The Cardinal Abstraction for Quantitative Information Flow

Mounir Assaf, Julien Signoles, Eric Totel, Frédéric Tronel

The Cardinal Abstraction for Quantitative Information Flow

Mounir Assaf* Julien Signoles† Éric Totel‡ Frédéric Tronel‡

* Stevens Institute of Technology
Hoboken, US
first.last@stevens.edu

† CEA LIST, Software Reliability Lab.
Gif-sur-Yvette, France
first.last@cea.fr

‡ CentraleSupélec/Inria
Rennes, France
first.last@centralesupelec.fr

***Abstract*—Qualitative information flow aims at detecting information leaks, whereas the emerging quantitative techniques target the estimation of information leaks. Quantifying information flow in the presence of low inputs is challenging, since the traditional techniques of approximating and counting the reachable states of a program no longer suffice.**

This paper proposes an automated quantitative information flow analysis for imperative deterministic programs with low inputs. The approach relies on a novel abstract domain, the cardinal abstraction, in order to compute a precise upper-bound over the maximum leakage of batch-job programs. We prove the soundness of the cardinal abstract domain by relying on the framework of abstract interpretation. We also prove its precision with respect to a flow-sensitive type system for the two-point security lattice.

I. Introduction

Secure information flow is a challenging problem in program analysis. Earlier techniques focus on enforcing qualitative security properties such as non-interference properties [1], [2]. Such qualitative techniques aim at detecting and avoiding information leaks by relying, for instance, on type systems [3], [4], [5], program logic [6], [7], [8] or monitoring [9], [10].

Motivated by the observation that many programs cannot comply with strict security policies preventing information leaks, recent works propose to address the secure information flow challenges by relying on Quantitative Information Flow (QIF). These quantitative techniques propose to measure the amount of leaked information in order to decide whether it is tolerable.

Traditionally, the QIF literature defines the leakage of a program as the difference between an initial uncertainty about the secret prior to any program run, and the remaining uncertainty after attackers observe the outputs of a program:

$$\text{leakage} = \text{initial uncertainty} - \text{remaining uncertainty}.$$

Denning [11, Chapter 5] proposes the first quantitative measure of a program's leakage in terms of Shannon entropy [12]. Clark et al. [13] builds on Denning's work to characterize both the initial uncertainty and the remaining one in terms of Shannon entropy and conditional entropy, then defines the information leakage as the reduction of uncertainty about the secret. Clarkson et al. [14], [15] propose to define the leakage as a change in accuracy, rather than uncertainty, by defining an initial and remaining accuracy in terms of information belief [16]. In a different approach, Smith [17], [18] proposes to quantify information leaks wrt. min-entropy [19], that is the probability of attackers guessing the secret in one try. Smith then defines both the initial uncertainty and the remaining uncertainty in terms of the probability of attackers guessing the secret.

Since both the initial uncertainty and the remaining uncertainty depend on the initial probability distribution of the secret, Denning relies on *capacity* – the maximum leakage measured through Shannon entropy over all distributions of the secret –, to abstract away from the assumed distribution of the secret. Similarly, Smith also relies on *min-capacity*, that is the maximum leakage measured through min-entropy over all distributions of the secret. Additionally, Smith [17], [18] proves that for deterministic programs, capacity and min-capacity coincide, and they are given by the logarithm of the cardinal number of outputs a program may produce:

$$\mathcal{ML} = \log_2 |outputs|.$$

Smith's framework considers a program as a channel that accepts only high inputs. Therefore, a program $\mathtt{c}$ that accepts low inputs can be seen as inducing a family of channels accepting only high inputs, and the maximum leakage of $\mathtt{c}$ can be defined as the maximum leakage over all induced channels. Therefore, by assuming the input memories are partitioned into low input memories $\varrho_{L_0} \in \Sigma_{L_0}$ and high input memories $\varrho_{H_0} \in \Sigma_{H_0}$, the maximum leakage for programs accepting both low and high inputs is given by:

$$\mathcal{ML} = \max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 \left| outputs_{\varrho_{L_0}} \right|. \tag{1}$$

Let us consider a command $\mathtt{c}$ that outputs no intermediate steps of computation. Let us also assume a set of variables $\mathtt{v}_1, \mathtt{v}_2, \ldots \mathtt{v}_f$ attackers can observe at the end of command $\mathtt{c}$'s execution. Thus, the outputs attackers may observe are solely determined by the final environments ϱ resulting from the execution of command $\mathtt{c}$:

$$\begin{aligned} outputs_{\varrho_{L_0}} = \{ & (\varrho(v_1), \varrho(v_2) \ldots, \varrho(v_f)) \mid \\ & \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } [\![c]\!]\varrho_0 = \varrho \}. \end{aligned} \tag{2}$$

Therefore, computing the maximum leakage for deterministic batch-job programs requires maximizing the cardinality of the set introduced in Equation (2), over all low input memories ϱ_{L_0}.

Most existing QIF analyses tackle this problem by reducing it to a reachability problem on a self-composed program [7], [6], [8], [20]. Indeed, recent techniques [21], [22], [23] rely on software model checkers [24], [25], [26] in order to

find symbolic witnesses of an information leak, namely two low equivalent memories that deliver different observable outputs. Since for deterministic programs, each observable output induces an equivalence class over low equivalent input memories that are indistinguishable for that output, these approaches compute the maximum leakage by synthesising and enumerating the number of all such symbolic witnesses. Köpf and Rybalchenko [27] note that such an approach is prohibitively hard, and propose approximation-based approaches for QIF. These approaches include relying on a generic abstract interpretation [28] analysis in order to upper-bound the leakage of a program by approximating the reachable states of a program. However, observe from Equation (2) that in the presence of low inputs, one needs to approximate a set of sets of reachable states – one set of reachable states for each possible low inputs –, in order to precisely upper-bound the maximum leakage $\mathcal{ML}$. This latter remark means that while we can rely on traditional abstract domains to approximate the set of final reachable states irrespective of the low inputs, such an approach would deliver an imprecise over-approximation of the leakage.

The contributions of this paper include a novel non-relational abstract domain, the cardinal abstraction, for automated QIF. This abstract domain computes directly *variety* – the cardinal number of values variables may take – in order to upper-bound the maximum leakage of programs accepting both low and high inputs, instead of relying on an underlying reachability analysis. Section II introduces the intuition behind the cardinal abstract domain.

Section III defines the cardinal abstract domain and introduces its abstract semantics, all the while providing intuitive arguments for its soundness. We also characterize in Theorem 1 an upper-bound over the maximum leakage, by relying on the result of the cardinal abstract domain.

The cardinal abstract domain is sound. Section IV sketches this soundness proof, and defines the collecting semantics over a set of sets of reachable states, that is over an abstraction of the more general hyperproperties [29], [30]. While such a collecting semantics may be termed as "non-standard" in abstract interpretation, it appears explicitly in [31, Section 11], and implicitly in [32, Section 3]. We summarize how to build a Galois connection relating our concrete domain to the abstract one. We also gloss over the functional lifting of this abstraction in order to systematically derive the abstract semantics of the cardinal abstraction.

The cardinal abstract domain is also precise. Indeed, while quantitative security properties emerge as an alternative to qualitative properties, it is unclear how existing approximation-based QIF analyses fare wrt. the more robust qualitative analyses. To the best of our knowledge, the cardinal abstraction is the first approximation-based static analysis that provably computes a leakage of zero bits for programs that are "well-typed" by the traditional Hunt and Sands' [4], [5] flow-sensitive type system. Section V proves that the results of the cardinal abstract domain can be abstracted furthermore to yield a security labelling of variables that is at least as precise as the labelling obtained by Hunt and Sands' flow-sensitive type system for the two-point lattice. Thus, the cardinal abstraction provides an alternative to the above-mentioned flow-sensitive type system for proving non-interference, while providing additional quantitative information to guide declassification when needed.

Section VI introduces related work and Section VII concludes.

An appendix, providing detailed proofs for all results, can be found at the end of this technical report.

II. Overview

The cardinal abstraction over-approximates *variety* directly, that is the cardinal number of values variables may take, in order to upper-bound the maximum leakage.

a) Towards the cardinal abstraction: Consider for instance the program in Listing 1, where variable `s` is a high input, whereas variables `input` and `x` are low inputs. We assume these variables are integers of finite size κ, covering the range $[-2^{\kappa-1}, 2^{\kappa-1}-1]$. Let us assume that only variable `x` is a low **output** observable by attackers. Thus, the maximum leakage for this program is given by $\mathcal{ML} = \log_2(2) = 1\ bit$, which accounts for the intuition that only the parity bit of the secret leaks to attackers.

```
// {s ↦ 2^κ, input ↦ 1, x ↦ 1}
x := s mod 2; // {s ↦ 2^κ, input ↦ 1, x ↦ 2}
x := x + input; // {s ↦ 2^κ, input ↦ 1, x ↦ 2}
```

Listing 1. A program accepting both low and high inputs

In order to analyse the program in Listing 1, let us focus only on computing variety. Such an abstraction enables the computation of an over-approximation of the maximum leakage directly.

We annotate Listing 1 with the results of an analysis computing only variety. Equation (2) requires us to compute the variety for a fixed arbitrary low input, while the secret ranges throughout all possible values in $[-2^{\kappa-1}, 2^{\kappa-1}-1]$. Therefore, our analysis initially maps variable `s` to a cardinal number of 2^κ values, whereas it maps both variables `s` and `x` to a cardinal number of 1 value. Assuming attackers only observe variable `x`, we can deduce that they make at most 2 different observations for each low input, at the end of the program. Thus, we can also deduce that the maximum leakage of the program in Listing 1 is at most 1 bit. In this example, the analysis in Listing 1 is precise: it computes a tight upper-bound over the maximum leakage. Yet, abstract interpretation [28] in general may incur a loss of precision, depending on the underlying approximate representations.

In fact, an analysis that only keeps track of variety will lose precision whenever it encounters a conditional instruction. Consider for instance the program in Listing 2 that is annotated with an analysis that only tracks variety. Let us assume that variable `secret` is a high input, whereas variable `input` is a low input. At both Lines 2 and 5, the analysis determines that variable `x` has only one possible value since it is assigned a constant. However, how many values can variable `x` have at the merge point of the conditional at Line 6?

Unlike abstract domains aiming at approximating trace properties, the cardinal abstraction cannot simply compute

the union of both abstract states at the merge point of the conditional, since this would be unsound: with $\max$ as a union operator over the lattice $([0, 2^\kappa], \leq, 0, 2^\kappa, \max, \min)$ of natural numbers, the analysis would find that variable `x` has at most 1 value, which is obviously not sound. Instead, the cardinal abstraction must add both abstract values for variable `x`, in order to soundly deduce that variable `x` has at most 2 different values at the merge point of the conditional at Line 6. Note also that providing the set $[0, 2^\kappa]$ with the addition operator as a join operator cannot define a proper lattice structure, since addition is not idempotent. Similarly, variable `input` also has 1 possible value in each branch of the conditional. Therefore, without any additional information, the analysis must treat variable `input` the same way it treats variable `x`, and must deduce that variable `input` has at most 2 different values at the merge point of the conditional at Line 6. In order to retain precision for variable `input`, the analysis must know that variable `input` is not modified inside the conditional branches – otherwise, the abstract value of variable `input` will increase whenever the analysis encounters a branching instruction. We achieve this by letting the cardinal abstraction track both the cardinal number of values variables may take, and the program points where variables may have been last assigned.

```
0 // {secret ↦ 2^κ, input ↦ 1, x ↦ 1}
1 if (secret > input){
2   x := 0; // {secret ↦ 2^κ, input ↦ 1, x ↦ 1}
3 }
4 else {
5   x := 1; // {secret ↦ 2^κ, input ↦ 1, x ↦ 1}
6 } // {secret ↦ 2^κ, input ↦ 2, x ↦ 2}
```

Listing 2. An analysis keeping track of only the cardinal number of values

b) The cardinal abstraction: Listing 3 annotates the program in Listing 2 with the results of the cardinal abstraction. We denote by pp_k the program point of the command at Line k. Unlike the previous analysis in Listing 2, the cardinal abstraction retains some precision for variables that are not modified inside conditional branches. As a result, the cardinal abstraction is able to determine that, at Line 6, variable `input` does indeed have only 1 possible value, whereas variable `x` has 2 possible values.

```
0 // {secret ↦ ({pp0}, 2^κ), input ↦ ({pp0}, 1),
      x ↦ ({pp0}, 1)}
1 if (secret > input){
2   x := 0;// {secret ↦ ({pp0}, 2^κ),
        input ↦ ({pp0}, 1), x ↦ ({pp2}, 1)}
3 }
4 else {
5   x := 1;// {secret ↦ ({pp0}, 2^κ),
        input ↦ ({pp0}, 1), x ↦ ({pp5}, 1)}
6 } // {secret ↦ ({pp0}, 2^κ), input ↦ ({pp0}, 1),
      x ↦ ({pp2, pp5}, 2)}
```

Listing 3. The results of the cardinal abstraction on the program in Listing 2

III. Abstract Semantics

This section formalizes the cardinal abstraction. We consider a deterministic `While` language [33] introduced in Figure 1. Expressions include integers n of finite size κ, variables `id`, binary arithmetic operations (bop) as well as comparison operations (cmp). Commands are instructions identified by a unique program point $pp \in \mathbb{P}$.

Exp:	$a ::= n$	(constants)
	$\mid id$	(variables)
	$\mid a_1\ bop\ a_2$	(binary operators)
	$\mid a_1\ cmp\ a_2$	(comparison operators)
Com:	$c ::= {}^{pp}skip$	(empty instruction)
	$\mid {}^{pp}id := a$	(assignment)
	$\mid c_1;\ c_2$	(sequence)
	$\mid {}^{pp}if\ (a)\ c_1\ else\ c_2$	(conditional)
	$\mid {}^{pp}while\ (a)\ c$	(loop)

Fig. 1. Abstract syntax of `While`

A. Abstract Domain

The cardinal abstract domain computes an over-approximation of variety, namely the cardinal number of values variables may take when attackers provide an arbitrary low input memory.

