The Cardinal Abstraction for Quantitative Information Flow
Mounir Assaf, Julien Signoles, Eric Totel, Frédéric Tronel

To cite this version:

HAL Id: hal-01334604
https://inria.hal.science/hal-01334604
Submitted on 5 Aug 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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:

\[ \mathcal{M}_L = \log_2 |\text{outputs}|. \]

Smith’s framework considers a program as a channel that accepts only high inputs. Therefore, a program \( c \) that accepts low inputs can be seen as inducing a family of channels accepting only high inputs, and the maximum leakage of \( c \) 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:

\[ \mathcal{M}_L = \max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 |\text{outputs}_{\varrho_{L_0}}|. \]

Let us consider a command \( c \) that outputs no intermediate steps of computation. Let us also assume a set of variables \( v_1, v_2, \ldots, v_\ell \) attackers can observe at the end of command \( c \)’s execution. Thus, the outputs attackers may observe are solely determined by the final environments \( \varrho \) resulting from the execution of command \( c \):

\[ \text{outputs}_{\varrho_{L_0}} = \{ (\varrho(v_1), \varrho(v_2), \ldots, \varrho(v_\ell)) \mid \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } [c]_{\varrho_0} = \varrho \}. \]

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 \( \varrho_{L_0} \).

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 pro-
hibitively 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 \( \text{variety} \) – the cardinal
number of values variables may take – in order to upper-bound
the maximum leakage of programs accepting both low and
d 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 intro-
duces 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 summarise 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 approxima-
tion-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 con-
cludes.

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

\[
\begin{align*}
// & \{ s \mapsto 2^\kappa, \text{input} \mapsto 1, x \mapsto 1 \} \\
x & := s \mod 2; // \{ s \mapsto 2^\kappa, \text{input} \mapsto 1, x \mapsto 2 \} \\
x & := x + \text{input}; // \{ s \mapsto 2^\kappa, \text{input} \mapsto 1, x \mapsto 2 \}
\end{align*}
\]

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 \( 2^{\kappa} \) 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 \( 1 \, \text{bit} \). 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 \( \text{secret} \) is a high input, whereas variable \( \text{input} \)
is a low input. At both Lines 2 and 5, the analysis determines
whether 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 \([0, 2^n] \cup [0, 2^n], \leq, 0, 2^n, \max, \min\) 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 \([0, 2^n]\) 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.

```plaintext
6 // \{secret \mapsto 2^n, input \mapsto 1, x \mapsto 1\}
if (secret > input) {
  x := 0; // \{secret \mapsto 2^n, input \mapsto 1, x \mapsto 1\}
}
else {
  x := 1; // \{secret \mapsto 2^n, input \mapsto 1, x \mapsto 1\}
} // \{secret \mapsto 2^n, input \mapsto 2, x \mapsto 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.

```plaintext
6 // \{secret \mapsto \{\text{pp}_0\}, 2^n, input \mapsto \{\text{pp}_0\}, 1, x \mapsto \{\text{pp}_0\}, 1\}
if (secret > input) {
  x := 0; // \{secret \mapsto \{\text{pp}_0\}, 2^n\},
          input \mapsto \{\text{pp}_0\}, 1, x \mapsto \{\text{pp}_2\}, 1\}
}
else {
  x := 1; // \{secret \mapsto \{\text{pp}_0\}, 2^n\},
          input \mapsto \{\text{pp}_0\}, 1, x \mapsto \{\text{pp}_0\}, 1\}
} // \{secret \mapsto \{\text{pp}_0\}, 2^n, input \mapsto \{\text{pp}_0\}, 1, x \mapsto \{\text{pp}_2, \text{pp}_0\}, 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 \texttt{While} language \cite{33} introduced in Figure 1. Expressions include integers \(n\) of finite size \(\kappa\), variables \(id\), binary arithmetic operations \(\text{bop}\) as well as comparison operations \(\text{cmp}\). Commands are instructions identified by a unique program point \(pp \in \mathbb{P}\).

\begin{align*}
\text{Exp:} & \quad a ::= n \quad \text{(constants)} \\
& \quad \mid id \quad \text{(variables)} \\
& \quad \mid a_1 \text{bop} a_2 \quad \text{(binary operators)} \\
& \quad \mid a_1 \text{cmp} a_2 \quad \text{(comparison operators)} \\
\text{Com:} & \quad c ::= pp\text{skip} \quad \text{(empty instruction)} \\
& \quad \mid \text{ppid} := a \quad \text{(assignment)} \\
& \quad \mid c_1; c_2 \quad \text{(sequence)} \\
& \quad \mid \text{ppif} (a) c_1 \text{else} c_2 \quad \text{(conditional)} \\
& \quad \mid \text{ppwhile} (a) c \quad \text{(loop)}
\end{align*}

Fig. 1. Abstract syntax of \texttt{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 \(2^n\) 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 \(n \in [0, 2^n]\). 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 \([0, 2^n]\), 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

\[
(D_{\kappa}^{\mathcal{C}}, \subseteq_{\mathcal{C}}, \emptyset, 0, (\mathbb{P}, 2^n), \cup_{\mathcal{C}}, \cap_{\mathcal{C}})
\]

where:

\[
D_{\kappa}^{\mathcal{C}} \triangleq \mathcal{P}(\mathbb{P}) \times [0, 2^n] \\
\subseteq_{\mathcal{C}} \triangleq \subseteq \times \leq \\
\cup_{\mathcal{C}} \triangleq \cup \times \max \\
\cap_{\mathcal{C}} \triangleq \cap \times \min
\]

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 \( \text{id} \) may have been last assigned, whereas \( n \) represents the cardinal number of values variable \( \text{id} \) may take.

B. Abstract Semantics of Expressions

We denote by \( \varrho^e \in \text{Var} \rightarrow D^\ell_C \) an abstract environment that maps each variable \( \text{id} \) to an abstract value \( (\varrho_{\text{pp}}, n) \in D^\ell_C \). We also denote by \( \text{proj}_i(\cdot) \) the projection onto the \( i \)th component of a pair \( (i = 1, 2) \). Figure 2 introduces the abstract semantics of expressions \( A^x \in \text{Exp} \rightarrow (\text{Var} \rightarrow D^\ell_C) \rightarrow \{0, 2^n\} \).

\[
\begin{align*}
A^x[n]\varrho^e &= 1 \\
A^x[id]\varrho^e &= \text{proj}_2(\varrho^e(id)) \\
A^x[a_1 \text{ bop } a_2]\varrho^e &= \min (A^x[a_1]\varrho^e \times A^x[a_2]\varrho^e, 2^n) \\
A^x[a_1 \text{ mod } n]\varrho^e &= \min (A^x[a_1]\varrho^e, n) \\
A^x[a_1 \text{ cmp } a_2]\varrho^e &= \min (A^x[a_1]\varrho^e \times A^x[a_2]\varrho^e, 2^n) 
\end{align*}
\]

Fig. 2. Abstract semantics of expressions

The abstract semantics of expressions \( A^x \) evaluates an expression \( a \) in an abstract environment \( \varrho^e \), in order to yield a cardinal \( n \in \{0, 2^n\} \) representing the cardinal number of values expression \( a \) may evaluate to. Constant expressions always evaluate to a single value. Binary arithmetic operations \( a_1 \text{ bop } a_2 \) yield at most the product of the cardinal of \( a_1 \) and the cardinal of \( a_2 \), or \( 2^n \) when this product overflows. Comparison operations \( a_1 \text{ cmp } a_2 \) evaluate to at most 2 different values (true or false), and only 1 when both \( a_1 \) and \( a_2 \) 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 \( a_1 \text{ mod } n \) produces at most \( n \) different values or \( A^x[a_1]\varrho^e \) different values when \( a_1 \) 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 \( a_2 \) has a maximum value of 10 for instance, then the cardinal abstraction can soundly conclude that expression \( a_1 \text{ mod } a_2 \) may evaluate to at most 10 different values or \( A^x[a_1]\varrho^e \) different values when \( a_1 \) 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 \( \llbracket c \rrbracket^f \in (\text{Var} \rightarrow D^\ell_C) \rightarrow (\text{Var} \rightarrow D^\ell_C) \) evaluates a command \( c \) in an abstract environment \( \varrho^e \), then yields a new abstract environment.

