Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 388 (2007) 53–82
Theoretical Computer Science
www.elsevier.com/locate/tcs

Optimality and condensing of information flow through linear refinement

Fausto Spoto*

Dipartimento di Informatica, Università di Verona, Strada le Grazie, 15, 37134 Verona, Italy

Received 2 August 2006; received in revised form 3 May 2007; accepted 7 May 2007

Communicated by R. Gorrieri

Abstract

Detecting *information flows* inside a program is useful to check *non-interference* or *independence* of program variables, an important aspect of software security. In this paper we present a new abstract domain C expressing *constancy* of program variables. We then apply Giacobazzi and Scozzari's *linear refinement* to build a domain $\mathsf{C} \to \mathsf{C}$ which contains all input/output dependences between the constancy of program variables. We show that $\mathsf{C} \to \mathsf{C}$ is optimal, in the sense that it cannot be further linearly refined, and condensing, in the sense that a compositional, input-independent static analysis over $\mathsf{C} \to \mathsf{C}$ has the same precision as a non-compositional, input-driven analysis. Moreover, we show that $\mathsf{C} \to \mathsf{C}$ has a *natural* representation in terms of Boolean formulas, which is important since it allows one to use the efficient binary decision diagrams in its implementation. We then prove that $\mathsf{C} \to \mathsf{C}$ coincides with Genaim, Giacobazzi and Mastroeni's IF domain for information flows and with Amtoft and Banerjee's Independ domain for independence. This lets us extend to IF and Independ the properties that we proved for $\mathsf{C} \to \mathsf{C}$: optimality, condensing and representation in terms of Boolean formulas. As a secondary result, it lets us conclude that IF and Independ are actually the same abstract domain, although completely different static analyses have been based on them.

Keywords: Information flow; Linear refinement; Abstract interpretation; Static analysis

1. Introduction

Language-based security is recognised as an important aspect of modern programming languages design and implementation [17]. One of its aspects is *non-interference* or *independence*, which determines the pairs of program variables that do not affect each other's values during the execution of a program. From non-interference it is then possible to study the confinement of confidential information injected in the program through *some* input variables, usually called *high* variables. The other variables are called *low* variables. The intuition here is that high variables are allowed to contain information related to some *secret*, such as credit card codes or medical or industrial reserved knowledge. This notion of *confidentiality* as non-interference has been introduced in [8]. It assumes that an attacker can only observe low variables, so that confidential information injected from high variables must not affect the final value

* Tel.: +39 3204352527; fax: +39 0458027068.
E-mail address: fausto.spoto@univr.it.

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.05.004

of low variables (*i.e.,* their value at the end of the execution of a statement). For instance, in the statement $\mathtt{y} := \mathtt{x} + 1$ the initial value of y is non-interfering with the final value of y, while the initial value of x is interfering with the final values of y and x itself, since the final value of y is the initial value of x plus 1 and the final value of x is exactly the initial value of x. In this situation, it is normally required that if x is a high variable then also y is a high variable. It is also admitted that x is low while y is high, but it is forbidden that y is low and x is high, because this would allow information from a high variable to flow into a low variable. In the example above there is an *explicit* link between the values of x and y. This is different from what happens in the statement $\mathtt{if}\ \mathtt{y} = 0\ \mathtt{then}\ \mathtt{x} := 4\ \mathtt{else}\ \mathtt{x} := 5$, where the initial value of y is interfering with the final value of x while the initial value of x is non-interfering with the final value of x. In this case, the link between y and x is *implicit*, in the sense that it is a consequence of the use of y in the guard of a conditional whose branches affect the final value of x.

Non-interference or independence is frequently implemented over an *information-flow* analysis, often based on a type system, which tracks the flows of information in a program [22,16,18,5,17,9]. From this set of flows it is then possible to check non-interference of program variables. A more direct approach consists in tracking, explicitly, the pairs of independent variables, as done in [1].

Information flows as well as pairs of independent variables in a program can be computed through Cousot and Cousot's abstract interpretation [6]. The idea underlying abstract interpretation is that of executing the program over a *description* of the actual, concrete data. In our case this description should be somehow related to the high or low security level of each variable. This description is called *abstract domain*. One such domain is called IF in this paper and has been defined by Genaim, Giacobazzi and Mastroeni [9] to represent sets of flows in a program. Abstract interpretation consists here in executing the program over the description of the concrete data as provided by IF. Correctness states that if a program features a flow, then this flow must be included in the set of flows that the analysis computes. The domain IF has been implemented by using Boolean formulas. This leads to an efficient analysis [10], already implemented for the whole Java bytecode. The analysis uses the efficient binary decision diagrams [4] to implement such formulas. These are data structures which represent Boolean functions in an efficient way, so that logical operations over those functions can be performed quickly. Another abstract domain developed through abstract interpretation, this time however modelling independence of pairs of variables, is the domain Independ of [3]. It is used to decorate the program with Hoare-like assertions. Correctness means here that if two variables are deemed independent by the analysis then they are actually independent in the program.

Both the analyses in [9] and [3] are *compositional*, in the sense that the analysis of a complex command is derived from the analysis of its subcommands, in a bottom-up fashion. This is both elegant and efficient, since the analysis of a piece of code which is called from many program points, such as a library function, can be performed just once and then *plugged* in each calling context. This is sensible since both IF and Independ *seem* to enjoy some form of *input-independency*. That is, the analysis of a piece of code can be performed only once, without any assumption on the input provided to the program. Those input variables which contain confidential information can be specified *after* the analysis is performed, when the result of the analysis of the piece of code is plugged into the calling contexts. An *input-driven* analysis, instead, would require the input to be available *before* the analysis, which must then be re-executed, in a top-down rather than bottom-up fashion, for each different input, so that it is not possible to analyse a library independently from the applications that use it. It was not clear, up to now, if this input-independency was achieved at the price of precision or not. If precision is not sacrified, then those abstract domains deserve the qualification of *condensing* [12]. We exemplify these notions of compositionality, input-independency and condensing in Section 2. We provide their formal definitions in Section 6.

In this paper we define a new abstract domain C which expresses sets of variables which are constant: their value is the same whichever input is provided to the program. We then use Giacobazzi and Scozzari's *linear refinement* [13] to build a domain $\mathsf{C} \to \mathsf{C}$ which expresses *propagation* of constancy inside a computer program. We hence prove that $\mathsf{C} \to \mathsf{C}$ cannot be further linearly refined, which means that $\mathsf{C} \to \mathsf{C}$ is *optimal i.e.,* it contains all possible dependences of constancy. Moreover, we prove that this also entails that $\mathsf{C} \to \mathsf{C}$ is condensing, by extending to imperative programs a similar result already proved in [12] for logic programs. Then we show that there is a simple *normal* form for the elements of $\mathsf{C} \to \mathsf{C}$ in terms of Boolean formulas. These results are then lifted to IF and Independ by showing that they both coincide with $\mathsf{C} \to \mathsf{C}$ and hence with each other.

In conclusion, the contribution of this paper can be summarised as follows:

– we show that both IF and Independ are reformulations of a domain $\mathsf{C} \to \mathsf{C}$ for constancy propagation;

– we show that IF and Independ are actually the same domain, although they have been used in radically different static analyses;
– we extend to imperative programs the result that abstract domains closed by linear refinement are condensing, so that an input-indendent static analysis and an input-driven one coincide;
– we show that both IF and Independ are condensing and optimal w.r.t. linear refinement;
– we show that both IF and Independ can be represented in terms of Boolean formulas, which is interesting in view of an implementation based on the efficient binary decision diagrams [4].

The rest of this paper is organised as follows. Section 2 discusses, informally, the notions of compositionality, input-independency and condensing. Section 3 presents the preliminaries, the abstract interpretation notions used in the paper and the domain C for constancy. Section 4 formalises the abstract domain IF which expresses sets of information flows. Section 5 defines the abstract domain Independ which expresses sets of pairs of independent variables. Section 6 defines concrete and abstract non-compositional (*operational*) and compositional (*denotational*) semantics for imperative programs and shows that they coincide for abstract domains which satisfy the condensing property. Section 7 introduces the linear refinement technique and shows that if an abstract domain is closed by linear refinement then it is condensing. Section 8 proves that the linear refinement $\mathsf{C} \to \mathsf{C}$ of C is closed w.r.t. further linear refinements and is hence optimal and condensing. Section 9 provides a representation of the elements of $\mathsf{C} \to \mathsf{C}$ in terms of Boolean formulas. Section 10 proves that IF coincides with $\mathsf{C} \to \mathsf{C}$ and hence inherits its properties of being optimal and condensing and has a representation in terms of Boolean formulas. Section 11 proves that Independ coincides with $\mathsf{C} \to \mathsf{C}$, and hence with IF, so that it enjoys the properties of being optimal and condensing and has a representation in terms of Boolean formulas. Section 12 presents related work and concludes.

A preliminary and partial version of this paper appeared in [21].

2. Compositionality, input-independency and condensing

In this section we discuss the notions of compositionality, input-independency and condensing.

Consider the following program:

```
void main() {
  float h1 := readHighFloat();
  float l1 := readLowFloat();
  float l2 := readLowFloat();
  int h2 := hashcode(h1,l1);
  printInt(hashcode(l1,l2));
}

int hashcode(float f1, float f2) {
  return ((int)f1 + (int)f2) mod 100;
}
```

The function `main` reads three floating point numbers from the keyboard. We assume that the high read is performed without presenting any feedback to the screen, as if a password were entered. Then `main` computes two hashcodes of pairs of floating point numbers and prints the second hashcode. We assume that `h1` and `h2` are high, secret variables, so that for instance they are stored inside a memory which is not accessible from the external environment; variables `l1` and `l2` are instead low, so that their value can be safely read from the external environmnent, for instance by an attacker. Also the message printed on the screen is accessible from the external environment. We can say that the secret information contained in `h1` is *confined* inside this program since that information, and any information derived from it, does not flow into any low variable and is not printed on the screen.

If we had to type each variable with a fixed security level (high or low) then we can only type `f1` as high since, in the two successive calls to `hashcode`, the first parameter is first `h1`, high, and then `l1`, low. Thus the only consistent choice is to assume high. But then the return value of `hashcode` can only be typed as high, since it is the result of a computation which involves the high variable `f1`. As a consequence, the assignment to `h2` inside `main` is legal but printing on the screen a high value is recognised as illegal. It is obvious, however, that the information printed here on

the screen is not related to any high variable, so that this printing should be legal. The problem, here, is that typing a library function (such as `hashcode`) with fixed security levels reduces the precision of the type system as soon as the library function is called from many different contexts.

The previous problem has been solved in [9] and [3] by using compositional, input-independent typings (called *abstract denotations* in [9]). For instance, in [9] the function `hashcode` gets typed with the set of information flows $\texttt{f1} \rightsquigarrow r, \texttt{f2} \rightsquigarrow r$, where r is a pseudo-variable which stands for the result value of `hashcode`. Those flows state that the result value of `hashcode` holds secret information if at least one between `f1` and `f2` holds secret information at the beginning of the execution of the function `hashcode`. This typing can then be *plugged* (with suitable variable renamings) into each calling context of `hashcode` inside `main` and the analysis of the latter can be performed compositionally (or *bottom-up*) as the composition of the typings (abstract denotations) of each statement in `main`. Since the typing of `hashcode` is now parametric, we are able to verify that the assignment to `h2` is legal since that variable takes there a high value (the parameter `h1` is high); we can also verify that printing on the screen the result value of the second call to `hashcode` is legal, since that second call yields a low value (both the parameters `l1` and `l2` are low). In [3] the authors use parametric Hoare-like assertions instead of an abstract domain of information flows. In any case, parametric typings mean that the analysis of functions such as `hashcode` is *input-independent*, in the sense that no hypothesis is made about the input to the function: the result of the analysis must be parametric w.r.t. that input, which is only provided later, when the analysis is plugged into each calling context. Moreover, these parametric typings can be efficiently implemented through binary decision diagrams [4], which are data structures which implement Boolean formulas and operations over Boolean formulas in an efficient way. For instance, the set of flows $\texttt{f1} \rightsquigarrow r, \texttt{f2} \rightsquigarrow r$ seen above can be represented by the Boolean formula $(\texttt{f1} \Rightarrow r) \wedge (\texttt{f2} \Rightarrow r)$, implemented efficiently as a binary decision diagram. This is the idea underlying the implementation in [10].

A compositional, input-independent analysis has been hence defined in [9] and [3] since it is more precise, elegant and efficient than an analysis based on fixed types. It must be noted, however, that at least the same precision should be attainable by executing the program, starting from the beginning of `main`, by using a type environment which binds each variable to its current security level. For instance, we start the analysis of `main` from the empty type environment [] and we proceed through the `read` statements yielding the type environment $[\texttt{h1} \mapsto \mathit{high}, \texttt{l1} \mapsto \mathit{low}, \texttt{l2} \mapsto \mathit{low}]$. We now analyse the first call to `hashcode` by using the last type environment as a calling context. The result value is then high and gets assigned to `h2`, yielding the type environment $[\texttt{h1} \mapsto \mathit{high}, \texttt{h2} \mapsto \mathit{high}, \texttt{l1} \mapsto \mathit{low}, \texttt{l2} \mapsto \mathit{low}]$. We then analyse the second call to `hashcode` by using the last type environment as a calling context. We can conclude that the result value of `hashcode` is low here, so that it is safe to print it on the screen. This other approach to the verification of `main` needs, for more complex programs, some form of *memoisation* or *caching* in order to guarantee termination. Moreover, it analyses `hashcode` twice, with two different calling contexts; it cannot be applied to `hashcode` alone, but only to the whole program; it is not compositional. As a consequence, we think that this second approach, called *operational* or *top-down* or *input-driven*, is less attractive than those in [9] and [3], called *denotational* or *bottom-up* or *input-independent*.

The only motivation for the use of an input-driven analysis is that it can be, in principle, more precise than an input-independent one. This happens when parametric typings are not able to express *all* the input/output dependences featured by a program function. In terms of the theory of abstract domains developed in abstract interpretation, this means that the set of parametric typings (the *abstract domain*) is not closed w.r.t. linear refinement, which is the domain counterpart of input/output dependence [13]. It follows that for those domains which are instead closed w.r.t. linear refinement, an input-driven analysis has exactly the same precision as an input-independent one (*condensing* property), as proved in [12] for the case of logic programs; in Section 6 we extend this result to imperative programs. The latter domains are hence considered *optimal*, in the sense that they cannot be further enriched with input/output dependences. Up to now, there were no proofs of optimality and condensing for the parametric typings (the abstract domains) used in [9] and [3]. This proof is one of the goals of this paper.

3. Preliminaries

3.1. Functions and ordered sets

We denote a total function by $\mapsto$ and a partial function by $\rightarrow$. We denote the *domain* of a function f by $dom(f)$. We let $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ denote the function f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$.

Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ we denote the *restriction* of f to $s \subseteq dom(f)$. A *poset* is a set S with a reflexive, transitive and antisymmetric relation $\leq$. An *upper (respectively, lower) bound* of $S' \subseteq S$ is an element $u \in S$ such that $u' \leq u$ (respectively, $u' \geq u$) for every $u' \in S'$. A *complete lattice* is a poset where *least* upper bounds ($\sqcup$) and *greatest* lower bounds ($\sqcap$) always exist. The *cardinality* of a set S is denoted by $\#S$.

3.2. Denotations and constancy

We model the *state* of an interpreter of a computer program at a given program point as a function from variables to *values*. We consider integers as values, but any other domain of values would do.

Definition 1 (*State*). Let V be a finite set of variables (this will be assumed in the rest of the paper). A *state* over V is a total function from V into integer *values*. The set of states over V is Σ_V, where V is often omitted. □

Example 1. An example of state $\sigma \in \Sigma_V$ is such that $\sigma(v) = 3$ for each $v \in V$. □

In general, we write variables in italic such as v above. However, we write `v` when we want to refer to a variable in a given program or command.

A denotational semantics associates a *denotation* to each piece of code *i.e.,* a function from input states to output states. Possible divergence is traditionally modelled by using partial functions as denotations. We prefer here to use a distinguished constant *undefined* which states explicitly when a denotation is undefined. This simplifies some definitions such as Definition 6 and some proofs.

Definition 2 (*Denotation*). A *denotation* over V is a partial function $\delta : \Sigma_V \to \Sigma_V$. The set of denotations is Δ_V, where V is usually omitted. Let $\sigma \in \Sigma_V$. If $\delta(\sigma)$ is not defined, then we let $\delta(\sigma) = \mathit{undefined}$. If $\delta(\sigma) = \mathit{undefined}$ then for every $x \in V$ we define $\delta(\sigma)(x) = \mathit{undefined}$. This lets us consider denotations as total maps from now on. Denotations δ_1 and δ_2 are *sequentially composed* into $\delta_1; \delta_2$ by defining $(\delta_1; \delta_2)(\sigma) = \delta_2(\delta_1(\sigma))$ if $\delta_1(\sigma) \neq \mathit{undefined}$; otherwise we let $(\delta_1; \delta_2)(\sigma) = \mathit{undefined}$. □

Denotations can be used to define both an *operational* and a *denotational* semantics for a program, as we will see in Section 6.