We assume that the set of variables of analysed programs are partitioned into low and high inputs. All variables may cover the range $[-2^{\kappa-1}, 2^{\kappa-1} - 1]$ of integers. Therefore, initially at program point pp_0, low input variables have only 1 possible value, whereas high input variables have 2^κ possible values as illustrated by Listing 3 for the program point pp_0.

Definition 1 introduces the cardinal abstract domain. Abstract values are pairs of a set $s_{pp} \subseteq \mathbb{P}$ of program points and a cardinal number $n \in [0, 2^\kappa]$. The resulting lattice is actually the cartesian product of 2 lattices:

1) the set $\mathcal{P}(\mathbb{P})$ of all subsets of $\mathbb{P}$, ordered via set inclusion $\subseteq$, with set union $\cup$ as a join operator and set intersection $\cap$ as a meet operator, as well as
2) the set of natural numbers $[0, 2^\kappa]$, ordered via the natural order $\leq$ over integers, with $\max$ as a join operator and $\min$ as a meet operator.

Definition 1 (Cardinal abstract domain)**.**
The cardinal abstract domain is defined as the lattice

$$\left(D_C^\sharp, \subseteq_\otimes, (\varnothing, 0), (\mathbb{P}, 2^\kappa), \cup_\otimes, \cap_\otimes\right)$$

where:

$$D_C^\sharp \triangleq \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa] \qquad \subseteq_\otimes \triangleq \subseteq \times \leq$$
$$\cup_\otimes \triangleq \cup \times \max \qquad \cap_\otimes \triangleq \cap \times \min$$

Our analysis maps each variable `id` to a pair of a set s_{pp} of program points and a cardinal number n. The set s_{pp} represents

the program points where variable id may have been last assigned, whereas n represents the cardinal number of values variable id may take.

B. Abstract Semantics of Expressions

We denote by $\varrho^\sharp \in Var \rightharpoonup D_C^\sharp$ an abstract environment that maps each variable id to an abstract value $(s_{pp}, n) \in D_C^\sharp$. We also denote by $proj_i()$ the projection onto the ith component of a pair ($i = 1, 2$). Figure 2 introduces the abstract semantics of expressions $\mathbb{A}^\sharp \in Exp \to (Var \rightharpoonup D_C^\sharp) \to [0, 2^\kappa]$.

$$\mathbb{A}^\sharp[\![n]\!]\varrho^\sharp = 1 \qquad \mathbb{A}^\sharp[\![id]\!]\varrho^\sharp = proj_2(\varrho^\sharp(id))$$

$$\mathbb{A}^\sharp[\![a_1 \ bop \ a_2]\!]\varrho^\sharp = \min\left(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp, 2^\kappa\right)$$

$$\mathbb{A}^\sharp[\![a_1 \ mod \ n]\!]\varrho^\sharp = \min\left(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp, n\right)$$

$$\mathbb{A}^\sharp[\![a_1 \ cmp \ a_2]\!]\varrho^\sharp = \min\left(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp, 2\right)$$

Fig. 2. Abstract semantics of expressions

The abstract semantics of expressions $\mathbb{A}^\sharp$ evaluates an expression a in an abstract environment $\varrho^\sharp$, in order to yield a cardinal $n \in [0, 2^\kappa]$ representing the cardinal number of values expression a may evaluate to. Constant expressions always evaluate to a single value. Binary arithmetic operations $\mathtt{a}_1$ bop $\mathtt{a}_2$ yield at most the product of the cardinal of $\mathtt{a}_1$ and the cardinal of $\mathtt{a}_2$, or 2^κ when this product overflows. Comparison operations $\mathtt{a}_1$ cmp $\mathtt{a}_2$ evaluate to at most 2 different values (true or false), and only 1 when both $\mathtt{a}_1$ and $\mathtt{a}_2$ evaluate to only one possible value.

In the case where the second argument of the modulo binary operator is a literal, the cardinal abstraction tries to be more precise. Indeed, the modulo operation $\mathtt{a}_1$ mod n produces at most n different values or $\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp$ different values when $\mathtt{a}_1$ may take less than n values.

One advantage of formalizing our analysis as an abstract interpretation approach is the possibility of refining the precision of our analysis by relying on extent abstract domains [34], [35], [36], [37], [38], [39]. For instance, if a numerical abstraction determines that an expression $\mathtt{a}_2$ has a maximum value of 10 for instance, then the cardinal abstraction can soundly conclude that expression $\mathtt{a}_1$ mod $\mathtt{a}_2$ may evaluate to at most 10 different values or $\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp$ different values when $\mathtt{a}_1$ may take less than 10 values. The process of combining different abstract domains in order to refine the precision of an abstraction is called a reduced product [40]. In general, relying on reduced products using traditional abstract domains can refine most of the cardinal abstract domain definitions.

C. Abstract Semantics of Commands

Figure 3 introduces the forward abstract semantics of commands. This abstract semantics $[\![c]\!]^\sharp \in (Var \rightharpoonup D_C^\sharp) \to (Var \rightharpoonup D_C^\sharp)$ evaluates a command c in an abstract environment $\varrho^\sharp$, then yields a new abstract environment.

Commands ^{pp}skip do not modify input abstract environments. Assignments ^{pp}id := a map the abstract value $(\{pp\}, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)$ to variable id since id can take as many values as expression a, and is assigned at program point pp. A sequence of commands $\mathtt{c}_1$; $\mathtt{c}_2$ composes the abstract semantics of the second command $\mathtt{c}_2$ with the abstract semantics of the first one $\mathtt{c}_1$.

Recall that the cardinal abstract domain computes the cardinal number of values variables may take, when attackers provide an arbitrary low input memory. Recall also that we assume a partition of the set of variables into low and high input variables, so that all variables are initialized at the initial program point pp_0. Therefore, initially at the entry program point pp_0, low input variables have only 1 possible value. On the opposite, high input variables range over the interval $[-2^{\kappa-1}, 2^{\kappa-1} - 1]$ of integers. Therefore, high input variables have 2^κ possible values at the entry program point pp_0.

When the cardinal abstraction determines that a variable id has at most 1 possible value, it also implicitly determines that variable id may only depend on low input variables. Indeed, if variable id depended on high input memories, then there would exist two low equivalent input memories that yield two different values for variable id. Therefore, the cardinal abstraction would determine that variable id has a cardinal number of values greater or equal to 2, since it over-approximates the cardinal number of values variables may take, for a fixed arbitrary low input. This remark will shortly prove useful when defining the abstract semantics of conditionals.

a) Conditional instructions: The abstract semantics of conditional commands considers two different cases depending on the abstract value of the conditional guard.

First, if the conditional guard has only one possible value, then it may only depend on low inputs. Hence, the concrete evaluation of the conditional always executes the same conditional branch for each fixed low input. Therefore, the join operator $\cup_\otimes$ lifted over environments soundly over-approximates the semantics of conditionals, by computing the set union over the set of program points where each variable may be last assigned, as well as the maximum cardinal over both branches for each variable. Indeed, if a variable id can take at most n_1 values (resp. n_2 values) in the 'then' branch (resp. the 'else' branch), and assuming low inputs are fixed, the cardinal abstract domain soundly concludes that variable id can take at most $\max(n_1, n_2)$ values after the conditional.

Second, if the conditional guard has more than one value, then it may depend on high inputs. Hence, the concrete evaluation of the conditional may execute both conditional branches for each fixed low input. Assuming that variable id can have at most n_1 values (resp. n_2 values) in the 'then' branch (resp. the 'else' branch), the cardinal abstract domain may soundly conclude that variable id can take at most $n_1 + n_2$ values after the conditional, or 2^k values if the latter sum overflows. However, the abstract semantics uses the operator $\cup_{add(c_1,c_2)}$ to retain precision for the variables V that are assigned neither in $\mathtt{c}_1$ nor in $\mathtt{c}_2$, by computing the join $\cup_\otimes$ over their abstract values. Therefore, for variables V, the cardinal abstract domain simply computes the set union over sets of program points, where these variables may be last assigned, and the maximum over both their cardinals.

In order to determine which variables may be modified

$$[\![^{pp}skip]\!]^\sharp\varrho^\sharp \triangleq \varrho^\sharp \qquad [\![^{pp}id := a]\!]^\sharp\varrho^\sharp \triangleq \varrho^\sharp[id \mapsto (\{pp\}, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)] \qquad [\![c_1;\ c_2]\!]^\sharp\varrho^\sharp \triangleq [\![c_2]\!]^\sharp([\![c_1]\!]^\sharp\varrho^\sharp)$$

$$[\![^{pp}if\ (a)\ c_1\ else\ c_2]\!]^\sharp\varrho^\sharp \triangleq \text{let } n = \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \text{ in let } \varrho_1^\sharp = [\![c_1]\!]^\sharp\varrho^\sharp \text{ in let } \varrho_2^\sharp = [\![c_2]\!]^\sharp\varrho^\sharp \text{ in } \lambda id. \begin{cases} \varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id) & \text{if } n = 1 \\ \varrho_1^\sharp(id) \cup_{add(c_1,c_2)} \varrho_2^\sharp(id) & \text{otherwise} \end{cases}$$

$$[\![^{pp}while\ (a)\ c]\!]^\sharp\varrho^\sharp \triangleq \text{let } F^\sharp \triangleq \lambda\varrho^\sharp.\varrho_0^\sharp \dot\cup_\otimes \left([\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp\varrho^\sharp\right) \text{ in } \mathrm{lfp}^{\dot\subseteq_\otimes} F^\sharp$$

$$(s_{pp}, n) \cup_{add(c_1,c_2)} (s'_{pp}, n') \triangleq \begin{cases} (s_{pp}, n) \cup_\otimes (s'_{pp}, n') & \text{if } PP(c_1; c_2) \cap (s_{pp} \cup s'_{pp}) = \varnothing \\ \left(s_{pp} \cup s'_{pp}, min(n + n', 2^\kappa)\right) & \text{otherwise} \end{cases}$$

$$PP(^{pp}skip) \triangleq \{pp\} \qquad PP(^{pp}id := a) \triangleq \{pp\} \qquad PP(c_1;\ c_2) \triangleq PP(c_1) \cup PP(c_2)$$

$$PP(^{pp}while\ (a)\ c) \triangleq \{pp\} \cup PP(c) \qquad PP(^{pp}if\ (a)\ c_1\ else\ c_2) \triangleq \{pp\} \cup PP(c_1) \cup PP(c_2)$$

Fig. 3. Abstract semantics of commands

within an instruction `c`, the cardinal abstract domain relies on the set of program points where variables may be last assigned as well as the operator $PP(c)$, that we define as the set of program points appearing in command `c`. Hence, instruction `c` does not modify variable `id` if $PP(c) \cap s_{pp} = \varnothing$, supposing s_{pp} over-approximates the set of program points where `id` may be last assigned. Note that this condition is only a sufficient one since s_{pp} is an over-approximation of the set of program points where variable `id` may be last assigned.

In general, abstract domains retain precision in conditional branches by relying on the conditional guard to reduce the abstract environments entering both the 'then' branch and the 'else' branch. For instance, let us assume that an interval analysis determines that variable `x` ranges over the interval $[0, 9]$, whereas variable `y` ranges over the interval $[2, 5]$. Then, in the case of a conditional guard (`x == y`) testing the equality of variables `x` and `y`, the interval analysis can soundly reduce both abstract values mapped to `x` and `y` in the 'then' branch to the intersection $[2, 5]$ of both intervals. For simplicity of presentation, the cardinal abstract semantics of conditionals presented in Figure 3 does not attempt to leverage on such reductions. We leave this improvement as future work.

b) Loop instructions: Let us denote by $\dot\subseteq_\otimes$ (resp. by $\dot\cup_\otimes$) the pointwise lifting of the partial order relation $\subseteq_\otimes$ (resp. of the join operator $\cup_\otimes$) over environments:

$$\text{Partial order:} \quad \varrho_1^\sharp \dot\subseteq_\otimes \varrho_2^\sharp \iff \forall id, \varrho_1^\sharp(id) \subseteq_\otimes \varrho_2^\sharp(id)$$
$$\text{Join operator:} \quad \varrho_1^\sharp \dot\cup_\otimes \varrho_2^\sharp = \lambda id.\varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id).$$

The abstract semantics of loops requires computing an abstract environment $\varrho^\sharp$ that is a loop invariant $\dot\subseteq_\otimes$-greater than the initial abstract environment $\varrho_0^\sharp$. This loop invariant $\varrho^\sharp$ satisfies the following fixpoint equation:

$$\varrho^\sharp = \varrho_0^\sharp \dot\cup_\otimes \left([\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp\varrho^\sharp\right).$$

Therefore, we can define the abstract semantics of loops as the least fixpoint of $F^\sharp$, where $F^\sharp$ is defined by:

$$F^\sharp \triangleq \lambda\varrho^\sharp.\varrho_0^\sharp \dot\cup_\otimes \left([\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp\varrho^\sharp\right).$$

The least fixpoint of $F^\sharp$ exists, since $F^\sharp$ is monotonic and the lattice of abstract environments is complete. Additionally, this least fixpoint can be computed iteratively [41], [42] by defining a sequence $(x_n)_{n\geq 0}$ as follows:

$$x_0 = \varrho_0^\sharp$$
$$x_{n+1} = \varrho_0^\sharp \dot\cup_\otimes \left([\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp x_n\right).$$

The least fixpoint of $F^\sharp$ is therefore equal to the limit x_∞ of the sequence $(x_n)_{n\geq 0}$.