Commands \( \text{pp}\text{skip} \) do not modify input abstract environments. Assignments \( \text{pp}\text{id} := a \) map the abstract value \( \{\text{pp}\}, A^x[a]\varrho^e \) to variable \( \text{id} \) since \( \text{id} \) can take as many values as expression \( a \), and is assigned at program point \( pp \). A sequence of commands \( c_1; c_2 \) composes the abstract semantics of the second command \( c_2 \) with the abstract semantics of the first one \( c_1 \).

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 \( 2^{\kappa} \) possible values at the entry program point \( pp_0 \).

When the cardinal abstraction determines that a variable \( \text{id} \) has at most 1 possible value, it also implicitly determines that variable \( \text{id} \) may only depend on low input variables. Indeed, if variable \( \text{id} \) depended on high input memories, then there would exist two low equivalent input memories that yield two different values for variable \( \text{id} \). Therefore, the cardinal abstraction would determine that variable \( \text{id} \) 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 \( \cup \) 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 \( \text{id} \) 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 \( \text{id} \) 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 \( \text{id} \) 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 \( \text{id} \) 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 \( \cup_{\text{add}(c_1, c_2)} \) to retain precision for the variables \( V \) that are assigned neither in \( c_1 \) nor in \( c_2 \), by computing the join \( \cup \) 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
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 \(\text{id}\) if \(PP(c) \cap s_{pp} = \emptyset\), supposing
\(s_{pp}\) over-approximates the set of program points where \(\text{id}\)
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 \(\text{id}\) 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 \((x \leq y)\) 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 \(\hat{\sqcup}_{\otimes}\) (resp. by \(\sqcup_{\otimes}\))
the pointwise lifting of the partial order relation \(\subseteq_{\otimes}\) (resp. of
the join operator \(\sqcup_{\otimes}\)) over environments:

- Partial order: \(\hat{g}_1^\otimes \sqcup_{\otimes} \hat{g}_2^\otimes \iff \forall \text{id}, \hat{g}_1^\otimes(\text{id}) \subseteq_{\otimes} \hat{g}_2^\otimes(\text{id})\)
- Join operator: \(\hat{g}_1^\otimes \sqcup_{\otimes} \hat{g}_2^\otimes = \lambda \text{id}. \hat{g}_1^\otimes(\text{id}) \sqcup_{\otimes} \hat{g}_2^\otimes(\text{id})\).

The abstract semantics of loops requires computing an abstract
environment \(\hat{g}^\otimes\) that is a loop invariant \(\hat{g}^\otimes\) greater
than the initial abstract environment \(g_0^\otimes\). This loop invariant \(\hat{g}^\otimes\)
satisfies the following fixpoint equation:

\[
\hat{g}^\otimes = g_0^\otimes \sqcup_{\otimes} \left(\left[PP\text{if } (a) c \text{ else } PP\text{skip}\right]^\otimes g^\otimes\right).
\]

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

\[
F^\otimes = \lambda g^\otimes. g_0^\otimes \sqcup_{\otimes} \left(\left[PP\text{if } (a) c \text{ else } PP\text{skip}\right]^\otimes g^\otimes\right).
\]

The least fixpoint of \(F^\otimes\) exists, since \(F^\otimes\) 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:

\[
\begin{align*}
x_0 &= g_0^\otimes \\
x_{n+1} &= g_0^\otimes \sqcup_{\otimes} \left(\left[PP\text{if } (a) c \text{ else } PP\text{skip}\right]^\otimes x_n\right).
\end{align*}
\]

The least fixpoint of \(F^\otimes\) is therefore equal to the limit \(x_{\infty}\)
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:

\[
\exists \theta_H : \Sigma_{H_0} : \theta_0 \triangleq \left(\theta_{L_0}, \theta_{H_0}\right) \text{ and } [c] \theta_0 = \theta.
\]

Recall that we denote by \(v_1, v_2, \ldots, v_f\) 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.
For deterministic batch-job programs, the maximum leakage (over-approximation of Theorem 1) is upper-bounded by:

\[ \mathcal{ML} \leq \log_2 \left( \prod_{1 \leq i \leq f} \text{proj}_2(\varrho_{0}^f(v_i)) \right) \]

where \( \varrho_{0}^f \) denotes the abstract environment computed at the end of the program, and \( v_1, v_2, \ldots v_f \) 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:

\[ \llbracket pp := a \rrbracket \varrho = \varrho(id \mapsto (pp, A[a] \varrho)) \]

Therefore, concrete environments \( \varrho \in \text{Var} \rightarrow \mathbb{P} \times \mathbb{V} \) 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 \( \kappa \). The denotation \( \llbracket e \rrbracket \varrho \) yields an environment \( \varrho' \). Additionally, we also denote by \( A[a] \varrho \) the evaluation of an expression \( a \) in an environment \( \varrho \), 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 erased 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 \( m^\sharp \hat{=} \{ x \mapsto 2 \} \) determining that a variable \( x \) may take at most 2 different values. What would be the concrete memories \( m \in \text{Var} \rightarrow \mathbb{V} \) 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 \( v \in \mathbb{V} \):

\[ \{ V \in \mathbb{P}(\mathbb{V}) : |V| \leq 2 \} \in \mathbb{P}(\mathbb{P}(\mathbb{V})) . \]

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

\[ R = \{ r \in \mathbb{P}(\text{Var} \rightarrow \mathbb{V}) : |\{ m(x) : m \in r \}| \leq 2 \} . \]

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 \( c \) 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 $\{c\}_c \in \mathcal{P} \left( \mathcal{P} \left( \mathcal{P} (\mathcal{P} (\mathcal{P} \times \forall)) \rightarrow \mathcal{P} (\mathcal{P} (\mathcal{P} (\mathcal{P} \times \forall))) \right) \rightarrow \mathcal{P} (\mathcal{P} (\mathcal{P} (\mathcal{P} \times \forall))) \right)$ of the concrete collecting semantics to functions $\partial_c$ of abstract environments.

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.

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 $v \in \forall$.

