and $\forall f \in \mathcal{F} : \xi_1(l_1).f =_\sigma^{\xi_1,\xi_2} \xi_2(l_2).f$
- Two environments ρ_1 and ρ_2 are σ-equivalent ($\rho_1 =_\sigma^E \rho_2$) iff
  $\forall t \in \mathcal{B}$: either $\rho_1(t) \sqsubseteq_\mathcal{L} \sigma$ and $\rho_2(t) \sqsubseteq_\mathcal{L} \sigma$ or $\rho_1(t) \not\sqsubseteq_\mathcal{L} \sigma$ and $\rho_2 \not\sqsubseteq_\mathcal{L} \sigma$.
- Two concrete pairs memory-heap (μ_1, ξ_1) and (μ_2, ξ_2) are σ-equivalent ($(\mu_1, \xi_1) =_\sigma^{\mathcal{M},\Xi} (\mu_2, \xi_2)$) iff $\forall x : \mu_1(x) =_\sigma^{\xi_1,\xi_2} \mu_2(x)$.
- Two concrete channels $a_1 = (\delta_1, \sigma_1)$ and $a_2 = (\delta_2, \sigma_2)$ are σ-equivalent
  ($a_1 =_\sigma^\mathcal{C} a_2$) if one of the following cases hold:
  $\sigma_1 \sqsubseteq_\mathcal{L} \sigma, \sigma_2 \sqsubseteq_\mathcal{L} \sigma$ and $\delta_1 = \delta_2$
  $\sigma_1 \not\sqsubseteq_\mathcal{L} \sigma$ and $\sigma_2 \not\sqsubseteq_\mathcal{L} \sigma$
- Two concrete states $q_1 = \langle t_1, \rho_1, \mu_1, \xi_1, c_1 \rangle$ and $q_2 = \langle t_2, \rho_2, \mu_2, \xi_2, c_2 \rangle$ are σ-equivalent ($q_1 =_\sigma^\mathcal{Q} q_2$) iff

$$t_1 = t_2 \quad \rho_1 =_\sigma^E \rho_2 \quad (\mu_1, \xi_1) =_\sigma^{\mathcal{M},\Xi} (\mu_2, \xi_2)$$
$$\forall a \in \texttt{Names} : c_1(a) =_\sigma^\mathcal{C} c_2(a)$$

The relation $=_\sigma^\mathcal{Q} \in \mathcal{Q} \times \mathcal{Q}$ is an equivalence relation.

Lemma 1 *Let be $\mu_1, \mu_2 \in \mathcal{M}$ and $\xi_1, \xi_2 \in \Xi$, such that $(\mu_1, \xi_1) =_\sigma^{\mathcal{M},\Xi} (\mu_2, \xi_2)$ and an expression $E \in Expr$. If*

$\langle E, \mu_1, \xi_1 \rangle \xrightarrow{E} v_1$, *and*

$\langle E, \mu_2, \xi_2 \rangle \xrightarrow{E} v_2$,

then,

$v_1 =_\sigma^{\xi_1,\xi_2} v_2$

Proof By induction on the structure of expressions. □

The following lemma states that a) the execution of a non-conditional instruction preserves σ-equivalence; and b) σ-equivalence is also preserved with a conditional instruction, provided that the tested condition is less secret than σ.

Lemma 2 *Let be $q_1 = \langle t, \rho_1, \mu_1, \xi_1, c_1 \rangle =_\sigma^\mathcal{Q} q_2 = \langle t, \rho_2, \mu_2, \xi_2, c_2 \rangle$. If $q_1 \longrightarrow q_1' = \langle t_1', \rho_1', \mu_1', \xi_1', c_1' \rangle$ and $q_2 \longrightarrow q_2' = \langle t_2', \rho_2', \mu_2', \xi_2', c_2' \rangle$, then*

$$\rho_1' =_\sigma^E \rho_2' \quad (\mu_1, \xi_1) =_\sigma^{\mathcal{M},\Xi} (\mu_2, \xi_2) \quad \forall a \in \texttt{Names} : c_1'(a) =_\sigma^\mathcal{C} c_2'(a)$$

Moreover, if

- *t is neither an* `if` *nor a* `while` *instruction; or*
- *t is either an* `if` *or a* `while` *instruction and the guard E is such that*

$$\langle E, \mu_1, \xi_1 \rangle \xrightarrow{E} (v_1^e, \sigma_1') \text{ with } \sigma_1' \sqsubseteq_\mathcal{L} \sigma$$

then it is also true that $t_1' = t_2'$ (and therefore $q_1' =_\sigma^\mathcal{Q} q_2'$)

Proof By examining all possible kinds of instruction.

($\mathbf{Assign}_{\texttt{t:x=e}}$) The rule affects only the memory. By Lemma 1, we have that the expression E, once evaluated, returns two σ-equivalent values, let say $v_1 = (v_1^e, \sigma_1)$ and $v_2 = (v_2^e, \sigma_2)$ (we suppose for simplicity that v_1 and v_2 are simple values and not references). Moreover, suppose $\rho_1(t) = \tau_1$ and $\rho_2(t) = \tau_2$. The memories μ_1' and μ_2' differ from μ_1 and μ_2 only for the values $\mu_1'(x)$ and $\mu_2'(x)$ that are updated with $v_1' = (v_1^e, \sigma_1 \sqcup_\mathcal{L} \tau_1)$ and $v_2' = (v_2^e, \sigma_2 \sqcup_\mathcal{L} \tau_2)$, respectively. Now, we have two cases:

- $\sigma_1 \sqsubseteq_\mathcal{L} \sigma$ and $\sigma_2 \sqsubseteq_\mathcal{L} \sigma$, $\tau_1 \sqsubseteq_\mathcal{L} \sigma$ and $\tau_2 \sqsubseteq_\mathcal{L} \sigma$
  Since $v_1 =_\sigma^{\xi_1,\xi_2} v_2$ and $(\sigma_1 \sqcup_\mathcal{L} \tau_1) \sqsubseteq_\mathcal{L} \sigma$ and $(\sigma_2 \sqcup_\mathcal{L} \tau_2) \sqsubseteq_\mathcal{L} \sigma$. Therefore, also $v_1' =_\sigma^{\xi_1,\xi_2} v_2'$.
- (σ_1 and $\sigma_2 \not\sqsubseteq_\mathcal{L} \sigma$) or ($\tau_1$ and $\tau_2 \not\sqsubseteq_\mathcal{L} \sigma$) Then $(\sigma_1 \sqcup_\mathcal{L} \tau_1) \not\sqsubseteq_\mathcal{L} \sigma$ and $(\sigma_2 \sqcup_\mathcal{L} \tau_2) \not\sqsubseteq_\mathcal{L} \sigma$. Therefore, $v_1' =_\sigma^{\xi_1,\xi_2} v_2'$.

Then, it holds that $(\mu_1', \xi_1) =_\sigma^{\mathcal{M},\Xi} (\mu_2', \xi_2)$.

($\mathbf{Assign}_{\texttt{t:x.f=e}}$) This rule affects only the heap. Again, since by Lemma 1 the expression E returns two σ-equivalent values and $\rho_1 =_\sigma^E \rho_2$, we can make a reasoning similar to the previous case.

($\mathbf{New}_{\texttt{t:x=new S}}$) Here, both the memory and the heap change. Suppose that $\text{fresh}(\xi_1) = \ell_1$ and $\text{fresh}(\xi_2) = \ell_2$. It holds that $\forall y \in Var, y \neq x, \mu_1'(y) = \mu_1(y) =_\sigma^{\xi_1,\xi_2} \mu_2(y) = \mu_2'(y)$. Then $\mu_1'(y) =_\sigma^{\xi_1,\xi_2} = \mu_2'(y)$. Since the function fresh returns two clean addresses, it holds also that $\mu_1'(y) =_\sigma^{\xi_1',\xi_2'} \mu_2'(y)$. Moreover $\mu_1'(x) = (\ell_1, t, \rho_1(t)) =_\sigma^{\xi_1,\xi_2} \mu_2'(y) = (\ell_2, t, \rho_2(t))$ since the two new instances are equal and $\rho_1 =_\sigma^E \rho_2$.

($\mathbf{Input}_{\texttt{t:a?x}}$) This rule modifies the channel a and the variable x. However, the thesis trivially derives from the σ-equivalence of memories, environments and channels before the input operation.

(**Output**$_{\mathtt{t:b?E}}$) When applying this rule, only the channel b is modified. Again, the thesis holds from the σ-equivalence of the environments and of the two results of the expression E (by Lemma 1)

(**If**, **While**) Here, only the program counter and the environment can change. Let $\langle E, \mu_1, \xi_1\rangle \xrightarrow{E} (k_1, \sigma_1')$ and $\langle E, \mu_2, \xi_2\rangle \xrightarrow{E} (k_2, \sigma_2')$ be the two transitions that happen when evaluating the guards. By Lemma 1, only two cases may apply:

- $\sigma_1' \sqsubseteq_{\mathcal{L}} \sigma$ **and** $\sigma_2' \sqsubseteq_{\mathcal{L}} \sigma$ Then, it holds that the execution values of the guards are the same ($k_1 = k_2$) and therefore $t_1' = t_2'$. Moreover the value of the environment for instructions in the scope may be updated, in both cases, with a level $\sqsubseteq_{\mathcal{L}} \sigma$, thus preserving σ-equivalence.
- $\sigma_1' \not\sqsubseteq_{\mathcal{L}} \sigma$ **and** $\sigma_2' \not\sqsubseteq_{\mathcal{L}} \sigma$ The value of the environment for instructions in the scope is upgraded, in both cases, with a level $\not\sqsubseteq_{\mathcal{L}} \sigma$, thus preserving σ-equivalence. □

The following theorem states the soundness of the collecting semantics. It proves that the management of data performed by the concrete semantics correctly represent the level of the information flows of the program.

Theorem 1 (soundness of the concrete semantics) *A program P has secure information flow under a security policy $\mathcal{S}$ if for each concrete state $\langle t, \rho, \mu, \xi, c\rangle \in sem$, for each channel a, if $c(a) = (\delta, \sigma), \delta \in \mathbb{Z}^\star$, then $\sigma \sqsubseteq \mathcal{S}(a)$.*

Proof (sketch) The proof is made by proving σ-security for a generic σ. Consider two concrete executions starting from the same configurations of the input channels belonging to $\mathtt{Names}^{\sqsubseteq\sigma}$. Until a conditional or iterative instruction has not been reached with a high conditional expression (i.e. with level $\not\sqsubseteq \sigma$), by Lemma 2 the two executions perform the same instructions and reach at each step σ-equivalent states. If an input or output instruction is executed on a channel a such that $\mathcal{S}(a) \sqsubseteq_{\mathcal{L}} \sigma$, by the hypothesis the level of the channel is not upgraded to a level $\not\sqsubseteq_{\mathcal{L}} \sigma$ and thus the output values are σ-equivalent and have equal numeric part. If no **if** or **while** with high guard is reached, the property is satisfied. If, instead, an **if** or **while** command is reached, say t, with a high guard, then the two executions can be made up of different sequences of instructions. Let $q_1 = \langle t, \rho_1, \mu_1, \xi_1, c_1\rangle$ and $q_2 = \langle t, \rho_2, \mu_2, \xi_2, c_2\rangle$ the states reached by the two computations respectively before the execution of t and let $q_1^1 = \left\langle t_1^1, \rho_1^1, \mu_1, \xi_1, c_1\right\rangle$ and $q_2^1 = \left\langle t_2^1, \rho_2^1, \mu_2, \xi_2, c_2\right\rangle$, respectively, the states after the execution of the rule **If** or **While** with label t. By Lemma 2 it holds that $\rho_1^1 =_\sigma^E \rho_2^1$. As a consequence, by the **If** and **While** rules, for each instruction $t' \in scope(t)$, it holds $\rho_1^1(t') \not\sqsubseteq_{\mathcal{L}} \sigma$ and $\rho_2^1(t') \not\sqsubseteq_{\mathcal{L}} \sigma$. While instructions that belong to $scope(t)$ are executed, since the annotation of values is upgraded to the level of the environment of the instructions, it holds: i) if a variable or a field of a structure is updated, then the stored value has annotation $\not\sqsubseteq_{\mathcal{L}} \sigma$ in both executions; and ii) no input or output channel a with $\mathcal{S}(a) \sqsubseteq_{\mathcal{L}} \sigma$ is affected, otherwise *sem* would not respect the condition imposed by the theorem (since the environment of the instruction is $\not\sqsubseteq_{\mathcal{L}} \sigma$, if a channel is updated, the annotation of the channel would rise to a level $\not\sqsubseteq_{\mathcal{L}} \sigma$). Two cases are possible:

Case 1 At least one subsequent instruction of the program does not belong to $scope(t)$ (there is not a loop). Let $\bar{t}$ be the first instruction not belonging to $scope(t)$, that is $\bar{t}$ is the join point of the two branches. Both computations reach $\bar{t}$. Let $\bar{q_1} = \langle \bar{t}, \bar{\rho_1}, \bar{\mu_1}, \bar{\xi_1}, \bar{c_1}\rangle$ and $\bar{q_2} = \langle \bar{t}, \bar{\rho_2}, \bar{\mu_2}, \bar{\xi_2}, \bar{c_2}\rangle$, respectively, the corresponding states. Note that, since the definition of rule **If** and **While** upgrades the environment of all instructions in $scope(t)$, it holds $\bar{\rho_1} =_\sigma^E \rho_1 1 =_\sigma^E \bar{\rho_2} =_\sigma^E \rho_2 1$. Let i and j be the minimum indexes of the chain sem_n such that $\bar{q_1} \in sem_i$ and $\bar{q_2} \in sem_j$. Due to the align operation applied by next, there are two states in $sem_{\max(i,j)}(\bar{t})$, $\hat{q_1} = \left\langle \bar{t}, \hat{\rho_1}, \hat{\mu_1}, \hat{\xi_1}, \hat{c_1}\right\rangle$ and $\hat{q_2} = \left\langle \bar{t}, \hat{\rho_2}, \hat{\mu_2}, \hat{\xi_2}, \hat{c_2}\right\rangle$, corresponding resp. to the alignment of $\bar{q_1}$ and $\bar{q_2}$, i.e. with the same execution values resp. of $\bar{q_1}$ and $\bar{q_2}$, but with the security levels upgraded to the lub of all the states in $sem_{\max(i,j)}(\bar{t})$. We have that $(\hat{\mu_1}, \hat{\xi_1}) =_\sigma^{\mathcal{M},\Xi} (\hat{\mu_2}, \hat{\xi_2})$ since $(\mu_1, \xi_1) =_\sigma^{\mathcal{M},\Xi} (\mu_2, \xi_2)$ and a variable or a structure field in $\hat{q_1}, \hat{q_2}$ has level $\sqsubseteq_{\mathcal{L}} \sigma$ after align only if has not been affected by either of the two computations. A similar reasoning can be made for channels. Thus it holds that $\hat{q_1} =_\sigma^{\mathcal{Q}} \hat{q_2}$. The above reasoning can be iterated following the two computations from resp. $\hat{q_1}$ and $\hat{q_2}$.

Case 2 All subsequent instructions of the program belong to $scope(t)$ (there is a loop). By hypothesis, channels with security level $\sqsubseteq_{\mathcal{L}} \sigma$ cannot be affected from this point on by either of the two computations and thus the property is satisfied. □

5 Abstract domains

In this section we define the abstract domains of the abstract interpretation and we prove that they are connected to the concrete domains by means of a correctness condition. Let us recall the definition of Galois Insertion.

Definition 6 (*Galois Insertion*) Let $(C, \subseteq)$ and $(A, \sqsubseteq)$ be two complete lattices. Two functions $\alpha : C \mapsto A$ and

Fig. 10 Lattice of abstract values

$$\mathcal{A}^\sharp = \wp(\texttt{New}),\ \mathcal{V}^\sharp = (\{\bullet\} \cup \mathcal{A}^\sharp) \times \mathcal{L} \text{ ranged over by } v_1^\sharp, v_2^\sharp, \ldots$$

$$\sqsubseteq_\mathcal{V} :\ v_1^\sharp \sqsubseteq_\mathcal{V} v_2^\sharp \text{ iff } v_1^\sharp = (T_1,\sigma_1) \wedge v_2^\sharp = (T_2,\sigma_2) \wedge T_1 \subseteq T_2 \ \wedge\ \sigma_1 \sqsubseteq_\mathcal{L} \sigma_2 \vee \vee\ v_1^\sharp = (\bullet,\sigma_1) \wedge v_2^\sharp = (\bullet,\sigma_2) \wedge \sigma_1 \sqsubseteq_\mathcal{L} \sigma_2$$

$$\sqcup_\mathcal{V} :\ v_1^\sharp \sqcup_\mathcal{V} v_2^\sharp = \begin{cases} (T_1 \cup T_2, \sigma_1 \sqcup_\mathcal{L} \sigma_2) & \text{if } v_1^\sharp = (T_1,\sigma_1) \wedge v_2^\sharp = (T_2,\sigma_2) \\ (\bullet, \sigma_1 \sqcup_\mathcal{L} \sigma_2) & \text{if } v_1^\sharp = (\bullet,\sigma_1) \wedge v_1^\sharp = (\bullet,\sigma_2) \\ \top_\mathcal{V} & \text{otherwise} \end{cases}$$

$$\sqcap_\mathcal{V} :\ v_1^\sharp \sqcap_\mathcal{V} v_2^\sharp = \begin{cases} (T_1 \cap T_2, \sigma_1 \sqcap_\mathcal{L} \sigma_2) & \text{if } v_1^\sharp = (T_1,\sigma_1) \wedge v_2^\sharp = (T_2,\sigma_2) \\ (\bullet, \sigma_1 \sqcap_\mathcal{L} \sigma_2) & \text{if } v_1^\sharp = (\bullet,\sigma_1) \wedge v_1^\sharp = (\bullet,\sigma_2) \\ \bot_\mathcal{V} & \text{otherwise} \end{cases}$$

$$\alpha^1_\mathcal{V}(v) = \begin{cases} (\bullet,\sigma) & v = (k,\sigma), k \in \mathbb{Z} \\ (\{t\},\sigma) & v = ((\ell,t),\sigma), (\ell,t) \in \mathcal{A} \end{cases} \qquad y \in \wp(\mathcal{V}), \alpha_\mathcal{V}(y) = \bigsqcup_{v_i \in y}^\mathcal{V} \alpha^1_\mathcal{V}(v_i)$$

$$\gamma_\mathcal{V}(v^\sharp) = \begin{cases} \{(k,\sigma') | k \in \mathbb{Z}, \sigma' \sqsubseteq_\mathcal{L} \sigma\} & v^\sharp = (\bullet,\sigma) \\ \{((\ell,t),\sigma') | t \in T, (\ell,t) \in \mathcal{A}, \sigma' \sqsubseteq_\mathcal{L} \sigma\} & v^\sharp = (T,\sigma) \\ \mathcal{V} & v^\sharp = \top_\mathcal{V} \\ \emptyset & v^\sharp = \bot_\mathcal{V} \end{cases}$$

$\gamma : A \mapsto C$ form a *Galois insertion* between $(C, \subseteq)$ and $(A, \sqsubseteq)$, iff all the following conditions hold:

- α**-Monotonicity**: $\forall y, y' \in C.\ y \subseteq y' \Rightarrow \alpha(y) \sqsubseteq \alpha(y')$
- γ**-Monotonicity**: $\forall a, a' \in A.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')$
- **Galois**: $\forall y \in C.\ y \subseteq \gamma(\alpha(y))$
- **Insertion**: $\forall a \in A.\ \alpha(\gamma(a)) = a$

Showing that the concrete and the abstract domains are connected by a Galois Insertion will be necessary for proving that the abstract flow equations converge to a fixpoint [25]. The abstract domains are obtained by eliminating from the concrete values both the execution values and execution addresses. Every value maintains instead its security annotation. Simple values (`int`) are no longer held and are represented with a $\bullet$ symbol. In order to make the heap finite we abstract onto the same element different structures created at the same label. Moreover, an abstract address $\ell^\sharp$ is composed of a set of labels in `New`. In this way $\ell^\sharp$ records all the possible creation points of the structures pointed to by it during the computation. The operations defined on the lattice of abstract values $(\mathcal{V}^\sharp, \sqsubseteq^\sharp_\mathcal{V}, \sqcup^\sharp_\mathcal{V}, \sqcap^\sharp_\mathcal{V}, \bot^\sharp_\mathcal{V}, \top^\sharp_\mathcal{V})$ are reported in Fig. 10. The abstraction of a set of simple values is the lub of their security levels. We assume that $\alpha_\mathcal{V}$ returns the bottom element of $\mathcal{V}^\sharp$ if applied to the the empty set. Dually for the concretization function. The same for the other abstraction functions. The abstraction of a set of concrete references is an abstract reference that contains both the lub of their security levels and a set T of instruction points. T contains all the instructions at which the structure referenced is created. For example, if $v_1 = ((\ell_1,t_1),\sigma_1)$ and $v_2 = ((\ell_2,t_2),\sigma_2)$, then $v^\sharp = \alpha_\mathcal{V}(\{v_1,v_2\}) = (\{t_1,t_2\}, \sigma_1 \sqcup_\mathcal{L} \sigma_2)$.

An abstract memory $\mu^\sharp \in \mathcal{M}^\sharp_X = X \to \mathcal{V}^\sharp$ maps every variable in the set X to an abstract value (see Fig. 11). Two abstract memories can be compared only if their domains are the same. When $X = \mathit{Var}$ we omit the subscript and indicate the domain with $\mathcal{M}^\sharp$.

$$\begin{aligned} \mathcal{M}^\sharp_X &= X \to (\mathcal{L} \cup \mathcal{A}^\sharp) \text{ ranged over by } \mu_1, \mu_2, \ldots \\ \sqsubseteq_\mathcal{M} &: \mu_1 \sqsubseteq_\mathcal{M} \mu_2 \text{ iff } \forall x \in X. \mu_1(x) \sqsubseteq_\mathcal{A} \mu_2(x) \\ \sqcup_\mathcal{M} &: (\mu_1 \sqcup_\mathcal{M} \mu_2)(x) = \mu_1(x) \sqcup_\mathcal{V} \mu_2(x) \\ \sqcap_\mathcal{M} &: (\mu_1 \sqcap_\mathcal{M} \mu_2)(x) = \mu_1(x) \sqcap_\mathcal{V} \mu_2(x) \\ \bot_\mathcal{M} &: \forall x \in X. \bot_\mathcal{M}(x) = \bot_\mathcal{V} \\ \top_\mathcal{M} &: \forall x \in X. \top_\mathcal{M}(x) = \top_\mathcal{V} \\ \alpha_\mathcal{M}(y)(x) &= \alpha_\mathcal{V}(\{\mu(x) | \mu \in y\}) \\ \gamma_\mathcal{M}(\mu^\sharp) &= \{\mu \in \mathcal{M} | \forall x \in X. \mu(x) \in \gamma_\mathcal{V}(\mu(x))\} \end{aligned}$$

Fig. 11 Lattice of abstract memories

$$\begin{aligned} \Xi^\sharp &= \texttt{New} \to \mathcal{M}^\sharp_{\texttt{struct}} \text{ ranged over by } \xi_1, \xi_2, \ldots \\ \sqsubseteq_\Xi &: \xi_1 \sqsubseteq_\Xi \xi_2 \text{ iff } \forall t \in \texttt{New}. \xi_1(t) \sqsubseteq_\mathcal{M} \xi_2(t) \\ \sqcup_\Xi &: (\xi_1 \sqcup_\Xi \xi_2)(t) = \xi_1(t) \sqcup_\mathcal{M} \xi_2(t) \\ \sqcap_\Xi &: (\xi_1 \sqcap_\Xi \xi_2)(t) = \xi_1(t) \sqcap_\mathcal{M} \xi_2(t) \\ \bot_\Xi &: \forall t \in \texttt{New}. \bot_\Xi(t) = \bot_\mathcal{M} \\ \top_\Xi &: \forall t \in \texttt{New}. \top_\Xi(t) = \top_\mathcal{M} \\ \alpha_\Xi(y)(t) &= \alpha_\mathcal{M}(\{\xi(\ell,t) | \xi \in y, \ell \in \mathcal{A}^e\}) \\ \gamma_\Xi(\xi^\sharp) &= \{\xi \in \Xi | \forall (\ell,t) \in \mathcal{A}. \xi(\ell,t) \in \gamma_\mathcal{M}(\xi^\sharp(t))\} \end{aligned}$$

Fig. 12 Lattice of abstract heaps

An abstract heap $\xi^\sharp \in \Xi^\sharp = \texttt{New} \to \mathcal{M}^\sharp_{\texttt{struct}}$ is a map from structure creation points to abstract memories representing fields contents. Two heaps $\xi^\sharp_1, \xi^\sharp_2$ can be compared only if each abstract address points to structures of the same type, i.e. $\forall t \in \texttt{New}$, $\xi^\sharp_1(t)$ and $\xi^\sharp_2(t)$ are comparable memories (see Fig. 12).