D. Over-approximating the Maximum Leakage

The cardinal abstract domain computes an over-approximation of the number of values variables may take, when attackers provide a low input memory. Hence, we can upper-bound the cardinal number of outputs attackers may observe when providing an arbitrary low input memory, in order to obtain an over-approximation of the maximum leakage.

Equation (2) characterizes the observations attackers make for batch-job programs `c` after providing a low input memory as follows :

$$\begin{aligned} outputs_{\varrho_{L_0}} = \{(\varrho(v_1), \varrho(v_2) \ldots, \varrho(v_f)) \mid \\ \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } [\![c]\!]\varrho_0 = \varrho\}. \end{aligned} \tag{2}$$

Recall that we denote by $\mathtt{v_1}, \mathtt{v_2}, \ldots \mathtt{v_f}$ the variables that attackers may observe at the end of the program's execution. Notice also that the partition of variables into low and high outputs may be completely different from the partition of variables into low and high inputs.

Let us denote by R the set of sets of final reachable states for each fixed low input memory, for a batch-job program $\texttt{c}$:

$$R \triangleq \big\{\{\varrho \mid \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } [\![c]\!]\varrho_0 = \varrho\} \mid \varrho_{L_0} \in \Sigma_{L_0}\big\}.$$

Then, we can prove that the maximum cardinal number of outputs attackers may observe, when providing a low input memory, can be over-approximated by relying on the cardinal abstraction. Indeed, assuming that $\varrho^\sharp_\Downarrow$ is the final abstract environment computed for the program $\texttt{c}$, the cardinal number of outputs attackers may observe is given by the product of cardinals computed by $\varrho^\sharp_\Downarrow$, over all low observable variables $\texttt{v}_1, \texttt{v}_2, \ldots \texttt{v}_f$:

$$\begin{aligned}
2^{\mathcal{ML}} &= \max_{r \in R} |\{(\varrho(v_1), \varrho(v_2) \ldots, \varrho(v_f)) \mid \varrho \in r\}| \\
&\leq \text{(by loosing relations in each set } r) \\
&\quad \max_{r \in R} \prod_{1 \leq i \leq f} |\{\varrho(v_i) \mid \varrho \in r\}| \\
&\leq \text{(by loosing relations in } R) \\
&\quad \prod_{1 \leq i \leq f} \max_{r \in R} |\{\varrho(v_i) \mid \varrho \in r\}| \\
&\leq \text{(by soundness of the cardinal abstraction)} \\
&\quad \prod_{1 \leq i \leq f} proj_2(\varrho^\sharp_\Downarrow(v_i)) \quad \square
\end{aligned}$$

The latter proof yields Theorem 1 which provides an over-approximation of the maximum leakage for batch-job programs by relying on the cardinal abstract domain.

Theorem 1 (Over-approximation of $\mathcal{ML}$ for batch-job programs)**.**
For deterministic batch-job programs, the maximum leakage $\mathcal{ML}$ is upper-bounded by:

$$\mathcal{ML} \leq \log_2 \left(\prod_{1 \leq i \leq f} proj_2(\varrho^\sharp_\Downarrow(v_i)) \right)$$

where $\varrho^\sharp_\Downarrow$ denotes the abstract environment computed at the end of the program, and $\texttt{v}_1, \texttt{v}_2, \ldots \texttt{v}_f$ denote the low output variables attackers are allowed to observe.

Note that the proof of Theorem 1 relies on the assumption that the cardinal abstract domain is sound. This same proof sets the stage for the proof of soundness of the cardinal abstraction:

1) define a semantics over a set R of sets of reachable states for each fixed low input memory, then
2) define an abstraction that ignores relations between variables in each set $r \in R$
3) define an abstraction computing the cardinal number of values.

Therefore, the next section will sketch a soundness proof of the cardinal abstract domain.

IV. Soundness

Abstract interpretation generally requires defining a standard semantics for the considered language, describing the result of executing a command over a concrete environment. Then, abstract interpretation focuses on defining a collecting semantics [28] describing the details that are relevant to the properties of interest. These properties are in general not computable. Therefore, abstract interpretation frameworks introduce abstract representations aimed at approximating them. Finally, by linking the properties of interest to the abstract representations through a Galois connection, a sound abstract semantics can be systematically [43] derived in order to tractably compute approximations of the properties of interest.

The denotational semantics of the `While` language of Figure 1 is fairly standard. Thus, we omit it for concision. We make explicit only the denotational semantics of assignments. Indeed, since the cardinal abstract domain tracks the set of program points where variables may have been last assigned, we instrument the denotational semantics for assignments in order to track these program points as well:

$$[\![^{pp} id := a]\!]\varrho = \varrho[id \mapsto (pp, \mathbb{A}[\![a]\!]\varrho)]$$

Therefore, concrete environments $\varrho \in Var \rightharpoonup \mathbb{P} \times \mathbb{V}$ map a variable `id` to a pair of a program point $pp \in \mathbb{P}$ and an integer value $v \in \mathbb{V}$ of size κ. The denotation $[\![c]\!]\varrho$ yields an environment ϱ'. Additionally, we also denote by $\mathbb{A}[\![a]\!]\varrho$ the evaluation of an expression `a` in an environment ϱ, that yields a value $v \in \mathbb{V}$. Notice that the instrumentation of assignments to track the program points where variables are last assigned does not modify the semantics over values. An erasure of the program points yields the standard denotational semantics of a `While` language [33].

A. Collecting Semantics

The choice of a collecting semantics depends on the problem of interest. Indeed, a collecting semantics must at least describe program behaviours that are relevant. Ideally, the collecting semantics should also abstract away from the details that are not relevant to the studied problem. Therefore, how should we define the collecting semantics to prove the soundness of the cardinal abstraction?

To answer this question intuitively, let us assume the simpler case of an abstract memory $m^\sharp \triangleq \{x \mapsto 2\}$ determining that a variable `x` may take at most 2 different values. What would be the concrete memories $m \in Var \rightharpoonup \mathbb{V}$ that are represented by $m^\sharp$? Since variable `x` may take at most 2 different values, the possible values `x` may take are given by a set of sets of values $v \in \mathbb{V}$:

$$\{V \in \mathcal{P}(\mathbb{V}) : |V| \leq 2\} \in \mathcal{P}(\mathcal{P}(\mathbb{V})).$$

Therefore, the concrete memories represented by $m^\sharp$ are also given by a set $R \in \mathcal{P}(\mathcal{P}(Var \rightharpoonup \mathbb{V}))$ of sets of memories $m \in Var \rightharpoonup \mathbb{V}$:

$$R = \big\{r \in \mathcal{P}(Var \rightharpoonup \mathbb{V}) : |\{m(x) : m \in r\}| \leq 2\big\}.$$

Intuitively, in each set r of memories, the memories $m \in r$ map a variable `x` to at most 2 different values. Thus, we are going to define the collecting semantics over a set of sets of environments. Interestingly, if we consider a program $\texttt{c}$ that accepts both low and high inputs, such a collecting semantics

also enables us to describe the set of sets of final reachable states, one set of final reachable states for each low input memory, in accordance with Equations (1) and (2).

Such structures implying an additional level of sets over environments naturally arise when dealing with security policies such as information flow policies. Indeed, Clarkson and Schneider [29], [30] note that important classes of security policies are best described by *hyperproperties* which they define as sets of legal sets of traces – legal wrt. a security policy –, in contrast to *properties* that are defined as sets of legal traces.

a) Forward collecting semantics of commands: Equation (3) defines the forward collecting semantics for commands. This collecting semantics computes the set R' of sets of final reachable environments when a command $\subset$ is executed over a set R of sets of initial environments.

$$\begin{aligned}&[\![c]\!]_c \in \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right)\\ &[\![c]\!]_c R \triangleq \{\{[\![c]\!]\varrho \mid \varrho \in r\} \mid r \in R\}.\end{aligned} \tag{3}$$

Notice that our collecting semantics $[\![c]\!]_c$ is defined over an abstraction of hyperproperties. Indeed, Clarkson and Schneider [29], [30] define a hyperproperty as a set of sets of traces, where each trace is a sequence of states. They also define a relational hyperproperty as a set of sets of relational traces, where each relational trace is a pair of an initial state and a final state. Relational hyperproperties are an abstraction of the more general hyperproperties, since they forget about the intermediate states and keep only the initial and final states. Similarly, our set of sets of reachable states abstracts relational hyperproperties, by forgetting about the initial states and keeping only the final states. One can also compose an element-wise abstraction [44] with the reachability abstraction [45] in order to directly abstract a hyperproperty into a set of sets of reachable states.

Cousot and Cousot [31, Section 11] stress down that the notion of a collecting semantics "is relative to a set of questions. It defines exactly which questions can be answered about programs". They also define a collecting semantics over a set of sets of states, similar to the one we define in Equation (3), while observing that this collecting semantics is more general than the basic collecting semantics defined over a set of states. Such a collecting semantics with an additional level of sets also implicitly appears in Amtoft and Banerjee's [32, Section 3] work, since the Galois connection they introduce relates a set of sets of relational traces to their abstract domain computing independences between variables.

b) Forward collecting semantics of expressions: Similarly to Equation (3), Equation (4) defines the forward collecting semantics for expressions over a set of sets of environments. This semantics then yields a set of sets of values $v \in \mathbb{V}$.

$$\begin{aligned}&\mathbb{A}_c[\![a]\!] \in \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right)\\ &\mathbb{A}_c[\![a]\!]R \triangleq \{\{\mathbb{A}[\![a]\!]\varrho \mid \varrho \in r\} \mid r \in R\}.\end{aligned} \tag{4}$$

B. Sketch of the Soundness Proof

Once we define a collecting semantics for commands, we need to construct a Galois connection that relates the lattice of concrete objects $R \in \mathcal{P}\left(\mathcal{P}\left(Var \mapsto \mathbb{P} \times \mathbb{V}\right)\right)$ to the lattice of abstract ones $\varrho^\sharp \in Var \mapsto D_C^\sharp$. With such a Galois connection, we can derive [43] a sound abstract semantics $[\![c]\!]^\sharp$ for commands as well as a sound abstract semantics for expressions $\mathbb{A}^\sharp[\![a]\!]$. For the sake of pedagogical presentation, let us for now postulate the existence of such a Galois connection $(\dot\alpha, \dot\gamma)$:

$$\begin{aligned}&\left\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \right\rangle\\ &\qquad \underset{\dot\alpha}{\overset{\dot\gamma}{\leftrightarrows}}\\ &\left\langle Var \rightharpoonup \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^\kappa]; \dot\subseteq_\otimes, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^\kappa), \dot\cup_\otimes, \dot\cap_\otimes \right\rangle\end{aligned} \tag{5}$$

Then, in order to derive a sound abstract semantics for commands, we can consider the functional abstraction $\alpha^\triangleright_{com}$ defined in Equation (6), in order to transpose functions $[\![c]\!]_c$ of the concrete collecting semantics to functions $[\![c]\!]^\sharp$ of the abstract semantics. Note that Equation (6) denotes by Env the set $Var \rightharpoonup \mathbb{P} \times \mathbb{V}$ of concrete environments, and by $Env^\sharp$ the set $Var \rightharpoonup D_C^\sharp$ of abstract environments.

$$\begin{aligned}&\alpha^\triangleright_{com} \in \left(\mathcal{P}\left(\mathcal{P}\left(Env\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(Env\right)\right)\right) \to \left(Env^\sharp \to Env^\sharp\right)\\ &\alpha^\triangleright_{com}([\![c]\!]_c) \triangleq \dot\alpha \circ [\![c]\!]_c \circ \dot\gamma\end{aligned} \tag{6}$$

Finally, note that $\alpha^\triangleright_{com}([\![c]\!]_c)$ provides the best abstraction of the collecting semantics of commands. However, this abstraction might not be computable in general. Therefore, soundness only requires that the abstract semantics $[\![c]\!]^\sharp$ of commands over-approximates the functional abstraction $\alpha^\triangleright_{com}([\![c]\!]_c)$ of the collecting semantics:

$$\alpha^\triangleright_{com}([\![c]\!]_c)\varrho^\sharp \mathrel{\dot\subseteq_\otimes} [\![c]\!]^\sharp\varrho^\sharp.$$

Similarly, we also construct a functional abstraction $\alpha^\triangleright_{exp}$ in order to derive a sound abstract semantics for expressions. The functional abstraction $\alpha^\triangleright_{exp}$ transposes functions $\mathbb{A}_c[\![a]\!]$ of the concrete collecting semantics of expressions to functions $\mathbb{A}^\sharp[\![a]\!]$ of the abstract semantics of expressions:

$$\alpha^\triangleright_{exp} \in \left(\mathcal{P}\left(\mathcal{P}\left(Env\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right)\right) \to \left(Env^\sharp \to [0, 2^\kappa]\right).$$

Additionally, the abstract semantics of expressions $\mathbb{A}^\sharp[\![a]\!]$ is sound if it verifies the following condition:

$$\alpha^\triangleright_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp.$$

Figure 4 summarizes the steps towards building the Galois connection $(\dot\alpha, \dot\gamma)$ assumed in Equation (5). We mainly rely on the fact that composing Galois connections yields a new Galois connection. Thus, composing a non-relational abstraction (α_r, γ_r) with the pointwise abstraction (α_c, γ_c) of values yields the desired Galois connection $(\dot\alpha, \dot\gamma)$. The complete details can be found in the appendix.

Theorems 2 and 3 state the soundness of the cardinal abstract domain.

Theorem 2 (Soundness of the abstract semantics $\mathbb{A}^\sharp[\![a]\!]$)**.** *The abstract semantics of expressions in Figure 2 is sound:*

$$\alpha^\triangleright_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp.$$

Theorem 3 (Soundness of the abstract semantics $[\![c]\!]^\sharp$)**.** *The abstract semantics of commands in Figure 3 is sound:*

$$\alpha^\triangleright_{com}([\![c]\!]_c)\varrho^\sharp \mathrel{\dot\subseteq_\otimes} [\![c]\!]^\sharp\varrho^\sharp.$$

Both proofs can be found in the appendix.