Equations (3) and (4) define a sound abstract semantics for expressions and commands, respectively. For the sake of pedagogical presentation, let us now postulate the existence of such a Galois connection, we can derive [43] a sound abstract semantics $\|\cdot\|_c \leq$ for expressions $\Lambda^d[a]$ of the concrete collecting semantics to functions $\partial_c$ of abstract environments. Then, in order to derive a sound abstract semantics for commands, we can consider the functional abstraction $\alpha^{\text{com}}$ defined in Equation (6), in order to transpose functions $\|c\|_c$ of the concrete collecting semantics to functions $\|c\|_c$ of the abstract semantics. Note that Equation (6) denotes by $\text{Env}$ the set $\mathcal{V}$ of all environments, and by $\text{Env}^d$ the set $\mathcal{V}$ of abstract environments.

Finally, note that $\alpha^{\text{com}}$ 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 $\|\cdot\|_c$ of commands over-approximates the functional abstraction $\alpha^{\text{com}}$ of the collecting semantics:

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

Additionally, the abstract semantics of expressions $\Lambda^d[a]$ is sound if it verifies the following condition:

Figure 4 summarizes the steps towards building the Galois connection $(\hat{\alpha}, \hat{\gamma})$ 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 $(\hat{\alpha}, \hat{\gamma})$ with the pointwise abstraction $(\hat{\alpha}_c, \hat{\gamma}_c)$ of values yields the desired Galois connection $(\hat{\alpha}, \hat{\gamma})$. 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 $\Lambda^d[a]$). The abstract semantics of expressions in Figure 2 is sound:

**Theorem 3** (Soundness of the abstract semantics $\|\cdot\|_c$). The abstract semantics of commands in Figure 3 is sound:

Both proofs can be found in the appendix.
that forgets about the program points:

\[ \alpha_{@L,H} \in (V a r \rightarrow D^\gamma_C) \mapsto (V a r \rightarrow \{L, H\}) \]

\[ \alpha_{@L,H}(\vartheta^t) \triangleq \lambda x. \alpha_{@L,H}(\pi_j(\vartheta^t(x))) \]

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 \( \alpha \), for all abstract environments \( \vartheta^t \in V a r \rightarrow D^\gamma_C \), for all type environments \( \Gamma \in V a r \rightarrow \{L, H\} \), such that: \( \check{\kappa}^t[\alpha] \vartheta^t = n \) and \( \Gamma \vdash \alpha : t \).

It holds that:

\[ \alpha_{@L,H}(\vartheta^t) \sqsubseteq \Gamma \implies \alpha_{@L,H}(\vartheta^t) \sqsubseteq \Gamma \]

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 \( \sqsubseteq \) 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 \( \vartheta^t_0, \vartheta^t \), and type environments \( \Gamma_0, \Gamma \) such that: \( \check{\kappa}^{\gamma_0}_c \vartheta^t_0 = \vartheta^t \) and \( \Gamma \vdash \Gamma_0[c] \Gamma \).

It holds that:

\[ \alpha_{@L,H}(\vartheta^t_0) \sqsubseteq \Gamma_0 \implies \alpha_{@L,H}(\vartheta^t) \sqsubseteq \Gamma \]
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 $\text{secret \ mod \ 1}$ 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 $\text{secret}$ 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 $\text{secret \ mod \ 1}$ as high $H$. Another corner case is the expression $\text{secret \ mod \ 0}$ 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 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 $v_1, v_2, \ldots, v_\ell$ that attackers may observe.

**Corollary 1** (Zero leakage for well-typed programs). For all commands $c$, for all abstract environments $\mathcal{G}_0, \mathcal{G}^\sharp$, and type environments $\Gamma_0, \Gamma$ such that: $\{c\}^{\sharp} \mathcal{G}_0 = \mathcal{G}^\sharp$, and $L \vdash \Gamma_0(c) \Gamma$, and for all $i \in [1, f]$, $\text{proj}_i(\mathcal{G}^\sharp(v_i)) \neq 0$.

It holds that:

$$\alpha_{\text{LR}}(\mathcal{G}_0^{\sharp}) \sqsubseteq \Gamma_0 \text{ and } (\forall i \in [1, f], \Gamma(v_i) = L) \Rightarrow \log_2 \left( \prod_{1 \leq i \leq f} \text{proj}_i(\mathcal{G}^\sharp(v_i)) \right) = 0.$$

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. 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 build on Denning’s work in order to propose a measure of information leakage, based on Shannon entropy 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. 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. propose a quantitative measure, based on information belief. 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, a pseudo-metric defined over probability distributions.

Smith 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. 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.

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

Alvim et al. 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.
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 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.

REFERENCES


---

\(^1\)https://github.com/moun/Fhamator
\(^2\)http://www.irisa.fr/celtique/teaching/PAS/while_analyser.tgz
This section builds the Galois connection assumed in Equation (5):

\[
\langle P (P (Var \to P \times V)) ; \subseteq, \emptyset, P (Var \to P \times V) \rangle \xrightarrow{\gamma} \langle P (Var \to P (P \times V)) ; \subseteq, \emptyset, \lambda x. \emptyset, \lambda x. P (2^V), \emptyset \rangle
\]

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:

\[
\alpha (r) \triangleq \lambda x. \{ q(x) \mid q \in r \}
\]

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

\[
\alpha_\emptyset (R) \triangleq \{ \alpha (r) \mid r \in R \}
\]

\[
= \{ \lambda x. \{ q(x) \mid q \in r \} \mid r \in R \}
\]

\[
\gamma_\emptyset (Q) \triangleq \{ r \mid \alpha (r) \in Q \}
\]

Therefore, we obtain a Galois connection:

\[
\langle P (P (Var \to P \times V)) ; \subseteq, \emptyset, P (Var \to P \times V) \rangle \xrightarrow{\gamma_\emptyset} \langle P (Var \to P (P \times V)) ; \subseteq, \emptyset, Var \to P (P \times V) \rangle
\]

The abstraction \(\alpha_\emptyset\) 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:

\[
R_0 \triangleq \{ [x \mapsto \{ pp_0, 0\}, y \mapsto \{ pp_0, 2\}], [x \mapsto \{ pp_0, 0\}, y \mapsto \{ pp_0, 3\}], [x \mapsto \{ pp_0, 1\}, y \mapsto \{ pp_0, 4\}], [x \mapsto \{ pp_0, 1\}, y \mapsto \{ pp_0, 5\}] \}
\]

Then, the abstraction \(\alpha_\emptyset (R_0)\) yields:

\[
\alpha_\emptyset (R_0) = \{ [x \mapsto \{ pp_0, 0\}, y \mapsto \{ pp_0, 2\}, \{ pp_0, 3\}], [x \mapsto \{ pp_0, 1\}, y \mapsto \{ pp_0, 4\}, \{ pp_0, 5\}] \}
\]

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

\[
\alpha^\emptyset (P) \triangleq \lambda x. \{ p(x) \mid p \in P \}
\]

\[
\gamma^\emptyset (Q) \triangleq \{ p \mid \alpha^\emptyset (p) \in Q \}
\]

Therefore, \((\alpha^\emptyset, \gamma^\emptyset)\) yields a Galois connection [31]:

\[
\langle P (Var \to P (P \times V)) ; \subseteq, \emptyset, Var \to P (P \times V) \rangle \xrightarrow{\gamma^\emptyset} \langle P (Var \to P (P \times V)) ; \subseteq, \emptyset, \lambda x. \emptyset, \lambda x. P (P \times V) \rangle
\]