Example 2. The denotation for the assignment $\mathtt{y} := \mathtt{x} + 1$ is δ_1 such that $\delta_1(\sigma) = \sigma[\mathtt{y} \mapsto \sigma(\mathtt{x}) + 1]$ for all $\sigma \in \Sigma$. That is, the successor of the input or *initial* value of `x` is stored in the output or *final* value of `y`. The other variables are not modified. □

Example 3. The denotation of the assignment $\mathtt{x} := 4$ is δ_2 such that $\delta_2(\sigma) = \sigma[\mathtt{x} \mapsto 4]$ for all $\sigma \in \Sigma$. That is, the output value of `x` is constantly bound to 4. The other variables are not modified. □

Example 4. The denotation of

```
if y = 0 then x := 4 else while true do skip
```

is δ_4, *compositionally* defined as

$$\delta_4(\sigma) = \begin{cases} \delta_2(\sigma) & \text{if } \sigma(\mathtt{y}) = 0 \\ \delta_3(\sigma) & \text{if } \sigma(\mathtt{y}) \neq 0, \end{cases}$$

where δ_2 is the denotation of $\mathtt{x} := 4$ (Example 3) and δ_3 is the denotation of `while true do skip`, which is *always* undefined. □

Example 5. The denotation of

```
if y = 0 then x := 4 else x := 5
```

is δ_5, *compositionally* defined as

$$\delta_5(\sigma) = \begin{cases} \delta_2(\sigma) & \text{if } \sigma(\mathtt{y}) = 0 \\ \delta_6(\sigma) & \text{if } \sigma(\mathtt{y}) \neq 0, \end{cases}$$

where δ_2 is the denotation of $\mathtt{x} := 4$ (Example 3) and δ_6 is the denotation of $\mathtt{x} := 5$, that is $\delta_6(\sigma) = \sigma[\mathtt{x} \mapsto 5]$. We conclude that

$$\delta_5(\sigma) = \begin{cases} \sigma[\mathtt{x} \mapsto 4] & \text{if } \sigma(\mathtt{y}) = 0 \\ \sigma[\mathtt{x} \mapsto 5] & \text{if } \sigma(\mathtt{y}) \neq 0. \end{cases} \quad \square$$

Example 6. The denotation of $\mathtt{x} := 4;\ \mathtt{y} := \mathtt{x} + 1$ is the functional composition $\delta_2; \delta_1$ (Examples 2 and 3) which is such that

$$(\delta_2; \delta_1)(\sigma) = \sigma[\mathtt{x} \mapsto 4, \mathtt{y} \mapsto 5]$$

for every $\sigma \in \Sigma$. In general, ; is the semantical counterpart of the syntactical sequential composition of commands. $\square$

Constancy is a property of denotations. Namely, a variable v is *constant* in a denotation δ when δ always binds v to the same value.

Definition 3 (*Constancy*). Let $\delta \in \Delta$. The set of variables which are *constant* in δ is

$$const(\delta) = \left\{ v \in V \;\middle|\; \begin{array}{l} \text{for all } \sigma_1, \sigma_2 \in \Sigma \\ \text{we have } \delta(\sigma_1)(v) = \delta(\sigma_2)(v) \end{array} \right\}. \quad \square$$

Example 7 (*Constancy*). The denotation δ_1 of Example 2 copies $\mathtt{x} + 1$ into $\mathtt{y}$. Hence $const(\delta_1) = \varnothing$. The denotation δ_2 of Example 3 binds $\mathtt{x}$ to 4. Then $const(\delta_2) = \{\mathtt{x}\}$. The denotation δ_4 of Example 4 always binds $\mathtt{x}$ to 4 whenever it is defined. However, it can also be undefined. For instance, when $\sigma(\mathtt{y}) = 1$ we have $\delta(\sigma)(\mathtt{x}) = \mathit{undefined} \neq 4$. Hence $\mathtt{x} \notin const(\delta_4)$. $\square$

Constancy is closed w.r.t. composition of denotations.

Lemma 1 (*Constancy is Closed by Composition*). *Let $\delta, \overline{\delta} \in \Delta$ and $v \in V$. If $v \in const(\delta)$ then $v \in const(\overline{\delta}; \delta)$.*

Proof. Let $\sigma_1, \sigma_2 \in \Sigma_V$. Since $v \in const(\delta)$, we have $(\overline{\delta}; \delta)(\sigma_1)(v) = \delta(\overline{\delta}(\sigma_1))(v) = \delta(\overline{\delta}(\sigma_2))(v) = (\overline{\delta}; \delta)(\sigma_2)(v)$ *i.e.,* $v \in const(\overline{\delta}; \delta)$. $\square$

3.3. Abstract domains and abstract interpretation. The abstract domain C

Let C be a complete lattice w.r.t. $\leq$, playing the role of the *concrete* domain of computation. For instance, in this paper C will be the powerset $\wp(\Delta)$ of the *concrete* denotations of Section 3.2, ordered by set-inclusion. Each element of C is an *abstract property*. For instance, the set of concrete denotations which bind $\mathtt{x}$ to 4 is an element of $\wp(\Delta)$ expressing the property: "x holds 4 in the output of the denotation". An *abstract domain* A is a collection of abstract properties *i.e.,* a subset of C. This is exemplified below for the case of an abstract domain C for constancy.

Definition 4 (*The Abstract Domain* C). Let us use $\wp(\Delta_V)$, ordered by set-inclusion, as concrete domain and let

$$\mathbf{v_1} \cdots \mathbf{v_n} = \{\delta \in \Delta_V \mid v_i \in const(\delta) \text{ for } 1 \leq i \leq n\}.$$

The abstract element $\mathbf{v_1} \cdots \mathbf{v_n}$ is hence a *notation* or *representation* for a set of concrete denotations. This set does not change by changing the ordering of the variables in $\mathbf{v_1} \cdots \mathbf{v_n}$. An abstract domain of $\wp(\Delta)$ is

$$\mathsf{C}_V = \{\mathbf{v_1} \cdots \mathbf{v_n} \mid \{v_1, \ldots, v_n\} \subseteq V\},$$

where V is usually omitted. Since C_V is a subset of $\wp(\Delta)$, it is ordered by set-inclusion. Equivalently, the representations of its elements, written through the notation $\mathbf{v_1} \cdots \mathbf{v_n}$, are ordered by inverse inclusion of sets of variables. The top element is represented by the empty set of variables, written as $\mathbf{O}$ to avoid confusion with $\varnothing$. The abstract domain C_V expresses the properties of *being constant* for a set of variables in a denotation. $\square$

Example 8. From Example 7 we conclude that $\delta_1 \in \mathbf{O}$ and $\delta_2 \in \mathbf{x}$. However, $\delta_2 \notin \mathbf{xy}$ since $\mathtt{y}$ is not constant in δ_2 (Example 3). $\square$

Abstract interpretation theory [6] requires the abstract domain A to be meet-closed, which guarantees the existence in A of a *best approximation* $\rho(c)$ for each element of $c \in C$. That is, A must be a *Moore family* of C *i.e.,* a complete meet-sublattice of C (for any $Y \subseteq A$ we have $\sqcap Y \in A$). Note that A is not, in general, a complete sublattice of C, since the join over A might be different from that over C. The function ρ is an *upper closer operator*:

- *monotonic*: for every $c_1, c_2 \in C$, if $c_1 \leq c_2$ then $\rho(c_1) \leq \rho(c_2)$;
- *expansive*: for every $c \in C$ we have $\rho(c) \geq c$;
- *idempotent*: for every $c \in C$ we have $\rho(\rho(c)) = \rho(c)$.

It is also known as the *abstraction function* from C to A. It is induced by A as follows: for every $c \in C$ we have

$$\rho(c) = \sqcap\{a \in A \mid a \geq c\}.$$

Proposition 1 (C *is an Abstract Domain*). *The set* C *of Definition 4 is closed w.r.t. intersection i.e., the* $\sqcap$ *operation on* $\wp(\Delta)$. *Hence* C *deserves the name of* abstract domain.

Proof. We have $(\mathbf{v_1} \cdots \mathbf{v_n}) \cap (\mathbf{w_1} \cdots \mathbf{w_m}) = \mathbf{x_1} \cdots \mathbf{x_p}$, where

$$\{x_1, \ldots, x_p\} = \{x \mid x \in \{v_1, \ldots, v_n\} \text{ or } x \in \{w_1, \ldots, w_m\}\}. \quad \square$$

For any $X \subseteq C$, we denote by $\curlywedge X = \{\sqcap I \mid I \subseteq X\}$ the *Moore closure* of X *i.e.,* the least Moore family of C containing X. Hence the operation $\curlywedge$ *constructs* the smallest abstract domain which includes the set of properties X. An example of use of $\curlywedge$ is in Definition 5.

Definition 5 (*Constancy as a Moore Closure*). We write the set of denotations where x is constant as

$$\mathbf{x} = \{\delta \in \Delta \mid x \in const(\delta)\}.$$

The abstract domain of Definition 4 can be constructed as

$$\mathsf{C}_V = \curlywedge\{\mathbf{x} \mid x \in V\}.$$

We write the elements of C as $\mathbf{v_1} \cdots \mathbf{v_n}$, standing for $\cap\{\mathbf{v_i} \mid 1 \leq i \leq n\}$. If $vs \subseteq V$ then by $\mathbf{vs}$ we mean $\cap\{\mathbf{v} \mid v \in vs\}$. $\square$

Once an abstract domain A is defined, abstract interpretation theory provides the abstract semantics induced by each given concrete semantics. It is constructed from the concrete semantics by substituting each concrete operation op over the concrete domain with the induced abstract operation $op^A = \rho(op)$ over A (Section 6). For instance, the composition operation ; is substituted by $;^A$ defined as $\lambda a_1, a_2.\rho(a_1; a_2)$. Hence, from a theoretical point of view, given a concrete semantics, the abstract domain is an exhaustive definition of an abstract semantics for a programming language, which can then be implemented and used for static analysis.

In Sections 4 and 5 we give two more examples of abstract domain, which formalise two already existing abstract domains for information flow analysis.

4. The abstract domain IF

We present here an abstract domain for information flow analysis, originally defined in [9]. It expresses which *termination-sensitive* flows [5,17] are allowed in a denotation. This domain, which we will call IF, has already been used to implement an information-flow analysis for Java bytecode by using Boolean formulas to represent its elements [10]. We will show in Section 10 that IF coincides with $\mathsf{C} \to \mathsf{C}$. Hence Section 9 can be seen as a formal justification of the use of Boolean formulas to represent the elements of IF.

A denotation δ *features* a flow from x to y if the input value of x can affect in δ the output value of y.

Definition 6 (*Termination Sensitive Information-Flow*). Let $\delta \in \Delta$ and $x, y \in V$. We say that δ features a termination sensitive information flow from x to y if there exist $\sigma_1, \sigma_2 \in \Sigma$ such that

1. $\sigma_1|_{V\setminus x} = \sigma_2|_{V\setminus x}$ (σ_1 and σ_2 agree on variables different from x);
2. $\delta(\sigma_1)(y) \neq \delta(\sigma_2)(y)$ (the input value of x affects the output value of y). $\square$

Definition 6 entails that $\sigma_1(x) \neq \sigma_2(x)$. Moreover, if exactly one between $\delta(\sigma_1)$ and $\delta(\sigma_2)$ is defined, then by Definition 2 the condition $\delta(\sigma_1)(y) \neq \delta(\sigma_2)(y)$ holds. This is why Definition 6 formalises termination-sensitive information flows.

Example 9. The denotation δ_1 of Example 2 is such that $\delta_1(\sigma) = \sigma[\mathtt{y} \mapsto \sigma(\mathtt{x}) + 1]$ for every $\sigma \in \Sigma$. Let σ_1 and σ_2 be such that $\sigma_1(v) = 0$ for every $v \in V$, $\sigma_2(\mathtt{x}) = 1$ and $\sigma_2(v) = 0$ for every $v \in V \setminus \mathtt{x}$. We have $\sigma_1|_{V\setminus\mathtt{x}} = \sigma_2|_{V\setminus\mathtt{x}}$, $\delta_1(\sigma_1) = \sigma_1[\mathtt{y} \mapsto 1]$, $\delta_1(\sigma_2) = \sigma_2[\mathtt{y} \mapsto 2]$ and hence $\delta_1(\sigma_1)(\mathtt{y}) = 1 \neq 2 = \delta_1(\sigma_2)(\mathtt{y})$. Then δ_1 features a flow from x to y. Moreover, $\delta_1(\sigma_1)(\mathtt{x}) = 0$ and $\delta_1(\sigma_2)(\mathtt{x}) = 1$. Then δ_1 features a flow from x to x. These are both *explicit* flows [17] *i.e.,* generated by copying input values into output values in a denotation. They are the only flows featured by δ_1. For instance, δ_1 does not feature any flow from y to y, since for every $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1|_{V\setminus\mathtt{y}} = \sigma_2|_{V\setminus\mathtt{y}}$ we have $\delta_1(\sigma_1)(\mathtt{y}) = \sigma_1(\mathtt{x}) + 1 = \sigma_2(\mathtt{x}) + 1 = \delta_1(\sigma_2)(\mathtt{y})$. □

Example 10. The denotation δ_5 of Example 5 features a flow from y to x. Let σ_1 and σ_2 be such that $\sigma_1(v) = 0$ for every $v \in V$, $\sigma_2(v) = 0$ for every $v \in V \setminus \mathtt{y}$ and $\sigma_2(\mathtt{y}) = 1$. We have $\sigma_1|_{V\setminus\mathtt{y}} = \sigma_2|_{V\setminus\mathtt{y}}$ and $\delta_5(\sigma_1)(\mathtt{x}) = 4 \neq 5 = \delta_5(\sigma_2)(\mathtt{x})$. This flow is called *implicit* [17] since it arises from the conditional execution of program statements on the basis of the initial value of some variables, in this case y. Note that if the assignment on the `else` branch of the conditional statement in Example 5 were changed into $\mathtt{x} := 4$ then δ_5 would feature no flow from y to x. □

Example 11. The denotation δ_4 of Example 4 is such that

$$\delta_4(\sigma) = \begin{cases} \sigma[\mathtt{x} \mapsto 4] & \text{if } \sigma(\mathtt{y}) = 0 \\ \textit{undefined} & \text{if } \sigma(\mathtt{y}) \neq 0. \end{cases}$$

It features a flow from y to x. Namely, take σ_1 and σ_2 such that $\sigma_1(v) = 0$ for every $v \in V$, $\sigma_2(v) = 0$ for every $v \in V \setminus \mathtt{y}$ and $\sigma_2(\mathtt{y}) = 1$. We have $\sigma_1|_{V\setminus\mathtt{y}} = \sigma_2|_{V\setminus\mathtt{y}}$, $\delta_4(\sigma_1)(\mathtt{x}) = 4 \neq \textit{undefined} = \delta_4(\sigma_2)(\mathtt{x})$. Since we consider termination-sensitive flows, the denotation δ_4 actually features a flow from y to *any* variable $v \in V$, since the initial value of y determines the termination of the conditional statement in Example 4. □

The abstract domain for information flow analysis is the powerset of the set of flows. Each abstract element expresses which flows a denotation is allowed to feature.

Definition 7 (*Abstract Domain* IF). Let $x_i, y_i \in V$ for $i = 1, \ldots, n$. We define

$$x_1 \rightsquigarrow y_1, \ldots, x_n \rightsquigarrow y_n = \left\{ \delta \in \Delta_V \;\middle|\; \begin{array}{l} \text{if } \delta \text{ features a flow} \\ \text{from } x \text{ to } y \text{ then} \\ \text{there exists } i \text{ such that} \\ x \equiv x_i \text{ and } y \equiv y_i \end{array} \right\}.$$

The abstract domain for information flow analysis is

$$\mathsf{IF}_V = \left\{ x_1 \rightsquigarrow y_1, \ldots, x_n \rightsquigarrow y_n \;\middle|\; \begin{array}{l} n \geq 0 \text{ and } x_i, y_i \in V \\ \text{for every } i = 1, \ldots, n \end{array} \right\}$$

where V is usually omitted. It is ordered by set-inclusion of denotations *i.e.,* by inverse inclusion of sets of flows. □

Each element of IF is a set of denotations. In order to justify the name of *abstract domain* for IF, we must prove that the set of its elements is closed by intersection.

Proposition 2 (IF *is a Moore Family*). *The set* IF *is a Moore family of* $\wp(\Delta)$.

Proof. Let $f^i = x^i_1 \rightsquigarrow y^i_1, \ldots, x^i_{n^i} \rightsquigarrow y^i_{n^i} \in \mathsf{IF}$ with $I \subseteq \mathbb{N}$ and $i \in I$. We prove that $X = \{x \rightsquigarrow y \mid x \rightsquigarrow y \in f^i \text{ for all } i \in \mathbb{N}\}$ (which belongs to IF) is their intersection. We have $\delta \in \cap_{i\in I} f^i$ if and only if $\delta \in f^i$ for all $i \in I$, if and only if whenever δ features a flow from x to y then $x \rightsquigarrow y \in f^i$ for all $i \in I$, if and only if whenever δ features a flow from x to y then $x \rightsquigarrow y \in X$, if and only if $\delta \in X$. □

Fig. 1 shows the abstract domain $\mathsf{IF}_{\{x,y\}}$. The top of the domain allows denotations to feature any flow, and hence coincides with Δ.

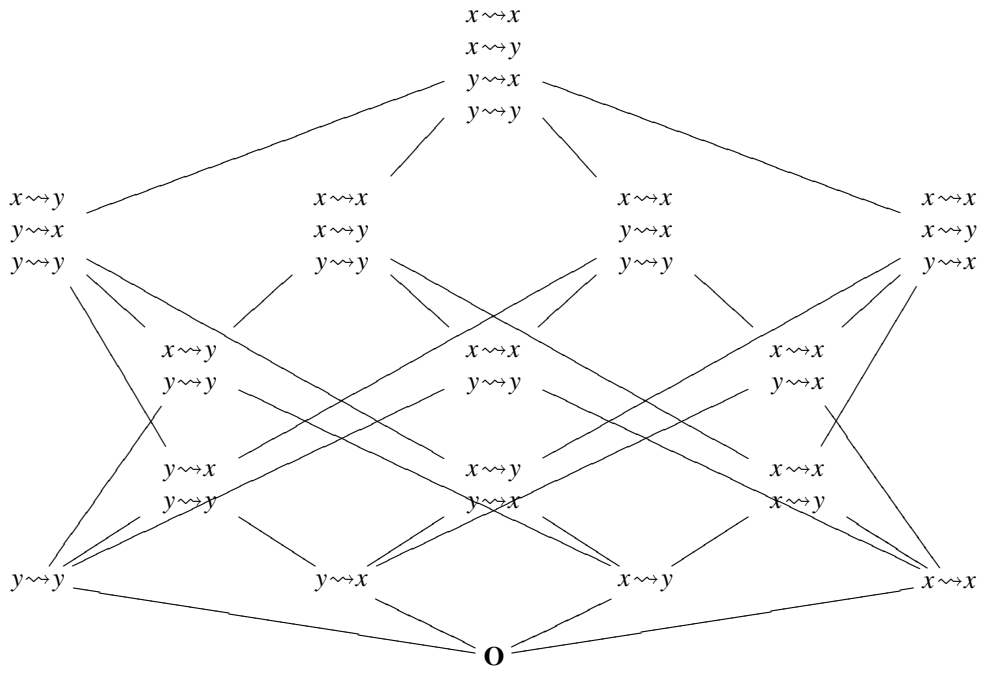

Fig. 1. The abstract domain $\mathsf{IF}_{\{x,y\}}$.