Input and output channels are represented in the abstract domain $\mathcal{C}^\sharp = \texttt{Names} \to \mathcal{L}$ with tuples of security levels, one for each channel (see Fig. 13).

Proposition 2 *$\alpha_\mathcal{V}$ and $\gamma_\mathcal{V}$ form a Galois Insertion.*

Proof (α**-Monotonicity**) Let $y, y' \in \wp(\mathcal{V})$. Since $\alpha_\mathcal{V}(y) = \bigsqcup_{v_i \in y}^\mathcal{V} \alpha^1_\mathcal{V}(v_i)$ and $y \subseteq y'$, we can write

$$
\begin{array}{ll}
\mathcal{C}^\sharp & = \texttt{Names}_I \to \mathcal{L} \text{ ranged over by } c_1, c_2 \ldots \\
\sqsubseteq_{\mathcal{C}} & : \; c_1 \sqsubseteq_{\mathcal{C}} c_2 \text{ iff } \forall a \in \texttt{Names}. c_1(a) \sqsubseteq_{\mathcal{L}} c_2(a) \\
\sqcup_{\mathcal{C}} & : \; (c_1 \sqcup_{\mathcal{C}} c_2)(a) = c_1(a) \sqcup_{\mathcal{L}} c_2(a) \\
\sqcap_{\mathcal{C}} & : \; (c_1 \sqcap_{\mathcal{C}} c_2)(a) = c_1(a) \sqcap_{\mathcal{M}} c_2(a) \\
\bot_{\mathcal{C}} & : \; \forall a \in \texttt{Names}. \bot_{\mathcal{C}}(a) = \bot_{\mathcal{L}} \\
\top_{\mathcal{C}} & : \; \forall a \in \texttt{Names}. \top_{\mathcal{C}}(a) = \top_{\mathcal{L}}
\end{array}
$$

$$
\begin{array}{ll}
\alpha_{\mathcal{C}}(y)(a) & = \alpha_{\mathcal{L}}\left(\{c(a) | c \in y\}\right) \\
\gamma_{\mathcal{C}}(c^\sharp) & = \left\{c \in \mathcal{C} | \forall a \in \texttt{Names}. c(a) = (s, \sigma), s \in \mathbb{Z}^\star, \sigma \sqsubseteq_{\mathcal{L}} c^\sharp(a)\right\}
\end{array}
$$

Fig. 13 Lattice of abstract channels

$\alpha_{\mathcal{V}}(y') = \alpha_{\mathcal{V}}(y) \sqcup_{\mathcal{V}} \bigsqcup_{\mathcal{V}}_{v_i \in y \setminus y'} \alpha^1_{\mathcal{V}}(v_i)$. The properties of the lub $\sqcup_{\mathcal{V}}$ assure therefore that $\alpha_{\mathcal{V}}(y) \sqsubseteq_{\mathcal{A}} \alpha_{\mathcal{V}}(y')$.

(γ**-Monotonicity**) Let $v_1^\sharp, v_2^\sharp \in \mathcal{V}^\sharp$. If $v_1^\sharp \sqsubseteq_{\mathcal{V}} v_2^\sharp$ then either both $v_1^\sharp$ and $v_2^\sharp$ are addresses or both are simple values. Let us consider the former case, in which $v_1^\sharp = (T_1, \sigma_1)$ and $v_2^\sharp = (T_2, \sigma_2)$. We will prove $\gamma_{\mathcal{V}}(v_1^\sharp) \subseteq \gamma_{\mathcal{V}}(v_2^\sharp)$ showing that every $v = (\ell, t, \sigma) \in \gamma_{\mathcal{V}}(v_1^\sharp)$ belongs also to $\gamma_{\mathcal{V}}(v_2^\sharp)$. $t \in T \wedge \sigma \sqsubseteq_{\mathcal{L}} \sigma_1$ since $v \in \gamma_{\mathcal{V}}(v_1)$. On the other hand, $T_1 \subseteq T_2 \wedge \sigma_1 \sqsubseteq_{\mathcal{L}} \sigma_2$, since $v_1^\sharp \sqsubseteq_{\mathcal{V}} v_2^\sharp$. Then, $t \in T_1 \subseteq T_2 \Rightarrow t \in T_2$ and $\sigma \sqsubseteq_{\mathcal{L}} \sigma_1 \sqsubseteq_{\mathcal{L}} \sigma_2 \Rightarrow \sigma \sqsubseteq_{\mathcal{L}} \sigma_2$, thus proving that $v \in \gamma_{\mathcal{V}}(v_2^\sharp)$. If $v_1^\sharp$ and $v_2^\sharp$ are simple values, the proof is similar.

(**Galois**) Let $y \in \wp(\mathcal{V})$ and $v \in y$. We will show that $v \in \gamma_{\mathcal{V}}(\alpha_{\mathcal{V}}(y))$ to prove the thesis.

$$
\begin{array}{l}
v \in y \Rightarrow \{v\} \subseteq y \Rightarrow \text{(by monotonicity of } \alpha_{\mathcal{V}}) \\
\qquad \Rightarrow \alpha_{\mathcal{V}}(\{v\}) \sqsubseteq_{\mathcal{V}} \alpha_{\mathcal{V}}(y) \\
\qquad \Rightarrow \text{(by monotonicity of } \gamma_{\mathcal{V}}) \\
\qquad \Rightarrow \gamma_{\mathcal{V}}(\alpha_{\mathcal{V}}(\{v\})) \subseteq \gamma_{\mathcal{V}}(\alpha_{\mathcal{V}}(y)).
\end{array}
$$

Thus, it suffices to prove that $v \in \gamma_{\mathcal{V}}(\alpha_{\mathcal{V}}(\{v\})) = \gamma_{\mathcal{V}}(\alpha^1_{\mathcal{V}}(v))$ Suppose that y contains only addresses: then $\alpha^1_{\mathcal{V}}(v) = \alpha^1_{\mathcal{V}}((\ell, t, \sigma)) = v^\sharp = (t, \sigma)$. It is straightforward, by definition of $\gamma_{\mathcal{V}}$, that $v \in \gamma_{\mathcal{V}}(v^\sharp)$. Analogously for the case in which y contains only simple values. If y contains both addresses and simple values, $\alpha_{\mathcal{V}}(y) = \top_{\mathcal{V}}$ and $\gamma_{\mathcal{V}}(\top_{\mathcal{V}}) = \wp(\mathcal{V})$ that is the top element of the lattice $(\mathcal{V}, \subseteq)$.

(**Insertion**) Let $v \in \mathcal{V}^\sharp, v = (T, \sigma), y = \gamma_{\mathcal{V}}(v)$. Then, by definition of abstraction and concretization functions,

$$
\begin{array}{l}
\alpha_{\mathcal{V}}(y) = \bigsqcup_{\mathcal{V}}_{v_i \in y} \alpha_{\mathcal{V}}(v_i) = \bigsqcup_{\mathcal{V}}_{t \in T \sigma' \sqsubseteq_{\mathcal{L}} \sigma} \alpha_{\mathcal{V}}((\ell, t), \sigma') \\
\qquad = \bigsqcup_{\mathcal{V}}_{t \in T \sigma' \sqsubseteq_{\mathcal{L}} \sigma} (\{t\}, \sigma') = (T, \sigma)
\end{array}
$$

Similarly for a $v = (\bullet, \sigma)$. □

Proposition 3 *$\alpha_{\mathcal{M}}$ and $\gamma_{\mathcal{M}}$ form a Galois Insertion.*

Proof (α**-Monotonicity**) Let $y, y' \in \wp(\mathcal{M}_X)$ and $x \in X$.

$$
\begin{array}{l}
y \subseteq y' \Rightarrow \{\mu(x) | \mu \in y\} \subseteq \{\mu(x) | \mu \in y\} \\
\qquad \Rightarrow \text{ (by monotonicity of } \alpha_{\mathcal{V}}) \\
\qquad \Rightarrow \alpha_{\mathcal{V}}\left(\{\mu(x) | \mu \in y\}\right) \sqsubseteq_{\mathcal{V}} \alpha_{\mathcal{V}}\left(\{\mu(x) | \mu \in y\}\right) \\
\qquad \Rightarrow \text{(by definition of } \alpha_{\mathcal{M}}) \\
\qquad \Rightarrow \alpha_{\mathcal{M}}(y)(x) \sqsubseteq_{\mathcal{V}} \alpha_{\mathcal{M}}(y')(x)
\end{array}
$$

(γ**-Monotonicity**) Let $\mu, \mu' \in \mathcal{M}^\sharp$.

$$
\begin{array}{l}
\mu \sqsubseteq_{\mathcal{M}} \mu' \Leftrightarrow \forall x \in X. \mu(x) \sqsubseteq_{\mathcal{V}} \mu'(x) \\
\qquad \Rightarrow \text{ (by monotonicity of } \gamma_{\mathcal{V}}) \\
\qquad \Rightarrow \forall x \in X. \gamma_{\mathcal{V}}(\mu(x)) \subseteq \gamma_{\mathcal{V}}(\mu'(x)) \\
\qquad \Rightarrow \text{ (by definition of } \gamma_{\mathcal{M}}) \\
\qquad \Rightarrow \gamma_{\mathcal{M}}(\mu) \subseteq \gamma_{\mathcal{M}}(\mu').
\end{array}
$$

(**Galois**) Let $y \in \wp(\mathcal{M}_X)$ and $\mu \in y$.

$$
\begin{array}{l}
\{\mu\} \subseteq y \\
\quad \Rightarrow \text{(by monotonicity of } \alpha_{\mathcal{M}}) \\
\quad \Rightarrow \alpha_{\mathcal{M}}(\{\mu\}) \sqsubseteq_{\mathcal{M}} \alpha_{\mathcal{M}}(y) \\
\quad \Rightarrow \text{(by monotonicity of } \gamma_{\mathcal{M}}) \\
\quad \Rightarrow \gamma_{\mathcal{M}}(\alpha_{\mathcal{M}}(\{\mu\})) \subseteq \gamma_{\mathcal{M}}(\alpha_{\mathcal{M}}(y)).
\end{array}
$$

Thus, it suffices to show that $\mu \in \gamma_{\mathcal{M}}(\alpha_{\mathcal{M}}(\{\mu\}))$ to prove the thesis.