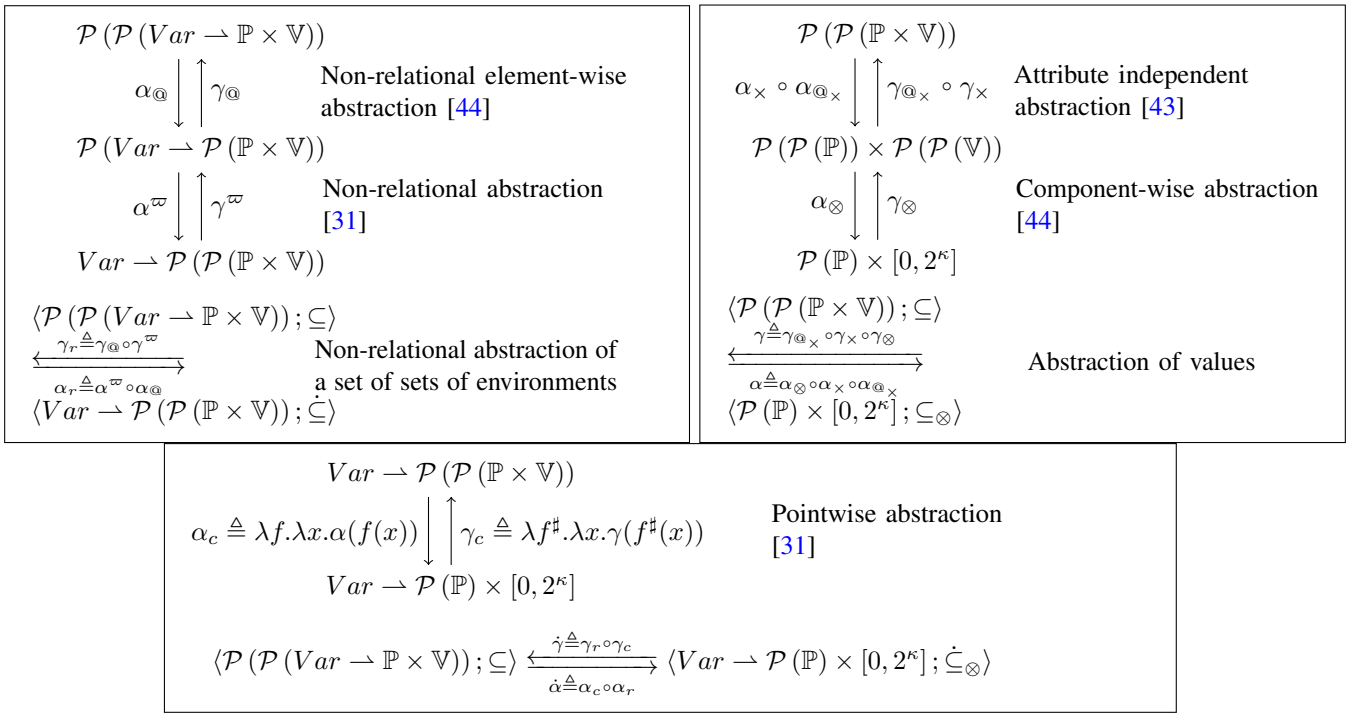

Fig. 4. Building the Galois connection $(\dot{\alpha}, \dot{\gamma})$

V. Precision

Throughout our developments of the cardinal abstraction, we consider some optimizations in order to enhance the precision of our analysis, while we leave out some others for future work. For instance, the cardinal abstract domain tracks the program points where variables may have been last assigned, in order to improve its precision in the case of branching instructions. However, we leave off optimizations such as reducing abstract environments by relying on conditional guards, or combining the cardinal abstraction with numerical abstract domains [46], [28]. These choices are motivated by making the precision of the cardinal abstraction reach a first checkpoint; indeed, the cardinal abstraction is at least as precise as a flow-sensitive type system [4], [5] for the two point lattice $\langle \{L, H\}; \sqsubseteq, L, H, \sqcup, \sqcap \rangle$.

Intuitively, abstracting further the results of cardinal abstract domain, by mapping variables having a cardinal number of at most 1 to a low security level L and mapping variables having a cardinal number strictly greater than 1 to a high security level H, yields an analysis that is at least as precise as Hunt and Sands' type system.

Let us first introduce an abstraction operator $@_{LH}$ mapping a cardinal number n to a security label L or H:

$$@_{LH} \in [0, 2^{\kappa}] \mapsto \{L, H\}$$
$$@_{LH}(n) \triangleq \begin{cases} L \text{ if } n \leq 1 \\ H \text{ otherwise} \end{cases}$$

This abstraction can be lifted to abstract environments through a pointwise abstraction [31] as well as a projection that forgets about the program points:

$$\alpha_{@_{LH}} \in (Var \rightharpoonup D_C^{\sharp}) \mapsto (Var \rightharpoonup \{L, H\})$$
$$\alpha_{@_{LH}}(\varrho^{\sharp}) \triangleq \lambda x.@_{LH}\left(proj_2(\varrho^{\sharp}(x))\right)$$

Lemma 1 proves that the abstract semantics of expressions is as precise as the type labelling of expressions. A proof of Lemma 1, by structural induction on expressions, is presented in the appendix.

Lemma 1 (The abstract semantics of expressions is at least as precise as the type labelling of expressions)**.**
For all expressions a*, for all abstract environments* $\varrho^{\sharp} \in Var \rightharpoonup D_C^{\sharp}$*, for all type environments* $\Gamma \in Var \rightharpoonup \{L, H\}$*, such that:* $\mathbb{A}^{\sharp}[\![a]\!]\varrho^{\sharp} = n$ *and* $\Gamma \vdash a : t$*.*
It holds that:

$$\alpha_{@_{LH}}(\varrho^{\sharp}) \sqsubseteq \Gamma \implies @_{LH}(n) \sqsubseteq t$$

Theorem 4 proves that the abstract semantics of commands is at least as precise as the type labelling of commands. We let the partial order $\dot{\sqsubseteq}$ denote the pointwise lifting of the partial order over security labels to type environments. A proof of Theorem 4 is presented in the appendix. This proof is by structural induction on commands and relies on Lemma 1.

Theorem 4 (The abstract semantics of commands is at least as precise as the type labelling of commands)**.**
For all commands c*, for all abstract environments* $\varrho_0^{\sharp}, \varrho^{\sharp}$*, and type environments* Γ_0, Γ *such that:* $[\![c]\!]^{\sharp}\varrho_0^{\sharp} = \varrho^{\sharp}$ *and* $L \vdash \Gamma_0\{c\}\Gamma$*.*
It holds that:

$$\alpha_{@_{LH}}(\varrho_0^{\sharp}) \sqsubseteq \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^{\sharp}) \dot{\sqsubseteq} \Gamma$$

The cardinal abstraction is almost exactly as precise as Hunt and Sands flow-sensitive type system. The only case that stands out is the following expression `secret mod 1` that we uncovered while trying to prove that the cardinal abstraction is exactly as precise as Hunt and Sands' flow-sensitive type system. Indeed, assuming that variable `secret` may take more than 2 values, the cardinal abstraction still determines that this expression has at most 1 possible value, whereas the typing judgement for expressions reasons on the free variables of that expression, and labels the expression `secret mod 1` as high H. Another corner case is the expression `secret mod 0` where the cardinal abstraction determines as having 0 value, which is still sound considering that this expression is undefined. This also warrants one of our design decisions to formalize our analysis as an abstract domain, since we can rely on reduced products [40] to not only enhance the precision of the cardinal abstract domain, but also to prove the absence of runtime errors in analysed programs.

As a direct result of Theorem 4, Corollary 1 states that the cardinal abstraction computes a leakage of zero bits for programs that are "well-typed" by Hunt and Sands' flow-sensitive type system. By "well-typed", we mean that the final typing environment computed by the flow-sensitive type system labels as low all output variables $\mathtt{v}_1, \mathtt{v}_2, \ldots \mathtt{v}_f$ that attackers may observe.

Corollary 1 (Zero leakage for well-typed programs)**.** *For all commands c, for all abstract environments $\varrho_0^\sharp, \varrho^\sharp$, and type environments Γ_0, Γ such that: $[\![c]\!]^\sharp \varrho_0^\sharp = \varrho^\sharp$, and $L \vdash \Gamma_0\{c\}\Gamma$, and for all $i \in [1, f]$, $proj_2(\varrho^\sharp(v_i)) \neq 0$.*

It holds that:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) \sqsubseteq \Gamma_0 \text{ and } (\forall i \in [1, f], \Gamma(v_i) = L) \implies \log_2 \left(\prod_{1 \leq i \leq f} proj_2(\varrho^\sharp(v_i)) \right) = 0.$$

Since the cardinal abstraction is at least as precise as a flow-sensitive type system for the two point lattice, it is appealing to rely on this abstract domain even when interested in proving that a program complies with a qualitative security policy such as termination-insensitive non-interference [2]. Indeed, in addition to labelling variables as low or high, the cardinal abstraction also provides quantitative information that may assist in making better informed decisions when declassification is necessary.

Unlike Hunt and Sands' [4], [5] type system, the cardinal abstraction quantifies information flow only for the two-point lattice. Future work includes a generalization of the cardinal abstract domain to handle the case of arbitrary multilevel security lattices.

VI. Related Work

Clark, Hunt and Malacaria [47] build on Denning's [11] work in order to propose a measure of information leakage, based on Shannon entropy [12] and mutual information. In particular, they prove in the deterministic setting that the leakage of a program can be measured by the conditional Shannon entropy of the observable outputs, knowing the public inputs. They also prove that this latter quantity equals zero in the case of a deterministic program iff. the program is non-interferent. Clark et al. [48] also propose a type system to quantify information flow for deterministic batch-job programs with low inputs. They also prove that if attackers can observe non-termination, their analysis under-estimates the leakage by at most one bit. The same argument can be used to prove that the cardinal abstraction also under-estimates the leakage by at most one bit if attackers can observe non-termination. Indeed, the observation of non-termination by attackers can be modelled as an additional observable output $\perp$, which means that the upper-bound computed by the cardinal abstraction is under-estimated by at most 1 bit.

Clarkson et al. [14], [15] propose a quantitative measure, based on information belief [16]. This measure reasons on attackers' belief, namely the probability distribution of the confidential input that attackers may assume. Clarkson et al. model how attackers revise their belief after observing outputs of a program. The revised attackers' belief as well as the initial belief provide a way to measure the improvement in the accuracy of attackers' belief. Clarkson et al. also propose to measure this improvement in accuracy by relying on relative entropy [49], a pseudo-metric defined over probability distributions.

Smith [17] notes that Shannon entropy and mutual information are unsuitable metrics for estimating information flow leakage in particular scenarios. Indeed, Shannon entropy of a random variable can be arbitrarily high despite being highly vulnerable to being guessed in one try [50]. Smith then proposes the use of min-entropy as a measure for quantifying information flow. This measure estimates the probability that attackers guess the confidential inputs in one try, after observing a program run. Additionally, Smith proves that the maximum leakage over all a priori distributions measured either with min-entropy (namely, min-capacity) or Shannon entropy (namely, capacity) coincide for deterministic programs, and conjecture that min-capacity upper-bounds capacity for probabilistic programs [18].

Braun, Chatzikokolakis and Palamidessi [51] propose an additive notion of information leakage based on the probability that attackers guess the wrong confidential inputs. They also investigate how to compute supremums for their additive notion as well as for min-entropy [17]. Particularly, they prove that the supremum for min-entropy is reached when the confidential inputs are uniformly distributed [51], [18].

Alvim et al. [52] propose a generalization of min-entropy using gain functions. Gain functions model a variety of scenarios for attackers, such as the advantage attackers gain from guessing part of the secret or making a set of guesses. They also prove in the probabilistic setting that min-capacity is an upper bound on both Shannon capacity and capacity defined using gain functions. Hence, min-capacity offers a way to abstract from a priori distributions of the confidential inputs and to bound various entropy-based information flow metrics. Note however that different quantitative information flow metrics are appropriate for modelling different attacks scenarios [17].

Backes, Köpf and Rybalchenko [21] propose to synthesize

the equivalence classes induced by outputs over low equivalent memories by relying on software model checkers. Their approach characterizes the maximum leakage $\mathcal{ML}$ by counting the number of such equivalence classes. Additionally, they also estimate the size of such equivalence classes in order to compute other quantitative information flow metrics such as Shannon entropy [12] and the Guessing entropy [53]. Heusser and Malacaria [22] also rely on a similar technique to quantify information flow for database queries. Köpf and Rybalchenko [54] note that the exact computation of information-theoretic characteristics is prohibitively hard, and propose instead to rely on approximation-based analyses.

In particular, Köpf and Rybalchenko [54] rely on both abstract interpretation and model checking to compute over-approximations and under-approximations of the equivalence classes over the low equivalent memories. Therefore, they can compute an upper-bound and a lower-bound over the maximum leakage $\mathcal{ML}$. Additionally, they propose a randomization technique based on sampling in order to bound Shannon entropy [12]. Their abstract interpretation approach relies on over-approximating the set of reachable states, which delivers an imprecise upper-bound of the leakage for programs accepting both low and high inputs. Köpf and Rybalchenko also propose to rely on self-composition [6], [7], [8], [20] to model a scenario where attackers may refine their knowledge by influencing the low inputs. Klebanov [23] relies on a similar technique to handle programs with low inputs, and uses two different approaches to quantify information leaks. The first approach computes a functional specification of programs using user-supplied invariants, and the second one relies on self-composition to automatically compute the equivalence classes over low input memories. Klebanov also proposes the use of polyhedra to synthesize linear constraints over variables. He then relies on symbolic procedures to compute both the number and size of the equivalent classes over low input memories. However, this approach does not support programs with arbitrary expressions.