Example 12. Assume $V = \{\mathtt{x}, \mathtt{y}\}$. The denotation δ_1 of Example 2 belongs to $\mathtt{x}\rightsquigarrow\mathtt{x}$, $\mathtt{x}\rightsquigarrow\mathtt{y}$ since it only features flows from x to y and from x to x (Example 9). It also belongs to the upper bound $\mathtt{x}\rightsquigarrow\mathtt{x}$, $\mathtt{x}\rightsquigarrow\mathtt{y}$, $\mathtt{y}\rightsquigarrow\mathtt{y}$. However, δ_1 does not belong to $\mathtt{x}\rightsquigarrow\mathtt{x}$ since δ_1 features a flow from x to y (Example 9), not allowed in $\mathtt{x}\rightsquigarrow\mathtt{x}$. □

5. The abstract domain Independ

We define here the abstract domain Independ for termination-sensitive independence of program variables [3], originally defined in [2] in its termination-insensitive version. The elements of this domain are sets of pairs of variables. A pair $[o\#i]$ represents the denotations where the final value of o is constrained to be *independent* from the initial value of i.

Definition 8 (*The Abstract Domain* Independ). Let $i, o \in V$. We define

$$[o\#i]_V = \left\{ \delta \in \Delta_V \;\middle|\; \begin{array}{l} \text{for all } \sigma_1, \sigma_2 \in \Sigma_V \text{ such that } \sigma_1|_{V\setminus i} = \sigma_2|_{V\setminus i} \\ \text{we have } \delta(\sigma_1)(o) = \delta(\sigma_2)(o) \end{array} \right\}.$$

The notation $[o_1\#i_1]_V \cdots [o_n\#i_n]_V$ stands for $\cap_{j=1,\ldots,n}[o_j\#i_j]_V$. When $n = 0$, we write **O** for the empty intersection *i.e.,* Δ_V. The abstract domain for *independence* of program variables is

$$\mathsf{Independ}_V = \{[o_1\#i_1]_V \cdots [o_n\#i_n]_V \mid n \geq 0 \text{ and } i, o \in V\}.$$

The subscripts will be usually omitted. □

Note that the elements of $\mathsf{Independ}_V$ are isomorphic to sets of independence pairs $[o\#i]_V$. They are ordered by set-inclusion of denotations *i.e.,* inverse inclusion of sets of independence pairs. Fig. 2 shows the abstract domain $\mathsf{Independ}_{\{x,y\}}$.

Example 13. Consider denotation δ_1 from Example 2. We have $\delta_1 \in [\mathtt{x}\#\mathtt{y}]$ since the final value of x is not affected by the initial value of y. Formally, for every $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1|_{V\setminus\mathtt{y}} = \sigma_2|_{V\setminus\mathtt{y}}$ we have $\delta_1(\sigma_1)(\mathtt{x}) = \sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x}) = \delta_1(\sigma_2)(\mathtt{x})$. We also have $\delta_1 \in [\mathtt{y}\#\mathtt{y}]$ and $\delta_1 \in [\mathtt{x}\#\mathtt{x}]$, so that $\delta_1 \in [\mathtt{x}\#\mathtt{x}][\mathtt{x}\#\mathtt{y}][\mathtt{y}\#\mathtt{y}]$. However, we have $\delta_1 \notin [\mathtt{y}\#\mathtt{x}]$ since the initial value of x can affect the final value of y: take $\sigma_1, \sigma_2 \in \Sigma$ such that

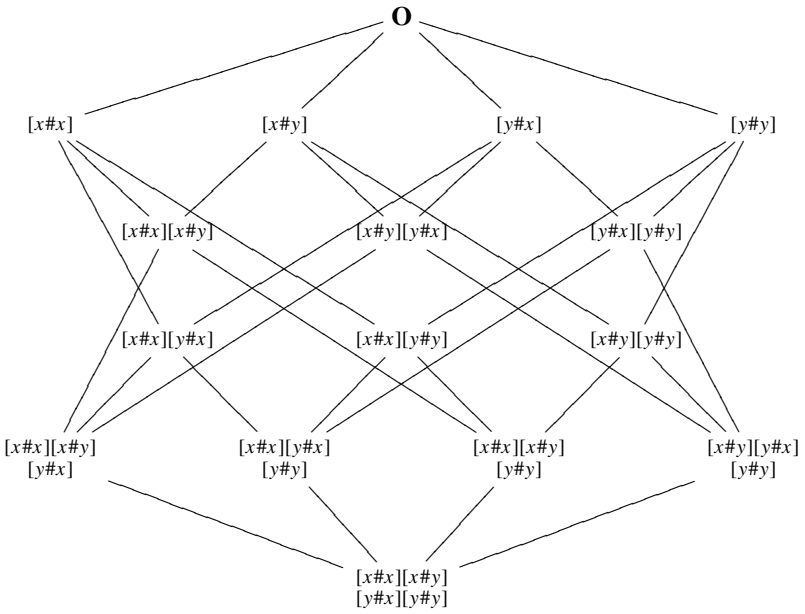

Fig. 2. The abstract domain $\mathsf{Independ}_{\{x,y\}}$.

$\sigma_1(v) = 0$ for every $v \in V$, $\sigma_2(v) = 0$ for every $v \in V \setminus \mathtt{x}$ and $\sigma_2(\mathtt{x}) = 1$. We have $\sigma_1|_{V\setminus\mathtt{x}} = \sigma_2|_{V\setminus\mathtt{x}}$ but $\delta_1(\sigma_1)(\mathtt{y}) = \sigma_1(\mathtt{x}) + 1 = 1 \neq 2 = \sigma_2(\mathtt{x}) + 1 = \delta_1(\sigma_2)(\mathtt{y})$. □

Definition 8 formalises termination-sensitive independence since if δ terminates for some values of i and diverges for others then $\delta \notin [o\#i]$ for any $o \in V$. The notion of independence considered in [2] is instead termination-insensitive *i.e.,* only terminating computations are considered in order to prove independence. In [3] termination-sensitive independence is split in two, by defining an abstract domain which expresses both termination-insensitive independence of o w.r.t. i and independence of termination w.r.t. i.

6. Operational and denotational semantics. Condensing

In this section we define a generic input-driven, non-compositional *operational* and a generic input-independent, compositional *denotational* semantics for imperative programs. Their *concrete* versions specify the behaviour of a computer program as it is observed on a machine running that program. Their *abstract* versions mimick this behaviour over an abstract domain, which approximates properties of the concrete behaviour. We show that our abstract operational and denotational semantics have the same precision when the abstract domain satisfies a property called *condensing*. In Section 7 we show that each abstract domain closed by *linear refinement* satisfies the condensing property.

We start by defining the syntax of our imperative language.

Definition 9 (*Command*). A *command* is a portion of code whose execution modifies the state. The set of commands is defined by the grammar:

$$
\begin{aligned}
\mathit{com} ::= {} & \mathtt{if\ v\ then}\ \mathit{com}\ \mathtt{else}\ \mathit{com} \\
& \mid \mathit{com}; \mathit{com} \\
& \mid \mathtt{v_0} := \mathtt{f(v_1, \ldots, v_m)} \\
& \mid \mathtt{bc}
\end{aligned}
$$

that is, a command is either a conditional, a sequence of commands, a function call or a *basic command*. The set of basic commands is left unspecified (it normally includes assignments to variables and to fields of objects). □

Definition 10 (*Program*). A *program* P is a set of *function definitions*

$$\mathtt{f}(\mathtt{w}_1, \ldots, \mathtt{w}_m)\ body_{\mathtt{f}}$$

where $\mathtt{w}_1, \ldots, \mathtt{w}_m$ are distinct variables and $body_{\mathtt{f}}$ is a command. We assume that P is type-checked and that all functions called in P are also defined in P. In the following, we assume that P is given. □

The syntax of our language is very abstract, in order to simplify the theoretical results, but it is not restrictive. In particular, there are no loops: they must be simulated through recursive functions; local variables cannot be defined in the body of a function: they must be included as dummy parameters, whose value at call time is irrelevant; a function `f` returns the value which is stored, at the end of its execution, in the variable called `f`. At the same time, that syntax is very general: it includes a generic set of basic commands. We assume that the semantics of each basic command `bc` is specified by a set of denotations $bc \subseteq \Delta$. A basic command can even be non-deterministic, in which case bc contains more than one denotation. Also the notion of *state* for our language is completely generic. In our examples we will always use maps from variables to values (Definition 1). However, one can also use more complex states including a *memory*, which are useful if our language has basic commands for updates of fields of objects.

Example 14. Most imperative languages allow commands of the form $\mathtt{y} := \mathtt{x} + \mathtt{1}$ whose semantics $y := x + 1$ is $\{\delta_1\}$, where δ_1 is the denotation of Example 2. □

We can now give semantics to a command in an *operational* and in a *denotational* way. In both cases we use three operators over sets of denotations.

Definition 11 (*Semantical Operators*). We define three *semantical operators* over sets of denotations:

1. the point-wise extension ; of the sequential composition of denotations of Definition 2;
2. the union $\cup$ of sets of denotations;
3. the return from function operator. Given $v_0 \in V$ and f function name, we define it as the point-wise extension to sets of denotations of

$$return_{f,v_0} \lambda\delta_1.\lambda\delta_2.\lambda\sigma.\delta_1(\sigma)[v_0 \mapsto \delta_2(\sigma)(f)].$$

It restores the state $\delta_1(\sigma)$ of the caller but stores into v_0 the return value $\delta_2(\sigma)(f)$ of the callee. □

The following result will be useful in Section 6.4.

Lemma 2. *Let $\delta_0, \delta_1, \delta_2 \in \Delta$, f be a function name and $v_0 \in V$. We have*

$$\delta_0; return_{f,v_0}(\delta_1, \delta_2) = return_{f,v_0}((\delta_0; \delta_1), (\delta_0; \delta_2)).$$

Proof. Let $\sigma \in \Sigma$. We have

$$\begin{aligned}(\delta_0; return_{f,v_0}(\delta_1, \delta_2))(\sigma) &= (\delta_0; \lambda\sigma.\delta_1(\sigma)[v_0 \mapsto \delta_2(\sigma)(f)])(\sigma)\\ &= \delta_1(\delta_0(\sigma))[v_0 \mapsto \delta_2(\delta_0(\sigma))(f)]\\ &= (\delta_0; \delta_1)(\sigma)[v_0 \mapsto (\delta_0; \delta_2)(\sigma)(f)]\\ &= return_{f,v_0}((\delta_0; \delta_1), (\delta_0; \delta_2))(\sigma). \quad \square\end{aligned}$$

6.1. Operational semantics

The *operational semantics* of a command c reflects its execution by an actual virtual machine for our language. Hence it can be used for an implementation of that virtual machine. It is given as a total transition function: $\langle d_i \,||\, c\rangle \Rightarrow d_o$ with $d_i, d_o \subseteq \Delta$. It means that if we have already run some code whose behaviour is described by the denotations d_i and then continue with c, we get an overall run whose behaviour is described by the denotations d_o. By taking $d_i = \{\iota\}$, where ι is the identity denotation, this generalises the traditional operational semantics over states [23].

Definition 12 (*Operational Semantics*). The (big step) *operational semantics* for our language is a transition relation $\Rightarrow$ from *configurations* to sets of denotations, where a configuration is a pair $\langle c \,||\, d\rangle$ of a command c and of a set of denotations $d \subseteq \Delta$. It is defined by the following transition rules:

$$\frac{}{\langle \mathtt{bc} \,||\, d_0\rangle \Rightarrow d_0; bc}$$

that is, the evaluation of a basic command uses the semantics of the basic command.

$$\frac{\langle com_1 \,||\, d_0\rangle \Rightarrow d_1 \quad \langle com_2 \,||\, d_1\rangle \Rightarrow d_2}{\langle com_1; com_2 \,||\, d_0\rangle \Rightarrow d_2}$$

that is, the execution of a sequence of commands runs, sequentially, the first and then the second command.

$$\frac{\langle com_1 \,||\, d_0; is\_true_{\mathtt{v}}\rangle \Rightarrow d_1 \quad \langle com_2 \,||\, d_0; is\_false_{\mathtt{v}}\rangle \Rightarrow d_2}{\langle \mathtt{if\ v\ then}\ com_1\ \mathtt{else}\ com_2 \,||\, d_0\rangle \Rightarrow d_1 \cup d_2}$$

that is, the execution of a conditional evaluates its condition and then selects the right branch to run through generic $is\_true_{\mathtt{v}}$ and $is\_false_{\mathtt{v}}$ denotations.

$$\frac{\langle body_{\mathtt{f}} \,||\, d_0; call_{\mathtt{f}(\mathtt{v}_1,\ldots,\mathtt{v}_m)}\rangle \Rightarrow d_1}{\langle \mathtt{v}_0 := \mathtt{f}(\mathtt{v}_1,\ldots,\mathtt{v}_m) \,||\, d_0\rangle \Rightarrow return_{\mathtt{f},\mathtt{v}_0}(d_0, d_1)}$$

i.e., a function call creates a scope $d_0; call_{f(v_1,\ldots,v_m)}$ where the body of the function can run. At return time it stores the return value of the function into v_0. □

Example 15. The actual definition of is_true_v and is_false_v is irrelevant in this paper. For states allowing Boolean values, they are usually defined as

$$is\_true_v = \{\delta_t\} \quad \text{where } \delta_t = \lambda\sigma.\begin{cases}\sigma & \text{if } \sigma(v) = true\\ undefined & \text{otherwise}\end{cases}$$

$$is\_false_v = \{\delta_f\} \quad \text{where } \delta_f = \lambda\sigma.\begin{cases}\sigma & \text{if } \sigma(v) = false\\ undefined & \text{otherwise.}\end{cases}$$ □

Example 16. The actual definition of *call* is irrelevant in this paper. However, it can be interesting to see its usual definition over the states of Definition 1. It is $call_{f(v_1,\ldots,v_m)} = \{\delta_c\}$ where

$$\delta_c = \lambda\sigma.\sigma[w_1 \mapsto \sigma(v_1), \ldots, w_m \mapsto \sigma(v_m)]$$

i.e., $call_{f(v_1,\ldots,v_m)}$ renames variables $v_1, \ldots, v_m$ into $w_1, \ldots, w_m$. □

6.2. Denotational semantics

Denotational semantics describes the behaviour of a command in terms of the behaviours of its components. Hence it is compositional, a feature which is highly appreciated when it comes to specifying and implementing a static analysis.

The denotational semantics of a command c is a set of denotations $[\![c]\!]_I$, where I is an *interpretation* providing the semantics of the program functions.

Definition 13 (*Denotation of a Command*). Let I be an *interpretation i.e.,* a collection $I_{f_1}, \ldots, I_{f_q}$ of sets of denotations, one for each function name f_i in P, with $1 \le i \le q$. The *denotation of a command c* in I is a set of denotations $[\![c]\!]_I$ defined as

$$\begin{aligned}
[\![\mathtt{bc}]\!]_I &= bc\\
[\![com_1; com_2]\!]_I &= [\![com_1]\!]_I; [\![com_2]\!]_I\\
[\![\mathtt{if\ v\ then}\ com_1\ \mathtt{else}\ com_2]\!]_I &= (is_true_{\mathtt{v}}; [\![com_1]\!]_I) \cup (is_false_{\mathtt{v}}; [\![com_2]\!]_I)\\
[\![\mathtt{v}_0 := \mathtt{f}(\mathtt{v}_1,\ldots,\mathtt{v}_m)]\!]_I &= return_{\mathtt{f},\mathtt{v}_0}(\{\iota\}, (call_{\mathtt{f}(\mathtt{v}_1,\ldots,\mathtt{v}_m)}; I_{\mathtt{f}}))
\end{aligned}$$

where ι is the identity denotation. □

Because of recursion, the denotational semantics is computed as a fixpoint computation.