$$
\begin{array}{ll}
\text{(by definition of } \alpha_{\mathcal{M}}) & \forall x \in X \; \alpha_{\mathcal{M}}(\{\mu\})(x) = \alpha_{\mathcal{V}}(\{\mu(x)\}) \\
\text{(by Proposition 2)} & \forall x \in X \{\mu(x)\} \subseteq \gamma_{\mathcal{V}}(\alpha_{\mathcal{V}}(\{\mu(x)\})) \Rightarrow \\
\text{(by definition of } \gamma_{\mathcal{M}}) & \Rightarrow \quad \mu \in \gamma_{\mathcal{M}}(\alpha_{\mathcal{M}}(\{\mu\}))
\end{array}
$$

(**Insertion**) Let $\mu^\sharp \in \mathcal{M}^\sharp_X$ and let $x \in X$.

$$
\begin{array}{ll}
\alpha_{\mathcal{M}}(\gamma_{\mathcal{M}}(\mu^\sharp))(x) & = \text{(by definition of } \alpha_{\mathcal{M}}) \\
& = \alpha_{\mathcal{V}}(\{\mu(x) | \mu \in \gamma_{\mathcal{M}}(\mu^\sharp)\}) \\
& = \text{(by definition of } \gamma_{\mathcal{M}}) \\
& = \alpha_{\mathcal{V}}(\{\mu(x) | \mu \in \{\mu^{c'} | \forall x' \in X. \mu^{c'}(x') \\
& \qquad \in \gamma_{\mathcal{V}}(\mu^\sharp(x'))\}\}) \\
& = \text{(by set definition)} \\
& = \alpha_{\mathcal{V}}(\{\mu(x) | \mu(x) \in \gamma_{\mathcal{V}}(\mu^\sharp(x))\}) \\
& = \text{(by set definition)} \\
& = \alpha_{\mathcal{V}}(\gamma_{\mathcal{V}}(\mu^\sharp(x))) \\
& = \text{(by Proposition 2)} \\
& = \mu^\sharp(x)
\end{array}
$$

□

Proposition 4 *α_Ξ and γ_Ξ form a Galois Insertion.*

Proposition 5 *$\alpha_{\mathcal{C}}$ and $\gamma_{\mathcal{C}}$ form a Galois Insertion.*

We omit proofs for Propositions 4 and 5 since they are similar to the previous.

The abstract domain of states is $\mathcal{Q}^\natural = \mathcal{B} \to (\mathcal{L} \times \mathcal{M}^\natural \times \Xi^\natural \times \mathcal{C}^\natural)$. It contains all functions associating the instruction labels $\mathcal{B}$ with elements in $(\mathcal{L} \times \mathcal{M}^\natural \times \Xi^\natural \times \mathcal{C}^\natural)$. Given an abstract state $q^\natural \in \mathcal{Q}^\natural$, and an instruction label $t \in \mathcal{B}$, $q^\natural(t) = \langle \sigma, \mu^\natural, \xi^\natural, c^\natural \rangle$ is a tuple composed of a security level representing the security environment of t, an abstract memory, heap and channels. We use $q^\natural(t).env$ to denote σ. We denote by

$$dom(q^\natural) = \Big\{ t \mid q^\natural(t) = \langle \sigma, \mu^\natural, \xi^\natural, c^\natural \rangle \wedge \mu^\natural \neq \bot^\natural_{\mathcal{M}} \wedge \xi^\natural \neq \bot^\natural_{\Xi} \wedge c^\natural \neq \bot^\natural_{\mathcal{C}} \Big\}$$

the instruction addresses to which $q^\natural$ assigns a defined value for memory, heap and channels. We have that $(\mathcal{Q}^\natural, \sqsubseteq_{\mathcal{Q}})$ is a lattice, where, the operation $\sqsubseteq_{\mathcal{Q}}$ is defined as the pointwise application of the corresponding operation on the fields of the abstract states. Let us now consider the abstraction and concretization functions between the concrete and abstract domains of the states.

$\alpha_{\mathcal{Q}} : \wp(\mathcal{Q}) \to \mathcal{Q}^\natural$ is defined as follows. Let Q be a set of concrete states in $\mathcal{Q} = \mathcal{B} \times Env \times \mathcal{M} \times \Xi \times \mathcal{C}$. For each $t \in \mathcal{B}$, it is $\alpha_{\mathcal{Q}}(Q)(t) = \langle \sigma, \mu^\natural, \xi^\natural, c^\natural \rangle$ where

$$\sigma = \bigsqcup_{\mathcal{L}} \{\rho(t) \mid \langle t', \rho, \mu, \xi, c \rangle \in Q\},$$
$$\mu^\natural = \alpha_{\mathcal{M}}(\{\mu \mid \langle t, \rho, \mu, \xi, c \rangle \in Q\}),$$
$$\xi^\natural = \alpha_{\Xi}(\{\xi \mid \langle t, \rho, \mu, \xi, c \rangle \in Q\}),$$
$$c^\natural = \alpha_{\mathcal{C}}(\{c \mid \langle t, \rho, \mu, \xi, c \rangle \in Q\}).$$

If an instruction t does not occur in Q, then the abstraction functions $\alpha_{\mathcal{M}}$, α_{Ξ} and $\alpha_{\mathcal{C}}$ will produce bottom values, excluding t from $dom(\alpha_{\mathcal{Q}}(Q))$. Note that the security environment of an instruction t (whether t is in $dom(\alpha_{\mathcal{Q}}(Q))$ or not) in the abstract state is the lub of the security environments assigned to t by all states in Q, while the abstract memory, heap and channels associated with t are the lub of the abstractions of the concrete memories, heaps and channels, respectively, occurring the states of Q corresponding to the execution of the instruction with label t. For the concretization function $\gamma_{\mathcal{Q}} : \mathcal{Q}^\natural \to \wp(\mathcal{Q})$ we have:

$$\gamma_{\mathcal{Q}}(q^\natural) = \{\langle t, \rho, \mu, \xi, c \rangle \mid t \in dom(q^\natural), \forall t' \in \mathcal{B}, \rho(t') \sqsubseteq q^\natural(t').env, q^\natural(t) = \langle \sigma, \mu^\natural, \xi^\natural, c^\natural \rangle, \mu = \gamma_{\mathcal{M}}(\mu^\natural), \xi = \gamma_{\Xi}(\xi^\natural), c = \gamma_{\mathcal{C}}(c^\natural)\}$$

Theorem 2 *$\alpha_{\mathcal{Q}}$ and $\gamma_{\mathcal{Q}}$ form a Galois Insertion.*

Proof (**α-Monotonicity**) We want to prove that, $\forall Q, Q' \in \wp(\mathcal{Q})$, such that $Q \subseteq Q'$, $\alpha_{\mathcal{Q}}(Q) \sqsubseteq_{\mathcal{Q}} \alpha_{\mathcal{Q}}(Q')$. From the definition of $\alpha_{\mathcal{Q}}$ it is straightforward to prove $dom(\alpha_{\mathcal{Q}}(Q)) \subseteq dom(\alpha_{\mathcal{Q}}(Q'))$. It remains to prove that, given $t \in \mathcal{B}$ with $\alpha_{\mathcal{Q}}(Q)(t) = \langle \sigma, \mu^\natural, \xi^\natural, c^\natural \rangle$, $\alpha_{\mathcal{Q}}(Q')(t) = \langle \sigma, \mu^{\natural\prime}, \xi^{\natural\prime}, c^{\natural\prime} \rangle$, it is: (1) $\sigma \sqsubseteq_{\mathcal{L}} \sigma'$, (2) $\mu^\natural \sqsubseteq_{\mathcal{M}} \mu^{\natural\prime}$, (3) $\xi^\natural \sqsubseteq_{\Xi} \xi^{\natural\prime}$ and (4) $c^\natural \sqsubseteq_{\mathcal{C}} c^{\natural\prime}$. The first inequality holds by definition of $\alpha_{\mathcal{Q}}$ and the set inclusion between Q and Q'. The second, the third and the fourth are proven by the fact $Q \subseteq Q' \Rightarrow \{\langle t, \rho, \mu, \xi, c \rangle \in Q\} \subseteq \{\langle t, \rho, \mu, \xi, c \rangle \in Q'\}$.

(**γ-Monotonicity**) To prove this property, we must show that $\forall q_1^\natural, q_2^\natural \in \mathcal{Q}^\natural, q_1^\natural \sqsubseteq_{\mathcal{Q}} q_2^\natural, \gamma_{\mathcal{Q}}(q_1^\natural) \subseteq \gamma_{\mathcal{Q}}(q_2^\natural)$. Firstly, we note that $q_1^\natural \sqsubseteq_{\mathcal{Q}} q_2^\natural$ implies $dom(q_1^\natural) \subseteq dom(q_2^\natural)$. Other consequences of the hypothesis are that, if $q_1^\natural(t) = \langle \sigma_1, \mu_1^\natural, \xi_1^\natural, c_1^\natural \rangle$ and $q_2^\natural(t) = \langle \sigma_2, \mu_2^\natural, \xi_2^\natural, c_2^\natural \rangle$, then $\sigma_1 \sqsubseteq_{\mathcal{L}} \sigma_2$, $\mu_1^\natural \sqsubseteq_{\mathcal{M}} \mu_2^\natural$, $\xi_1^\natural \sqsubseteq_{\Xi} \xi_2^\natural$ and $c_1^\natural \sqsubseteq_{\mathcal{C}} c_2^\natural$. By $\gamma-$monotonicity for $\mathcal{M}$, Ξ and $\mathcal{C}$, we have that $\gamma_{\mathcal{M}}(\mu_1^\natural) \subseteq \gamma_{\mathcal{M}}(\mu_2^\natural)$, $\gamma_{\Xi}(\xi_1^\natural) \subseteq \gamma_{\Xi}(\xi_2^\natural)$ and $\gamma_{\mathcal{C}}(c_1^\natural) \subseteq \gamma_{\mathcal{C}}(c_2^\natural)$. All these facts suffice to state that, by construction of $\gamma_{\mathcal{Q}}$, $\gamma_{\mathcal{Q}}(q_1^\natural) \subseteq \gamma_{\mathcal{Q}}(q_2^\natural)$.

(**Galois**) Let $Q \in \wp(\mathcal{Q})$ and $q \in Q$, $q = \langle t, \rho, \mu, \xi, c \rangle$. We will show that $q \in \gamma_{\mathcal{V}}(\alpha_{\mathcal{V}}(Q))$ to prove that $Q \subseteq \gamma_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(Q))$.

$$\begin{aligned} q \in Q &\Rightarrow \{q\} \subseteq Q \Rightarrow \text{(by monotonicity of } \alpha_{\mathcal{Q}}) \\ &\Rightarrow \alpha_{\mathcal{Q}}(\{q\}) \sqsubseteq_{\mathcal{Q}} \alpha_{\mathcal{Q}}(Q) \Rightarrow \text{(by monotonicity of } \gamma_{\mathcal{Q}}) \\ &\Rightarrow \gamma_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(\{q\})) \subseteq \gamma_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(Q)). \end{aligned}$$

Thus, it suffices to prove that $q \in \gamma_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(\{q\}))$. We have that $\alpha_{\mathcal{Q}}(\{q\}) = q^\natural$, with $dom(q^\natural) = \{t\}$ and $\alpha_{\mathcal{Q}}(\{q\})(t) = \langle \rho(t), \alpha_{\mathcal{M}}(\mu), \alpha_{\Xi}(\xi), \alpha_{\mathcal{C}}(c) \rangle$. We have that $q \in \gamma_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(\{q\})) = \{\langle t', \rho', \mu', \xi', c' \rangle \mid \forall t' \in \mathcal{B}, \rho'(t) \sqsubseteq_{\mathcal{L}} \rho(t), \mu' \in \gamma_{\mathcal{M}}(\alpha_{\mathcal{M}}(\mu)), \xi' \in \gamma_{\Xi}(\alpha_{\Xi}(\xi)), c' \in \gamma_{\mathcal{C}}(\alpha_{\mathcal{C}}(c))\}$ that, by Propositions 3, 4 and 5 clearly contains q.