VII. Conclusion

This paper proposes a novel analysis, the cardinal abstraction, for QIF in batch-job programs. Following the abstract interpretation framework [28], we formalize the cardinal abstract domain and prove its soundness.

Unlike previous approaches aimed at QIF, our approach does not rely on an underlying reachability analysis. Instead, the cardinal abstraction approximates variety directly to precisely upper-bound the maximum leakage. We also prove the precision of our analysis by proving that the cardinal abstraction computes a maximum leakage of zero for programs that are "well-typed" wrt. a traditional flow-sensitive type system [4], [5].

We also propose an implementation of the cardinal abstraction [1] by modifying an existing prototype abstract interpreter[2] for an imperative `While` language. This implementation enhances the precision of the cardinal abstraction by relying on a reduced product [40] of both the cardinal abstract domain and an interval abstract domain [34]. Indeed, one advantage of formalizing our analysis as an abstract interpretation approach is the possibility of improving its precision by combining it with existing domains [39], [35], [36], [37], [38]. In particular, we plan on building on the cardinal abstraction and Miné's [38] previous work in order to enhance the cardinal abstraction with relational constraints, while still supporting arbitrary non-linear expressions.

Future work also includes generalizing the cardinal abstraction for QIF in the case of a multilevel security lattice. By doing so, we also hope to gain more insights towards understanding abstract interpretation-based verification methods for hyperproperties. The cardinal abstraction is but a first step towards that goal. Indeed, we are aware that the bounding problem in QIF the cardinal abstraction targets is not a k-safety property, but a hypersafety [29], [30], [55], [56], [57] one.

Additionally, we would also like to extend our approach to deal with real-world programs. In ongoing work, we are extending the restricted batch-job computation model to a more general one [58], and will move from there to target richer programming languages.

[1] https://github.com/moun/Fhamator

[2] http://www.irisa.fr/celtique/teaching/PAS/while_analyser.tgz

References

[1] J. A. Goguen and J. Meseguer, "Security Policies and Security Models." *IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.

[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[3] D. Volpano, C. Irvine, and G. Smith, "A Sound Type System for Secure Flow Analysis," *Journal in Computer Security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[4] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, New York, USA, Jan. 2006, pp. 79–90.

[5] ——, "From Exponential to Polynomial-Time Security Typing via Principal Types." in *ESOP*. City University London, 2011, pp. 297–316.

[6] Á. Darvas, R. Hähnle, and D. Sands, "A Theorem Proving Approach to Analysis of Secure Information Flow," in *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 193–209.

[7] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 2004, pp. 100–114.

[8] H. Unno, N. Kobayashi, and A. Yonezawa, "Combining type-based analysis and model checking for finding counterexamples against non-interference," in *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*. New York, New York, USA: ACM Press, 2006, p. 17.

[9] G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *ASIAN'06: Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*. Springer-Verlag, Dec. 2006.

[10] A. Russo and A. Sabelfeld, "Dynamic vs. Static Flow-Sensitive Security Analysis," in *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*. IEEE, 2010, pp. 186–199.

[11] D. E. R. Denning, *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[12] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[13] D. Clark, S. Hunt, and P. Malacaria, "Quantitative Analysis of the Leakage of Confidential Data." *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 3, pp. 238–251, 2002.

[14] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Belief in information flow," *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pp. 31–45, 2005.

[15] ——, "Quantifying information flow with beliefs," *Journal of Computer Security*, vol. 17, pp. 655–701, 2009.

[16] J. Y. Halpern, *Reasoning about uncertainty*. MIT press Cambridge, 2003.

[17] G. Smith, "On the foundations of quantitative information flow," in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. FOSSACS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 288–302.

[18] ——, "Quantifying information flow using min-entropy," in *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*. IEEE, 2011, pp. 159–167.

[19] A. Rényi, "On measures of entropy and information," in *the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, 1961.

[20] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition." *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[21] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 141–153.

[22] J. Heusser and P. Malacaria, "Applied Quantitative Information Flow and Statistical Databases." in *Formal Aspects in Security and Trust*, 2009, pp. 96–110.

[23] V. Klebanov, "Precise quantitative information flow analysis - a symbolic approach." *Electr. Notes Theor. Comput. Sci. ()*, vol. 538, pp. 124–139, 2014.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *the 31st ACM SIGPLAN-SIGACT symposium*. New York, New York, USA: ACM Press, 2004, pp. 232–244.

[25] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *the ACM SIGPLAN 2001 conference*. New York, New York, USA: ACM Press, 2001, pp. 203–213.

[26] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs." *TACAS*, vol. 2988, no. Chapter 15, pp. 168–176, 2004.

[27] B. Köpf and A. Rybalchenko, "Automation of Quantitative Information-Flow Analysis." *SFM*, vol. 7938, no. Chapter 1, pp. 1–28, 2013.

[28] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, New York, USA: ACM Request Permissions, Jan. 1977, pp. 238–252.

[29] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pp. 51–65, 2008.

[30] ——, "Hyperproperties." *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[31] P. Cousot and R. Cousot, "Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages)," in *Computer Languages, 1994., Proceedings of the 1994 International Conference on*. IEEE Comput. Soc. Press, 1994, pp. 95–112.

[32] T. Amtoft and A. Banerjee, "Information Flow Analysis in Logical Form," in *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 100–115.

[33] G. Winskel, *The formal semantics of programming languages: an introduction*, Feb. 1993.

[34] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the second International Symposium on Programming*. Paris, 1976, pp. 106–130.

[35] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program." *POPL*, pp. 84–96, 1978.

[36] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *ESOP'05: Proceedings of the 14th European conference on Programming Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, Apr. 2005, pp. 5–20.

[37] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, Mar. 2006.

[38] ——, "Symbolic Methods to Enhance the Precision of Numerical Abstract Domains," in *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 2006, pp. 348–363.

[39] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "Combination of Abstractions in the ASTRÉE Static Analyzer," in *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 272–300.

[40] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1979, pp. 269–282.

[41] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[42] P. Cousot and R. Cousot, "Constructive versions of Tarski's fixed point theorems," *Pacific journal of Mathematics*, vol. 82, no. 1, pp. 43–57, 1979.

[43] P. Cousot, "The calculational design of a generic abstract interpreter," *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, vol. 173, pp. 421–506, 1999.

[44] J. Midtgaard and T. P. Jensen, "A Calculational Approach to Control-Flow Analysis by Abstract Interpretation." *SAS*, vol. 5079, no. Chapter 23, pp. 347–362, 2008.

[45] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Static Analysis and Verification of Aerospace Software by Abstract Interpretation," in *AIAA Infotech@Aerospace 2010*. Reston, Virigina: American Institute of Aeronautics and Astronautics, Jun. 2012.

[46] P. Granger, "Static analysis of arithmetical congruences," *International Journal of Computer Mathematics*, 1989.

[47] D. Clark, S. Hunt, and P. Malacaria, "Quantitative Information Flow, Relations and Polymorphic Types." *Journal of Logic and Computation*, vol. 15, no. 2, pp. 181–199, 2005.

[48] ——, "A static analysis for quantifying information flow in a simple imperative language," *Journal of Computer Security*, vol. 15, no. 3, pp. 321–371, 2007.

[49] T. M. Cover and J. A. Thomas, *Elements of Information Theory 2nd Edition*. Wiley-Interscience, 2006.

[50] B. Espinoza and G. Smith, "Min-entropy as a resource," *Information and Computation*, vol. 226, pp. 57–75, May 2013.

[51] C. Braun, K. Chatzikokolakis, and C. Palamidessi, "Quantitative notions of leakage for one-try attacks," *Electronic Notes in Theoretical Computer Science*, vol. 249, pp. 75–91, 2009.

[52] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith, "Measuring Information Leakage Using Generalized Gain Functions." in *2012 IEEE 25th Computer Security Foundations Symposium (CSF)*. IEEE, 2012, pp. 265–279.

[53] J. L. Massey, "Guessing and entropy," in *1994 IEEE International Symposium on Information Theory*. IEEE, 1994, p. 204.

[54] B. Köpf and A. Rybalchenko, "Approximation and Randomization for Quantitative Information-Flow Analysis," in *CSF '10: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*. IEEE Computer Society, Jul. 2010, pp. 3–14.

[55] H. Yasuoka and T. Terauchi, "On Bounding Problems of Quantitative Information Flow," in *Computer Security – ESORICS 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 2010, pp. 357–372.

[56] ——, "Quantitative Information Flow - Verification Hardness and Possibilities," *2010 IEEE 23rd Computer Security Foundations Symposium (CSF)*, pp. 15–27, 2010.

[57] ——, "On bounding problems of quantitative information flow." *Journal of Computer Security*, vol. 19, no. 6, pp. 1029–1082, 2011.

[58] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-Insensitive Noninterference Leaks More Than Just a Bit," in *Computer Security - ESORICS 2008*, ser. Lecture Notes in Computer Science, vol. 5283, 2008.

APPENDIX

This section builds the Galois connection assumed in Equation (5):

$$\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \langle Var \rightharpoonup \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^{\kappa}]; \dot{\subseteq}_{\otimes}, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^{\kappa}), \dot{\cup}_{\otimes}, \dot{\cap}_{\otimes} \rangle$$

The steps towards building such a Galois connection are summarized in Figure 4.

A *Non-relational Abstraction of Environments*

The cardinal abstraction is a non-relational abstraction. Thus, we start by defining a first Galois connection that ignores the relationships among variables.

In the case of environment properties, the following abstraction function [43] ignores relations between variables:

$$\begin{aligned} @ &\in \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right) \to \left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right) \\ @(r) &\triangleq \lambda x.\{\varrho(x) \mid \varrho \in r\} \end{aligned}$$

Since our collecting semantics is defined over hyperproperties – sets of environment properties –, we lift the previous abstraction $@$ over sets of properties through an element-wise abstraction [44] denoted by $\alpha_{@}$:

$$\begin{aligned} \alpha_{@}(R) &\triangleq \{@(r) \mid r \in R\} \\ &= \{\lambda x.\{\varrho(x) \mid \varrho \in r\} \mid r \in R\} \\ \gamma_{@}(Q) &\triangleq \{r \mid @(r) \in Q\} \end{aligned}$$

Therefore, we obtain a Galois connection:

$$\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle \underset{\alpha_{@}}{\overset{\gamma_{@}}{\leftrightarrows}} \langle \mathcal{P}\left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle$$

The abstraction $\alpha_{@}$ only forgets relationships among variable in each set $r \in R$ of environments. For instance, let us assume a set R_0 of sets of environments defined as:

$$R_0 \triangleq \Big\{ \begin{array}{l} \{[x \mapsto (pp_0, 0); y \mapsto (pp_0, 2)], [x \mapsto (pp_0, 0); y \mapsto (pp_0, 3)]\}, \\ \{[x \mapsto (pp_0, 1); y \mapsto (pp_0, 4)], [x \mapsto (pp_0, 1); y \mapsto (pp_0, 5)]\} \end{array} \Big\} \tag{7}$$

Then, the abstraction $\alpha_{@}(R_0)$ yields:

$$\alpha_{@}(R_0) = \Big\{ \begin{array}{l} [x \mapsto \{(pp_0, 0)\}; y \mapsto \{(pp_0, 2), (pp_0, 3)\}], \\ [x \mapsto \{(pp_0, 1)\}; y \mapsto \{(pp_0, 4), (pp_0, 5)\}] \end{array} \Big\}$$

In this latter set, we can further ignore relationships among variables through an additional non-relational abstraction:

$$\begin{aligned} \alpha^{\varpi}(P) &\triangleq \lambda x.\{p(x) \mid p \in P\} \\ \gamma^{\varpi}(Q) &\triangleq \{p \mid \alpha^{\varpi}(p) \in Q\} \end{aligned}$$

Therefore, $(\alpha^{\varpi}, \gamma^{\varpi})$ yields a Galois connection [31]:

$$\langle \mathcal{P}\left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle \underset{\alpha^{\varpi}}{\overset{\gamma^{\varpi}}{\leftrightarrows}} \langle Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \dot{\subseteq}, \lambda x.\varnothing, \lambda x.\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \dot{\cup}, \dot{\cap} \rangle$$

Since composing Galois connections yields a Galois connection, we define $(\alpha_r, \gamma_r) \triangleq (\alpha^{\varpi} \circ \alpha_{@}, \gamma^{\varpi} \circ \gamma_{@})$ to obtain a non-relational abstraction over hyperproperties:

$$\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle \underset{\alpha_r \triangleq \alpha^{\varpi} \circ \alpha_{@}}{\overset{\gamma_r \triangleq \gamma_{@} \circ \gamma^{\varpi}}{\leftrightarrows}} \langle Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \dot{\subseteq}, \lambda x.\varnothing, \lambda x.\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \dot{\cup}, \dot{\cap} \rangle \tag{8}$$

If we recall the example set R_0 of sets of environments defined in Equation (7), then $\alpha_r(R_0)$ ignores all relationships among variables:

$$\alpha_r(R_0) = \big[x \mapsto \big\{\{(pp_0, 0)\}, \{(pp_0, 1)\}\big\}; \\ y \mapsto \big\{\{(pp_0, 2), (pp_0, 3)\}, \{(pp_0, 4), (pp_0, 5)\}\big\}\big]$$

As we have defined a non-relational abstraction (α_r, γ_r) over hyperproperties, we can now focus on defining an abstraction over sets of sets of values as shown in Section IV-B. This latter abstraction can then be lifted over environments as illustrated by Section IV-B.

B Abstraction of Values

Similarly to the previous section, where we lift a non-relational abstraction to a set of sets of environments, we can also lift an *attribute independent abstraction* [43], that forgets about relationships between components of pairs, to a set of sets of pairs through an element-wise abstraction.