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

\[
\langle P (P (Var \to P \times V)) ; \subseteq, \emptyset, P (Var \to P \times V) \rangle \xrightarrow{\gamma_r \circ \gamma_\emptyset} \langle P (Var \to P (P \times V)) ; \subseteq, \emptyset, \lambda x. \emptyset, \lambda x. P (P \times V) \rangle
\]

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

As we have defined a non-relational abstraction \((\alpha_r, \gamma_r)\) 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 \( \overline{\otimes}_x \) is given by:
\[
\overline{\otimes}_x \in \mathcal{P}(\mathcal{P}(\mathcal{P} \times \mathcal{V})) \rightarrow \mathcal{P}(\mathcal{P} \times \mathcal{P}(\mathcal{V}))
\]
\[
\overline{\otimes}_x(p) \triangleq (\Pi_1(p), \Pi_2(p))
\]
\[
\Pi_i(p) \triangleq \{ \text{proj}_i(x) | x \in p \}
\]

Therefore, we define an element-wise abstraction \( \alpha_{\overline{\otimes}_x} \) over a set of sets of pairs as follows:
\[
\alpha_{\overline{\otimes}_x}(P) \triangleq \{ \overline{\otimes}_x(p) | p \in P \}
\]
\[
= \{ (\Pi_1(p), \Pi_2(p)) | p \in P \}
\]
\[
\gamma_{\overline{\otimes}_x}(Q) \triangleq \{ p | \overline{\otimes}_x(p) \in Q \}
\]

Therefore, \((\alpha_{\overline{\otimes}_x}, \gamma_{\overline{\otimes}_x})\) defines a Galois connection:
\[
\langle \mathcal{P}(\mathcal{P}(\mathcal{P} \times \mathcal{V})); \subseteq, \emptyset, \mathcal{P}(\mathcal{P} \times \mathcal{V}), \cup, \cap \rangle \xrightarrow{\alpha_{\overline{\otimes}_x}} \mathcal{P}(\mathcal{P}(\mathcal{P} \times \mathcal{P}(\mathcal{V}))); \subseteq, \emptyset, \mathcal{P}(\mathcal{P} \times \mathcal{P}(\mathcal{V})), \cup, \cap \rangle \quad (9)
\]

Let us assume that \( P_0 \) is a set of sets of pairs defined as follows:
\[
P_0 \triangleq \{ \{(pp_0, 2), \{(pp_0, 3), \{(pp_0, 4), \{(pp_0, 5)\}\}\}\}\}
\quad (10)
\]

Then \( \alpha_{\overline{\otimes}_x}(P_0) \) is given by:
\[
\alpha_{\overline{\otimes}_x}(P_0) = \{ \{(pp_0, 2), \{(pp_0, 3)\}\}, \{(pp_0, 4), \{(pp_0, 5)\}\}\}
\]

In this latter set, we can further ignore relationships among values through an additional attribute independent abstraction:
\[
\alpha_x(Q) \triangleq (\Pi_1(Q), \Pi_2(Q))
\]
\[
\gamma_x((X, Y)) \triangleq X \times Y
\]

Therefore, we obtain a Galois connection \([43]\):
\[
\langle \mathcal{P}(\mathcal{P}(\mathcal{P} \times \mathcal{P}(\mathcal{V}))); \subseteq, \emptyset, \mathcal{P}(\mathcal{P} \times \mathcal{P}(\mathcal{V})), \cup, \cap \rangle \xrightarrow{\alpha_x} \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{V})))))); \subseteq, \emptyset, \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{V}))))), \cup, \cap \rangle \quad (11)
\]

with the component-wise ordering, join and meet:
\[
\subseteq_x \triangleq \subseteq \times \subseteq
\]
\[
\cup_x \triangleq \cup \times \cup
\]
\[
\cap_x \triangleq \cap \times \cap
\]

For instance, if we recall the example set \( P_0 \) defined in Equation (10), then \( \alpha_x \circ \alpha_{\overline{\otimes}_x}(P_0) \) is given by:
\[
\alpha_x \circ \alpha_{\overline{\otimes}_x}(P_0) = \alpha_x(\{(pp_0), \{(2, 3), (pp_0), \{(4, 5)\}\}\})
\]
\[
= \{(pp_0), \{(2, 3), (4, 5)\}\}\]
Therefore, we obtain a Galois connection:

$$\langle P (P (V)) ; \subseteq, \emptyset, P (P (V)), \cup, \cap \rangle \xleftarrow{\gamma_\gamma} \langle [0, 2^n] ; \leq, 0, 2^n, \text{max}, \text{min} \rangle$$

$$\langle P (P (P)) ; \subseteq, \emptyset, P (P (P)), \cup, \cap \rangle \xleftarrow{\gamma_{\alpha \gamma}} \langle P (P) ; \subseteq, \emptyset, P (P), \cup, \cap \rangle$$

Then, a component-wise abstraction \( (\alpha_0, \gamma_\emptyset) \) defined as follows:

$$\alpha_0((s_{pp}, s_v)) \triangleq (\alpha_{pp}(s_{pp}), \alpha_v(s_v))$$

$$\gamma_\emptyset((s_{pp}, n)) \triangleq (\gamma_{pp}(s_{pp}), \gamma_v(n))$$

yields a Galois connection:

$$\langle P (P (P)) \times P (P (V)) ; \subseteq, (\emptyset, \emptyset), P (P) \times P (V), \cup, \cap \rangle \xleftarrow{\gamma_{\emptyset}} \langle P (P) \times [0, 2^n] ; \subseteq_\emptyset, (\emptyset \times 0), (P \times 2^n), \cup_\emptyset, \cap_\emptyset \rangle \quad (12)$$

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

$$\langle P (P \times V) ; \subseteq, \emptyset, P (P \times V), \cup, \cap \rangle \xleftarrow{\gamma_\gamma} \langle P (P) \times [0, 2^n] ; \subseteq_\emptyset, (\emptyset \times 0), (P \times 2^n), \cup_\emptyset, \cap_\emptyset \rangle \quad (13)$$

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

**B0c Abstraction of program points:** The abstraction \(\alpha_{pp}\) merges all the sets of program points:

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

$$\gamma_{pp}(s_{pp}) \triangleq P(s_{pp})$$

Therefore, we obtain a Galois connection:

$$\langle P (P (P)) ; \subseteq, \emptyset, P (P), \cup, \cap \rangle \xleftarrow{\gamma_{\alpha_{pp}, \gamma_{pp}}} \langle P (P) ; \subseteq, \emptyset, P, \cup, \cap \rangle$$

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

$$\alpha_{pp}(S) \subseteq s_{pp} \iff \bigcup_{s \in S} s \subseteq s_{pp}$$

$$\iff \forall s \in S, s \in P(s_{pp})$$

$$\iff S \subseteq P(s_{pp})$$

$$\iff S \subseteq \gamma_{pp}(s_{pp}) \quad \square$$

**B0d Abstraction of concrete values:** The abstraction \(\alpha_v\) computes the maximum cardinal of values over the sets \(s_v \in S_v:\)

$$\alpha_v(S_v) \triangleq \max_{s_v \in S_v} |s_v|$$

$$\gamma_v(n) = \{ V \in P (V) \mid |V| \leq n \}$$

Therefore, \((\alpha_v, \gamma_v)\) is a Galois connection:

$$\langle P (P (V)) ; \subseteq, \emptyset, P (V), \cup, \cap \rangle \xleftarrow{\gamma_{\alpha_v, \gamma_v}} \langle [0, 2^n] ; \leq, 0, 2^n, \text{max}, \text{min} \rangle$$