(**Insertion**) We have to prove that $\forall q^\natural \in \mathcal{Q}^\natural, \alpha_{\mathcal{Q}}(\gamma_{\mathcal{Q}}(q^\natural)) = q^\natural$. We will show that the thesis hold analyzing each field of a generic abstract state for the instruction t. For the environment we have:

$$\begin{aligned} \alpha_{\mathcal{Q}}(\gamma_{\mathcal{Q}}(q^\natural))(t).env &= \text{(by definition of } \alpha_{\mathcal{Q}}) \\ &= \bigsqcup_{\mathcal{L}} \{\rho(t) \mid \langle t', \rho(t), \mu, c \rangle \in \gamma_{\mathcal{Q}}(q^\natural)\} \\ &= \text{(by definition of } \gamma_{\mathcal{Q}}) \\ &= \bigsqcup_{\mathcal{L}} \{\rho(t) \sqsubseteq_{\mathcal{L}} q^\natural.env\} = q^\natural.env \end{aligned}$$

and for the memory:

$$\begin{aligned} \alpha_{\mathcal{Q}}(\gamma_{\mathcal{Q}}(q^\natural))(t).mem &= \text{(by definition of } \alpha_{\mathcal{Q}}) \\ &= \alpha_{\mathcal{M}}\left(\{\mu \mid \langle t, \rho(t), \mu, c \rangle \in \gamma_{\mathcal{Q}}(q^\natural)\}\right) \\ &= \text{(by definition of } \gamma_{\mathcal{Q}}) \\ &= \alpha_{\mathcal{M}}\left(\gamma_{\mathcal{Q}}(q^\natural.mem)\right) \\ &= \text{(by Proposition 3)} \\ &= q^\natural.mem \end{aligned}$$

where, given $q^\sharp \in \mathcal{Q}^\sharp$ and $t \in \mathcal{B}$ with $q^\sharp(t) = \langle \rho, \mu^\sharp, \xi^\sharp, c^\sharp \rangle$, $q^\sharp(t).mem$ indicates $\mu^\sharp$. The heap and channels parts are omitted because similar to the memory case. □

6 Abstract semantics and correctness

In this section we give an abstract semantics of the language that allows us to finitely execute the program in the abstract domain.

Figure 14 describes the abstract semantics of expressions. The rules of the abstract semantics for instructions are shown in Fig. 15. They define a relation $\xrightarrow{C^\sharp}$ between the abstract states: if the premise of the rule is true, the rule transforms the state $q^\sharp$ in the way described by the rule. There is only one rule for `if` and `while`: in both cases, besides propagating the state unchanged to the successors, the field *env* of all the instructions in *scope*(t) are updated. Rules **Value**$_{x.f}$ and **Assign**$_{x.f=e}$ need some explanations. In the abstract semantics, the structure addresses are lost and the references, besides the security level, contain the set T of possible creation points. Then, in order to obtain the abstract value `x.f` needed by Rule **Value**$_{x.f}$, it is necessary to compute the lub of $\xi^\sharp(t_i)(f)$ for all the t_i in the set T. Similarly, to execute Rule **Assign**$_{x.f=e}$, an assignment must be performed for each abstract structure that x might refer to.

Definition 7 (next$^\sharp$ *operator*) Given an abstract state $q^\sharp$, the application of the next$^\sharp$ operator yields the state reached in one step of computation from each instruction:

$$\mathsf{next}^\sharp(q^\sharp) = \bigsqcup\nolimits_{\mathcal{Q}} \left\{ \bar{q}^\sharp | q^\sharp \xrightarrow{C^\sharp} \bar{q}^\sharp \right\}$$

Proposition 6 (monotonicity of next$^\sharp$) next$^\sharp$ *is monotone in* $(\mathcal{Q}^\sharp, \sqsubseteq^\sharp)$.

Definition 8 (*initial abstract state* $q_0^\sharp$) For the initial state $q_0^\sharp$ we have $dom(q_0^\sharp) = \{t_0\}$ and $q_0^\sharp(t_0) = \left\langle \bot_{\mathcal{L}}, \bot_{\mathcal{M}}, \bot_\Xi, c_0^\sharp \right\rangle$, where for all $a \in \mathtt{Names}_I, c_0^\sharp(a) = \mathcal{S}(a)$ and for all $a \in \mathtt{Names}_O, c_0^\sharp(a) = \bot_{\mathcal{L}}$.

Definition 9 (*abstract semantics*) The abstract semantics $sem^\sharp \in \mathcal{Q}^\sharp$ is the least upper bound in $(\mathcal{Q}^\sharp, \sqsubseteq^\sharp)$ of the following increasing chain, defined for all $n \in \mathbb{N}$:

$$sem_0^\sharp = q_0^\sharp$$
$$sem_{n+1}^\sharp = \mathsf{next}^\sharp(sem_n)$$

Fig. 14 Abstract expressions semantics

$$\textbf{Const}\ \frac{}{\langle k, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (\bullet, \bot_{\mathcal{L}})} \qquad \textbf{Op}\ \frac{\langle E_1, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (\bullet, \tau_1), \langle E_2, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (\bullet, \tau_2)}{\langle E_1\ op\ E_2, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (\bullet, \tau_1 \sqcup_{\mathcal{L}} \tau_2)}$$

$$\textbf{Value}_{\mathtt{x}}\ \frac{}{\langle x, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} \mu^\sharp(x)} \qquad \textbf{Value}_{\mathtt{x.f}}\ \frac{\mu^\sharp(x) = (T, \sigma),\ \bigsqcup_{\forall, t \in T} \xi^\sharp(t)(f) = (w, \tau)}{\langle x.f, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (w, \tau \sqcup_{\mathcal{L}} \sigma)}$$

Fig. 15 Abstract semantics of commands

$$\textbf{Skip}_{\mathtt{t:skip}}\ \frac{q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ q^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle}{\bar{q}^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp, \xi'^\sharp \sqcup_\Xi \xi^\sharp, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp \rangle}$$

$$\textbf{Assign}_{\mathtt{t:x=e}}\ \frac{q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ q^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle,\ \langle E, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (w, \tau)}{\bar{q}^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp\ [x \leftarrow (w, \sigma \sqcup_{\mathcal{L}} \tau)], \xi'^\sharp \sqcup_\Xi \xi^\sharp, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp \rangle}$$

$$\textbf{Assign}_{\mathtt{t:x.f=e}}\ \frac{\begin{array}{c} q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ q^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle,\ \langle E, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (w, \tau_1), \\ \mu^\sharp(x) = (T, \tau_2),\ \tau_3 = \sigma \sqcup_{\mathcal{L}} \tau_1 \sqcup_{\mathcal{L}} \tau_2 \end{array}}{\bar{q}^\sharp(succ(t)) = \left\langle \sigma', \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp, \xi'^\sharp \sqcup_\Xi \xi^\sharp[t_j.f \leftarrow (w, \tau_3)]_{\forall t_j \in T}, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp \right\rangle}$$

$$\textbf{New}_{\mathtt{t:x=new\ S}}\ \frac{q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ q^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle}{\bar{q}^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp\ [x \leftarrow (t, \sigma)], \xi'^\sharp \sqcup_\Xi \xi^\sharp, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp \rangle}$$

$$\textbf{Input}_{\mathtt{t:a?x}}\ \frac{q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ q^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle,\ \tau = c^\sharp(a) \sqcup_{\mathcal{L}} \sigma}{\bar{q}^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp\ [x \leftarrow \tau], \xi'^\sharp \sqcup_\Xi \xi^\sharp, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp\ [a \leftarrow \tau] \rangle}$$

$$\textbf{Output}_{\mathtt{t:b!E}}\ \frac{q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ q^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle,\ \langle E, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (\bullet, \tau)}{\bar{q}^\sharp(succ(t)) = \langle \sigma', \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp, \xi'^\sharp \sqcup_\Xi \xi^\sharp, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp\ [b \leftarrow \sigma \sqcup_{\mathcal{L}} \tau] \rangle}$$

$$\textbf{If, while}_{\begin{array}{l}\mathtt{t:if\ (E)\ C} \\ \mathtt{else\ C} \\ \mathtt{while\ E\ C}\end{array}}\ \frac{\begin{array}{c} q^\sharp(t) = \langle \sigma, \mu^\sharp, \xi^\sharp, c^\sharp \rangle,\ \langle E, \mu^\sharp, \xi^\sharp \rangle \xrightarrow{E^\sharp} (\bullet, \tau), \\ q^\sharp(succ_{true}(t)) = \langle \sigma', \mu'^\sharp, \xi'^\sharp, c'^\sharp \rangle,\ q^\sharp(succ_{false}(t)) = \langle \sigma'', \mu''^\sharp, \xi''^\sharp, c''^\sharp \rangle, \\ \bar{q}^\sharp(succ_{true}(t)) = \langle \sigma' \sqcup_{\mathcal{L}} \tau, \mu'^\sharp \sqcup_{\mathcal{M}} \mu^\sharp, \xi'^\sharp \sqcup_\Xi \xi^\sharp, c'^\sharp \sqcup_{\mathcal{C}} c^\sharp \rangle \\ \bar{q}^\sharp(succ_{false}(t)) = \langle \sigma'' \sqcup_{\mathcal{L}} \tau, \mu''^\sharp \sqcup_{\mathcal{M}} \mu^\sharp, \xi''^\sharp \sqcup_\Xi \xi^\sharp, c''^\sharp \sqcup_{\mathcal{C}} c^\sharp \rangle, \\ \forall t' \in scope(t) : \bar{q}^\sharp(t').env = \tau \sqcup_{\mathcal{L}} q^\sharp(t').env \end{array}}{}$$

Proposition 7 (Expression semantics approximation)

$$\forall \langle E,\mu,\xi\rangle : \langle E,\mu,\xi\rangle \xrightarrow{E} v \Rightarrow \langle E,\alpha_{\mathcal{M}}(\mu),\alpha_{\Xi}(\xi)\rangle \xrightarrow{E^\sharp} v^\sharp \quad \textit{with } \alpha^1_{\mathcal{V}}(v) \sqsubseteq_{\mathcal{V}} v^\sharp$$

Proof We prove the correspondences between each pair of rules in concrete and abstract semantics for expressions.

(**Const**) $v^\sharp = \alpha^1_{\mathcal{V}}(v)$ is true since $\alpha^1_{\mathcal{V}}((k,\bot_{\mathcal{L}})) = \bot_{\mathcal{L}}$.

(**Value**$_x$) $\mu^\sharp = \alpha_{\mathcal{M}}(\mu) \Rightarrow \mu^\sharp(x) = \alpha^1_{\mathcal{V}}(\mu(x))\ \forall x \in Var$ by definition of $\alpha_{\mathcal{M}}$.

(**Value**$_{x.f}$) We want to prove $\alpha^1_{\mathcal{V}}(v^e,\sigma_1 \sqcup_{\mathcal{L}} \sigma_2) \sqsubseteq_{\mathcal{V}} (w,\sigma \sqcup_{\mathcal{L}} \tau)$. First of all, since $\mu^\sharp = \alpha_{\mathcal{M}}(\mu)$ then $T = \{t\}$ and $\sigma = \sigma_1$. Moreover, $v = (w,\tau) = \xi^\sharp(t)(f) = \alpha_{\Xi}(\xi)(t)(f) = \bigsqcup_{\mathcal{V}}_{(\ell_i,t)\in dom(\xi)} \xi(\ell_i,t)(f) \sqsubseteq_{\mathcal{V}} \xi(\ell)(f) = (v^e,\sigma_2)$, by which the thesis.