Definition 14 (*Denotational Semantics*). Given a program P, the *immediate consequence operator* T_P is the transformer on interpretations defined as

$$(T_P(I))_f = [\![body_f]\!]_I \quad \text{for every function name } f \text{ in } P.$$

Its least fixpoint is the *denotational semantics* $\mathcal{D}_P$ of P. □

6.3. Abstract semantics

Both our operational and denotational semantics work over sets of denotations. As a consequence, they are already *collecting semantics* in the sense of [6]. Sets of denotations express properties of denotations. They can be substituted with an abstract domain A: each element of A stands for a set of denotations. We get an (operational or denotational) correct abstract semantics which approximates the properties expressed by A. They use the best possible approximations over A of the semantical operators of Definition 11 (Section 3.3):

$$\begin{aligned} ;^A &= \lambda a_1, a_2 \in A.\rho(a_1; a_2) \\ \cup^A &= \lambda a_1, a_2 \in A.\rho(a_1 \cup a_2) \\ return^A_{f,v_0} &= \lambda a_1, a_2 \in A.\rho(return_{f,v_0}(a_1, a_2)). \end{aligned}$$

Definition 15 (*Abstract Operational Semantics*). Given an abstract domain A approximating sets of denotations, with the induced upper closure operator (abstraction function) ρ mapping sets of denotations into their best approximation inside A, the (big step) *abstract operational semantics* for our language is a transition relation $\Rightarrow^A$ from *abstract configurations* to A, where an abstract configuration is a pair $\langle c \,||\, a\rangle$ of a command c and of an element $a \in A$. It is defined by the following transition rules:

$$\frac{}{\langle \mathtt{bc} \,||\, a_0\rangle \Rightarrow^A \rho(a_0; bc)}$$

$$\frac{\langle com_1 \,||\, \rho(a_0; is\_true_{\mathtt{v}})\rangle \Rightarrow^A a_1 \quad \langle com_2 \,||\, \rho(a_0; is\_false_{\mathtt{v}})\rangle \Rightarrow^A a_2}{\langle \mathtt{if\ v\ then}\ com_1\ \mathtt{else}\ com_2 \,||\, a_0\rangle \Rightarrow^A a_1 \cup^A a_2}$$

$$\frac{\langle com_1 \,||\, a_0\rangle \Rightarrow^A a_1 \quad \langle com_2 \,||\, a_1\rangle \Rightarrow^A a_2}{\langle com_1; com_2 \,||\, a_0\rangle \Rightarrow^A a_2}$$

$$\frac{\langle body_{\mathtt{f}} \,||\, \rho(a_0; call_{\mathtt{f}(\mathtt{v}_1,\ldots,\mathtt{v}_m)})\rangle \Rightarrow^A a_1}{\langle \mathtt{v}_0 := \mathtt{f}(\mathtt{v}_1, \ldots, \mathtt{v}_m) \,||\, a_0\rangle \Rightarrow^A return^A_{\mathtt{f},\mathtt{v}_0}(a_0, a_1)}. \quad \square$$

It is important to note that the abstract operational semantics performs the abstractions through ρ as the computation proceeds.

Definition 16 (*Abstract Denotation of a Command*). Let I be an *abstract interpretation i.e.,* a collection $I_{f_1}, \ldots, I_{f_q}$ of elements of A, one for each function name f_i in P, with $1 \le i \le q$. The *abstract denotation of a command* c in I is $[\![c]\!]^A_I \in A$, defined as

$$\begin{aligned} [\![\mathtt{bc}]\!]^A_I &= \rho(bc) \\ [\![com_1; com_2]\!]^A_I &= [\![com_1]\!]^A_I \ ;^A\ [\![com_2]\!]^A_I \\ [\![\mathtt{if\ v\ then}\ com_1\ \mathtt{else}\ com_2]\!]^A_I &= (\rho(is\_true_{\mathtt{v}});^A [\![com_1]\!]^A_I) \cup^A (\rho(is\_false_{\mathtt{v}});^A [\![com_1]\!]^A_I) \\ [\![\mathtt{v}_0 := \mathtt{f}(\mathtt{v}_1, \ldots, \mathtt{v}_m)]\!]^A_I &= return^A_{\mathtt{f},\mathtt{v}_0}(\rho(\{\iota\}), (\rho(call_{\mathtt{f}(\mathtt{v}_1,\ldots,\mathtt{v}_m)});^A I_{\mathtt{f}})) \end{aligned}$$

where ι is the identity denotation. □

The abstract immediate consequence operator T^A_P and the abstract denotational semantics $\mathcal{D}^A_P$ are defined similarly to Definition 14. We observe that the abstract denotational semantics, for compositionality, abstracts the basic

denotations *before* composing them, which is the main difference w.r.t. the abstract operational semantics. Because of this, the abstract operational semantics is in general more precise than the abstract denotational semantics.

Example 17. Consider the abstract domain C of Definition 4. The abstract operational semantics of the command $\mathtt{v} := 2; \mathtt{w} := \mathtt{v}$ from the top approximation $\mathbf{O}$ correctly captures the fact that at its end both variables $\mathtt{v}$ and $\mathtt{w}$ are constant. This is because $\rho(\mathbf{O}; v := 2) = \mathbf{v}$ and $\rho(\mathbf{v}; w := v) = \mathbf{wv}$, so that

$$\frac{\langle \mathtt{v} := 2 \,||\, \mathbf{O}\rangle \Rightarrow^{\mathsf{C}} \mathbf{v} \quad \langle \mathtt{w} := \mathtt{v} \,||\, \mathbf{v}\rangle \Rightarrow^{\mathsf{C}} \mathbf{vw}}{\langle \mathtt{v} := 2; \mathtt{w} := \mathtt{v} \,||\, \mathbf{O}\rangle \Rightarrow^{\mathsf{C}} \mathbf{vw}}.$$

However, the abstract denotational semantics of the same command is much more imprecise, since

$$\begin{aligned}[\![\mathtt{v} := 2; \mathtt{w} := \mathtt{v}]\!]_I^{\mathsf{C}} &= [\![\mathtt{v} := 2]\!]_I^{\mathsf{C}} ;^{\mathsf{C}} [\![\mathtt{w} := \mathtt{v}]\!]_I^{\mathsf{C}} \\ &= \rho(v := 2);^{\mathsf{C}} \rho(w := v) \\ &= \mathbf{v};^{\mathsf{C}} \mathbf{O} = \mathbf{O}.\end{aligned}$$

The explanation of this imprecision is that the abstract domain C is not able to express the dependency between the constancy of $\mathtt{v}$ and that of $\mathtt{w}$ which is featured by $w := v$, nor the fact that $w := v$ does not modify the constancy of $\mathtt{v}$. □

6.4. The condensing property

The *condensing* property formalises the fact that a concrete element and its abstraction convey *exactly* the same abstract information w.r.t. composition.

Definition 17 (*Condensing Property*). Let $\wp(\Delta)$ be our concrete domain, A be an abstract domain with induced abstraction function ρ. If for every $d_1, d_2 \in \wp(\Delta)$ we have

$$\rho(\rho(d_1); d_2) = \rho(\rho(d_1); \rho(d_2)) = \rho(d_1; \rho(d_2))$$

then A enjoys the *condensing property* or is *condensing*. □

Definition 17 is called *weak-completeness* in [12]. In Theorem 1 we show that it entails that our abstract operational and denotational semantics have the same precision. We need some lemmas to that purpose. The following one lifts to $;^A$ the commutativity of $;$.

Lemma 3. *If A is a condensing abstract domain then $;^A$ is commutative.*

Proof. Let $a_1, a_2, a_3 \in A$. We have

$$\begin{aligned}a_1;^A (a_2;^A a_3) &= \rho(a_1; \rho(a_2; a_3)) \\ \text{(Definition 17)} &= \rho(a_1; (a_2; a_3)) \\ \text{(; is commutative)} &= \rho((a_1; a_2); a_3) \\ \text{(Definition 17)} &= \rho(\rho(a_1; a_2); a_3) = (a_1;^A a_2);^A a_3.\end{aligned}$$ □

The following lemma states that $;^A$ distributes over $\cup^A$.

Lemma 4. *Let A be a condensing abstract domain and $a_0; a_1, a_2 \in A$. We have*

$$a_0;^A (a_1 \cup^A a_2) = (a_0;^A a_1) \cup^A (a_0;^A a_2).$$

Proof. We have

$$\begin{aligned}a_0;^A (a_1 \cup^A a_2) &= \rho(a_0; \rho(a_1 \cup a_2)) \\ \text{(Definition 17)} &= \rho(a_0; (a_1 \cup a_2)) \\ \text{(; distributes over } \cup) &= \rho((a_0; a_1) \cup (a_0; a_2))\end{aligned}$$

By expansivity and monotonicity of ρ we have $\rho((a_0; a_1) \cup (a_0; a_2)) \subseteq \rho(\rho(a_0; a_1) \cup \rho(a_0; a_2))$. Moreover, by monotonicity we have $\rho(a_0; a_1) \subseteq \rho((a_0; a_1) \cup (a_0; a_2))$ and $\rho(a_0; a_2) \subseteq \rho((a_0; a_1) \cup (a_0; a_2))$ so that $\rho(a_0; a_1) \cup \rho(a_0; a_2) \subseteq \rho((a_0; a_1) \cup (a_0; a_2))$. Then $\rho(\rho(a_0; a_1) \cup \rho(a_0; a_2)) \subseteq \rho(\rho((a_0; a_1) \cup (a_0; a_2)))$ and by idempotency $\rho(\rho(a_0; a_1) \cup \rho(a_0; a_2)) \subseteq \rho((a_0; a_1) \cup (a_0; a_2))$. In conclusion

$$\begin{aligned}\rho((a_0; a_1) \cup (a_0; a_2)) &= \rho(\rho(a_0; a_1) \cup \rho(a_0; a_2)) \\ &= (a_0;^A a_1) \cup^A (a_0;^A a_2). \quad \square\end{aligned}$$

Lemma 5 states that $;^A$ distributes over $return^A$. It requires that A is *compatible* with *return*.

Definition 18 (*Compatibility with Return*). Let A be an abstract domain. We say that A is *compatible* with *return* if for every function name f in P, $v_0 \in V$ and $d_1, d_2 \subseteq \Delta$ we have

$$\rho\left(return_{f,v_0}(\rho(d_1), \rho(d_2))\right) = \rho(return_{f,v_0}(d_1, d_2)). \quad \square$$

Lemma 5. *Let A be a condensing abstract domain compatible with return, $a_0; a_1, a_2 \in A$, f be a function name in P and $v_0 \in V$. We have*

$$a_0;^A (return^A_{f,v_0}(a_1, a_2)) = return^A_{f,v_0}((a_0;^A a_1), (a_0;^A a_2)).$$

Proof. We have

$$\begin{aligned}a_0;^A (return^A_{f,v_0}(a_1, a_2)) &= \rho(a_0; \rho(return_{f,v_0}(a_1, a_2))) \\ \text{(Definition 17)} &= \rho(a_0; return_{f,v_0}(a_1, a_2)) \\ \text{(Lemma 2)} &= \rho(return_{f,v_0}((a_0; a_1), (a_0; a_2))) \\ \text{(Definition 18)} &= \rho(return_{f,v_0}(\rho(a_0; a_1), \rho(a_0; a_2))) \\ &= return^A_{f,v_0}((a_0;^A a_1), (a_0;^A a_2)). \quad \square\end{aligned}$$

We can now prove the main result of this section. It says that, for a condensing abstract domain compatible with *return*, the abstract operational (input-driven) and the abstract denotational (input-independent) semantics have the same precision.

Theorem 1 (*Equivalence of the Abstract Operational and Denotational Semantics*). *Given a condensing abstract domain A compatible with return, $a_0, a \in A$ and a command c, we have $\langle c \,||\, a_0\rangle \Rightarrow^A a$ if and only if $a_0;^A [\![c]\!]^A_{\mathcal{D}^A_P} = a$.*

Proof. Since both $\Rightarrow^A$ and $\lambda c'.[\![c']\!]^A_{\mathcal{D}^A_P}$ are total maps, it is enough to prove that if $\langle c \,||\, a_0\rangle \Rightarrow^A a$ then $a_0;^A [\![c]\!]^A_{\mathcal{D}^A_P} = a$. Let then $\langle c \,||\, a_0\rangle \Rightarrow^A a$. We proceed by rule induction on the derivation of $\langle c \,||\, a_0\rangle \Rightarrow^A a$ (Definition 15). If $c = \texttt{bc}$ then

$$\begin{aligned}a_0;^A [\![\texttt{bc}]\!]^A_{\mathcal{D}^A_P} &= a_0;^A \rho(bc) \\ &= \rho(a_0; \rho(bc)) \\ \text{(Definition 17)} &= \rho(a_0; bc) = a.\end{aligned}$$

If $c = com_1; com_2$ we have $\langle com_1 \,||\, a_0\rangle \Rightarrow^A a_1$ and $\langle com_2 \,||\, a_1\rangle \Rightarrow^A a$ and by the inductive hypothesis $a_0;^A [\![com_1]\!]^A_{\mathcal{D}^A_P} = a_1$ and $a_1;^A [\![com_2]\!]^A_{\mathcal{D}^A_P} = a$. Hence

$$\begin{aligned}a_0;^A ([\![com_1; com_2]\!]^A_{\mathcal{D}^A_P}) &= a_0;^A ([\![com_1]\!]^A_{\mathcal{D}^A_P};^A [\![com_2]\!]^A_{\mathcal{D}^A_P}) \\ \text{(Lemma 3)} &= (a_0;^A [\![com_1]\!]^A_{\mathcal{D}^A_P});^A [\![com_2]\!]^A_{\mathcal{D}^A_P} \\ &= a_1;^A [\![com_2]\!]^A_{\mathcal{D}^A_P} = a.\end{aligned}$$

If $c = \texttt{if v then } com_1 \texttt{ else } com_2$ then $\langle com_1 \,||\, \rho(a_0; is\_true_{\mathtt{v}})\rangle \Rightarrow^A a_1$, $\langle com_2 \,||\, \rho(a_0; is\_false_{\mathtt{v}})\rangle \Rightarrow^A a_2$ and $a = a_1 \cup^A a_2$. By the inductive hypothesis we have $\rho(a_0; is\_true_{\mathtt{v}});^A [\![com_1]\!]^A_{\mathcal{D}^A_P} = a_1$ and $\rho(a_0; is\_false_{\mathtt{v}});^A [\![com_2]\!]^A_{\mathcal{D}^A_P} = a_2$. Then

$$
\begin{aligned}
& a_0;^A [\![\texttt{if v then } com_1 \texttt{ else } com_2]\!]^A_{\mathcal{D}^A_P} \\
&= a_0;^A \begin{pmatrix} (\rho(is\_true_{\mathtt{v}});^A [\![com_1]\!]^A_{\mathcal{D}^A_P}) \cup^A \\ (\rho(is\_false_{\mathtt{v}});^A [\![com_2]\!]^A_{\mathcal{D}^A_P}) \end{pmatrix} \\
\text{(Lemma 4)} &= \begin{pmatrix} (a_0;^A \rho(is\_true_{\mathtt{v}});^A [\![com_1]\!]^A_{\mathcal{D}^A_P}) \cup^A \\ (a_0;^A \rho(is\_false_{\mathtt{v}});^A [\![com_2]\!]^A_{\mathcal{D}^A_P}) \end{pmatrix} \\
&= \begin{pmatrix} (\rho(a_0; \rho(is\_true_{\mathtt{v}}));^A [\![com_1]\!]^A_{\mathcal{D}^A_P}) \cup^A \\ (\rho(a_0; \rho(is\_false_{\mathtt{v}}));^A [\![com_2]\!]^A_{\mathcal{D}^A_P}) \end{pmatrix} \\
\text{(Definition 17)} &= \begin{pmatrix} (\rho(a_0; is\_true_{\mathtt{v}});^A [\![com_1]\!]^A_{\mathcal{D}^A_P}) \cup^A \\ (\rho(a_0; is\_false_{\mathtt{v}});^A [\![com_2]\!]^A_{\mathcal{D}^A_P}) \end{pmatrix} \\
&= a_1 \cup^A a_2 = a.
\end{aligned}
$$

If $c = (\mathtt{v_0} := \mathtt{f}(\mathtt{v_1}, \ldots, \mathtt{v_m}))$ let $ca = call_{\mathtt{f}(\mathtt{v_1},\ldots,\mathtt{v_m})}$. We know that $\langle body_{\mathtt{f}} \,||\, \rho(a_0; ca)\rangle \Rightarrow^A a_1$ and $a = return_{\mathtt{f},\mathtt{v_0}}(a_0, a_1)$. By the inductive hypothesis we have $\rho(a_0; ca);^A [\![body_{\mathtt{f}}]\!]^A_{\mathcal{D}^A_P} = a_1$. Since $\mathcal{D}^A_P$ is the least fixpoint of T^A_P then $(\mathcal{D}^A_P)_{\mathtt{f}} = [\![body_{\mathtt{f}}]\!]^A_{\mathcal{D}^A_P}$ and $\rho(a_0; ca);^A (\mathcal{D}^A_P)_{\mathtt{f}} = a_1$. Hence

$$
\begin{aligned}
& a_0;^A [\![\mathtt{v_0} := \mathtt{f}(\mathtt{v_1}, \ldots, \mathtt{v_m})]\!]^A_{\mathcal{D}^A_P} \\
&= a_0;^A return^A_{\mathtt{f},\mathtt{v_0}} \left(\rho(\{\iota\}), (\rho(ca);^A (\mathcal{D}^A_P)_{\mathtt{f}})\right) \\
\text{(Lemma 5)} &= return_{\mathtt{f},\mathtt{v_0}}((a_0;^A \rho(\{\iota\})), (a_0;^A \rho(ca);^A (\mathcal{D}^A_P)_{\mathtt{f}})) \\
&= return_{\mathtt{f},\mathtt{v_0}}(\rho(a_0; \rho(\{\iota\})), (\rho(a_0; \rho(ca));^A (\mathcal{D}^A_P)_{\mathtt{f}})) \\
\text{(Definition 17)} &= return_{\mathtt{f},\mathtt{v_0}}(\rho(a_0; \{\iota\}), (\rho(a_0; ca);^A (\mathcal{D}^A_P)_{\mathtt{f}})) \\
&= return_{\mathtt{f},\mathtt{v_0}}(\rho(a_0), (\rho(a_0; ca);^A (\mathcal{D}^A_P)_{\mathtt{f}})) \\
&= return_{\mathtt{f},\mathtt{v_0}}(a_0, a_1) = a. \quad \square
\end{aligned}
$$

Section 7 introduces the *linear refinement* of abstract domains and links the closure under linear refinement with the condensing property and the compatibility with *return*.