Indeed, let us prove that \((\alpha_v, \gamma_v)\) is a Galois connection:

$$\alpha_v(S_v) \leq n \iff \max_{s_v \in S_v} |s_v| \leq n$$

$$\iff \forall s_v \in S_v, s_v \in P\alpha (V)$$

$$\iff S_v \subseteq \gamma_v(n) \quad \square$$
Abstract of Environments

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

Indeed, let us define a pointwise abstraction \((\alpha_c, \gamma_c)\) as follows:

\[
\alpha_c \triangleq \lambda f. \lambda x. \alpha(f(x)) \\
\gamma_c \triangleq \lambda f. \lambda x. \gamma(f^\sharp(x))
\]

Then, \((\alpha_c, \gamma_c)\) yields a Galois connection:

\[
\langle \mathcal{V} \times \mathcal{P} (\mathcal{P} \times \mathcal{V}) ; \subseteq, \lambda x. \emptyset, \lambda x. \mathcal{P} (\mathcal{P}, \emptyset, \cap) \rangle 
\xrightarrow{\gamma_c} \langle \mathcal{V} \times \mathcal{P} \times \mathcal{P} (\mathcal{P} \times \mathcal{V}), \cup, \cap \rangle
\]

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 \((\dot{\alpha}, \dot{\gamma})\):

\[
\dot{\alpha} \triangleq \alpha_c \circ \alpha_r \\
\dot{\gamma} \triangleq \gamma_r \circ \gamma_c
\]

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

\[
\langle \mathcal{P} (\mathcal{P} \times \mathcal{V}) ; \subseteq, \emptyset, \mathcal{P} (\mathcal{P} \times \mathcal{V}), \cup, \cap) \rangle 
\xrightarrow{\dot{\gamma}} \langle \mathcal{P} \times \mathcal{P} (\mathcal{P} \times \mathcal{V}) ; \subseteq, \lambda x. (\emptyset, 0), \lambda x. (\mathcal{P} \times \mathcal{P} (\mathcal{P} \times \mathcal{V})), \cup, \cap \rangle
\]

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 \(A^c[J_aK]\) of expressions, we can define the following functional abstraction [43]:

\[
\alpha^{\exp}_{exp} \in (\mathcal{P} (\mathcal{P} (\mathcal{E}n)) \rightarrow \mathcal{P} (\mathcal{P} (\mathcal{V}))) \mapsto (\mathcal{E}n^\sharp \rightarrow [0, 2^\kappa]) \\
\alpha^{\exp}_{exp}(\phi) \triangleq \alpha_v \circ \phi \circ \dot{\gamma}
\]

Therefore, the abstract semantics \(A^d[[a]]\) of expressions is sound wrt. the collecting semantics \(A_c[[a]]\) of expressions if:

\[
\alpha^{\exp}_{exp}(A_c[[a]]) \varrho^\sharp \leq A^d[[a]] \varrho^\sharp
\]

Theorem 2 (Soundness of the abstract semantics \(A^d[[a]]\)).

The abstract semantics of expressions in Figure 2 is sound:

\[
\alpha^{\exp}_{exp}(A_c[[a]]) \varrho^\sharp \leq A^d[[a]] \varrho^\sharp.
\]

Let us derive an abstract semantics \(A^d[[a]] R\) for expressions:

— Case \(a = n\):

\[
\alpha^{\exp}_{exp}(A_c[[n]]) \varrho^\sharp \triangleq \alpha_v \circ A_c[[n]] \circ \dot{\gamma}(\varrho^\sharp) \\
= \alpha_v \left( \{ \{ v \in \mathcal{V} \mid \exists \varrho \in r, A[[n]] \varrho = v \} \mid r \in \dot{\gamma}(\varrho^\sharp) \} \right) \\
= \alpha_v(\{ \{ n \} \}) \\
= 1 \\
\triangleq A^d[[n]] \varrho^\sharp
\]
— Case $a = \text{id}$:

$$α_{\exp}^\circ(\mathcal{C}_v[\text{id}])g^δ \triangleq α_v \circ [\text{id}]^\circ \circ \hat{σ}(g^δ)$$

$$= α_v (\{ v | \exists q ∈ r, \mathcal{A}[\text{id}]q = v \} | r ∈ \hat{σ}(g^δ))$$

$$= α_v (\{ \text{proj}(q(\text{id})) | \exists q ∈ r \} | r ∈ \hat{σ}(g^δ))$$

$$= (\text{By definition of } \hat{σ})$$

$$α_v \circ α_v (q^δ(\text{id}))$$

$$≤ (α_v \circ α_v \circ \hat{σ}^\circ(\text{id}))$$

$$= A^\circ[\text{id}]q^δ$$

— Case $a = a_1 \mod n$:

$$α_{\exp}^\circ(\mathcal{C}_v[a])g^δ \triangleq α_v (\{ v_1 \mod n | \exists q ∈ r, \mathcal{A}[a_1]q = v_1 \} | r ∈ \hat{σ}(g^δ))$$

$$= \min (α_v (\{ v_1 | \exists q ∈ r, \mathcal{A}[a_1]q = v_1 \} | r ∈ \hat{σ}(g^δ)), n)$$

$$= \min (α_v \circ \mathcal{C}_v[a_1] \circ \hat{σ}(g^δ), n)$$

$$≤ (\text{By induction hypothesis})$$

$$\min (A^\circ[a_1]g^δ, n)$$

$$= A^\circ[a_1 \mod n]g^δ$$

— Case $a = a_1 \text{ bop } a_2$:

$$α_{\exp}^\circ(\mathcal{C}_v[a_1 \text{ bop } a_2])g^δ \triangleq α_v (\{ v_1 \text{ bop } v_2 | \exists q ∈ r, \mathcal{A}[a_1]q = v_1$$

$$\land \mathcal{A}[a_2]q = v_2 \} | r ∈ \hat{σ}(g^δ))$$

$$≤ (\text{By loosing relationships among variables in } r)$$

$$α_v (\{ v_1 \text{ bop } v_2 | \exists q_1 ∈ r_1, \exists q_2 ∈ r_2, \mathcal{A}[a_1]q_1 = v_1$$

$$\land \mathcal{A}[a_2]q_2 = v_2 \} | r_1, r_2 ∈ \hat{σ}(g^δ))$$

$$= (\text{By definition of } α_v)$$

$$\max_{r_1, r_2 ∈ \hat{σ}(g^δ)} (\{ v_1 \text{ bop } v_2 | \exists q_1 ∈ r_1, \mathcal{A}[a_1]q_1 = v$$

$$\land q_2 ∈ r_2, \mathcal{A}[a_2]q_2 = v_2 \} | r_1, r_2 ∈ \hat{σ}(g^δ))$$

$$≤ (\text{values } v ∈ V \text{ are finite, of size } 2^κ)$$

$$\min (2^κ, \max_{r_1, r_2 ∈ \hat{σ}(g^δ)} (\{ v_1 | \exists q_1 ∈ r_1, \mathcal{A}[a_1]q_1 = v_1 \})$$

$$\times \{ v_2 | \exists q_2 ∈ r_2, \mathcal{A}[a_2]q_2 = v_2 \})$$

$$= \min (2^κ, \max_{r_1 ∈ \hat{σ}(g^δ)} (\{ v_1 | \exists q_1 ∈ r_1, \mathcal{A}[a_1]q_1 = v_1 \})$$

$$\times \max_{r_2 ∈ \hat{σ}(g^δ)} (\{ v_2 | \exists q_2 ∈ r_2, \mathcal{A}[a_2]q_2 = v_2 \}))$$

$$≤ (\text{By induction hypothesis in Equation (15)})$$

$$\min (2^κ, A^\circ[a_1]g^δ × A^\circ[a_2]g^δ)$$

$$\triangleq A^\circ[a_1 \text{ bop } a_2]g^δ$$

— Case $a = a_1 \text{ cmp } a_2$: this case is similar to the previous case, apart that expression $a_1 \text{ cmp } a_2$ may evaluate to only 2 different boolean values.