(**Op**) The thesis is $\alpha^1_{\mathcal{V}}(v^e_1\ op\ v^e_2,\sigma_1 \sqcup_{\mathcal{L}} \sigma_2) = (\bullet,\sigma_1 \sqcup_{\mathcal{L}} \sigma_2) \sqsubseteq_{\mathcal{V}} (\bullet,\tau_1 \sqcup_{\mathcal{L}} \tau_2)$. The proof can be conducted by structural induction. Supposing that $\alpha^1_{\mathcal{V}}(v^e_i,\sigma_i) \sqsubseteq_{\mathcal{V}} (\bullet,\tau_i), i = 1,2$, then $\sigma_i \sqsubseteq_{\mathcal{L}} \tau_i$, leading to $(\sigma_1 \sqcup_{\mathcal{L}} \sigma_2) \sqsubseteq_{\mathcal{L}} (\tau_1 \sqcup_{\mathcal{L}} \tau_2)$. □

Proposition 8 (Command semantics correspondence)

$$\forall q \in \mathcal{Q} : q \longrightarrow q' \Rightarrow \alpha_{\mathcal{Q}}(\{q\}) \xrightarrow{C^\sharp} q^{\sharp\prime} \quad \textit{such that } \alpha_{\mathcal{Q}}(\{q'\}) \sqsubseteq_{\mathcal{Q}} q^{\sharp\prime}.$$

Proof As in the previous proposition, we will prove that each abstract rule correctly approximates its correspondent concrete version. From the definition of abstraction function we have that $\forall i \neq t, \alpha_{\mathcal{Q}}(\{q\})(i) = \langle \rho(i), \bot_{\mathcal{M}^\sharp}, \bot_{\mathcal{C}^\sharp}\rangle$. Please note that the application of an abstract semantics rule upgrades only the part of the abstract state referring to the successor(s) of t and that, in every case, the environment is only upgraded. Let us suppose $q = \langle t,\rho,\mu,\xi,c\rangle$, $q' = \langle t',\rho',\mu',\xi',c'\rangle$, $q^{\sharp\prime}(t') = \langle \sigma',\mu^{\sharp\prime},\xi^{\sharp\prime},c^{\sharp\prime}\rangle$ We proceed by cases depending on the type of instruction with label t.

(**Assign**$_{\mathtt{t:x=e}}$) Only the memory status is affected by this rule. Therefore we have to prove that if $\mu' = \mu[x \leftarrow (v^e,\rho(t)\sqcup\sigma)] \wedge \mu^\sharp = \alpha_{\mathcal{M}}(\mu)$, then $\alpha_{\mathcal{M}}(\mu') \sqsubseteq_{\mathcal{M}} \mu^\sharp[x \leftarrow (w,\rho(t) \sqcup \tau)]$. This is true, since by the premises of the rules and Prop. 7, we can suppose $\alpha^1_{\mathcal{V}}((v^e,\sigma)) \sqsubseteq_{\mathcal{V}} (w,\tau)$.

(**Assign**$_{\mathtt{t:x.f=e}}$) In this case, only the heap state is affected, therefore we have to show that $\alpha_{\Xi}(\xi') \sqsubseteq_{\Xi} \xi^{\sharp\prime}$. Since $\mu^{\sharp\prime} = \alpha_{\mathcal{M}}(\mu)$, then $T = t_1$ and $\tau_2 = \sigma_2$. Moreover, by Prop. 7 applied to the premises of the rules, we have $\alpha_{\mathcal{V}}(v^e,\sigma_1) \sqsubseteq_{\mathcal{V}} (w,\tau_1)$. Then we can conclude

$$\alpha_{\mathcal{V}}((v^e,\sigma_3)) \sqsubseteq_{\mathcal{V}} (w,\tau_3) \qquad (*)$$

$\alpha_{\Xi}(\xi')(t) = \alpha_{\Xi}(\xi[\ell, f \leftarrow (v^e,\sigma_3)])(t) = \xi^\sharp(t),\ \forall t \in \mathtt{New},\ t \neq t_1$, since only the memory for an object created at label t_1 is changed by application of the concrete rule. On the other hand,

$$\alpha_{\Xi}(\xi')(t_1) = \alpha_{\mathcal{M}}(\{\xi'(\ell_j,t_1) | \ell_j \neq \ell\}) \sqcup_{\mathcal{M}} \alpha_{\mathcal{M}}(\xi'(\ell,t_1)) = \xi^\sharp(t_1) \sqcup_{\mathcal{M}} \alpha_{\mathcal{M}}(\xi'(\ell,t_1))$$

Therefore, since $\xi^{\sharp\prime}(t_1) = \xi^\sharp(t_1) \sqcup_{\mathcal{M}} \xi^\sharp(t_1)[f \leftarrow (w,\tau_3)]$, it remains to prove that:

$$\alpha_{\mathcal{M}}(\xi'(\ell,t_1)) \sqsubseteq_{\mathcal{M}} \xi^\sharp(t_1)[f \leftarrow (w,\tau_3)] \qquad (**)$$

However, we can note that only the field f is changed by the application of the rules, then, using (*), it is easy to show that also (**) and the thesis hold.

(**New**$_{\mathtt{t:x=new\ S}}$) While in the concrete rule both the memory and the heap change, in the abstract rule the heap state remains the same. That happens because, in the concrete rule, we create a new object in a fresh location, filling its fields with default (bottom) values. Since in the abstraction $\xi^\sharp(t)$ is the lub between all instances created at label t, the new object does not give any contribution. It remains to prove that:

$$\alpha_{\mathcal{M}}(\mu[x \leftarrow (\ell,t,\rho(t))]) \sqsubseteq_{\mathcal{M}} \alpha_{\mathcal{M}}(\mu)[x \leftarrow (t,\rho(t))]$$

which is straightforward from: $\alpha_{\mathcal{V}}((\ell,t,\rho(t))) = (t,\rho(t))$.

(**Input**$_{\mathtt{t:a?x}}$) In this case the memory and the input states change. For the memory it suffices to note that $\sigma = \tau \wedge \alpha_{\mathcal{V}}(k,\sigma \sqcup_{\mathcal{L}} \rho(t)) = (\bullet,\tau \sqcup_{\mathcal{L}} \rho(t))$. For the input we must show that: $\alpha_{\mathcal{C}}(c) \sqsubseteq_{\mathcal{C}} c^{\sharp\prime}$, that is true since only the a channel is changed and $c'(a) = (s,\rho(t) \sqcup_{\mathcal{L}} \sigma) \wedge c^{\sharp\prime}(a) = \rho(t) \sqcup_{\mathcal{L}} \tau$.

(**Output**$_{\mathtt{t:b?E}}$) From the premises and Proposition 7 we have that:

$$\alpha_{\mathcal{V}}(k,\sigma_1) \sqsubseteq_{\mathcal{V}} (\bullet,\tau) \Rightarrow \sigma_1 \sqsubseteq_{\mathcal{L}} \tau_1$$

From $c^\sharp = \alpha_{\mathcal{C}}(c)$ we have $\sigma_2 = \tau_2$. Then $\alpha_{\mathcal{C}}(c') \sqsubseteq_{\mathcal{C}} c^{\sharp\prime}$ is true since only the b channel is changed and $c'(b) = (s,\rho(t) \sqcup_{\mathcal{L}} \sigma_1 \sqcup_{\mathcal{L}} \sigma_2) \wedge c^{\sharp\prime}(b) = \rho(t) \sqcup_{\mathcal{L}} \tau_1 \sqcup_{\mathcal{L}} \tau_2$.

(**If**, **While**) We have $q' = \langle t',\rho',\mu,\xi,c\rangle$ with $\rho'(\bar{t}) = \rho(\bar{t})$ for $\bar{t} \notin scope(t)$, $\rho'(\bar{t}) = \rho(\bar{t}) \sqcup_{\mathcal{L}} \tau$ if $\bar{t} \notin scope(t)$ and the tested condition evaluates to $(\underline{\text{true}},\tau)$ or $(\underline{\text{false}},\tau)$. Now, since by Prop. 7 the tested condition in the abstract case must evaluate to τ' with $\tau \sqsubseteq_{\mathcal{L}} \tau'$ we have that $\alpha_{\mathcal{Q}}(q')(\bar{t}).env \sqsubseteq_{\mathcal{L}} q^{\sharp\prime}(\bar{t}).env$. The abstract state $\alpha_{\mathcal{Q}}(q')$ will have bottom value for

memory, heap and channels for the instruction labels different from t'. If $\bar{t} = t'$ then the memory, heap and channels are unchanged, and $\alpha_{\mathcal{Q}}(q') \sqsubseteq_{\mathcal{Q}} q^{\sharp\prime}$. □

Proposition 9 *The application of* align *preserves abstraction. That is,* $\forall Q \in \wp(\mathcal{Q})$,

$$\alpha_{\mathcal{Q}}(\mathsf{align}(Q)) = \alpha_{\mathcal{Q}}(Q)$$

Proof (sketch) Since $\mathsf{align}(Q)$ contains all the states in Q, plus some states for which the security levels are lifted to the lub of states in Q, these latter states do not contribute to the lub due to the application of $\alpha_{\mathcal{Q}}$. □

Proposition 10 (Local correctness) $\mathsf{next}^\sharp$ *is a safe approximation of* next*:*

$$\forall Q \in \wp(\mathcal{Q}) : \mathsf{next}(Q) \subseteq \gamma(\mathsf{next}^\sharp(\alpha_{\mathcal{Q}}(Q)))$$

Proof It suffices to prove that:

$$\alpha_{\mathcal{Q}}(\mathsf{next}(Q)) \sqsubseteq_{\mathcal{Q}} \mathsf{next}^\sharp(\alpha_{\mathcal{Q}}(Q)) \qquad (\dagger)$$

Indeed, applying γ-monotonicity to (†), we can conclude:

$$\mathsf{next}(Q) \subseteq \gamma(\alpha_{\mathcal{Q}}(\mathsf{next}(Q))) \subseteq \gamma(\mathsf{next}^\sharp(\alpha_{\mathcal{Q}}(Q)))$$

where the first subset operation is given by the Galois property (Theorem 2). On the other hand, equation (†) can be derived directly from Proposition 8 and from the definitions of next and $\mathsf{next}^\sharp$ operators.

Using the definition of $\mathsf{next}^\sharp$, we can rewrite the right-hand member of (†) as:

$$\begin{aligned}\mathsf{next}^\sharp(\alpha_{\mathcal{Q}}(Q)) &= \mathsf{next}^\sharp\left(\bigsqcup_{q''\in Q}^{\mathcal{Q}} \alpha_{\mathcal{Q}}(\{q''\})\right)\\ &= \bigsqcup_{q^\sharp \to q^{\sharp\prime}, q^\sharp \in \alpha_{\mathcal{Q}}(Q)}^{\mathcal{Q}} q^{\sharp\prime}\end{aligned}$$

On the other hand, we can use the definition of next to obtain:

$$\begin{aligned}\alpha_{\mathcal{Q}}(\mathsf{next}(Q)) &= \text{(by Definition 3)}\\ &= \alpha_{\mathcal{Q}}(\mathsf{align}(Q \cup \{q' | \exists q \in Q : q \longrightarrow q'\}))\\ &= \text{(by Proposition 9)}\\ &= \alpha_{\mathcal{Q}}(Q \cup \{q' | \exists q \in Q : q \longrightarrow q'\})\\ &\sqsubseteq_{\mathcal{Q}} \text{(by monotonicity of } \alpha_{\mathcal{Q}})\\ &\sqsubseteq_{\mathcal{Q}} \bigsqcup_{q\to q', q\in Q}^{\mathcal{Q}} \alpha(\{q'\})\end{aligned}$$

From these two results we can rewrite (†) as:

$$\bigsqcup_{q\to q', q\in Q}^{\mathcal{Q}} \alpha(\{q'\}) \sqsubseteq_{\mathcal{Q}} \bigsqcup_{q^\sharp \to q^{\sharp\prime}, q^\sharp \in \alpha(Q)}^{\mathcal{Q}} q^{\sharp\prime} \qquad (\dagger\dagger)$$

Since Proposition 8 holds, each state contributing to the left-hand lub is surely less or equal of a a state contributing to the right-hand lub. The lattice properties ensure that, given a, b, c, d in a lattice $(A, \sqsubseteq)$, with $a \sqsubseteq b \wedge c \sqsubseteq d$, then $(a \sqcup c) \sqsubseteq (b \sqcup d)$. By applying this property, we can conclude (††) and therefore the thesis. □

Theorem 3 (Global correctness) $\alpha_{\mathcal{Q}}(sem) \sqsubseteq_{\mathcal{Q}} sem^\sharp$.