B0a Attribute independent abstraction: The attribute independent abstraction function $@_\times$ is given by:

$$@_\times \in \mathcal{P}(\mathbb{P} \times \mathbb{V}) \to \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}) \\ @_\times(p) \triangleq (\Pi_1(p), \Pi_2(p)) \\ \Pi_i(p) \triangleq \{proj_i(x) \mid x \in p\}$$

Therefore, we define an element-wise abstraction $\alpha_{@_\times}$ over a set of sets of pairs as follows:

$$\alpha_{@_\times}(P) \triangleq \{@_\times(p) \mid p \in P\} \\ = \{(\Pi_1(p), \Pi_2(p)) \mid p \in P\} \\ \gamma_{@_\times}(Q) \triangleq \{p \mid @_\times(p) \in Q\}$$

Therefore, $(\alpha_{@_\times}, \gamma_{@_\times})$ defines a Galois connection:

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P} \times \mathbb{V}), \cup, \cap\rangle \\ \underset{\alpha_{@_\times}}{\overset{\gamma_{@_\times}}{\leftrightarrows}} \\ \langle \mathcal{P}(\mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}), \cup, \cap\rangle \quad (9)$$

Let us assume that P_0 is a set of sets of pairs defined as follows:

$$P_0 \triangleq \big\{\{(pp_0, 2), (pp_0, 3)\}, \{(pp_0, 4), (pp_0, 5)\}\big\} \quad (10)$$

Then $\alpha_{@_\times}(P_0)$ is given by:

$$\alpha_{@_\times}(P_0) = \big\{(\{pp_0\}, \{2, 3\}), (\{pp_0\}, \{4, 5\})\big\}$$

In this latter set, we can further ignore relationships among values through an additional attribute independent abstraction:

$$\alpha_\times(Q) \triangleq (\Pi_1(Q), \Pi_2(Q)) \\ \gamma_\times((X, Y)) \triangleq X \times Y$$

Therefore, we obtain a Galois connection [43]:

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}), \cup, \cap\rangle \\ \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \\ \langle \mathcal{P}(\mathcal{P}(\mathbb{P})) \times \mathcal{P}(\mathcal{P}(\mathbb{V})); \subseteq_\times, (\varnothing, \varnothing), \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}), \cup_\times, \cap_\times\rangle \quad (11)$$

with the component-wise ordering, join and meet:

$$\subseteq_\times \triangleq \subseteq \times \subseteq \\ \cup_\times \triangleq \cup \times \cup \\ \cap_\times \triangleq \cap \times \cap$$

For instance, if we recall the example set P_0 defined in Equation (10), then $\alpha_\times \circ \alpha_{@_\times}(P_0)$ is given by:

$$\alpha_\times \circ \alpha_{@_\times}(P_0) = \alpha_\times\big(\big\{(\{pp_0\}, \{2, 3\}), (\{pp_0\}, \{4, 5\})\big\}\big) \\ = \big(\big\{\{pp_0\}\big\}, \big\{\{2, 3\}, \{4, 5\}\big\}\big)$$

B0b Component-wise abstraction: Finally, let us assume two Galois connections (α_v, γ_v) and $(\alpha_{pp}, \gamma_{pp})$ such that:

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{V}), \cup, \cap\rangle \underset{\alpha_v}{\overset{\gamma_v}{\leftrightarrows}} \langle [0, 2^\kappa]; \leq, 0, 2^\kappa, \max, \min\rangle$$
$$\langle \mathcal{P}(\mathcal{P}(\mathbb{P})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P}), \cup, \cap\rangle \underset{\alpha_{pp}}{\overset{\gamma_{pp}}{\leftrightarrows}} \langle \mathcal{P}(\mathbb{P}); \subseteq, \varnothing, \mathbb{P}, \cup, \cap\rangle$$

Then, a component-wise abstraction [44] $(\alpha_\otimes, \gamma_\otimes)$ defined as follows:

$$\alpha_\otimes\big((S_{pp}, S_v)\big) \triangleq \big(\alpha_{pp}(S_{pp}), \alpha_v(S_v)\big)$$
$$\gamma_\otimes\big((s_{pp}, n)\big) \triangleq \big(\gamma_{pp}(s_{pp}), \gamma_v(n)\big)$$

yields a Galois connection:

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{P})) \times \mathcal{P}(\mathcal{P}(\mathbb{V})); \subseteq_\times, (\varnothing, \varnothing), \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}), \cup_\times, \cap_\times\rangle \underset{\alpha_\otimes}{\overset{\gamma_\otimes}{\leftrightarrows}} \langle \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa]; \subseteq_\otimes, (\varnothing \times 0), (\mathbb{P} \times 2^\kappa), \cup_\otimes, \cap_\otimes\rangle \quad (12)$$

Consequently, the abstraction of values (α, γ) can be defined as the composition of the 3 Galois connections defined previously in Equations (9), (11) and (12):

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P} \times \mathbb{V}), \cup, \cap\rangle \underset{\alpha \triangleq \alpha_\otimes \circ \alpha_\times \circ \alpha_{@_\times}}{\overset{\gamma \triangleq \gamma_{@_\times} \circ \gamma_\times \circ \gamma_\otimes}{\leftrightarrows}} \langle \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa]; \subseteq_\otimes, (\varnothing, 0), (\mathbb{P}, 2^\kappa), \cup_\otimes, \cap_\otimes\rangle \quad (13)$$

Let us focus now on defining both Galois connections (α_v, γ_v) and $(\alpha_{pp}, \gamma_{pp})$.

B0c Abstraction of program points: The abstraction α_{pp} merges all the sets of program points:

$$\alpha_{pp}(S_{pp}) \triangleq \bigcup_{s \in S_{pp}} s$$
$$\gamma_{pp}(s_{pp}) \triangleq \mathcal{P}(s_{pp})$$

Therefore, we obtain a Galois connection:

$$(\mathcal{P}(\mathcal{P}(\mathbb{P})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P}), \cup, \cap) \underset{\alpha_{pp}}{\overset{\gamma_{pp}}{\leftrightarrows}} (\mathcal{P}(\mathbb{P}); \subseteq, \varnothing, \mathbb{P}, \cup, \cap)$$

Indeed, let us prove that $(\alpha_{pp}, \gamma_{pp})$ is a Galois connection:

$$\begin{aligned}
\alpha_{pp}(S) \subseteq s_{pp} &\iff \bigcup_{s \in S} s \subseteq s_{pp} \\
&\iff \forall s \in S, s \in \mathcal{P}(s_{pp}) \\
&\iff S \subseteq \mathcal{P}(s_{pp}) \\
&\iff S \subseteq \gamma_{pp}(s_{pp}) \quad \square
\end{aligned}$$

B0d Abstraction of concrete values: The abstraction α_v computes the maximum cardinal of values over the sets $s_v \in S_v$:

$$\alpha_v(S_v) \triangleq \max_{s_v \in S_v} |s_v|$$
$$\gamma_v(n) = \{V \in \mathcal{P}(\mathbb{V}) \mid |V| \leq n\}$$

Therefore, (α_v, γ_v) is a Galois connection:

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{V}), \cup, \cap\rangle \underset{\alpha_v}{\overset{\gamma_v}{\leftrightarrows}} \langle [0, 2^\kappa]; \leq, 0, 2^\kappa, \max, \min\rangle$$

Indeed, let us prove that (α_v, γ_v) is a Galois connection:

$$\begin{aligned}
\alpha_v(S_v) \leq n &\iff \max_{s_v \in S_v} |s_v| \leq n \\
&\iff \forall s_v \in S_v, s_v \in \mathcal{P}_n(\mathbb{V}) \\
&\iff S_v \subseteq \gamma_v(n) \quad \square
\end{aligned}$$

C Abstraction of Environments

As illustrated by Section IV-B, the abstraction (α, γ) of values – defined in Equation (13) – can be lifted through a pointwise abstraction [31], in order to abstract a set valued environment $\varrho \in Var \rightharpoonup \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V}))$.

Indeed, let us define a pointwise abstraction (α_c, γ_c) as follows:

$$\begin{aligned} \alpha_c &\triangleq \lambda f.\lambda x.\alpha\big(f(x)\big) \\ \gamma_c &\triangleq \lambda f^\sharp.\lambda x.\gamma\big(f^\sharp(x)\big) \end{aligned}$$

Then, (α_c, γ_c) yields a Galois connection:

$$\begin{array}{l} \langle Var \rightharpoonup \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V})) ; \dot{\subseteq}, \lambda x.\varnothing, \lambda x.\mathcal{P}(\mathbb{P} \times \mathbb{V}), \dot{\cup}, \dot{\cap} \rangle \\ \qquad\qquad \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} \\ \qquad\qquad \langle Var \rightharpoonup \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa] ; \dot{\subseteq}_\otimes, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^\kappa), \dot{\cup}_\otimes, \dot{\cap}_\otimes \rangle \end{array}$$

Therefore, by composing both the non-relational abstraction over environments and the pointwise abstraction of values defined in Equations (8) and (13), we can build the abstraction $(\dot{\alpha}, \dot{\gamma})$:

$$\begin{aligned} \dot{\alpha} &\triangleq \alpha_c \circ \alpha_r \\ \dot{\gamma} &\triangleq \gamma_r \circ \gamma_c \end{aligned}$$

Thus, the pair $(\dot{\alpha}, \dot{\gamma})$ yields a Galois connection:

$$\begin{array}{l} \langle \mathcal{P}(\mathcal{P}(Var \rightharpoonup \mathbb{P} \times \mathbb{V})) ; \subseteq, \varnothing, \mathcal{P}(Var \rightharpoonup \mathbb{P} \times \mathbb{V}), \cup, \cap \rangle \\ \qquad\qquad \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \\ \qquad\qquad \langle Var \rightharpoonup \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa] ; \dot{\subseteq}_\otimes, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^\kappa), \dot{\cup}_\otimes, \dot{\cap}_\otimes \rangle \end{array} \quad (14)$$

As mentioned in Section IV-A, once we construct a Galois connection relating concrete objects to abstract one, we can define functional abstractions in order to soundly approximate functions over concrete objects by functions over abstract ones.

In order to approximate the collecting semantics $\mathbb{A}_c[\![a]\!]$ of expressions, we can define the following functional abstraction [43]:

$$\begin{aligned} \alpha^{\triangleright}_{exp} &\in \big(\mathcal{P}(\mathcal{P}(Env)) \to \mathcal{P}(\mathcal{P}(\mathbb{V}))\big) \mapsto \big(Env^\sharp \to [0, 2^\kappa]\big) \\ \alpha^{\triangleright}_{exp}(\phi) &\triangleq \alpha_v \circ \phi \circ \dot{\gamma} \end{aligned}$$

Therefore, the abstract semantics $\mathbb{A}^\sharp[\![a]\!]$ of expressions is sound wrt. the collecting semantics $\mathbb{A}_c[\![a]\!]$ of expressions if:

$$\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \quad (15)$$

Theorem 2 (Soundness of the abstract semantics $\mathbb{A}^\sharp[\![a]\!]$)**.**
The abstract semantics of expressions in Figure 2 is sound:

$$\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp.$$

Let us derive an abstract semantics $\mathbb{A}^\sharp[\![a]\!]R$ for expressions:

— Case `a = n`:

$$\begin{aligned} \alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![n]\!])\varrho^\sharp &\triangleq \alpha_v \circ \mathbb{A}_c[\![n]\!] \circ \dot{\gamma}(\varrho^\sharp) \\ &= \alpha_v\left(\left\{\{v \in \mathbb{V} \mid \exists \varrho \in r, \mathbb{A}[\![n]\!]\varrho = v\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right) \\ &= \alpha_v(\{\{n\}\}) \\ &= 1 \\ &\triangleq \mathbb{A}^\sharp[\![n]\!]\varrho^\sharp \end{aligned}$$

— Case `a = id`:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![id]\!])\varrho^\sharp &\triangleq \alpha_v \circ [\![id]\!]^\sharp \circ \dot{\gamma}(\varrho^\sharp)\\
&= \alpha_v\left(\left\{\{v \mid \exists \varrho \in r, \mathbb{A}[\![id]\!]\varrho = v\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\
&= \alpha_v\left(\left\{\{proj_2(\varrho(id)) \mid \exists \varrho \in r\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\
&= \text{(By definition of } \dot{\gamma})\\
&\quad \alpha_v \circ \gamma_v(\varrho^\sharp(id))\\
&\leq (\alpha_v \circ \gamma_v \text{ is reductive})\\
&\quad \varrho^\sharp(id)\\
&\triangleq \mathbb{A}^\sharp[\![id]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = a₁ mod n`:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp &\triangleq \alpha_v\left(\left\{\{v_1 \ mod\ n \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1\} \mid \right.\right.\\
&\qquad \left.\left. r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\
&= \min\left(\alpha_v\left(\left\{\{v_1 \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1\} \mid \right.\right.\right.\\
&\qquad \left.\left.\left. r \in \dot{\gamma}(\varrho^\sharp)\right\}\right), n\right)\\
&= \min(\alpha_v \circ \mathbb{A}_c[\![a_1]\!] \circ \dot{\gamma}(\varrho^\sharp), n)\\
&\leq \text{(By induction hypothesis)}\\
&\quad \min(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp, n)\\
&\triangleq \mathbb{A}^\sharp[\![a_1 \ mod\ n]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = a₁ bop a₂`:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a_1 \ bop\ a_2]\!])\varrho^\sharp &\triangleq \alpha_v\left(\{\{v_1 \ bop\ v_2 \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1\right.\\
&\qquad \left.\wedge\, \mathbb{A}[\![a_2]\!]\varrho = v_2\} \mid r \in \dot{\gamma}(\varrho^\sharp)\}\right)\\
&\leq \text{(by loosing relationships among variables in } r)\\
&\quad \alpha_v\left(\left\{\{v_1 \ bop\ v_2 \mid \exists \varrho_1 \in r_1, \exists \varrho_2 \in r_2, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right.\right.\\
&\qquad \left.\left.\wedge\, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\} \mid r_1, r_2 \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\
&= \text{(by definition of } \alpha_v)\\
&\quad \max_{r_1, r_2 \in \dot{\gamma}(\varrho^\sharp)} \left|\left\{v_1 \ bop\ v_2 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v\right.\right.\\
&\qquad \left.\left.\wedge\, \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\\
&\leq \text{(values } v \in \mathbb{V} \text{ are finite, of size } 2^\kappa)\\
&\quad \min\left(2^\kappa, \max_{r_1, r_2 \in \dot{\gamma}(\varrho^\sharp)} \left|\left\{v_1 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right\}\right|\right.\\
&\qquad \left.\times \left|\left\{v_2 \mid \exists \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\right)\\
&= \min\left(2^\kappa, \max_{r_1 \in \dot{\gamma}(\varrho^\sharp)} \left|\left\{v_1 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right\}\right|\right.\\
&\qquad \left.\times \max_{r_2 \in \dot{\gamma}(\varrho^\sharp)} \left|\left\{v_2 \mid \exists \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\right)\\
&\leq \text{(By induction hypothesis in Equation (15))}\\
&\quad \min\left(2^\kappa, \mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp\right)\\
&\triangleq \mathbb{A}^\sharp[\![a_1 \ bop\ a_2]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = a₁ cmp a₂`: this case is similar to the previous case, apart that expression `a₁ cmp a₂` may evaluate to only 2 different boolean values.