$$α_{\exp}^\circ(\mathcal{C}_v[a])g^δ \leq \min (2, A^\circ[a_1]g^δ × A^\circ[a_2]g^δ)$$

$$\triangleq A^\circ[a_1 \text{ cmp } a_2]$$
The abstract semantics of commands in Figure 3 is sound:
\[ \alpha_{\text{com}}(\llbracket c \rrbracket_c) q^\# \subseteq \llbracket c \rrbracket_c q^\#
\]

Similarly to expressions, we define the functional abstraction \( \alpha_{\text{com}} \) in order to soundly approximate the collecting semantics \( \llbracket c \rrbracket_c \) of commands.

\[ \alpha_{\text{com}} \in (\mathcal{P}(\mathcal{P}(\text{Env})) \rightarrow \mathcal{P}(\mathcal{P}(\text{Env}))) \rightarrow (\text{Env}^\# \rightarrow \text{Env}^\#) \]

Therefore, the abstract semantics of commands \( \llbracket c \rrbracket_c^\# \) is sound wrt. the collecting semantics \( \llbracket c \rrbracket_c \) of commands if:

\[ \alpha_{\text{com}}(\llbracket c \rrbracket_c) q^\# \subseteq \llbracket c \rrbracket_c q^\# \] (16)

Theorem 3 (Soundness of the abstract semantics \( \llbracket c \rrbracket_c^\# \)).

The abstract semantics of commands in Figure 3 is sound:

\[ \alpha_{\text{com}}(\llbracket c \rrbracket_c) q^\# \subseteq \llbracket c \rrbracket_c q^\# \]

Let us derive an abstract semantics for instructions:

— Case \( c = \text{pp}skip \):

\[ \alpha_{\text{com}}(\llbracket \text{pp}skip \rrbracket_c) q^\# = \hat{\alpha} \circ \llbracket \text{pp}skip \rrbracket_c \circ \hat{\gamma}(q^\#)
\]

\[ = \hat{\alpha} \circ \hat{\gamma}(q^\#)
\]

\[ \subseteq_\odot (\hat{\alpha} \circ \hat{\gamma} \text{ is reductive})
\]

\[ q^\# = \llbracket \text{pp}skip \rrbracket_c q^\#
\]

— For instruction \( c = \text{pp}id := a \):

\[ \alpha_{\text{com}}(\llbracket \text{pp}id := a \rrbracket_c) q^\# = \hat{\alpha} \circ \llbracket \text{pp}id := a \rrbracket_c \circ \hat{\gamma}(q^\#)
\]

\[ = \hat{\alpha}(\{ \{ q' \mid \exists q \in r, [id := a] q = q' \} \mid r \in \hat{\gamma}(q^\#) \})
\]

Hence, for all \( x \in \text{Var} \) such that \( x \neq id \):

\[ (\alpha_{\text{com}}(\llbracket \text{pp}id := a \rrbracket_c) q^\#)(x) \subseteq_\odot q^\#(x)
\]

Additionnally, for \( x = id \):

\[ (\alpha_{\text{com}}(\llbracket \text{pp}id := a \rrbracket_c) q^\#)(\text{id}) = (\bigcup_{r' \in [c], \gamma(\text{id})} \Pi_1(\hat{\alpha}(r')(\text{id})), \max_{r' \in [c], \gamma(\text{id})} \Pi_2(\hat{\alpha}(r')(\text{id})))
\]

\[ = (\{pp\}, \max_{r' \in [c], \gamma(\text{id})} \Pi_2(\{\{q(id) : q \in r'\}\}))
\]

\[ = (\{pp\}, \text{av} \circ \Lambda_a \llbracket a \rrbracket q^\# \circ \hat{\gamma}(q^\#))
\]

\[ \subseteq_\odot (\{pp\}, \Lambda_a \llbracket a \rrbracket q^\#
\]

Hence,

\[ \alpha_{\text{com}}(\llbracket \text{pp}id := a \rrbracket c) q^\# \subseteq_\odot q^\#(id \mapsto (pp, \Lambda_a \llbracket a \rrbracket q^\#))
\]

\[ = \llbracket \text{pp}id := a \rrbracket c q^\#
\]

— For conditional instructions \( c = \text{pp}if \ (a) \ c_1 \ \text{else} \ c_0 \):

\[ \alpha_{\text{com}}(\llbracket c \rrbracket_c) q^\# = \hat{\alpha}(\{ \{ q' \mid \exists q \in r, [c] q = q' \} \mid r \in \hat{\gamma}(q^\#) \})
\]

\[ = \hat{\alpha}(\{ \{ q' : \exists q \in r, \exists v \in \{0,1\}, A[a] q = v \} \mid r \in \hat{\gamma}(q^\#) \})
\]
First, if $A\llbracket a \rrbracket g^x = 1$, then expression $a$ evaluates to at most one value in each set $r \in \hat{\gamma}(g^x)$:

$$\forall r \in \hat{\gamma}(g^x), \exists v \in \{0, 1\}, \forall \theta \in r, A\llbracket a \rrbracket g = v$$

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

$$\alpha^\com_{\hat{\gamma}}([c_1]_e)g^x = \hat{\alpha} (\{ \{ g' \mid \exists \theta \in r_1, [c_1]_e \theta = g' \} \mid r_1 \in \hat{\gamma}(g^x) \})$$

$$\cup \{ \{ g' \mid \exists \theta \in r_0, [c_0]_e \theta = g' \} \mid r_0 \in \hat{\gamma}(g^x) \})$$

$$\subseteq (\hat{\alpha} \text{ preserves joins})$$

$$\alpha (\{ \{ g' \mid \exists \theta \in r_1, [c_1]_e \theta = g' \} \mid r_1 \in \hat{\gamma}(g^x) \})$$

$$\cup \alpha (\{ \{ g' \mid \exists \theta \in r_0, [c_0]_e \theta = g' \} \mid r_0 \in \hat{\gamma}(g^x) \})$$

$$= (\hat{\alpha} \circ [c_1]_e \circ \hat{\gamma}(g^x)) \cup (\hat{\alpha} \circ [c_0]_e \circ \hat{\gamma}(g^x))$$

$$\subseteq (\text{by induction hypothesis Equation (16)})$$

$$\llbracket [c_1]^x g^x \cup [c_0]^x g^x \rrbracket (x)$$

Second, if $A\llbracket a \rrbracket g^x > 1$, then for variables $x$ that are modified in neither $c_1$ nor $c_0$, we have:

$$(\alpha^\com_{\hat{\gamma}}([c_1]_e)g^x)(x) = ([c_1]^x g^x \cup [c_0]^x g^x) (x)$$