Proof (sketch) The theorem can be proved by induction on the length of the chains sem_i and $sem_i^\sharp$, observing that $\alpha_{\mathcal{Q}}(sem_0) = sem_0^\sharp$ (base step) and applying Proposition 10 (induction step). □

A consequence of the above theorem is the following corollary. Its meaning is that we can use the abstract as a means to check secure information flow.

Corollary 1 *If, for any* $t \in \mathcal{B}$*, with* $sem^\sharp(t) = \langle t, \mu^\sharp, \xi^\sharp, c^\sharp \rangle$*, it holds that* $\forall a \in \mathtt{Names}$, $c^\sharp(a) \sqsubseteq_{\mathcal{L}} \mathcal{S}(a)$*, then the considered program has secure information flow.*

Proof (sketch) Combining Theorem 3 with Theorem 1. □

7 A prototype tool

A prototype tool (Iflow)[2] that, given a program, constructs its abstract semantics $sem^\sharp$, has been developed. Iflow accepts programs written in the language described in Sect. 3. The lattice $\mathcal{L}$ has been defined as the simplest two-level chain $\{L, H\}$, with $L \sqsubset_{\mathcal{L}} H$, but the tool can be easily extended to manage generic lattices. Iflow has been written in C++, using Flex [36] and Bison [20] as scanner and parser generators. After having parsed the input file, Iflow builds the initial abstract state $q_0^\sharp$. Then, starting from $q_0^\sharp$, it performs a least fixpoint computation using the Kildall working list algorithm [27]. Finally, it dumps $sem^\sharp$. Giving Iflow a "verbose" switch, it is possible to dump also each step of the fixpoint calculation.

As an example, consider the application of the algorithm to programs P5 and P8 in Fig. 4. In Fig. 16, we summarize the abstract execution, showing the result of the algorithm ($sem^\sharp$) in the two cases. Let us briefly explain how the state in Fig. 16a is computed for P5. Initially, the entry point of the program is inserted in the working list and abstractly executed. Every instruction brings its successor into the working list, and until instruction 4 is executed, the states are unchanged from their default value. Execution of instruction 4 aligns the value of x to

[2] Iflow is freely available at the URL: http://www.ing.unipi.it/ ~o1103499.

Fig. 16 Abstract semantics of the programs **a** P5 and **b** P8, calculated using lflow

Instruction	env	x	a	b	d
1:d?x	L	L	H	L	L
2:While (x > 0)	L	H	H	H	L
3:b!1	H	H	H	H	L
4:a?x	H	H	H	H	L
end:		H	H	H	L

(a)

Instruction	env	x	s1	s2	s3	1.f	1.g	2.f	2.g
1:s1=new S	L	L	$\emptyset, L$	$\emptyset, L$	$\emptyset, L$	L	L	L	L
2:s2=new S	L	L	$1, L$	$\emptyset, L$	$\emptyset, L$	L	L	L	L
3:a?x	L	L	$1, L$	$2, L$	$\emptyset, L$	L	L	L	L
4:if(x) 5: else 6:	L	H	$1, L$	$2, L$	$\emptyset, L$	L	L	L	L
5:s3=s1	H	H	$1, L$	$2, L$	$\emptyset, L$	L	L	L	L
6:s3=s2	H	H	$1, L$	$2\ L$	$\emptyset\ L$	L	L	L	L
7:s3.f=1	H	H	$1, L$	$2, L$	$\{1, 2\}, H$	L	L	L	L
8:???	H	H	$1, L$	$2, L$	$\{1, 2\}, H$	H	L	H	L

(b)

H. Then, when the while instruction is newly executed, the environment of all the instructions in its scope (3,4) is upgraded. The new execution of the loop lifts the security level of channel b to H (because of the environment, see Rule **Output**), thus making the program insecure. In Fig. 16b we show the abstract semantics for program P8. We can notice that, before executing instruction 7, `s3` may refer either to the object created at 1 or to the object created at 2. After the abstract execution of instruction 7, the field `f` of both the two abstract objects is upgraded.

7.1 The tax example

Reconsider the tax example described in Sect. 2, and suppose a program that implements the tax calculation. User data are stored in a list. Each element of the list is an instance of `User`, a structure (see Fig. 17) that contains three fields: `id` (the user ID), `incomeAmount` (the yearly income of the current user) and `next` (the reference to the next element of the list).

The program communicates, through I/O channels, with the users and with the net, that provides information about the current tax rates. In particular:

- the input channel `user` is used to receive the user id,
- the input channel `request` is used to receive the type of the user request,
- the input channel `income` is used to receive the user income,
- the input channel `from_net` is used to receive the tax rates and the threshold, from the net,
- the output channel `to_net` is used to request to the net:
  - the income threshold that discriminates between the application of the low and the high tax rate (0),
  - the low rate (1),
  - the high rate (2).
- the output channel `tax` is used to communicate to the user the tax rate he/she has to apply.

```
struct User {
  int id;
  int incomeAmount;
  struct User next;
};

struct User user_list,u,u1;
int n,x,flag,code,rate,tmp;

in user L;
in request L;
in income H;
in from_net L;
out to_net L;
out tax H;
```

Fig. 17 Declarations (tax example)

We set the security level of all the channels to **L** (low, public) except for the `tax` and the `income` channel, whose level is **H** (high, private). The `tax` channel holds confidential data since the applied rate depends on the user income. Recall that, according to our method, there is no need to assign a security level to the variables used in the program. The program is an infinite loop that waits for and serves requests from the users (see Fig. 18). The variable `tmp` holds the last value read from the channel `request` and contains the request code. Requests can be of four types:

- registration to the service (code 0),
- modification of the user income (code 1),
- request of the tax rate to be applied to the user income (code 2),
- un-registration to the service (code 3).

In the first case (Fig. 19) the user is inserted into the list, with a fictitious income of 0. In the second (see Fig. 20), the program, after having found the correct user instance in the list, sets the user income to the value taken from the confidential channel `income`. Otherwise, if the user was not registered, before setting the income, the program registers the user by adding a new element to the list. With the third type of request, an user asks the server which tax rate he/she must apply to his income (see Fig. 21). To answer, the program must contact the net that provides the current threshold and the two rates.

```
user_list = 0;
while (1) {
  user?x;
  request?tmp;                //   (⋆)
  if (tmp == 0) {
    // Registration of a new user (see Fig. 19)
  } else {
    if (tmp == 1)  {
      // Set the income for the user u.
      // Register the user if not registered yet.
      // (see Fig. 20)
    } else {
      if (tmp == 2)  {
        // Send the user the tax rate due (see Fig. 21)
      } else {
        if (tmp == 3 && user_list != 0) {
          // delete an user (see Fig. 22)
        } else {
          skip;
        };
      };
    };
  };
}; // end of the main loop
```

Fig. 18 Program structure (tax example)

```
if (tmp == 0) {
  u = new User;
  u.id = x;
  u.incomeAmount = 0;
  u.next = user_list;
  user_list = u;
} else { /* ... */ };
```

Fig. 19 The user registration (tax example)

```
if (tmp == 1) {
  u = user_list;  // search the user
  while (u != 0 && u.id != x ){ u = u.next; };
  if (u == 0){
    u = new User;          // create the user
    u.id = x;
    u.next = user_list;
    user_list = u;
  } else { skip; };
  income?tmp;              // get the income   (⋆⋆)
  u.incomeAmount = tmp;    // set the income
} else { /* ... */ };
```

Fig. 20 The modification of the income (tax example)

Finally, with the fourth type of request (see Fig. 22), an user can ask the program to delete his/her data from the list.

Analyzing the program with lflow, we can observe that, in the abstract semantics, all the channel but `to_net` respect their specification. In Fig. 23, the channels status, produced by the tool for the main loop instruction, is shown. The two security levels shown for each channel are the level specified by the policy and the level calculated by the abstract semantics, respectively. As the extract shows, the security level of `to_net` has

```
if (tmp == 2)  {
  u = user_list; // find the user
  while (u != 0 && u.id != x ) { u=u.next;};
  to_net!0;           // request the threshold
  from_net?n;         // get the threshold
  if (u != 0) {
    if (u.incomeAmount < n) {
      code = 1;       // request the low rate
    } else {
      code = 2;       // request the high rate
    };
    to_net!code;      // request the due tax rate
    from_net?rate;    // get the tax rate
    tax!rate;         // send the tax rate due
  } else { skip; };
} else { /* ... */ };
```

Fig. 21 The request of the tax rate (tax example)

```
if (tmp == 3 && user list != 0) {
  if (user_list.id == x) { // delete the head of the list
    user_list = user_list.next;
  } else {
    u = user_list;                  // find the user
    u1 = user_list.next;
    while (u1 != 0 && u1.id != x) {
      u = u1;
      u1 = u1.next;
    };
    if (u1 != 0) {
      u.next = u1.next;             // delete the user
    } else { skip; };
  };
} else { skip; };
```

Fig. 22 The deletion of an user (tax example)

```
State for instruction 106:While (Const){ 29 30 105 }
Var code          H
Var high_rate     L
Var low_rate      L
Var n             L
Var rate          L
Var tmp           H
Var u             L  { 34 49 }
Var u1            L  { 34 49 }
Var user_list     L  { 34 49 }
Var x             L
34.incomeAmount   H
34.label          L
34.next           L  {34 49 }
49.incomeAmount   H
49.label          L
49.next           L  {34 49 }
Channel from_net  L  L
Channel income    H  H
Channel tax       H  H
Channel to_net    L  H
Channel user      L  L
LFP iterations: 183
```

Fig. 23 Extract of the output of the lflow tool used with the tax example. The final channels state for the main while instruction is shown. The *last line* shows the number of iterations needed to reach the fixpoint

increased, in the abstract semantics, to the **H** security level. This is due to the part of the program that handles the request with code 2. Since the program requests

```
if (tmp == 2)  {
  u = user_list; // find the user
  while (u != 0 && u.id != x ) { u=u.next; };
  if (u != 0) {
    to_net!0;              // request the threshold
    from_net?n;            // get the threshold
    to_net!1;              // request the low rate
    from_net?low rate;     // get the low rate
    to_net!2;              // request the high rate
    from_net?high_rate;    // get the high rate
    if (u.incomeAmount < n){ // select the due rate
      rate = low_rate;
    } else {
      rate = high_rate;
    };
    tax!rate;              // send the rate
  } else { skip; };
} else { /* ... */ };
```

Fig. 24 The request of the tax rate - safe version (tax example)

the net either the high or the low rate through the public channel `to_net`, an external observer that has read access to this channel can infer if the user income amount is below the threshold or not, thus violating the policy. To overcome this problem, the code in Fig. 21 can be substituted by the code in Fig. 24. In this case, since the program requests both the low and the high rate, there is no way for the attacker to know which rate is communicated to the user, since the channel `tax` is confidential. Analyzing the new version of the program with lflow, the abstract semantics produced by the tool respects the policy. Please note that the variable `tmp` is used both to temporarily store the value read from the private channel `income` and to store the value read from the public channel `request`. Therefore, the variable `tmp` has a different security type after the instruction marked with ($\star$) (see Fig. 19) and after instruction ($\star\star$) (see Fig. 20). This would not have been possible with a Volpano-like typing [41], that would have assigned a high security type to `tmp` throughout the whole program, thus leading to deeming the program as insecure. To make this program typable in a Volpano-like typing, two different variables could be used in place of `tmp`: one to store private data, and another to store public data. In general, it may be not easy for the programmer to understand which variables need duplication and this may require a specific prior analysis. Moreover, if many variables needs duplication, the overhead may become significant, especially in embedded systems (e.g. Java Cards [12]) where the memory space is a concern.