7. Linear refinement

The definition of an appropriate abstract domain for a static analysis is not easy in general. Although a *basic* abstract domain A can be immediately constructed ($\curlywedge$) from the abstract properties one wants to model, there is no guarantee that the induced abstract semantics is precise enough to be useful. The intuition and experience of the abstract domain designer helps in determing what A is missing in order to improve its precision. To that goal, there are some methodological techniques which *refine* A to get a more precise domain.

Cousot and Cousot's *reduced product* [7] allows one to refine two abstract domains A_1 and A_2 into an abstract domain $A_1 \sqcap A_2 = \curlywedge(A_1 \cup A_2)$ which expresses the conjunction of the properties expressed by A_1 and A_2.

Giacobazzi and Scozzari's *linear refinement* [13] is another domain refinement operator. It allows one to enrich an abstract domain with information relative to the *propagation* of the abstract properties before and after the application of a concrete operator $\boxtimes$. It requires the concrete domain C to be a *quantale* w.r.t. $\boxtimes$ *i.e.,*

1. C must be a complete lattice;
2. $\boxtimes : C \times C \to C$ must be (in general partial and) associative;

3. for any $a \in C$ and $\{b_i\}_{i\in I} \subseteq C$ with $I \subseteq \mathbb{N}$ we must have $a \boxtimes (\sqcup_{i\in I} b_i) = \sqcup_{i\in I}\{a \boxtimes b_i\}$ and $(\sqcup_{i\in I} b_i) \boxtimes a = \sqcup_{i\in I}\{b_i \boxtimes a\}$.

For instance, the complete lattice $\wp(\Delta)$, ordered by set-inclusion, is a quantale w.r.t. the composition operator ; extended to sets of denotations as $d_1; d_2 = \{\delta_1; \delta_2 \mid \delta_1 \in d_1 \text{ and } \delta_2 \in d_2\}$.

Let $a, b \in C$. The abstract property $a \rightarrow^{\boxtimes} b$ which *transforms* every element of a into an element of b is

$$a \rightarrow^{\boxtimes} b = \bigsqcup\{c \in C \mid \text{if } a \boxtimes c \text{ is defined then } a \boxtimes c \leq b\}.$$

It is the set of concrete elements whose composition through $\boxtimes$ with a concrete element in a is undefined or is an element in b.

Definition 19 (*Linear Refinement [13]*). The (forward) *linear refinement* of an abstract domain $A_1 \subseteq C$ w.r.t. another abstract domain $A_2 \subseteq C$ is the abstract domain

$$A_1 \rightarrow^{\boxtimes} A_2 = \curlywedge\{a \rightarrow^{\boxtimes} b \mid a \in A_1 \text{ and } b \in A_2\}.$$

That is, $A_1 \rightarrow^{\boxtimes} A_2$ collects all possible arrows between elements of A_1 and elements of A_2. □

It is immediate to verify that $\rightarrow^{\boxtimes}$ is argument-wise monotonic.

We can instantiate $\rightarrow^{\boxtimes}$ over the quantale $\langle \wp(\Delta), ; \rangle$. The intuition under the choice of ; for $\boxtimes$ is that the denotational semantics of an imperative program is defined by *composing* smaller denotations to form larger ones (Section 6). Hence we must refine the composition operation if we want to improve the precision of an abstraction of $\wp(\Delta)$. We provide below an explicit definition for $\rightarrow^{;}$, stating that $d_1 \rightarrow^{;} d_2$ is the set of denotations whose composition with a denotation in d_1 is a denotation in d_2.

Proposition 3 (*Explicit Definition of* $\rightarrow^{;}$). *Let* $d_1, d_2 \subseteq \Delta$. *Then*

$$d_1 \rightarrow^{;} d_2 = \{\delta \in \Delta \mid \textit{for every } \overline{\delta} \in d_1 \textit{ we have } \overline{\delta}; \delta \in d_2\}.$$

Proof.

$$\begin{aligned} d_1 \rightarrow^{;} d_2 &= \bigcup\{d \in \wp(\Delta) \mid \text{if } d_1; d \text{ is defined then } d_1; d \subseteq d_2\} \\ &= \bigcup\{d \in \wp(\Delta) \mid d_1; d \subseteq d_2\} \\ &= \bigcup\{d \in \wp(\Delta) \mid \{\overline{\delta}; \delta \mid \overline{\delta} \in d_1 \text{ and } \delta \in d\} \subseteq d_2\} \\ &= \{\delta \in \Delta \mid \text{for every } \overline{\delta} \in d_1 \text{ we have } \overline{\delta}; \delta \in d_2\}. \quad \square \end{aligned}$$

Example 18. Consider the abstract domain C of Definition 4 and its elements **x** and **y**. The denotation δ_1 of Example 2 belongs to $\mathbf{x} \rightarrow^{;} \mathbf{y}$ since δ_1 stores the input value of x plus 1 in the output value of y, so if x is constant in δ_1's input then y is constant in δ_1's output. □

The following result states the behaviour of $\rightarrow^{;}$ when its right-hand side is the intersection of some properties of denotations.

Proposition 4. *Given* $d \in \wp(\Delta)$, $I \subseteq \mathbb{N}$ *and* $\{d_i\}_{i\in I} \subseteq \wp(\Delta)$, *we have* $d \rightarrow^{;} (\cap_{i\in I} d_i) = \cap_{i\in I}(d \rightarrow^{;} d_i)$.

Proof. We have

$$\begin{aligned} d \rightarrow^{;} \left(\bigcap_{i\in I} d_i\right) &= \left\{\delta \in \Delta \,\middle|\, \text{for every } \overline{\delta} \in d \text{ we have } \overline{\delta}; \delta \in \bigcap_{i\in I} d_i\right\} \\ &= \bigcap_{i\in I} \left\{\delta \in \Delta \,\middle|\, \text{for every } \overline{\delta} \in d \text{ we have } \overline{\delta}; \delta \in d_i\right\} \\ &= \bigcap_{i\in I} (d \rightarrow^{;} d_i). \quad \square \end{aligned}$$

From now on, we will omit ; in $\rightarrow^{;}$.

We prove now that any abstract domain A of $\wp(\Delta)$, closed by linear refinement, is condensing. The intuition is that in that case A embeds all possible dependences propagating the abstract property under composition, so that a concrete element c and its abstraction $\rho(c)$ over A are equivalent w.r.t. composition (Definition 17).

Theorem 2. *Let $\wp(\Delta)$ be the concrete domain and A be an abstract domain such that $A = A \rightarrow A$. Then A is condensing.*

Proof. We prove the condensing property of Definition 17. Let ρ be the abstraction map induced by A, which since $A = A \rightarrow A$ corresponds to the abstraction map induced by $A \rightarrow A$. By expansivity and monotonicity of ρ we have

$$\rho(\rho(d_1); d_2) \subseteq \rho(\rho(d_1); \rho(d_2)) \supseteq \rho(d_1; \rho(d_2)).$$

If we prove that for every $a \in A$ such that $a \supseteq \rho(d_1); d_2$ we have $a \supseteq \rho(d_1); \rho(d_2)$ we conclude that $\rho(\rho(d_1); d_2) = \cap\{a \in A \mid a \supseteq \rho(d_1); d_2\} \supseteq \cap\{a \in A \mid a \supseteq \rho(d_1); \rho(d_2)\} = \rho(\rho(d_1); \rho(d_2))$. Similarly, if we prove that for every $a \in A$ such that $a \supseteq d_1; \rho(d_2)$ we have $a \supseteq \rho(d_1); \rho(d_2)$ we conclude that $\rho(d_1; \rho(d_2)) = \cap\{a \in A \mid a \supseteq d_1; \rho(d_2)\} \supseteq \cap\{a \in A \mid a \supseteq \rho(d_1); \rho(d_2)\} = \rho(\rho(d_1); \rho(d_2))$. Hence we conclude that the condensing property holds:

$$\rho(\rho(d_1); d_2) = \rho(\rho(d_1); \rho(d_2)) = \rho(d_1; \rho(d_2)).$$

Let us prove those results then.

- Let $a \in A$ be such that $\rho(d_1); d_2 \subseteq a$. Then $d_2 \subseteq \rho(d_1) \rightarrow a$ and since $\rho(d_1) \rightarrow a \in A \rightarrow A = A$, we have $\rho(d_2) \subseteq \rho(d_1) \rightarrow a$ *i.e.,* $\rho(d_1); \rho(d_2) \subseteq a$.
- Let $a \in A$ be such that $d_1; \rho(d_2) \subseteq a$. The sets of denotations d_1 and $\rho(d_1)$ have the same abstract properties *i.e.,* by definition of ρ, for every $a' \in A$ we have $a' \supseteq d_1$ if and only if $a' \supseteq \rho(d_1)$. Since $\rho(d_2)$ belongs to $A \rightarrow A$ it has the form $(a_1^1 \rightarrow a_2^1) \cap \cdots \cap (a_1^q \rightarrow a_2^q)$ with $a_1^i, a_2^i \in A$ for every $i = 1, \ldots, q$. We conclude that $d_1; \rho(d_2) \subseteq a$ entails that $a \in \{a_2^i \mid 1 \leq i \leq q \text{ and } d_1 \subseteq a_1^i\} = \{a_2^i \mid 1 \leq i \leq q \text{ and } \rho(d_1) \subseteq a_1^i\}$ and hence $\rho(d_1); \rho(d_2) \subseteq a$. □

Theorem 1 requires compatibility with *return* to conclude that a condensing abstract domain induces abstract operational semantics and abstract denotational semantics of the same precision. The following result shows that, under certain conditions, if a domain is compatible with *return* then also its linear refinement is compatible with *return*.

Proposition 5. *Let A be an abstract domain compatible with return. If $A \subseteq A \rightarrow A$ and $A \rightarrow A$ is condensing then $A \rightarrow A$ is compatible with return.*

Proof. Let $a_1 \rightarrow a_2 \in A \rightarrow A$, with $a_1, a_2 \in A$, f be a function name in P, $v_0 \in V$ and $d_1, d_2 \subseteq \Delta$. Let ρ be the abstraction map induced by A and $\rho^{\rightarrow}$ the abstraction map induced by $A \rightarrow A$. We have

$$\begin{aligned}
& a_1 \rightarrow a_2 \supseteq \mathit{return}_{f,v_0}(\rho^{\rightarrow}(d_1), \rho^{\rightarrow}(d_2)) \\
& \text{iff } a_2 \supseteq a_1; \mathit{return}_{f,v_0}(\rho^{\rightarrow}(d_1), \rho^{\rightarrow}(d_2)) \\
\text{(Lemma 2)} \quad & \text{iff } a_2 \supseteq \mathit{return}_{f,v_0}((a_1; \rho^{\rightarrow}(d_1)), (a_1; \rho^{\rightarrow}(d_2))) \\
(*) \quad & \text{iff } a_2 \supseteq \mathit{return}_{f,v_0}((a_1; d_1), (a_1; d_2)) \\
\text{(Lemma 2)} \quad & \text{iff } a_2 \supseteq a_1; \mathit{return}_{f,v_0}(d_1, d_2) \\
& \text{iff } a_1 \rightarrow a_2 \supseteq \mathit{return}_{f,v_0}(d_1, d_2)
\end{aligned}$$

where $(*)$ follows since, by expansivity and monotonicity of $\rho^{\rightarrow}$, $return_{f,v_0}((a_1; \rho^{\rightarrow}(d_1)), (a_1; \rho^{\rightarrow}(d_2))) \supseteq return_{f,v_0}((a_1; d_1), (a_1; d_2))$; conversely,

$$\begin{aligned}
& a_2 \supseteq return_{f,v_0}((a_1; d_1), (a_1; d_2)) \\
& \Rightarrow \rho(a_2) \supseteq \rho(return_{f,v_0}((a_1; d_1), (a_1; d_2))) \\
(A \text{ is compatible}) \quad & \Rightarrow a_2 \supseteq \rho(return_{f,v_0}(\rho(a_1; d_1), \rho(a_1; d_2))) \\
(**) \quad & \Rightarrow a_2 \supseteq return_{f,v_0}(\rho^{\rightarrow}(a_1; d_1), \rho^{\rightarrow}(a_1; d_2)) \\
(A \rightarrow A \text{ condens.}) \quad & \Rightarrow a_2 \supseteq return_{f,v_0}(\rho^{\rightarrow}(a_1; \rho^{\rightarrow}(d_1)), \rho^{\rightarrow}(a_1; \rho^{\rightarrow}(d_2))) \\
(\rho^{\rightarrow} \text{ is expansive}) \quad & \Rightarrow a_2 \supseteq return_{f,v_0}((a_1; \rho^{\rightarrow}(d_1)), (a_1; \rho^{\rightarrow}(d_2)))
\end{aligned}$$

where $(**)$ follows since $A \subseteq A \rightarrow A$ entails that $\rho(d) \supseteq \rho^{\rightarrow}(d)$ for every $d \subseteq \Delta$.

Every element of $A \rightarrow A$ has the form $(a_1^1 \rightarrow a_2^1) \cap \cdots \cap (a_1^q \rightarrow a_2^q)$, so that

$$\begin{aligned}
& (a_1^1 \rightarrow a_2^1) \cap \cdots \cap (a_1^q \rightarrow a_2^q) \supseteq return_{f,v_0}(\rho^{\rightarrow}(d_1), \rho^{\rightarrow}(d_2)) \\
& \text{iff } a_1^i \rightarrow a_2^i \supseteq return_{f,v_0}(\rho^{\rightarrow}(d_1), \rho^{\rightarrow}(d_2)) \text{ for every } 1 \leq i \leq q \\
& \text{iff } a_1^i \rightarrow a_2^i \supseteq return_{f,v_0}(d_1, d_2) \text{ for every } 1 \leq i \leq q \\
& \text{iff } (a_1^1 \rightarrow a_2^1) \cap \cdots \cap (a_1^q \rightarrow a_2^q) \supseteq return_{f,v_0}(d_1, d_2).
\end{aligned}$$

We conclude that

$$\begin{aligned}
& \rho^{\rightarrow}(return_{f,v_0}(\rho^{\rightarrow}(d_1), \rho^{\rightarrow}(d_2))) \\
& = \cap\{a \in A \rightarrow A \mid a \supseteq return_{f,v_0}(\rho^{\rightarrow}(d_1), \rho^{\rightarrow}(d_2))\} \\
& = \cap\{a \in A \rightarrow A \mid a \supseteq return_{f,v_0}(d_1, d_2)\} \\
& = \rho^{\rightarrow}(return_{f,v_0}(d_1, d_2)). \quad \square
\end{aligned}$$

8. The linear refinement $\mathsf{C} \rightarrow \mathsf{C}$ is optimal and condensing

Definition 4 gives a basic domain C for constancy which models the set of variables which are constant in the output of a denotation. If we linearly refine C into $\mathsf{C} \rightarrow \mathsf{C}$, we obtain a new abstract domain for *constancy propagation*.

Example 19. The denotation δ_1 of Example 2 is such that by keeping x constant in the input variable y is constant in the output. Hence $\delta_1 \in \mathbf{x} \rightarrow \mathbf{y}$. The denotation δ_4 of Example 4 is such that $\delta_4 \in \mathbf{y} \rightarrow \mathbf{x}$ since if y is constant in the input of δ_4 then x is also constant in its output (if y is chosen in such a way to make δ_4 undefined, the value of x in the output of δ_4 is always *undefined*). The denotation δ_5 of Example 5 is such that $\delta_5 \in \mathbf{y} \rightarrow \mathbf{x}$. This is because by keeping y constant in the input of δ_5 variable x is constant in its output. Note that $\delta_5 \notin \mathbf{O} \rightarrow \mathbf{x}$ since by leaving y free in the input of δ_5 then variable x can have both values 4 and 5 in its output, so that it is not constant. $\square$

Lemma 6. *Let $v \in V$. We have $\mathbf{O} \rightarrow \mathbf{v} = \mathbf{v}$.*

Proof. Let $\delta \in \mathbf{O} \rightarrow \mathbf{v}$ and ι be the identity denotation, which is such that $\iota(\sigma) = \sigma$ for every $\sigma \in \Sigma$. We have $\iota \in \Delta = \mathbf{O}$, so that $\iota; \delta = \delta \in \mathbf{v}$. Conversely, let $\delta \in \mathbf{v}$. Constancy is closed by composition (Lemma 1), so $\overline{\delta}; \delta \in \mathbf{v}$ for every $\overline{\delta} \in \Delta = \mathbf{O}$. Hence $\delta \in \mathbf{O} \rightarrow \mathbf{v}$. $\square$

The domain $\mathsf{C} \rightarrow \mathsf{C}$ is shown in Fig. 3 for the case when $V = \{x, y\}$. The element $\mathbf{x}$ stands for $\mathbf{O} \rightarrow \mathbf{x}$. There are no arrows with more than one variable on the right since by Proposition 4 we can always factorise $\mathbf{L} \rightarrow \mathbf{xy}$ into $(\mathbf{L} \rightarrow \mathbf{x}) \cap (\mathbf{L} \rightarrow \mathbf{y})$. Moreover, there are no arrows with more than one variable on the left since otherwise, for the case $V = \{x, y\}$, we could only have $\mathbf{xy} \rightarrow \mathbf{x}$ and $\mathbf{xy} \rightarrow \mathbf{y}$ which are both tautological arrows and hence equivalent to the top $\Delta_{\{x,y\}}$ (if the initial values of *all* variables are constant then the final value of each variable can only be constant). Note that for larger V we would have non-tautological arrows such as $\mathbf{xy} \rightarrow \mathbf{x}$ and $\mathbf{xy} \rightarrow \mathbf{z}$.

We prove now that $\mathsf{C} \rightarrow \mathsf{C}$ is strictly more expressive than C *i.e.,* that it contains all the abstract properties expressed by C.