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp &\dot{\leq} \min\left(2, \mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp\right)\\
&\triangleq \mathbb{A}^\sharp[\![a_1 \ cmp\ a_2]\!]
\end{aligned}$$

Theorem 3 (Soundness of the abstract semantics $[\![c]\!]^\sharp$)**.**
The abstract semantics of commands in Figure 3 is sound:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \dot{\subseteq}_\otimes [\![c]\!]^\sharp\varrho^\sharp.$$

Similarly to expressions, we define the functional abstraction $\alpha^{\triangleright}_{com}$ in order to soundly approximate the collecting semantics $[\![c]\!]_c$ of commands.

$$\begin{aligned}\alpha^{\triangleright}_{com} &\in \big(\mathcal{P}\left(\mathcal{P}\left(Env\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(Env\right)\right)\big) \to \left(Env^\sharp \to Env^\sharp\right)\\ \alpha^{\triangleright}_{com}([\![c]\!]_c) &\triangleq \dot{\alpha} \circ [\![c]\!]_c \circ \dot{\gamma}\end{aligned}$$

Therefore, the abstract semantics of commands $[\![c]\!]^\sharp$ is sound wrt. the collecting semantics $[\![c]\!]_c$ of commands if:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \dot{\subseteq}_\otimes [\![c]\!]^\sharp\varrho^\sharp \tag{16}$$

Theorem 3 (Soundness of the abstract semantics $[\![c]\!]^\sharp$)**.**
The abstract semantics of commands in Figure 3 is sound:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \dot{\subseteq}_\otimes [\![c]\!]^\sharp\varrho^\sharp.$$

Let us derive an abstract semantics for instructions:
— Case `c = ` ^{pp}`skip`:

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![^{pp}skip]\!]_c)\varrho^\sharp &= \dot{\alpha} \circ [\![^{pp}skip]\!]_c \circ \dot{\gamma}(\varrho^\sharp)\\ &= \dot{\alpha} \circ \dot{\gamma}(\varrho^\sharp)\\ &\subseteq_\otimes (\dot{\alpha} \circ \dot{\gamma} \text{ is reductive})\\ &\quad \varrho^\sharp\\ &\triangleq [\![^{pp}skip]\!]^\sharp\varrho^\sharp\end{aligned}$$

— For instruction `c = ` ^{pp}`id := a`:

$$\begin{aligned}\alpha^{\triangleright}_{com}(c)\varrho^\sharp &= \dot{\alpha} \circ [\![^{pp}id := a]\!]_c \circ \dot{\gamma}(\varrho^\sharp)\\ &= \dot{\alpha}\left(\left\{\{\varrho' \mid \exists \varrho \in r, [\![id := a]\!]\varrho = \varrho'\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\ &= \dot{\alpha}\left(\left\{\{\varrho[id \mapsto (pp, v)] \mid \exists \varrho \in r, \mathbb{A}[\![a]\!]\varrho = v\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\end{aligned}$$

Hence, for all $x \in Var$ such that $x \neq id$:

$$(\alpha^{\triangleright}_{com}([\![^{pp}id := a]\!]_c)\varrho^\sharp)(x) \subseteq_\otimes \varrho^\sharp(x)$$

Additionnally, for $x = id$:

$$\begin{aligned}(\alpha^{\triangleright}_{com}(c)\varrho^\sharp)(id) &= \Big(\bigcup_{r' \in [\![c]\!]_c \circ \dot{\gamma}(\varrho^\sharp)} \Pi_1(@(r')(id)), \max_{r' \in [\![c]\!]_c \circ \dot{\gamma}(\varrho^\sharp)} |\Pi_2(@(r')(id))|\Big)\\ &= \big(\{pp\}, \max_{r' \in [\![c]\!]_c \circ \dot{\gamma}(\varrho^\sharp)} |\Pi_2(\{\varrho(id) : \varrho \in r'\})|\big)\\ &= \big(\{pp, \max_{r \in \dot{\gamma}(\varrho^\sharp)} |\{\mathbb{A}[\![a]\!]\varrho : \varrho \in r\}|\}\big)\\ &= \big(\{pp\}, \alpha_v \circ \mathbb{A}_c[\![a]\!]\varrho^\sharp \circ \dot{\gamma}(\varrho^\sharp)\big)\\ &\subseteq_\otimes \big(\{pp\}, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp\big)\end{aligned}$$

Hence,

$$\begin{aligned}\alpha^{\triangleright}_{com}(^{pp}id := a)\varrho^\sharp &\subseteq_\otimes \varrho^\sharp[id \mapsto (pp, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)]\\ &\triangleq [\![^{pp}id := a]\!]^\sharp\varrho^\sharp\end{aligned}$$

– For conditional instructions $c = {}^{pp}if\ (a)\ c_1\ else\ c_0$:

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp &= \dot{\alpha}\left(\left\{\{\varrho' \mid \exists \varrho \in r, [\![c]\!]\varrho = \varrho'\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\ &= \dot{\alpha}\left(\left\{\{\varrho' : \exists \varrho \in r, \exists v \in \{0,1\}, \mathbb{A}[\![a]\!]\varrho = v\right.\right.\\ &\qquad \left.\left. \wedge\ [\![c_v]\!]\varrho = \varrho'\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\end{aligned}$$

First, if $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp = 1$, then expression $\mathtt{a}$ evaluates to at most one value in each set $r \in \dot\gamma(\varrho^\sharp)$:

$$\forall r \in \dot\gamma(\varrho^\sharp), \exists v \in \{0,1\}, \forall \varrho \in r, \mathbb{A}[\![a]\!]\varrho = v$$

Therefore, the sets $r \in \dot\gamma(\varrho^\sharp)$ can be partitioned into sets r_1 (resp. r_0) where expression $\mathtt{a}$ evaluates to 1 (resp. evaluates to 0):

$$\begin{aligned}
\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp &= \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r_1, [\![c_1]\!]\varrho = \varrho'\} \mid r_1 \in \dot\gamma(\varrho^\sharp)\right\}\right.\\
&\qquad \left.\cup \left\{\{\varrho' \mid \exists \varrho \in r_0, [\![c_0]\!]\varrho = \varrho'\} \mid r_0 \in \dot\gamma(\varrho^\sharp)\right\}\right)\\
&\dot\subseteq_\otimes \text{ ($\dot\alpha$ preserves joins)}\\
&\quad \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r_1, [\![c_1]\!]\varrho = \varrho'\} \mid r_1 \in \dot\gamma(\varrho^\sharp)\right\}\right)\\
&\qquad \dot\cup_\otimes\, \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r_0, [\![c_0]\!]\varrho = \varrho'\} \mid r_0 \in \dot\gamma(\varrho^\sharp)\right\}\right)\\
&= \left(\dot\alpha \circ [\![c_1]\!]_c \circ \dot\gamma(\varrho^\sharp)\right) \dot\cup_\otimes \left(\dot\alpha \circ [\![c_0]\!]_c \circ \dot\gamma(\varrho^\sharp)\right)\\
&\dot\subseteq_\otimes \text{ (by induction hypothesis Equation (16))}\\
&\quad [\![c_1]\!]^\sharp\varrho^\sharp \,\dot\cup_\otimes\, [\![c_0]\!]^\sharp\varrho^\sharp
\end{aligned}$$

Second, if $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp > 1$, then for variables $\mathtt{x}$ that are modified in neither $\mathtt{c}_1$ nor $\mathtt{c}_0$, we have:

$$(\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp)(x) = \left([\![c_1]\!]^\sharp\varrho^\sharp \,\dot\cup_\otimes\, [\![c_0]\!]^\sharp\varrho^\sharp\right)(x)$$

Finally, for variables that are modified in either $\mathtt{c}_1$ or $\mathtt{c}_0$:

$$\begin{aligned}
(\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp)(x) &= \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r, \exists v \in \{0,1\}, \mathbb{A}[\![a]\!]\varrho = v\right.\right.\\
&\qquad \left.\left.\wedge\, [\![c_v]\!]\varrho = \varrho'\} \mid r \in \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&= \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r, \mathbb{A}[\![a]\!]\varrho = 1 \wedge [\![c_1]\!]\varrho = \varrho'\}\right.\right.\\
&\qquad \left.\left.\cup \{\varrho' \mid \exists \varrho \in r, \mathbb{A}[\![a]\!]\varrho = 0 \wedge [\![c_0]\!]\varrho = \varrho'\}\right\}\right)(x)\\
&\subseteq_\otimes \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r_2, \mathbb{A}[\![a]\!]\varrho = 1 \wedge [\![c_1]\!]\varrho = \varrho'\}\right.\right.\\
&\qquad \cup \{\varrho' \mid \exists \varrho \in r_1, \mathbb{A}[\![a]\!]\varrho = 0 \wedge [\![c_0]\!]\varrho = \varrho'\}\\
&\qquad \left.\left. \mid r_1, r_2 \in \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes \dot\alpha\left(\left\{\{\varrho' \mid \varrho' \in r_1'\} \cup \{\varrho' \mid \varrho' \in r_2'\} \mid r_1' \in [\![c_1]\!]_c \circ \dot\gamma(\varrho^\sharp),\right.\right.\\
&\qquad \left.\left. r_2' \in [\![c_0]\!]_c \circ \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes \text{ ($\dot\gamma \circ \dot\alpha$ is extensive, and $\dot\alpha$ is monotone)}\\
&\quad \dot\alpha\left(\left\{\{\varrho' \mid \varrho' \in r_1'\} \cup \{\varrho' \mid \varrho' \in r_2'\} \mid\right.\right.\\
&\qquad r_1' \in \dot\gamma \circ \dot\alpha \circ [\![c_1]\!]_c \circ \dot\gamma(\varrho^\sharp),\\
&\qquad \left.\left. r_2' \in \dot\gamma \circ \dot\alpha \circ [\![c_0]\!]_c \circ \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes \text{ (By hypothesis in Equation (16) and monotony of } \dot\alpha)\\
&\quad \dot\alpha\left(\left\{\{\varrho' \mid \varrho' \in r_1'\} \cup \{\varrho' \mid \varrho' \in r_2'\} \mid\right.\right.\\
&\qquad \left.\left. r_1' \in \dot\gamma([\![c_1]\!]^\sharp\varrho^\sharp), r_2' \in \dot\gamma([\![c_0]\!]^\sharp\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes \text{ (By definition of } (\dot\alpha, \dot\gamma))\\
&\quad \left(proj_1([\![c_1]\!]^\sharp\varrho^\sharp(x)) \cup proj_1([\![c_0]\!]^\sharp\varrho^\sharp(x)),\right.\\
&\qquad \left. proj_2([\![c_1]\!]^\sharp\varrho^\sharp(x)) + proj_2([\![c_0]\!]^\sharp\varrho^\sharp(x))\right)
\end{aligned}$$

Hence, the abstract semantics of conditionals is sound:

$$\begin{aligned}
[\![^{pp}if\ (a)\ c_1\ else\ c_2]\!]^\sharp\varrho^\sharp \triangleq\ &\text{let } n = \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \text{ in}\\
&\text{let } \varrho_1^\sharp = [\![c_1]\!]^\sharp\varrho^\sharp \text{ in}\\
&\text{let } \varrho_2^\sharp = [\![c_2]\!]^\sharp\varrho^\sharp \text{ in}\\
&\lambda id. \begin{cases} \varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id) & \text{if } n = 1\\ \varrho_1^\sharp(id) \cup_{add(c_1,c_2)} \varrho_2^\sharp(id) & \text{otherwise}\end{cases}
\end{aligned}$$

— For sequences $c_1;\ c_2$:

$$\begin{aligned}
\alpha^{\triangleright}_{com}(c_1;\ c_2)\varrho^\sharp &= \dot\alpha\big(\{\{\varrho_2 : \exists \varrho \in r, [\![c_1;\ c_2]\!]\varrho = \varrho_2\} : r \in \dot\gamma(\varrho^\sharp)\}\big)\\
&= \dot\alpha\big(\{\{\varrho_2 : \exists \varrho_1 \in r_1, [\![c_2]\!]\varrho_1 = \varrho_2\} : r_1 \in [\![c_1]\!]_c \dot\gamma(\varrho^\sharp)\}\big)\\
&\dot\subseteq_\otimes \dot\alpha\big(\{\{\varrho_2 : \exists \varrho_1 \in r_1, [\![c_2]\!]\varrho_1 = \varrho_2\} : r_1 \in \dot\gamma \circ \dot\alpha\left([\![c_1]\!]_c \dot\gamma(\varrho^\sharp)\right)\}\big)\\
&\dot\subseteq_\otimes \dot\alpha\big(\{\{\varrho_2 : \exists \varrho_1 \in r_1, [\![c_2]\!]\varrho_1 = \varrho_2\} : r_1 \in \dot\gamma\left([\![c_1]\!]^\sharp \varrho^\sharp\right)\}\big)\\
&\dot\subseteq_\otimes [\![c_2]\!]^\sharp\left([\![c_1]\!]^\sharp \varrho^\sharp\right)
\end{aligned}$$

— For while loops ${}^{pp}while\ (a)\ c$:

$$\begin{aligned}
\alpha^{\triangleright}_{com}({}^{pp}while\ (a)\ c)\varrho^\sharp &= \dot\alpha \circ [\![{}^{pp}while\ (a)\ c]\!]_c \circ \dot\gamma(\varrho^\sharp)\\
&= \text{(By characterizing the collecting semantics for loops}\\
&\quad \text{as a least fixpoint}\\
&\quad \text{with } F_{X_0} = \lambda X.X_0 \cup [\![{}^{pp}if\ (a)\ c\ else\ {}^{pp}skip]\!]_c\ )\\
&\quad \dot\alpha\left(\mathrm{lfp}^{\subseteq}_{\dot\gamma(\varrho^\sharp)} F\right)\\
&\dot\subseteq_\otimes \text{(By the fixpoint transfer theorem)}\\
&\quad \mathrm{lfp}^{\dot\subseteq_\otimes}_{\varrho^\sharp} [\![{}^{pp}if\ (a)\ c\ else\ {}^{pp}skip]\!]^\sharp
\end{aligned}$$

We prove that the cardinal abstraction refines a flow-sensitive type system labelling variables as either low or high.