Finally, for variables that are modified in either $c_1$ or $c_0$:

$$(\alpha^\com_{\hat{\gamma}}([c_1]_e)g^x)(x) = \hat{\alpha} (\{ \{ g' \mid \exists \theta \in r_1, [c_1]_e \theta = g' \} \mid r_1 \in \hat{\gamma}(g^x) \}) (x)$$

$$\cup \{ \{ g' \mid \exists \theta \in r_2, [c_0]_e \theta = g' \} \mid r_2 \in \hat{\gamma}(g^x) \}) (x)$$

$$\subseteq (\hat{\gamma} \circ \hat{\alpha} \text{ is extensive, and } \hat{\alpha} \text{ is monotone})$$

$$\hat{\alpha} (\{ \{ g' \mid \exists \theta \in r_1 \} \cup \{ \{ g' \mid \exists \theta \in r_2 \} \mid$$

$$r_1 \in \hat{\gamma}(\llbracket [c_1]^x g^x \rrbracket), r_2 \in \hat{\gamma}(\llbracket [c_0]^x g^x \rrbracket) \}) (x)$$

$$\subseteq (\text{By hypothesis in Equation (16) and monotony of } \hat{\alpha})$$

$$\hat{\alpha} (\{ \{ g' \mid \exists \theta \in r_1 \} \cup \{ \{ g' \mid \exists \theta \in r_2 \} \mid$$

$$r_1 \in \hat{\gamma}(\llbracket [c_1]^x g^x \rrbracket), r_2 \in \hat{\gamma}(\llbracket [c_0]^x g^x \rrbracket) \}) (x)$$

Hence, the abstract semantics of conditionals is sound:

$$[\text{pp} \text{ if } (a) \text{ c}_1 \text{ else } c_2]^x g^x \triangleq \text{ let } n = A\llbracket a \rrbracket g^x \text{ in }$$

$$\text{ let } g_1^x = [c_1]^x g^x \text{ in }$$

$$\text{ let } g_2^x = [c_2]^x g^x \text{ in }$$

$$\lambda id. \begin{cases} g_1^x(id) \cup \emptyset g_2^x(id) & \text{if } n = 1 \\ g_1^x(id) \cup add(c_1,c_2) g_2^x(id) & \text{otherwise} \end{cases}$$
— For sequences $c_1; c_2$:

$$\alpha^\text{com}_\text{pp}(c_1; c_2)\vartheta^\sharp = \hat{\alpha} (\{| \varrho_2 : \exists \varrho \in r_1, \| c_1; c_2 \| \varrho = \varrho_2 : r \in \hat{\gamma}(\vartheta^\sharp)\})$$

$$= \hat{\alpha} (\{| \varrho_2 : \exists \varrho_1 \in r_1, \| c_2 \varrho_1 = \varrho_2 : r_1 \in \| c_1 \| \hat{\gamma} (\vartheta^\sharp)\})$$

$$\subseteq^\otimes \hat{\alpha} (\{| \varrho_2 : \exists \varrho_1 \in r_1, \| c_2 \varrho_1 = \varrho_2 : r_1 \in \hat{\gamma} \circ \hat{\alpha} (\| c_1 \| \hat{\gamma} (\vartheta^\sharp))\})$$

$$\subseteq^\otimes \hat{\alpha} (\{| \varrho_2 : \exists \varrho_1 \in r_1, \| c_2 \varrho_1 = \varrho_2 : r_1 \in \hat{\gamma} (\| c_1 \|^2 \varrho_1^\sharp)\})$$

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

$$\alpha^\text{com}_\text{pp}(\text{pp} \text{while} (a) c)\vartheta^\sharp = \hat{\alpha} \circ \| \text{pp} \text{while} (a) c \| \circ \hat{\gamma} (\vartheta^\sharp)$$

$$= (\text{By characterizing the collecting semantics for loops})$$

as a least fixpoint

with $F_{X_0} = \lambda X : X_0 \cup \| \text{pp} \text{if} (a) c \text{ else } \text{pp} \text{skip} \|_c )$

$$\hat{\alpha} \left( \text{lfp}_{\gamma^\sharp}^\subseteq (\hat{\gamma}(\vartheta^\sharp) \ F) \right)$$

$$\subseteq^\otimes \text{lfp}_{\vartheta^\sharp}^\subseteq \| \text{pp} \text{if} (a) c \text{ else } \text{pp} \text{skip} \|^\sharp$$
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 \( H \):

\[
\@_{LH} \in [0, 2^n] \to \{L, H\}
\]

\[
\@_{LH}(n) \triangleq \begin{cases} 
L & \text{if } n = 1 \\
H & \text{otherwise}
\end{cases}
\]

Let us consider a flow-sensitive type system such as the one presented in [5]. Such a type system considers a type environment \( \Gamma \in \text{Var} \to \{L, H\} \) 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:

\[
\Gamma \vdash a : t \iff \bigcup_{x \in \text{fv}(a)} \Gamma(x)
\]

Derivation rules for instructions have the form:

\[
\text{pc} \vdash \Gamma_0\{c\}\Gamma_1
\]

where \( \text{pc} \) is a security label attached to the program counter, \( \Gamma_0 \) is an input type environment, and \( \Gamma \) 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 \( \varphi^a \in \text{Var} \to D^c \), for all type environments \( \Gamma \in \text{Var} \to \{L, H\} \), such that:

\[
\varphi^a[a] \varphi^a = n \quad \text{and} \quad \Gamma \vdash a : t.
\]

It holds that:

\[
\alpha_{\@_{LH}}(\varphi^a) \sqsubseteq \Gamma \implies \@_{LH}(n) \sqsubseteq t
\]

**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:

\[
\varphi^a[n] \varphi^a = 1, \Gamma \vdash n : L \quad \text{and} \quad \@_{LH}(1) \subseteq L
\]

— Case \( a = \text{id} \):

Assumption \( \alpha_{\@_{LH}}(\varphi^a) = \Gamma \) implies that \( \@_{LH}(n) = \Gamma(\text{id}) \)

\[
\alpha_{\@_{LH}}(\varphi^a) = \Gamma \implies \@_{LH}(n) \sqsubseteq \Gamma(\text{id})
\]

— Case \( a = a_1 \text{ bop } a_2 \):

Let \( n_1 = \varphi^a[a_1] \varphi^a \) and \( n_2 = \varphi^a[a_2] \varphi^a \). 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 \( a_1 \) and \( a_1 \), assumption \( \alpha_{\@_{LH}}(\varphi^a) = \Gamma \) implies that:

\[
\@_{LH}(n_1) \sqsubseteq t_1 \quad \text{and} \quad \@_{LH}(n_2) \sqsubseteq t_2
\]

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

— Cases \( a = a_1 \text{ cmp } a_2 \) and \( a = a_1 \text{ mod } n \) are both similar to case \( a = a_1 \text{ bop } a_2 \).