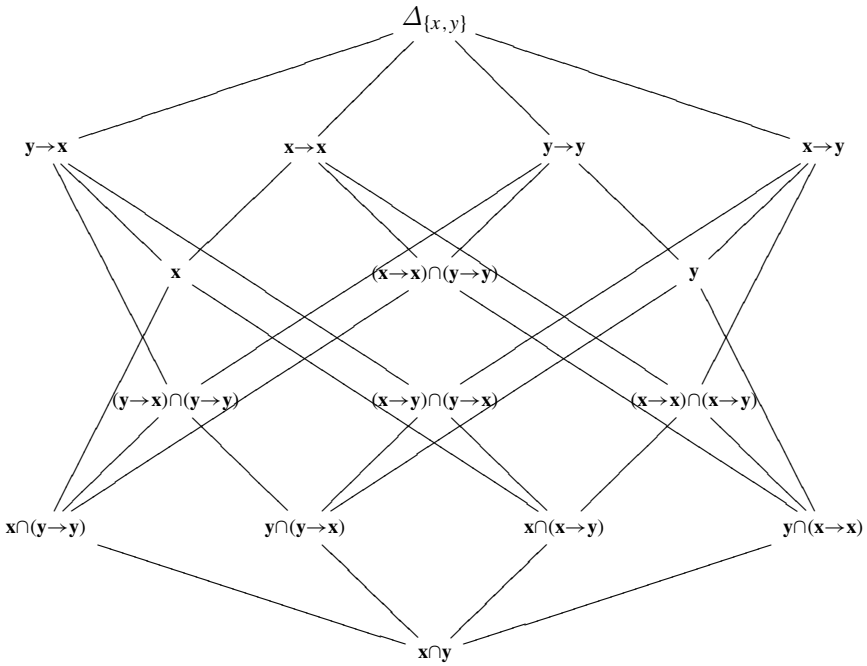

Fig. 3. The abstract domain $\mathsf{C}_{\{x,y\}} \to \mathsf{C}_{\{x,y\}}$.

Proposition 6 ($\mathsf{C} \to \mathsf{C}$ *is More Expressive than* C). *We have* $\mathsf{C} \subseteq \mathsf{C} \to \mathsf{C}$. *If* $\#V \geq 2$ *the inclusion is strict. If* $\#V = 1$ *we have* $\mathsf{C} = \mathsf{C} \to \mathsf{C}$.

Proof. Since $\mathbf{O} \in \mathsf{C}$ then for every $v \in V$ we have $\mathbf{O} \to \mathbf{v} \in \mathsf{C} \to \mathsf{C}$. By Lemma 6 we have that $\mathbf{O} \to \mathbf{v} = \mathbf{v}$ so that $\mathbf{v} \in \mathsf{C} \to \mathsf{C}$, that is, $\mathsf{C} \subseteq \mathsf{C} \to \mathsf{C}$.

Assume that $\#V \geq 2$. To prove the strict inclusion, let $x, y \in V$, $x \not\equiv y$. Let ι be the identity denotation. Since no variable is constant in ι, we have $\iota \in \mathbf{O}$ and $\iota \notin c$ for all $c \in \mathsf{C} \setminus \{\mathbf{O}\}$. We have $\iota \in \mathbf{x} \to \mathbf{x}$. This is because for all $\overline{\delta} \in \mathbf{x}$ we have $\overline{\delta}; \iota = \overline{\delta} \in \mathbf{x}$. To prove that $\mathsf{C} \subset \mathsf{C} \to \mathsf{C}$ it is then enough to show that $\mathbf{x} \to \mathbf{x} \neq \mathbf{O}$. Consider δ such that $\delta(\sigma) = \sigma[x \mapsto \sigma(y)]$. We have $\delta \in \mathbf{O}$ but $\delta \notin \mathbf{x} \to \mathbf{x}$, since if we take $\overline{\delta} \in \mathbf{x}$ such that $\overline{\delta}(\sigma) = \sigma[x \mapsto 0]$ we have $\overline{\delta}; \delta = \delta \notin \mathbf{x}$ (no variable is constant in δ).

If $\#V = 1$ then $\mathbf{v} \to \mathbf{v} = \mathbf{O} \to \mathbf{O} = \mathbf{v} \to \mathbf{O} = \mathbf{O}$ and $\mathbf{O} \to \mathbf{v} = \mathbf{v}$ (Lemma 6). Hence $\mathsf{C} \to \mathsf{C} = \{\mathbf{O}, \mathbf{O} \to \mathbf{v}\} = \{\mathbf{O}, \mathbf{v}\} = \mathsf{C}$. □

We show now that, if we linearly refine $\mathsf{C} \to \mathsf{C}$, we end up with $\mathsf{C} \to \mathsf{C}$ itself. Hence $\mathsf{C} \to \mathsf{C}$ contains already all possible dependences between constancy of variables and in this sense it is *optimal* [12]. To prove this result we need three technical lemmas.

Lemma 7. *Let* $vs_1, vs_2 \subseteq V$, $\sigma_1, \sigma_2 \in \Sigma_V$ *be such that* $\sigma_1|_{vs_1} = \sigma_2|_{vs_1}$ *and* $\delta \in \mathbf{vs_1} \to \mathbf{vs_2}$. *Then* $\delta(\sigma_1)|_{vs_2} = \delta(\sigma_2)|_{vs_2}$.

Proof. Define δ' such that $\delta'(\sigma) = \sigma[z \to \sigma_1(z) \mid z \in vs_1]$ for all $\sigma \in \Sigma_V$. We have $\delta'(\sigma_1) = \sigma_1$ and $\delta'(\sigma_2) = \sigma_2$. For all $\sigma_1', \sigma_2' \in \Sigma_V$ and $z \in vs_1$ we have $\delta'(\sigma_1')(z) = \sigma_1(z) = \delta'(\sigma_2')(z)$ so that $\delta' \in \mathbf{vs_1}$. Since $\delta \in \mathbf{vs_1} \to \mathbf{vs_2}$, we have $\delta'; \delta \in \mathbf{vs_2}$ and hence $\delta(\sigma_1)|_{vs_2} = \delta(\delta'(\sigma_1))|_{vs_2} = \delta(\delta'(\sigma_2))|_{vs_2} = \delta(\sigma_2)|_{vs_2}$. □

The second lemma states that two arrows with the same right hand side are equivalent to one arrow.

Lemma 8. *Let* $vs_1, vs_2 \subseteq V$, $vs = vs_1 \cap vs_2$ *and* $y \in V$. *We have*

$$(\mathbf{vs_1} \to \mathbf{y}) \cap (\mathbf{vs_2} \to \mathbf{y}) = \mathbf{vs} \to \mathbf{y}.$$

Proof. Let $\delta \in \mathbf{vs} \to \mathbf{y}$. We prove that $\delta \in (\mathbf{vs_1} \to \mathbf{y}) \cap (\mathbf{vs_2} \to \mathbf{y})$. Let hence $\overline{\delta} \in \mathbf{vs_1}$. Since $vs_1 \supseteq vs$ we have $\mathbf{vs_1} \subseteq \mathbf{vs}$. Then $\overline{\delta} \in \mathbf{vs}$ that is $\overline{\delta}; \delta \in \mathbf{y}$ so that $\delta \in \mathbf{vs_1} \to \mathbf{y}$. It can be proved symmetrically that $\delta \in \mathbf{vs_2} \to \mathbf{y}$. Then $\delta \in (\mathbf{vs_1} \to \mathbf{y}) \cap (\mathbf{vs_2} \to \mathbf{y})$.

Conversely, let $\delta \in (\mathbf{vs_1} \to \mathbf{y}) \cap (\mathbf{vs_2} \to \mathbf{y})$. We prove that $\delta \in \mathbf{vs} \to \mathbf{y}$.

We first prove the case when $vs = \varnothing$, which amounts to proving that $\delta \in \mathbf{y}$. Let $\sigma_1, \sigma_2 \in \Sigma_V$ be arbitrary. Define $\sigma_1' = \sigma_1[v \mapsto \sigma_2(v) \mid v \in vs_2]$. We have $\sigma_1'|_{vs_2} = \sigma_2|_{vs_2}$. Moreover, since $vs_1 \cap vs_2 = \varnothing$, we have $\sigma_1|_{vs_1} = \sigma_1'|_{vs_1}$. From the choice of δ and by Lemma 7 we conclude that $\delta(\sigma_1)(y) = \delta(\sigma_1')(y) = \delta(\sigma_2)(y)$. Since σ_1 and σ_2 are arbitrary, we have $\delta \in \mathbf{y}$.

Assume now that $vs \neq \varnothing$. Assume by contradiction that $\delta \notin \mathbf{vs} \to \mathbf{y}$. Then there is $\overline{\delta} \in \mathbf{vs}$ such that $\overline{\delta}; \delta \notin \mathbf{y}$, that is there are $\sigma_1, \sigma_2 \in \Sigma$ such that $\delta(\overline{\delta}(\sigma_1))(y) \neq \delta(\overline{\delta}(\sigma_2))(y)$. From $\overline{\delta} \in \mathbf{vs}$ and $vs \neq \varnothing$ we must have that $\overline{\delta}$ is always *undefined* or it is total (Definition 3). But $\overline{\delta}$ cannot be always *undefined*, since this would entail that $\delta(\overline{\delta}(\sigma_1))(y) = \textit{undefined} = \delta(\overline{\delta}(\sigma_2))(y)$, which is a contradiction. We conclude that $\overline{\delta}$ is total. We can hence define $\overline{\delta}_1(\sigma) = \overline{\delta}(\sigma)[v \mapsto \overline{\delta}(\sigma_2)(v) \mid v \in vs_2 \setminus vs]$. By construction, $\overline{\delta}$ and $\overline{\delta}_1$ coincide on vs_1 so that, from $\delta \in \mathbf{vs_1} \to \mathbf{y}$, we conclude that $\delta(\overline{\delta}(\sigma_1))(y) = \delta(\overline{\delta}_1(\sigma_1))(y)$ and $\delta(\overline{\delta}(\sigma_2))(y) = \delta(\overline{\delta}_1(\sigma_2))(y)$. Let us further define $\overline{\delta}_2(\sigma) = \overline{\delta}_1(\sigma)[v \mapsto \overline{\delta}(\sigma_1)(v) \mid v \in vs_1 \setminus vs]$. By construction, $\overline{\delta}_1$ and $\overline{\delta}_2$ coincide on vs_2 so that, from $\delta \in \mathbf{vs_2} \to \mathbf{y}$, we have $\delta(\overline{\delta}_1(\sigma_1))(y) = \delta(\overline{\delta}_2(\sigma_1))(y)$ and $\delta(\overline{\delta}_1(\sigma_2))(y) = \delta(\overline{\delta}_2(\sigma_2))(y)$. By construction, we have $\overline{\delta}_2 \in \mathbf{vs_1} \cup \mathbf{vs_2}$. From the choice of δ we have $\delta(\overline{\delta}_2(\sigma_1))(y) = \delta(\overline{\delta}_2(\sigma_2))(y)$. In conclusion we have $\delta(\overline{\delta}(\sigma_1))(y) = \delta(\overline{\delta}_1(\sigma_1))(y) = \delta(\overline{\delta}_2(\sigma_1))(y) = \delta(\overline{\delta}_2(\sigma_2)))(y) = \delta(\overline{\delta}_1(\sigma_2))(y) = \delta(\overline{\delta}(\sigma_2))(y)$, which is a contradiction. We conclude that $\delta \in \mathbf{vs} \to \mathbf{y}$. □

The third technical lemma shows that it is possible to write an element of $(\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C})$ as an equivalent element of $\mathsf{C} \to \mathsf{C}$.

Lemma 9. *Let $vs_2 \subset V$, $z \in V$, $I \subset \mathbb{N}$, $vs_1^i \subseteq V$ and $y^i \in V$ for every $i \in I$. Let the y^i be distinct and hence I be finite. Let $L = \{y^i \mid i \in I \text{ and } vs_1^i \subseteq vs_2\}$. We have*

$$\bigcap_{i \in I} (\mathbf{vs_1^i} \to \mathbf{y^i}) \to (\mathbf{vs_2} \to \mathbf{z}) = \mathbf{L} \to \mathbf{z}.$$

Proof. Let $\delta \in \mathbf{L} \to \mathbf{z}$. We prove that $\delta \in \cap_i (\mathbf{vs_1^i} \to \mathbf{y^i}) \to (\mathbf{vs_2} \to \mathbf{z})$. Let hence $\delta_1 \in \cap_i (\mathbf{vs_1^i} \to \mathbf{y^i})$ and $\delta_2 \in \mathbf{vs_2}$. We have to prove that $\delta_2; \delta_1; \delta \in \mathbf{z}$. From $\delta_2 \in \mathbf{vs_2}$ and since $\delta_1 \in \cap_i (\mathbf{vs_1^i} \to \mathbf{y^i})$ and $vs_1^i \subseteq vs_2$ for every $y^i \in L$, we have $\delta_2; \delta_1 \in \mathbf{L}$. By the choice of δ we have $\delta_2; \delta_1; \delta \in \mathbf{z}$, as desired.

Let conversely $\delta \in \cap_i (\mathbf{vs_1^i} \to \mathbf{y^i}) \to (\mathbf{vs_2} \to \mathbf{z})$. We prove that $\delta \in \mathbf{L} \to \mathbf{z}$. Let hence $\overline{\delta} \in \mathbf{L}$ and assume by contradiction that there are $\sigma_1, \sigma_2 \in \Sigma_V$ such that $(\overline{\delta}; \delta)(\sigma_1)(z) \neq (\overline{\delta}; \delta)(\sigma_2)(z)$, that is $\delta(\overline{\delta}(\sigma_1))(z) \neq \delta(\overline{\delta}(\sigma_2))(z)$. Let $Y = \{y^i \mid i \in I\}$ and let us define

$$\delta_2(\sigma)(v) = \begin{cases} 0 & \text{if } v \in vs_2 \\ 1 & \text{if } v \notin vs_2 \text{ and } \sigma = \sigma_1 \\ 2 & \text{if } v \notin vs_2 \text{ and } \sigma \neq \sigma_1 \end{cases}$$

$$\delta_1(\sigma)(v) = \begin{cases} \overline{\delta}(\sigma)(v) & \text{if } v \in L \\ \overline{\delta}(\sigma_1)(v) & \text{if } v \notin L,\ v = y^i \text{ for some } i \\ & \text{and } \sigma(w) = 1 \text{ for all } w \in vs_1^i \setminus vs_2 \\ \overline{\delta}(\sigma_2)(v) & \text{if } v \notin L,\ v = y^i \text{ for some } i \\ & \text{and } \sigma(w) \neq 1 \text{ for some } w \in vs_1^i \setminus vs_2 \\ \overline{\delta}(\sigma_1)(v) & \text{if } v \notin Y \text{ and } \sigma(w) = 1 \text{ for all } w \notin vs_2 \\ \overline{\delta}(\sigma_2)(v) & \text{if } v \notin Y \text{ and } \sigma(w) \neq 1 \text{ for some } w \notin vs_2. \end{cases}$$

The idea underlying these definitions is that δ_2 makes the variables in vs_2 constant while the other variables are constrained to 1 or 2 depending on the parameter σ, so that we can distinguish when δ_2 is applied to σ_1 or to σ_2. This distinction is possible since in the hypotheses of this lemma we have $vs_2 \subset V$, so that there is at least a variable which is not in vs_2. The definition of δ_1 is such that δ_1 makes the variables in L constant, since $\overline{\delta} \in \mathbf{L}$. Moreover, the variables $v = y^i \in Y \setminus L$ are bound to $\overline{\delta}(\sigma_1)(v)$ if there is a variable in $vs_1^i \setminus vs_2$ which is bound to 1, and to $\overline{\delta}(\sigma_2)(v)$ otherwise. Note that if $v = y^i \in Y \setminus L$ then $vs_1^i \setminus vs_2 \neq \varnothing$. The variables not in Y are treated similarly, by looking at the variables which are not in vs_2. As a consequence of these definitions, we have:

$\delta_2 \in \mathbf{vs_2}$: this is because δ_2 constantly binds to 0 the variables in vs_2;

$\delta_1 \in \cap_i(\mathbf{vs_1^i} \to \mathbf{y^i})$: let $\overline{\delta}' \in \mathbf{vs_1^i}$ and $\sigma' \in \Sigma$. If $y^i \in L$ then we have that $\delta_1(\overline{\delta}'(\sigma'))(y_i) = \overline{\delta}(\overline{\delta}'(\sigma'))(y^i)$ which is a constant by definition of $\overline{\delta}$; if $y^i \notin L$ then $\delta_1(\overline{\delta}'(\sigma'))(y^i) = \overline{\delta}(\sigma_1)(y^i)$, constantly, or $\delta_1(\overline{\delta}'(\sigma'))(y^i) = \overline{\delta}(\sigma_2)(y^i)$, constantly. In all cases we conclude that $\delta_1(\overline{\delta}'(\sigma')) \in \mathbf{y^i}$ so that $\delta_1 \in \mathbf{vs_1^i} \to \mathbf{y^i}$. Since this is true for every i, we have $\delta_1 \in \cap_i(\mathbf{vs_1^i} \to \mathbf{y^i})$;

$\delta_1(\delta_2(\sigma_1)) = \overline{\delta}(\sigma_1)$: Let $v \in V$. If $v \in L$ we have $\delta_1(\delta_2(\sigma_1))(v) = \overline{\delta}(\delta_2(\sigma_1))(v) = \overline{\delta}(\sigma_1)(v)$ since $\overline{\delta} \in \mathbf{L}$. If $v \notin L$ and $v = y^i$ for some i, since $\delta_2(\sigma_1)(w) = 1$ for all $w \notin vs_2$, we have $\delta_1(\delta_2(\sigma_1))(v) = \overline{\delta}(\sigma_1)(v)$. For the same reason, when $v \notin Y$, then $\delta_1(\delta_2(\sigma_1))(v) = \overline{\delta}(\sigma_1)(v)$. In conclusion, for every $v \in V$ we have $\delta_1(\delta_2(\sigma_1))(v) = \overline{\delta}(\sigma_1)(v)$;

$\delta_1(\delta_2(\sigma_2)) = \overline{\delta}(\sigma_2)$: symmetrically to the case above.