7.2 Complexity

Let us now give a short account of the complexity of our analysis: for space complexity, it is $O(N \cdot \log(M) \cdot n)$ where $N = \sharp(Var) + \sharp(\mathtt{New})$ if the maximum number of fields of each structure is constant, M is the number of elements in $\mathcal{L}$ and n is the number of program points. The time complexity is theoretically $O(N^2 \cdot M \cdot n)$: every application of an abstract rule has a linear complexity in N due to the lub operation on the abstract memory and heap, and, in the worst case, the abstract state of every instruction can assume up to $O(N \cdot M)$ different values during the verification process. However, in practice, the number of abstract executions is much smaller. As suggested in [30] the dataflow analysis can be conducted at the level of the basic blocks instead of single instructions, saving only the state for the beginning of each basic block and calculating the others on the fly. This helps to reduce the space complexity to $O(N \cdot \log(M) \cdot B)$, and the time complexity to $O(N^2 \cdot M \cdot B)$, where B is the number of basic blocks.

8 Related work and conclusions

A recent survey of works on secure information flow is [39]. The problem has been coped with mainly by means of typing. In type-based approaches, each variable is assigned a security level, which is part of the type of the variable and secure information flow is checked by means of a type system; see, for example, [1,8,33,41,43]. The work [4] handles secure information flow in object oriented languages. They extend the syntax of a Java-like language with security annotations and build a type system that enforce noninterference. Their language is richer than ours since it includes classes and methods. However, the pointer analysis is only syntactical, while our analysis can distinguish between object instances created at different instructions. In [37,38] references, exceptions and `let`-polymorphism are treated for a call-by-value λ-calculus.

As compared with typing, AI can give a finer inspection of information flows, since it executes the program, even though in an abstract way. In order to check input/output non-interference, it is not necessary to associate security levels to variables: a variable, during its life, can hold data with different security levels without affecting non-interference, provided that the output channels contain data with levels that are lower than or equal to the channel's level. Security typing can be obtained in our framework by collapsing the rows of the abstract semantics (corresponding to the instructions) into a unique row containing the pointwise lub of the rows. We think that also explicit declassification (as proposed, for instance, in [34]) can be suitably handled by abstract interpretation. The language could be extended with a new command $\mathtt{a}!^{\sigma}\ \mathtt{E}$ that sends the result of the expression `E` to the channel `a` forcing the

security annotation of the expression to be equal to σ. The concrete semantics rule could be:

$$\textbf{Declassification}_{\texttt{t:b!}^{\sigma}\ \texttt{E}} \quad \frac{\langle E, \mu, \xi\rangle \xrightarrow{E} (k, \tau_1),\ c(b) = (s, \tau_2),\ b \in \texttt{Names}_O}{\langle t, \rho, \mu, \xi, c\rangle \longrightarrow \left\langle succ(t), \rho, \mu, \xi, c\left[b \leftarrow (s \cdot k, \sigma \sqcup_{\mathcal{L}} \tau_2)\right]\right\rangle}$$

and the non-interference property of Definition 1 has to be changed in order to exclude the output performed using this new command.

Other papers based on AI [42,21] takes as abstract domain the lattice of levels and perform an AI with almost the same power of typing (in terms of class of certified programs). Thus they do not exploit all the power of abstract interpretation. For example, they can not certify program P7 above. On the other hand, the focus of [21] is the definition of a framework based on AI able to represent a parametrized notion of non-interference. Approaches that are able to cope with "temporary breaking of security", similar to that presented by program P7, are based on theorem proving [24,26]. AI is also exploited in [2] to annotate programs with pre and postconditions defining variable dependencies.

An early work for analyzing secure information flow in high-level languages was presented by Mizuno and Schmidt [32]. They describe the execution of programs in a rich language (non-void procedures with local and global variables are considered) by means of a denotational semantics that handles execution values and security levels. The full semantics is then approximated, using an AI, by an abstract semantics that handles only security levels. Though they provide a full proof that this approximation is correct, they do not prove that the full semantics enforces the security properties. On the contrary, we justify the use of the concrete semantics by means of Theorem 1.

A recent paper by Hunt and Sands [23] introduces a security typing system that is flow-sensitive, i.e. it allows variables to hold different security levels in different instructions of the program, as our analysis does. Even though their analysis does not deal neither with channels nor with dynamic structures, and therefore a straightforward comparison is not possible, it seems that they achieve the same precision of our method. Moreover, the paper presents an algorithm that transforms a program that is typable in their flow-sensitive analysis into another program that can be also typed in a Volpano-like typing [41].

Some previous papers of the team to which the authors belong cope with the definition of abstractions suitable to check secure information flow, based on the annotation of data with security levels. The works [6,7,9,10] handle secure information flow in stack based machine languages, while the papers [5,17] consider high level languages, including parallel ones. In these papers abstract transition systems are used, possibly having a high number of states: the same instruction may belong to different states, characterized by different security environments and memories. This corresponds to perform a less approximate abstract interpretation than instruction level security typing. The number of states being high, the abstraction is not suitable to be directly used for a definition of a tool for checking secure information flow. In fact there is a need for other techniques to be combined with this abstraction method: in the above papers we used model checking to complete the verification process (a similar combination of abstraction and model checking is used in [11]). In the present paper, instead, the abstract semantics is a table composed of a row for each program point and is built by an efficient fixpoint algorithm using the abstract rules. Finally, the previous papers of the authors do not cope with pointers and dynamic structures, here handled by a suitable abstract domain.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Proceedings, January 20–22, 1999, San Antonio, pp. 147–160. ACM, New York (1999)
2. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) SAS 2004 (11th Static Analysis Symposium), Verona, Italy, August 2004, Vol. 3148 of Lecture Notes in Computer Science, pp. 100–115. Springer, Berlin (2004)
3. Avvenuti, M., Bernardeschi, C., De Francesco, N.: Java bytecode verification for secure information flow. ACM SIGPLAN Notices **38**(12), 20–27 (2003)
4. Banerjee, A., Naumann, D.: Secure information flow and pointer confinement in a Java-like language. In: 15th IEEE Security Foundations Workshop (CSFW'02) Proceedings. IEEE, 2002
5. Barbuti, R., Bernardeschi, C., De Francesco, N.: Abstract interpretation of operational semantics for secure information flow. Inform. Process. Lett. **83**(2), 101–108 (2002)
6. Barbuti, R., Bernardeschi, C., De Francesco N.: Checking security of Java bytecode by abstract interpretation. In: SAC '02: Proceedings of the 2002 ACM symposium on Applied computing, March 10–14, 2002, Madrid, Spain, pp. 229–236. ACM New York (2002)
7. Barbuti, R., Bernardeschi, C., De Francesco, N.: Analyzing information flow properties in assembly code by abstract interpretation. Comput. J. **47**(1), 25–45 (2004)

8. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on types in languages design and implementation, pp. 103–112, New York, NY, USA, 2005. ACM, New York
9. Bernardeschi, C., De Francesco, N.: Combining abstract interpretation and model checking for analysing security properties of Java bytecode. In: Cortesi, A. (ed.) Third International Workshop on Verification, Model Checking and Abstract Interpretation Proceedings, VMCAI 2002, Venice, January 21–22, 2002, Proceedings, Vol. 2294 of Lecture Notes in Computer Science, pp. 1–15. Springer, Berlin (2002)
10. Bernardeschi, C., De Francesco, N., Lettieri, G.: An abstract semantics tool for secure information flow of stack-based assembly programs. Microprocess. Microsyst. **26**(8), 391–398 (2002)
11. Bieber, P., Cazin, J., Girard, P., Lanet, J.-L., Wiels, V., Zanon, G.: Checking secure interactions of smart card applets. In: ESORICS 2000 Proceedings (2000)
12. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer'S Guide. Addison-Wesley Longman Publishing, (2000)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Proceedings, pp. 238–252, Los Angeles, (1977)
14. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Logic Comput. **2**, 511–547 (1992)
15. Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretations. In: ACM (ed) Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92), Albuquerque, New Mexico, January 1992, pp. 83–94. ACM, New York (1992)
16. De Francesco, N., Martini, L.: Abstract interpretation to check secure information flow in programs with input–output security annotations. In: Dimitrakos, T., Martinelli, F., Ryan, P. Y., Schneider, S., (eds) Formal Aspects in Security and Trust: Third International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18–19, 2005, Revised Selected Papers, Lecture Notes in Computer Science, pp. 63–80. Springer, Berlin (2006)
17. De Francesco, N., Santone, A., Tesei, L.: Abstract interpretation and model checking for checking secure information flow in concurrent systems. Fundam. Inf. **54**(2–3), 195–211 (2003)
18. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
19. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun ACM **20**(7), 504–513 (1977)
20. Donnely, C., Stallman, R.: Bison, the YACC-compatible parser generator. Free Software Foundation, November (1995)
21. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Jones, N. D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '04), Venice, Italy, January 14–16, 2004, pp. 186–197, ACM, New York (2004)
22. Heintze, N., Riecke, J. G.: The SLam calculus: programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA, pp. 365–377. ACM, New York (1998)
23. Hunt, S., Sands, D.: On flow-sensitive security types. In: Morrisett, J. G., Jones, S. L. P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11–13, 2006, pp. 79–90. ACM, New York (2006)
24. Jacobs, B., Pieters, W., Warnier, M.: Statically checking confidentiality via dynamic labels. In: Workshop on Issues in the Theory of Security proceedings, Long Beach, CA, United States, January 20, 2005. ACM, New York (2005)
25. Jones, N. D., Nielson, F.: Abstract interpretation: a semantic based tool for program analysis. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, Vol. 4, 527–636. Oxford University Press, Oxford (1995)
26. Joshi, R., Leino, K.M.: A semantic approach to secure information flow. Sci. Comput. Programm. **37**(1–3), 113–138 (2000)
27. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, October 1973, pp. 194–206 (1973)
28. Kobayashi, N., Shirane, K.: Type-based information flow analysis for low-level languages. In: Informal Proceedings of the 3rd Asian Workshop on Programming Languages and Systems (APLAS'02) (2002)
29. Lampson, B.W.: A note on the confinement problem. Commun. ACM, **16**(10), 613–615 (1973)
30. Leroy, X.: Java bytecode verification: algorithms and formalizations. J. Automat. Reason. **30**(3–4), 235–269 (2003)
31. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing (1999)
32. Mizuno, M., Schmidt, D.A.: A security flow control algorithm and its denotational semantics correctness proof. Formal Aspects Comput. **4**, 727–754 (1992)
33. Myers, A.C.: Jflow: practical mostly-static information flow control. In: 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Proceedings, January 20–22, 1999, San Antonio, TX, pp. 228–241. ACM, New York (1999)
34. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles, pp. 129–142. ACM, New York (1997)
35. A guide to understanding Discretionary Access Control in trusted systems. Technical Report NCSC-TG-003 Version 1, National Computer Security Center, 1987
36. Paxson, V.: Flex, a fast scanner generator, version 2.5, March 1995
37. Pottier, F., Conchon, S.: Information flow inference for free. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000, SIGPLAN Notices **35**(9), pp. 46–57 (2000)
38. Pottier, F., Simonet, V.: Information flow inference for ML. In 29th ACM Symposium on Principles of Programming Languages Proceedings, Portland, January 16–18, 2002, pp. 319–330 (2002)

39. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Select. Areas Commun. **21**(1), 5–19 (2003)
40. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, pp. 1–10. ACM, New York (1998)
41. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(3), 167–187 (1996)
42. Zanotti, M.: Security typings by abstract interpretation. In: Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17–20, 2002, Proceedrings, Vol. 2477 of Lecture Notes in Computer Science, pp. 360–375, Springer, Berlin (2002)
43. Zdancewic, S., Myers, A.C.: Secure information flow via linear continuations. Higher Order Symbol. Comput. **15**(2–3), 209–234, Kluwer, Dordrecht (2002)