### 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 \( \sqsubseteq \) 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 \( \varphi^c[0] \varphi^c \), and type environments \( \Gamma_0, \Gamma \) such that:

\[
\varphi^c[0] \varphi^c = \varphi^c \quad \text{and} \quad L \vdash \Gamma_0\{c\}\Gamma.
\]

It holds that:

\[
\alpha_{\@_{LH}}(\varphi^c[0]) \sqsubseteq \Gamma_0 \implies \alpha_{\@_{LH}}(\varphi^c) \sqsubseteq \Gamma
\]

**Proof.** By structural induction on instructions:

— Case \( c = \text{pp} \text{ skip} \):
\( \phi_0^c = \phi^c \) and \( \Gamma_0 = \Gamma \). Therefore, assumption.

\[ \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \implies \alpha_{\theta, \lambda} (\phi^c) \subseteq \Gamma \]

— Case \( c = \text{pp id} := a \):
Let \( n = A^\sharp[a] \phi_0^c \) and \( t \) such that \( \Gamma_0 \vdash a : t \). By Lemma 1:

\[ \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \implies \varpi_{\theta, \lambda} (n) \subseteq t \]

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

\[ \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \implies \alpha_{\theta, \lambda} (\phi^c) \subseteq \Gamma \]

— Case \( c = \text{pp if} (a) c_1 \text{ else } c_2 \):
Let \( n = A^\sharp[a] \phi_0^c \) and \( t \) such that \( \Gamma_0 \vdash a : t \).
Let \( \phi_1^c = [c_1]^\sharp \phi_0^c \) and \( \phi_2^c = [c_2]^\sharp \phi_0^c \).
Let \( \Gamma_1 \) and \( \Gamma_2 \) such that \( \Gamma = \Gamma_0 \{c_1\} \Gamma_1 \) and \( \Gamma = \Gamma_0 \{c_2\} \Gamma_2 \).
By induction on both \( c_1 \) and \( c_2 \), we have:

\[ \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \implies \alpha_{\theta, \lambda} (\phi_1^c) \subseteq \Gamma_1 \] and \( \alpha_{\theta, \lambda} (\phi_2^c) \subseteq \Gamma_2 \) (17)

- If \( n = 1 \) and \( t = L \), we have \( \phi^c = \lambda id. \phi_1^c (id) \cup \phi_2^c (id) \) and \( \Gamma = \lambda id. \Gamma_1 (id) \cup \Gamma_2 (id) \). Therefore, since the union operator \( \cup \) computes the maximum over cardinal values, Equation (17) implies that

\[ \alpha_{\theta, \lambda} (\phi_0^c) \subseteq \Gamma_0 \implies \alpha_{\theta, \lambda} (\phi^c) \subseteq \Gamma \]

- If \( n > 1 \), then we have \( \phi^c = \lambda id. \phi_1^c (id) \cup_{\text{add}(c_1, c_2)} \phi_2^c (id) \). Plus, assuming that \( \alpha_{\theta, \lambda} (\phi_0^c) \subseteq \Gamma_0 \), Lemma 1 implies that \( n = L \).
Let \( \Gamma_1 \) and \( \Gamma_2 \) such that \( \Gamma = \Gamma_0 \{c_1\} \Gamma_1 \) and \( \Gamma = \Gamma_0 \{c_2\} \Gamma_2 \).
Let also \( \Gamma' \) such that \( \Gamma' = \lambda id. \Gamma_1 (id) \cup \Gamma_2 (id) \). Note that the only difference between \( \Gamma_1 \) and \( \Gamma_1 \) (resp. \( \Gamma_2 \) and \( \Gamma_2 \)) is that variables that may be modified inside instruction \( c_1 \) (resp. instruction \( c_2 \)) are set to \( H \). Therefore the only difference between \( \Gamma' \) and \( \Gamma \) is that variables that may be modified in either conditional branches are set to \( H \).
Thus, assuming \( \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \), for variables \( \text{id} \) that are not modified inside the conditional branches, Equation (17) yields:

\[ \varpi_{\theta, \lambda} (\phi^c (id)) \subseteq \Gamma (id) = \Gamma' (id) \]

Moreover, for variables \( \text{id} \) that may be modified inside the conditional branches, since \( \Gamma' \) maps them to \( H \), it holds that

\[ \varpi_{\theta, \lambda} (\phi^c (id)) \subseteq \Gamma' (id) \]

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

\[ \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \implies \alpha_{\theta, \lambda} (\phi^c) \subseteq \Gamma \]

— Case \( c_1 = c_2 \):
By a first induction on \( c_1 \), then a second induction on \( c_2 \), we obtain:

\[ \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \implies \alpha_{\theta, \lambda} (\phi^c) \subseteq \Gamma \]

— Case \( \text{pp while} (a) c \):
Let us assume \( \alpha_{\theta, \lambda} (\phi_0^c) = \Gamma_0 \). The output abstract environment is given by:

\[ \phi^c = \text{fix} \left( A^\sharp \phi_0^c \cup \left( \{\text{pp if} (a) \text{ c else } \text{pp skip}\} \phi_0^c \right) \right) \]

Additionally, the output type environment is given by:

\[ \Gamma = \text{fix} \left( \lambda \Gamma_v. \text{let } L \cup \Gamma_v (a) \vdash \Gamma_v \{c\} \Gamma' \text{ in } \Gamma' \cup \Gamma_0 \right) \]

Or, rewritten differently:

\[ \Gamma = \text{fix} \left( \lambda \Gamma_v. \text{let } L \cup \Gamma_v \{if (a) c \text{ else } \text{skip}\} \Gamma' \text{ in } \Gamma' \cup \Gamma_0 \right) \]
Let the sequences \((\varrho^n)\) and \(\Gamma_n\) be defined as follows:

\[
\varrho_{n+1} = \varrho_0 \cup_{\otimes} (\llbracket \text{pp if } (a) \text{ c else pp skip} \rrbracket^{\varrho_n})
\]

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

Then we prove by recurrence over \(n\) that \(\forall n, \alpha_{\text{LH}}(\varrho^n) \sqsubseteq \Gamma_n\):

- For the initial case \(n = 0\), it holds by assumption that \(\alpha_{\text{LH}}(\varrho_0) = \Gamma_0\).
- Let us assume \(\alpha_{\text{LH}}(\varrho^n) \sqsubseteq \Gamma_n\), and prove \(\alpha_{\text{LH}}(\varrho_{n+1}) \sqsubseteq \Gamma_{n+1}\).

By the same proof that is presented in the case of conditionals, we prove that:

\[
\alpha_{\text{LH}}(\llbracket \text{pp if } (a) \text{ c else pp skip} \rrbracket^{\varrho_n}) \sqsubseteq \Gamma'
\]

Additionally, since \(\alpha_{\text{LH}}(\varrho_0) = \Gamma_0\), and the join operator \(\cup_{\otimes}\) compute the maximum over cardinals, we have:

\[
\alpha_{\text{LH}}(\varrho_0 \cup_{\otimes} \llbracket \text{pp if } (a) \text{ c else pp skip} \rrbracket^{\varrho_n}) \sqsubseteq \Gamma' \sqcup \Gamma_0
\]

Therefore \(\alpha_{\text{LH}}(\varrho_{n+1}) \sqsubseteq \Gamma_{n+1}\).

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