As a consequence, by the choice of δ we have $\delta_2; \delta_1; \delta \in \mathbf{z}$ so that

$$\delta(\overline{\delta}(\sigma_1))(z) = \delta(\delta_1(\delta_2(\sigma_1)))(z) = \delta(\delta_1(\delta_2(\sigma_2)))(z) = \delta(\overline{\delta}(\sigma_2))(z)$$

which contradicts the choice of σ_1 and σ_2. We conclude that $\overline{\delta}; \delta \in \mathbf{z}$ that is $\delta \in \mathbf{L} \to \mathbf{z}$, as desired. □

It is easy now to prove that $\mathsf{C} \to \mathsf{C}$ is closed w.r.t. linear refinement.

Theorem 3 ($\mathsf{C} \to \mathsf{C}$ *is Closed w.r.t. Linear Refinement*). *We have* $(\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C}) = \mathsf{C} \to \mathsf{C}$.

Proof. We prove first that $\mathsf{C} \to \mathsf{C} \subseteq (\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C})$. Let $\mathbf{y^1} \cdots \mathbf{y^n} \to \mathbf{z} \in \mathsf{C} \to \mathsf{C}$. If we prove that $\mathbf{y^1} \cdots \mathbf{y^n} \to \mathbf{z} \in (\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C})$ we have this first inclusion by Proposition 4 and since both $\mathsf{C} \to \mathsf{C}$ and $(\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C})$ are Moore families. But this is true since we can choose $vs_2 = vs_1^i = \varnothing$ for all $i = 1, \ldots, n$ and from Lemma 9 we conclude that

$$\mathbf{y^1} \cdots \mathbf{y^n} \to \mathbf{z} = \bigcap_{i=0}^{n} (\mathbf{O} \to y^i) \to (\mathbf{O} \to \mathbf{z}) \in (\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C}).$$

We prove now the converse inclusion, that is $(\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C}) \subseteq \mathsf{C} \to \mathsf{C}$. Let $y^1, \ldots, y^m, z \in V$, $vs_1^1, \ldots, vs_1^m, vs_2 \subseteq V$ and consider $\cap_{i=0}^m(\mathbf{vs_1^i} \to \mathbf{y^i}) \to (\mathbf{vs_2} \to \mathbf{z}) \in (\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C})$. If we prove that it belongs to $\mathsf{C} \to \mathsf{C}$ then we have the thesis by Proposition 4 and since both $\mathsf{C} \to \mathsf{C}$ and $(\mathsf{C} \to \mathsf{C}) \to (\mathsf{C} \to \mathsf{C})$ are Moore families. But this is true when $vs_2 = V$, since in such a case $\mathbf{vs_2} \to \mathbf{z} = \mathbf{O}$ and $\cap_i(\mathbf{vs_1^i} \to \mathbf{y^i}) \to (\mathbf{vs_2} \to \mathbf{z}) = \cap_i(\mathbf{vs_1^i} \to \mathbf{y^i}) \to \mathbf{O} = \mathbf{O} \in \mathsf{C} \to \mathsf{C}$. If instead $vs_2 \subset V$, the thesis follows by Lemma 9. □

Theorem 3 entails that $\mathsf{C} \to \mathsf{C}$ enjoys the condensing property (Theorem 2). In order to conclude that the abstract denotational semantics over $\mathsf{C} \to \mathsf{C}$ has the same precision as the abstract operational semantics over $\mathsf{C} \to \mathsf{C}$, we still have to prove that $\mathsf{C} \to \mathsf{C}$ is compatible with *return* (Theorem 1).

Proposition 7. *The abstract domain* C *of Definition 5 is compatible with return.*

Proof. Let $v, v_0 \in V$, f be a function name in P and $d_1, d_2 \subseteq \Delta$. We have

$\mathbf{v} \supseteq \mathit{return}_{f,v_0}(\rho(d_1), \rho(d_2))$
iff for all $\delta \in \mathit{return}_{f,v_0}(\rho(d_1), \rho(d_2))$ we have $v \in \mathit{const}(\delta)$
iff $v \not\equiv v_0$ and for all $\delta_1 \in \rho(d_1)$ we have $v \in \mathit{const}(\delta_1)$
or $v \equiv v_0$ and for all $\delta_2 \in \rho(d_2)$ we have $f \in \mathit{const}(\delta_2)$
iff ($v \not\equiv v_0$ and $\mathbf{v} \supseteq \rho(d_1)$) or ($v \equiv v_0$ and $\mathbf{f} \supseteq \rho(d_2)$)
$(*)$ iff ($v \not\equiv v_0$ and $\mathbf{v} \supseteq d_1$) or ($v \equiv v_0$ and $\mathbf{f} \supseteq d_2$)
iff $\mathbf{v} \supseteq \mathit{return}_{f,v_0}(d_1, d_2)$

where $(*)$ follows from the fact that $\mathbf{v} \supseteq \rho(d_1)$ entails $\mathbf{v} \supseteq d_1$ by expansivity of ρ; conversely, $\mathbf{v} \supseteq d_1$ entails, by monotonicity of ρ, that $\rho(\mathbf{v}) \supseteq \rho(d_1)$ which, by idempotency of ρ, entails that $\mathbf{v} \supseteq \rho(d_1)$. The same proof holds for $\mathbf{f} \supseteq d_2$. Since every element $a \in \mathsf{C}$ is such that $a = \mathbf{v_1} \cap \cdots \cap \mathbf{v_k}$ (Definition 5), we conclude that

$$
\begin{aligned}
&a \supseteq \mathit{return}_{f,v_0}(\rho(d_1), \rho(d_2))\\
&\text{iff for every } 1 \leq i \leq k \text{ we have } \mathbf{v_i} \supseteq \mathit{return}_{f,v_0}(\rho(d_1), \rho(d_2))\\
&\text{iff for every } 1 \leq i \leq k \text{ we have } \mathbf{v_i} \supseteq \mathit{return}_{f,v_0}(d_1, d_2)\\
&\text{iff } a \supseteq \mathit{return}_{f,v_0}(d_1, d_2).
\end{aligned}
$$

As a consequence:

$$
\begin{aligned}
\rho(\mathit{return}_{f,v_0}(\rho(d_1), \rho(d_2))) &= \cap\{a \mid a \supseteq \mathit{return}_{f,v_0}(\rho(d_1), \rho(d_2))\}\\
&= \cap\{a \mid a \supseteq \mathit{return}_{f,v_0}(d_1, d_2)\}\\
&= \rho(\mathit{return}_{f,v_0}(d_1, d_2)). \quad \square
\end{aligned}
$$

Theorem 4 ($\mathsf{C} \to \mathsf{C}$ *is Condensing*). *The abstract domain* $\mathsf{C} \to \mathsf{C}$ *is condensing and compatible with return. Moreover, the abstract denotational semantics (input-independent) over* $\mathsf{C} \to \mathsf{C}$ *has the same precision as the abstract operational semantics (input-driven) over* $\mathsf{C} \to \mathsf{C}$.

Proof. By Theorems 2 and 3 we conclude that $\mathsf{C} \to \mathsf{C}$ is condensing. By Propositions 7, 6 and 5 we conclude that $\mathsf{C} \to \mathsf{C}$ is compatible with *return*. By Theorem 1 we conclude that the abstract denotational semantics over $\mathsf{C} \to \mathsf{C}$ has the same precision as the abstract operational semantics over $\mathsf{C} \to \mathsf{C}$. $\square$

Example 20. We have seen in Example 17 that the abstract operational semantics over C can be more precise than the abstract denotational semantics over C. If we consider $\mathsf{C} \to \mathsf{C}$ instead, the two semantics have the same precision. Namely, assuming $V = \{v, w\}$, we still have $\rho(\mathbf{O}; v := 2) = \mathbf{v}$ and $\rho(\mathbf{v}, w := v) = \mathbf{vw}$. Hence we have

$$
\frac{\langle \mathtt{v} := 2 \,||\, \mathbf{O}\rangle \Rightarrow^{\mathsf{C}\to\mathsf{C}} \mathbf{v} \quad \langle \mathtt{w} := \mathtt{v} \,||\, \mathbf{v}\rangle \Rightarrow^{\mathsf{C}\to\mathsf{C}} \mathbf{vw}}{\langle \mathtt{v} := 2; \mathtt{w} := \mathtt{v} \,||\, \mathbf{O}\rangle \Rightarrow^{\mathsf{C}\to\mathsf{C}} \mathbf{vw}}.
$$

The abstract denotational semantics of the same command yields the same result this time:

$$
\begin{aligned}
[\![\mathtt{v} := 2; \mathtt{w} := \mathtt{v}]\!]_I^{\mathsf{C}\to\mathsf{C}} &= [\![\mathtt{v} := 2]\!]_I^{\mathsf{C}\to\mathsf{C}} ;^{\mathsf{C}\to\mathsf{C}} [\![\mathtt{w} := \mathtt{v}]\!]_I^{\mathsf{C}\to\mathsf{C}}\\
&= \rho(v := 2);^{\mathsf{C}\to\mathsf{C}} \rho(w := v)\\
&= (\mathbf{v} \cap (\mathbf{w} \to \mathbf{w})) ;^{\mathsf{C}\to\mathsf{C}} (\mathbf{v} \to \mathbf{vw}) = \mathbf{vw}. \quad \square
\end{aligned}
$$

The conclusion of this section is that we have defined an abstract domain $\mathsf{C} \to \mathsf{C}$ for constancy propagation as the linear refinement of a basic domain C which expresses constancy of program variables. The domain $\mathsf{C} \to \mathsf{C}$ enjoys desirable properties such as optimality and condensing. We now provide a representation of the elements of $\mathsf{C} \to \mathsf{C}$ in terms of Boolean formulas, which can be efficiently implemented through binary decision diagrams [4].

9. A logical representation for $\mathsf{C} \to \mathsf{C}$

We show here that Boolean formulas can be used to represent the elements of $\mathsf{C} \to \mathsf{C}$. This will be achieved by defining a notion of *concretisation* of a Boolean formula into a set of denotations. All elements of $\mathsf{C} \to \mathsf{C}$ are sets of denotations and will be the concretisation of appropriate Boolean formulas.

Since the elements of $\mathsf{C} \to \mathsf{C}$ express dependences between constancy of variables in the input of a denotation and constancy of variables in its output, we need to distinguish such variables. Hence we write $\check{v}$ for the variable v in the input of a denotation, and $\hat{v}$ for the same variable in the output of a denotation, as in [9].

Definition 20 (*Input and Output Variables*). Let $vs \subseteq V$. We define $\check{vs} = \{\check{v} \mid v \in vs\}$ and $\hat{vs} = \{\hat{v} \mid v \in vs\}$. Let $vs \subseteq \{\check{v} \mid v \in V\} \cup \{\hat{v} \mid v \in V\}$. We use the notation $\wedge vs = \wedge\{v \mid v \in vs\}$. $\square$

Definition 21 (*Denotational Formulas*). The *denotational formulas* over V are the Boolean (propositional) formulas over the variables $\check{V} \cup \hat{V}$. A *model* of a denotation formula ϕ is a set $M \subseteq \check{V} \cup \hat{V}$ such that $M \models \phi$, where

$M \models true$
$M \models v$ if and only if $v \in M$
$M \models \phi_1 \wedge \phi_2$ if and only if $M \models \phi_1$ and $M \models \phi_2$
$M \models \phi_1 \wedge \phi_2$ if and only if $M \models \phi_1$ or $M \models \phi_2$
$M \models \phi_1 \Rightarrow \phi_2$ if and only if $M \not\models \phi_1$ or $M \models \phi_2$.

Two denotational formulas ϕ_1 and ϕ_2 are *equivalent* if and only if they have the same models. From now on, we consider denotational formulas modulo logical equivalence. □

Example 21. Given $V = \{x, y\}$, denotational formulas over V are $\hat{x} \wedge (\check{y} \Rightarrow \hat{y})$ as well as $\hat{x} \wedge (\hat{y} \Rightarrow \check{y})$. The formula $\hat{x} \wedge (\check{y} \Rightarrow \hat{y})$ is equivalent (and hence the same as) $(\check{y} \Rightarrow \hat{y}) \wedge \hat{x}$. □

We specify now the meaning or *concretisation* of a denotational formula ϕ. It is the set of denotations whose behaviour w.r.t. constancy is consistent with the propositional models of ϕ.

Definition 22 (*Concretisation of a Denotational Formula*). The *concretisation* of a denotational formula ϕ is

$$\gamma(\phi) = \{\delta \in \Delta \mid \text{for all } \overline{\delta} \in \Delta \text{ we have } \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models \phi\}. \quad \square$$

Example 22. Assume that $V = \{\mathrm{x}, \mathrm{y}\}$. The denotation δ_2 of Example 3 is such that $\delta_2(\sigma) = \sigma[\mathrm{x} \mapsto 4]$ for every $\sigma \in \Sigma$. Given $\overline{\delta} \in \Delta$, by Lemma 1 we have $\check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta_2) \supseteq \hat{const}(\delta_2) = \{\hat{\mathrm{x}}\}$. Moreover, if $\check{\mathrm{y}} \in \check{const}(\overline{\delta})$ then $\hat{\mathrm{y}} \in \hat{const}(\overline{\delta}; \delta_2)$. Hence $\check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta_2)$ is a model of $\hat{\mathrm{x}} \wedge (\check{\mathrm{y}} \Rightarrow \hat{\mathrm{y}})$ for all $\overline{\delta} \in \Delta$, that is $\delta_2 \in \gamma(\hat{\mathrm{x}} \wedge (\check{\mathrm{y}} \Rightarrow \hat{\mathrm{y}}))$. □

We prove that logical $\wedge$ is the abstraction of the intersection of sets of denotations.

Lemma 10 (*Concretisation of* $\wedge$). *Let ϕ_1, ϕ_2 be denotational formulas. Then $\gamma(\phi_1 \wedge \phi_2) = \gamma(\phi_1) \cap \gamma(\phi_2)$.*

Proof.

$$\begin{aligned}
\gamma(\phi_1 \wedge \phi_2) &= \left\{ \delta \in \Delta \,\middle|\, \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \text{ we have} \\ \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models (\phi_1 \wedge \phi_2) \end{array} \right\} \\
&= \left\{ \delta \in \Delta \,\middle|\, \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \text{ we have} \\ \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models \phi_1 \\ \text{and } \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models \phi_2 \end{array} \right\} \\
&= \left\{ \delta \in \Delta \,\middle|\, \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \text{ we have} \\ \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models \phi_1 \end{array} \right\} \\
&\quad \cap \left\{ \delta \in \Delta \,\middle|\, \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \text{ we have} \\ \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models \phi_2 \end{array} \right\} \\
&= \gamma(\phi_1) \cap \gamma(\phi_2). \quad \square
\end{aligned}$$

Example 23. By Lemma 10, $\gamma(\hat{x} \wedge (\check{y} \Rightarrow \hat{y})) = \gamma(\hat{x}) \cap \gamma(\check{y} \Rightarrow \hat{y})$. □

The concretisation of a formula which is just an output variable is the property expressing the constancy of that variable.

Lemma 11 (*Concretisation of* $\hat{x}$). *Let $x \in V$. We have $\gamma(\hat{x}) = \mathbf{x}$.*

Proof.

$$\begin{aligned}
\gamma(\hat{x}) &= \{\delta \in \Delta \mid \text{for all } \overline{\delta} \in \Delta \text{ we have } \check{const}(\overline{\delta}) \cup \hat{const}(\overline{\delta}; \delta) \models \hat{x}\} \\
&= \{\delta \in \Delta \mid \text{for all } \overline{\delta} \in \Delta \text{ we have } x \in const(\overline{\delta}; \delta)\} \\
&= \{\delta \in \Delta \mid x \in const(\delta)\} = \mathbf{x},
\end{aligned}$$

since if $x \in const(\overline{\delta}; \delta)$ for all $\overline{\delta} \in \Delta$ then $x \in const(\delta)$ since we can choose $\overline{\delta} = \iota$, the identity denotation. Conversely, if $x \in const(\delta)$ then by Lemma 1 we have $x \in const(\overline{\delta}; \delta)$. □

Example 24. By Lemma 11, $\gamma(\hat{x}) \cap \gamma(\check{y} \Rightarrow \hat{y}) = \mathbf{x} \cap \gamma(\check{y} \Rightarrow \hat{y})$. □

Logical implication corresponds to linear refinement.

Lemma 12 (*Concretisation of* $\Rightarrow$). *Let* $X = \{x_1, \ldots, x_n\}$, $X \subseteq V$ *and* $y \in V$. *Then* $\gamma(\wedge\check{X} \Rightarrow \hat{y}) = \gamma(\wedge\hat{X}) \to \gamma(\hat{y})$.

Proof.

$$\begin{aligned}
\gamma(\wedge\check{X} \Rightarrow \hat{y}) &= \left\{ \delta \in \Delta \;\middle|\; \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \text{ we have} \\ \check{con}st(\overline{\delta}) \cup \hat{con}st(\overline{\delta}; \delta) \models \wedge\check{X} \Rightarrow \hat{y} \end{array} \right\} \\
&= \left\{ \delta \in \Delta \;\middle|\; \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \\ (x_i \in const(\overline{\delta}) \text{ for all } i \text{ such that } 1 \leq i \leq n) \\ \text{entails } y \in const(\overline{\delta}; \delta) \end{array} \right\} \\
\text{(Lemma 11)} &= \left\{ \delta \in \Delta \;\middle|\; \begin{array}{l} \text{for all } \overline{\delta} \in \Delta \\ (\overline{\delta} \in \gamma(\hat{x}_i) \text{ for all } i \text{ such that } 1 \leq i \leq n) \\ \text{entails } \overline{\delta}; \delta \in \gamma(\hat{y}) \end{array} \right\} \\
\text{(Lemma 10)} &= \{\delta \in \Delta \mid \text{for all } \overline{\delta} \in \gamma(\wedge\hat{X}) \text{ we have } \overline{\delta}; \delta \in \gamma(\hat{y})\} \\
&= \gamma(\wedge\hat{X}) \to \gamma(\hat{y}). \quad \square
\end{aligned}$$

Example 25. By Lemma 12, $\mathbf{x} \cap \gamma(\check{y} \Rightarrow \hat{y}) = \mathbf{x} \cap (\gamma(\hat{y}) \to \gamma(\hat{y}))$, which by Lemma 11 is equal to $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$. □

The previous lemmas give us a *normal form*, in terms of denotational formulas, for the elements of $\mathsf{C} \to \mathsf{C}$.