Let us first introduce an abstraction mapping a cardinal number n to a security label L or H:

$$@_{LH} \in [0, 2^\kappa] \mapsto \{L, H\}$$
$$@_{LH}(n) \triangleq \begin{cases} L \text{ if } n = 1 \\ H \text{ otherwise} \end{cases}$$

Let us consider a flow-sensitive type system such as the one presented in [5]. Such a type system considers a type environment $\Gamma \in Var \rightharpoonup \{L, H\}$ as well as a security label pc. Let us denote by $\sqcup$ the join operator over security labels, and by $\sqsubseteq$ the partial order over security labels.

Derivation rules for expressions have the form:

$$\Gamma \vdash a : t \text{ iff. } \bigsqcup_{x \in fv(a)} \Gamma(x).$$

Derivation rules for instructions have the form:

$$\mathrm{pc} \vdash \Gamma_0\{c\}\Gamma_1$$

where pc is a security label attached to the program counter, Γ_0 is an input type environment, and Γ is an output type environment.

D Precision of the Abstract Semantics of Expressions

Lemma 1 proves that the abstract semantics of expressions is as precise as the type labelling of expressions.

Lemma 1 (The abstract semantics of expressions is at least as precise as the type labelling of expressions)**.**
For all expressions a*, for all abstract environments* $\varrho^\sharp \in Var \rightharpoonup D_C^\sharp$*, for all type environments* $\Gamma \in Var \rightharpoonup \{L, H\}$*, such that:* $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp = n$ *and* $\Gamma \vdash a : t$*.*

It holds that:

$$\alpha_{@_{LH}}(\varrho^\sharp) \sqsubseteq \Gamma \implies @_{LH}(n) \sqsubseteq t$$

Proof. By structural induction on expressions:

— Case `a = n`:

The abstract semantics of expressions evaluates constants to 1. The type labelling evaluates constants to L. Therefore:

$$\mathbb{A}^\sharp[\![n]\!]\varrho^\sharp = 1, \Gamma \vdash n : L \text{ and } @_{LH}(1) \sqsubseteq L$$

— Case `a = id`:

Assumption $\alpha_{@_{LH}}(\varrho^\sharp) = \Gamma$ implies that $@_{LH}(n) = \Gamma(id)$

$$\alpha_{@_{LH}}(\varrho^\sharp) = \Gamma \implies @_{LH}(n) \sqsubseteq \Gamma(id)$$

— Case `a = a1 bop a2`:

Let $n_1 = \mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp$ and $n_2 = \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp$. Let also t_1 and t_2 such that $\Gamma \vdash a_1 : t_1$, and $\Gamma \vdash a_2 : t_2$.

By induction on $\mathtt{a}_1$ and $\mathtt{a}_1$, assumption $\alpha_{@_{LH}}(\varrho^\sharp) = \Gamma$ implies that:

$$@_{LH}(n_1) \sqsubseteq t_1 \text{ and } @_{LH}(n_2) \sqsubseteq t_2$$

Therefore, $\min(n_1 \times n_2, 2^\kappa) \sqsubseteq t_1 \sqcup t_2$.

— Cases `a = a1 cmp a2` and `a = a1 mod n` are both similar to case `a = a1 bop a2`.

E Precision of the Abstract Semantics of Instructions

Theorem 4 proves that the abstract semantics of instructions is as precise as the type labelling of commands. We let $\dot{\sqsubseteq}$ denote the pointwise lifting the partial order over security labels to type environments.

Theorem 4 (The abstract semantics of commands is at least as precise as the type labelling of commands)**.**
For all commands c*, for all abstract environments* $\varrho_0^\sharp, \varrho^\sharp$*, and type environments* Γ_0, Γ *such that:* $[\![c]\!]^\sharp\varrho_0^\sharp = \varrho^\sharp$ *and* $L \vdash \Gamma_0\{c\}\Gamma$*.*

It holds that:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) \sqsubseteq \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \Gamma$$

Proof. By structural induction on instructions:

— Case `c = ` $^{\mathrm{pp}}$`skip`:

$\varrho_0^\sharp = \varrho^\sharp$ and $\Gamma_0 = \Gamma$. Therefore, assumption

$$\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \sqsubseteq \Gamma$$

— Case `c = ppid := a`:
Let $n = \mathbb{A}^\sharp[\![a]\!]\varrho_0^\sharp$ and t such that $\Gamma_0 \vdash a : t$. By Lemma 1:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0 \implies @_{LH}(n) \sqsubseteq t$$

Since $\varrho^\sharp = \varrho^\sharp[id \mapsto (\{pp\}, n)]$, and $\Gamma = \Gamma_0[id \mapsto t]$, we have:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \Gamma$$

— Case $c = {}^{pp}if\ (a)\ c_1\ else\ c_2$:
Let $n = \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp$ and t such that $\Gamma_0 \vdash a : t$.
Let $\varrho_1^\sharp = [\![c_1]\!]^\sharp\varrho_0^\sharp$, and $\varrho_2^\sharp = [\![c_2]\!]^\sharp\varrho_0^\sharp$.
Let Γ_1 and Γ_2 such that $L \vdash \Gamma_0\{c_1\}\Gamma_1$ and $L \vdash \Gamma_0\{c_2\}\Gamma_2$.
By induction on both $\mathtt{c}_1$ and $\mathtt{c}_2$, we have:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0 \implies \alpha_{@_{LH}}(\varrho_1^\sharp) \dot{\sqsubseteq} \Gamma_1 \text{ and } \alpha_{@_{LH}}(\varrho_2^\sharp) \dot{\sqsubseteq} \Gamma_2 \tag{17}$$

- If $n = 1$ and $t = L$, we have $\varrho^\sharp = \lambda id.\varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id)$ and $\Gamma = \lambda id.\Gamma_1(id) \sqcup \Gamma_2(id)$. Therefore, since the union operator $\cup_\otimes$ computes the maximum over cardinal values, Equation (17) implies that

$$\alpha_{@_{LH}}(\varrho_0^\sharp) \dot{\sqsubseteq} \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \Gamma$$

- if $n = 1$ and $t = H$, we have $\varrho^\sharp = \lambda id.\varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id)$.
  Additionally, Let Γ_1' and Γ_2' such that $H \vdash \Gamma_0\{c_1\}\Gamma_1'$ and $H \vdash \Gamma_0\{c_2\}\Gamma_2'$.
  Note that $\Gamma = \lambda id.\Gamma_1'(id) \sqcup \Gamma_2'(id)$ is the resulting type environments, and $\Gamma_1 \dot{\sqsubseteq} \Gamma_1'$ and $\Gamma_2 \dot{\sqsubseteq} \Gamma_2'$. Therefore, by Equation (17) we have:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) \dot{\sqsubseteq} \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \lambda id.\Gamma_1(id) \sqcup \Gamma_2(id) \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \lambda id.\Gamma_1'(id) \sqcup \Gamma_2'(id)$$

- If $n > 1$, then we have $\varrho^\sharp = \lambda id.\varrho_1^\sharp(id) \cup_{add(c_1,c_2)} \varrho_2^\sharp(id)$. Plus, assuming that $\alpha_{@_{LH}}(\varrho_0^\sharp) \dot{\sqsubseteq} \Gamma_0$, Lemma 1 implies that $t = H$.
  Let Γ_1' and Γ_2' such that $H \vdash \Gamma_0\{c_1\}\Gamma_1'$ and $H \vdash \Gamma_0\{c_2\}\Gamma_2'$.
  Let also Γ' such that $\Gamma' = \lambda id.\Gamma_1'(id) \sqcup \Gamma_2'(id)$. Note that the only difference between Γ_1' and Γ_1 (resp. Γ_2' and Γ_2) is that variables that may be modified inside instruction $\mathtt{c}_1$ (resp. instruction $\mathtt{c}_2$) are set to H. Therefore the only difference between Γ' and Γ is that variables that may be modified in either conditional branches are set to H.
  Thus, assuming $\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0$, for variables `id` that are not modified inside the conditional branches, Equation (17) yields:

$$@_{LH}(\varrho^\sharp(id)) \sqsubseteq \Gamma(id) = \Gamma'(id)$$

  Moreover, for variables `id` that may be modified inside the conditional branches, since Γ' maps them to H, it holds that

$$@_{LH}(\varrho^\sharp(id)) \sqsubseteq \Gamma'(id)$$

  Finally, by noting that $\Gamma = \Gamma'$ is the resulting type environments in the case where $\mathbb{A}^\sharp[\![a]\!] \geq 1$, it holds that:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \Gamma$$

— Case $c_1;\ c_2$:
By a first induction on $\mathtt{c}_1$, then a second induction on $\mathtt{c}_2$, we obtain:

$$\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0 \implies \alpha_{@_{LH}}(\varrho^\sharp) \dot{\sqsubseteq} \Gamma$$

— Case ${}^{pp}while\ (a)\ c$:
Let us assume $\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0$. The output abstract environment is given by:

$$\varrho^\sharp = \text{fix}\left(\lambda\varrho_v^\sharp.\varrho_0^\sharp \dot{\cup}_\otimes \left([\![{}^{pp}if\ (a)\ c\ else\ {}^{pp}skip]\!]^\sharp\varrho_v^\sharp\right)\right)$$

Additionally, the output type environment is given by:

$$\Gamma = \text{fix}\left(\lambda\Gamma_v.\text{let } L \sqcup \Gamma_v(a) \vdash \Gamma_v\{c\}\Gamma' \text{ in } \quad \Gamma' \sqcup \Gamma_0\right)$$

Or, rewritten differently:

$$\Gamma = \text{fix}\left(\lambda\Gamma_v.\text{let } L \vdash \Gamma_v\{if\ (a)\ c\ else\ skip\}\Gamma' \text{ in } \quad \Gamma' \sqcup \Gamma_0\right)$$

Let the sequences $(\varrho_n^\sharp)$ and Γ_n be defined as follows:

$$\varrho_{n+1}^\sharp = \varrho_0^\sharp \dot{\cup}_\otimes \left([\![^{pp} if\ (a)\ c\ else\ ^{pp} skip]\!]^\sharp \varrho_n^\sharp\right)$$

$$\Gamma_{n+1} = \text{let } L \vdash \Gamma_n\{if\ (a)\ c\ else\ skip\}\Gamma' \text{ in } \quad \Gamma' \sqcup \Gamma_0$$

Then we prove by recurrence over n that $\forall n, \alpha_{@_{LH}}(\varrho_n^\sharp) \dot{\sqsubseteq} \Gamma_n$:

- For the initial case $n = 0$, it holds by assumption that $\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0$.
- Let us assume $\alpha_{@_{LH}}(\varrho_n^\sharp) \dot{\sqsubseteq} \Gamma_n$, and prove $\alpha_{@_{LH}}(\varrho_{n+1}^\sharp) \dot{\sqsubseteq} \Gamma_{n+1}$.
  By the same proof that is presented in the case of conditionals, we prove that:

  $$\alpha_{@_{LH}}\left([\![^{pp} if\ (a)\ c\ else\ ^{pp} skip]\!]^\sharp \varrho_n^\sharp\right) \dot{\sqsubseteq} \Gamma'$$

  Additionally, since $\alpha_{@_{LH}}(\varrho_0^\sharp) = \Gamma_0$, and the join operator $\dot{\cup}_\otimes$ compute the maximum over cardinals, we have:

  $$\alpha_{@_{LH}}\left(\varrho_0^\sharp \dot{\cup}_\otimes [\![^{pp} if\ (a)\ c\ else\ ^{pp} skip]\!]^\sharp \varrho_n^\sharp\right) \dot{\sqsubseteq} \Gamma' \sqcup \Gamma_0$$

  Therefore $\alpha_{@_{LH}}(\varrho_{n+1}^\sharp) \dot{\sqsubseteq} \Gamma_{n+1}$.

This concludes our proof for the case of loops, since both sequences converge, and their limit satisfy $\alpha_{@_{LH}}(\varrho_\infty^\sharp) \dot{\sqsubseteq} \Gamma_\infty$.