Theorem 5 (*Normal Form for* $\mathsf{C} \to \mathsf{C}$). *The domain* $\mathsf{C} \to \mathsf{C}$ *is isomorphic to the set of denotational formulas of the form* $\wedge(\wedge\check{vs} \Rightarrow \wedge\hat{ws})$ *with* $vs, ws \subseteq V$.

Proof. Any element of $\mathsf{C} \to \mathsf{C}$ is the intersection of arrows of the form $\mathbf{vs} \to \mathbf{ws}$ with $vs = \{v_1, \ldots, v_n\} \subseteq V$ and $ws = \{w_1, \ldots, w_m\} \subseteq V$. If we prove that each arrow can be represented by (*i.e.,* it is the concretisation of) a formula $\wedge\check{vs} \Rightarrow \wedge\hat{ws}$ we have the thesis by Lemma 10. We have

$$\begin{aligned}
\mathbf{vs} \to \mathbf{ws} &= \bigcap_{i=1}^{m} (\mathbf{vs} \to \mathbf{w_i}) \\
&= \bigcap_{i=1}^{m} ((\mathbf{v_1} \cap \cdots \cap \mathbf{v_n}) \to \mathbf{w_i}) \\
\text{(Lemma 11)} &= \bigcap_{i=1}^{m} ((\gamma(\hat{v}_1) \cap \cdots \cap \gamma(\hat{v}_n)) \to \gamma(\hat{w}_i)) \\
\text{(Lemma 10)} &= \bigcap_{i=1}^{m} (\gamma(\hat{v}_1 \wedge \cdots \wedge \hat{v}_n) \to \gamma(\hat{w}_i)) \\
\text{(Lemma 12)} &= \bigcap_{i=1}^{m} (\gamma((\check{v}_1 \wedge \cdots \wedge \check{v}_n) \Rightarrow \hat{w}_i)) \\
\text{(Lemma 10)} &= \gamma\left(\bigwedge_{i=1}^{m} ((\check{v}_1 \wedge \cdots \wedge \check{v}_n) \Rightarrow \hat{w}_i) \right) \\
&= \gamma(\wedge\check{vs} \Rightarrow \wedge\hat{ws}). \quad \square
\end{aligned}$$

Example 26. Examples 23, 25 and 24 show that $\hat{x} \wedge (\check{y} \Rightarrow \hat{y})$ is a representation, in normal form, of $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$. Note that $\hat{x} = \wedge\varnothing \Rightarrow \hat{x}$. □

Fig. 4 shows the Boolean representation of $\mathsf{C}_{\{x,y\}} \to \mathsf{C}_{\{x,y\}}$.

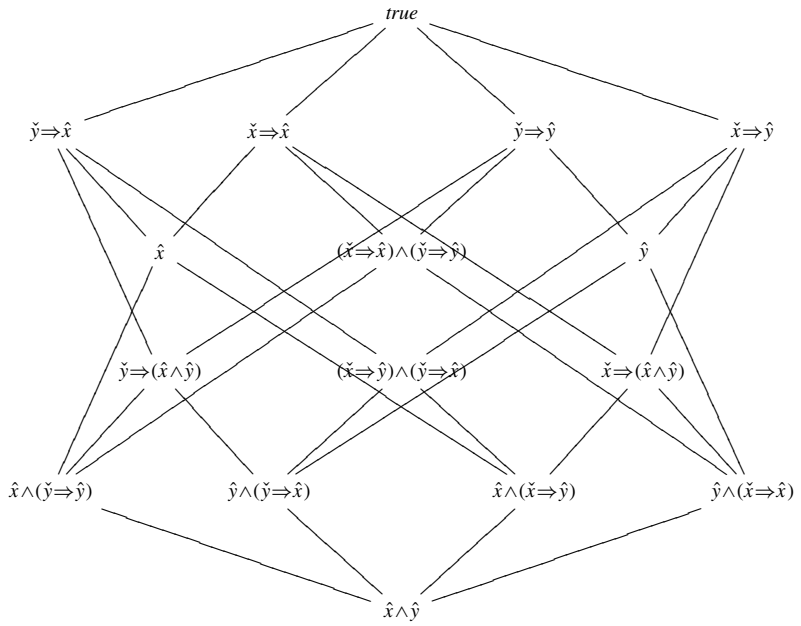

Fig. 4. The representation of $\mathsf{C}_{\{x,y\}} \to \mathsf{C}_{\{x,y\}}$ in terms of denotational formulas.

10. The domain IF coincides with C → C

In this section we prove that IF and C → C are the same abstract domain *i.e.,* that they express exactly the same properties of concrete denotations.

We start with a technical lemma which states that if a denotation does not feature any flow from a set of variables $V \setminus S$ into a given variable y, then y's value in the output of the denotation depends only on the input values of the variables in S.

Lemma 13. *Let $y \in V$, $S \subseteq V$, $\delta \in \Delta$, which does not feature any flow $v \rightsquigarrow y$ with $v \in V \setminus S$, and $\sigma_1, \sigma_2 \in \Sigma$. Then $\sigma_1|_S = \sigma_2|_S$ entails $\delta(\sigma_1)(y) = \delta(\sigma_2)(y)$.*

Proof. Let $V \setminus S = \{v_1, \ldots, v_n\}$. Define $\sigma_1^0 = \sigma_1$, $\sigma_2^0 = \sigma_2$ and $\sigma_1^i = \sigma_1^{i-1}[v_i \mapsto \max(\sigma_1(v_i), \sigma_2(v_i)]$, $\sigma_2^i = \sigma_2^{i-1}[v_i \mapsto \max(\sigma_1(v_i), \sigma_2(v_i)]$ for $1 \leq i \leq n$. Note that either $\sigma_1^i = \sigma_1^{i-1}$ or they differ at $v_i \in V \setminus S$ only. The same holds between σ_2^i and σ_2^{i-1}. Since δ does not feature any flow $v_i \rightsquigarrow y$, in both cases we have $\delta(\sigma_1^i)(y) = \delta(\sigma_1^{i-1})(y)$ and $\delta(\sigma_2^i)(y) = \delta(\sigma_2^{i-1})(y)$. Moreover, $\sigma_1^n = \sigma_2^n$. Hence $\delta(\sigma_1)(y) = \delta(\sigma_1^0)(y) = \delta(\sigma_1^1)(y) = \cdots = \delta(\sigma_1^n)(y) = \delta(\sigma_2^n)(y) = \cdots = \delta(\sigma_2^1)(y) = \delta(\sigma_2^0)(y) = \delta(\sigma_2)(y)$. □

We prove now that IF ⊆ C → C, by *implementing* each element of IF through an element of C → C.

Lemma 14. *Let $f = x_1 \rightsquigarrow y_1, \ldots, x_n \rightsquigarrow y_n \in \mathsf{IF}$. We have*

$$f = \cap\{\mathbf{S}(\mathbf{y}) \to \mathbf{y} \mid y \in V \text{ and } S(y) = \{x_i \mid x_i \rightsquigarrow y \in f\}\}.$$

Proof. Let $\delta \in x_1 \rightsquigarrow y_1, \ldots, x_n \rightsquigarrow y_n$. We prove that $\delta \in \mathbf{S}(\mathbf{y}) \to \mathbf{y}$ for each $y \in V$. Let $\overline{\delta} \in \mathbf{S}(\mathbf{y})$ and $\sigma_1, \sigma_2 \in \Sigma$. We have $\overline{\delta}(\sigma_1)|_{S(y)} = \overline{\delta}(\sigma_2)|_{S(y)}$. Moreover, δ does not feature any flow $v \rightsquigarrow y$ with $v \notin S(y)$. By Lemma 13 we have $(\overline{\delta}; \delta)(\sigma_1)(y) = \delta(\overline{\delta}(\sigma_1))(y) = \delta(\overline{\delta}(\sigma_2))(y) = (\overline{\delta}; \delta)(\sigma_2)(y)$ *i.e.,* $\overline{\delta}; \delta \in \mathbf{y}$.

Conversely, assume that $\delta \in \mathbf{S}(\mathbf{y}) \to \mathbf{y}$ for every $y \in V$. We show that if δ features a flow $v \rightsquigarrow w$ then $v \equiv x_i$ and $w \equiv y_i$ for some $1 \leq i \leq n$.

$w \in \{y_1, \ldots, y_n\}$. Let by contradiction $w \notin \{y_1, \ldots, y_n\}$. There exist $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1|_{V \setminus v} = \sigma_2|_{V \setminus v}$ and $\delta(\sigma_1)(w) \neq \delta(\sigma_2)(w)$. Since $S(w) = \varnothing$, we have $\delta \in \mathbf{O} \to \mathbf{w} = \mathbf{w}$. Then $\delta(\sigma_1)(w) = \delta(\sigma_2)(w)$, a contradiction.

$v \in \{x_i \mid x_i \rightsquigarrow w \in f\}$. Let by contradiction $v \notin \{x_i \mid x_i \rightsquigarrow w \in f\}$ *i.e.,* $v \notin S(w)$. There exist $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1|_{V\setminus v} = \sigma_2|_{V\setminus v}$ and $\delta(\sigma_1)(w) \neq \delta(\sigma_2)(w)$. Let $\overline{\delta}$ be such that $\overline{\delta}(\sigma) = \sigma_1[v \mapsto \sigma(v)]$. We have $\overline{\delta}(\sigma_1) = \sigma_1$, $\overline{\delta}(\sigma_2) = \sigma_2$. Moreover, we have $\overline{\delta} \in \mathbf{S(w)}$ since $v \notin S(w)$. We conclude that $\overline{\delta}; \delta \in \mathbf{w}$. But $(\overline{\delta}; \delta)(\sigma_1)(w) = \delta(\overline{\delta}(\sigma_1))(w) = \delta(\sigma_1)(w) \neq \delta(\sigma_2)(w) = \delta(\overline{\delta}(\sigma_2))(w) = (\overline{\delta}; \delta)(\sigma_2)(w)$, which is a contradiction. □

Example 27. Consider the abstract element $y \rightsquigarrow y$ over $V = \{x, y\}$. We have $S(x) = \varnothing$ and $S(y) = \{y\}$. Then $y \rightsquigarrow y = (\mathbf{O} \to \mathbf{x}) \cap (\mathbf{y} \to \mathbf{y}) = \mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$. □

We prove now that $\mathsf{C} \to \mathsf{C} \subseteq \mathsf{IF}$. We first show that each single arrow in $\mathsf{C} \to \mathsf{C}$ belongs to IF (Lemma 15) and then lift this result to arbitrary elements of $\mathsf{C} \to \mathsf{C}$ (Theorem 6).

Lemma 15. *Let $x_1, \ldots, x_n, y \in V$. We have*

$$\mathbf{x_1} \cdots \mathbf{x_n} \to \mathbf{y} = \{v \rightsquigarrow w \mid v \in V \text{ and } w \in V \setminus y\} \cup \{v \rightsquigarrow y \mid v \in \{x_1, \ldots, x_n\}\}.$$

Proof. Let $\delta \in \mathbf{x_1} \cdots \mathbf{x_n} \to \mathbf{y}$. Assume that δ features a flow $v \rightsquigarrow w$. If $w \not\equiv y$ then $v \rightsquigarrow w \in \{v \rightsquigarrow w \mid v \in V \text{ and } w \in V \setminus y\}$. Assume then $w \equiv y$. We must prove that $v \in \{x_1, \ldots, x_n\}$. Let by contradiction $v \notin \{x_1, \ldots, x_n\}$. There are $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1|_{V\setminus v} = \sigma_2|_{V\setminus v}$ and $\delta(\sigma_1)(y) \neq \delta(\sigma_2)(y)$. Let $\overline{\delta}(\sigma) = \sigma_1[v \mapsto \sigma(v)]$. We have $\overline{\delta}(\sigma_1) = \sigma_1$ and $\overline{\delta}(\sigma_2) = \sigma_2$. Moreover, since $v \notin \{x_1, \ldots, x_n\}$, we have $\overline{\delta} \in \mathbf{x_1} \cdots \mathbf{x_n}$. Then $\overline{\delta}; \delta \in \mathbf{y}$. But $(\overline{\delta}; \delta)(\sigma_1)(y) = \delta(\overline{\delta}(\sigma_1))(y) = \delta(\sigma_1)(y) \neq \delta(\sigma_2)(y) = \delta(\overline{\delta}(\sigma_2))(y) = (\overline{\delta}; \delta)(\sigma_2)(y)$, which is a contradiction.

Conversely, let δ feature flows in $\{v \rightsquigarrow w \mid v \in V \text{ and } w \in V \setminus y\} \cup \{v \rightsquigarrow y \mid v \in \{x_1, \ldots, x_n\}\}$ only. Let $\overline{\delta} \in \mathbf{x_1} \cdots \mathbf{x_n}$. We must prove that $\overline{\delta}; \delta \in \mathbf{y}$. Given $\sigma_1, \sigma_2 \in \Sigma$, we have $\overline{\delta}(\sigma_1)|_{\{x_1,\ldots,x_n\}} = \overline{\delta}(\sigma_2)|_{\{x_1,\ldots,x_n\}}$ since $\overline{\delta} \in \mathbf{x_1} \cdots \mathbf{x_n}$. Moreover, δ does not feature any flow from any $v \in V \setminus \{x_1, \ldots, x_n\}$ to y. By Lemma 13 we have $(\overline{\delta}; \delta)(\sigma_1)(y) = \delta(\overline{\delta}(\sigma_1))(y) = \delta(\overline{\delta}(\sigma_2))(y) = (\overline{\delta}; \delta)(\sigma_2)(y)$. Since σ_1 and σ_2 are arbitrary, we conclude that $\overline{\delta}; \delta \in \mathbf{y}$. □

Example 28. Consider the abstract element $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$ over $V = \{x, y\}$. By Lemma 15 we have $\mathbf{x} = \mathbf{O} \to \mathbf{x} = \{v \rightsquigarrow w \mid v \in V \text{ and } w \in V \setminus x\} \cup \{v \rightsquigarrow x \mid v \in \varnothing\} = \{x \rightsquigarrow y, y \rightsquigarrow y\} \cup \varnothing = \{x \rightsquigarrow y, y \rightsquigarrow y\}$. By the same lemma, $\mathbf{y} \to \mathbf{y} = \{v \rightsquigarrow w \mid v \in V \text{ and } w \in V \setminus y\} \cup \{v \rightsquigarrow y \mid v \in \{y\}\} = \{x \rightsquigarrow x, y \rightsquigarrow x\} \cup \{y \rightsquigarrow y\} = \{x \rightsquigarrow x, y \rightsquigarrow x, y \rightsquigarrow y\}$. The intersection of $\mathbf{x}$ and $\mathbf{y} \to \mathbf{y}$ is then $\{y \rightsquigarrow y\}$. This is the converse of Example 27. □

We can now prove that the domain IF for information flow analysis is the linear refinement of the basic domain for constancy.

Theorem 6 (IF *and* $\mathsf{C} \to \mathsf{C}$ *Coincide*). *We have*

$$\mathsf{IF} = \mathsf{C} \to \mathsf{C}.$$

Proof. By Lemma 14 we conclude that $\mathsf{IF} \subseteq \mathsf{C} \to \mathsf{C}$. Conversely every element of $\mathsf{C} \to \mathsf{C}$ is the intersection of arrows of the form $\mathbf{vs} \to \mathbf{vs}'$. We can assume that vs' is a single variable, since $\mathbf{vs} \to (\mathbf{vs_1} \cap \mathbf{vs_2}) = (\mathbf{vs} \to \mathbf{vs_1}) \cap (\mathbf{vs} \to \mathbf{vs_2})$ (Proposition 4). By Lemma 15 and since IF is closed by intersection (Proposition 2) we conclude that $\mathsf{C} \to \mathsf{C} \subseteq \mathsf{IF}$. □

As a consequence, one can pass from Fig. 1 to Fig. 3 by using Lemma 14 (as in Example 27) and from Fig. 3 to Fig. 1 by using Lemma 15 (as in Example 28).

Since $\mathsf{IF} = \mathsf{C} \to \mathsf{C}$, we can extend to IF the results already proved for $\mathsf{C} \to \mathsf{C}$.

Proposition 8 (*Optimality, Condensing and Boolean Representation of* IF). *The domain* IF *is closed by linear refinement and is hence optimal and condensing. Its elements can be represented through Boolean formulas.*

Proof. By Theorems 6, 3 and 5. □

By Theorems 6 and 5 we conclude that the elements of $\mathsf{IF}_{\{x,y\}}$, shown in Fig. 1, are each represented by the corresponding Boolean formula in Fig. 4.

Example 29. Consider $V = \{x, y\}$ and $y{\rightsquigarrow}y \in \mathsf{IF}_V$. In Example 27 we have seen that $y{\rightsquigarrow}y = \mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$. In Example 26 we have seen that $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$, and hence also $y{\rightsquigarrow}y$, is represented by the Boolean formula $\hat{x} \wedge (\check{y} \Rightarrow \hat{y})$. □

11. The domain Independ coincides with C → C

We prove here that the domain Independ of Definition 8 coincides with the linear refinement C → C and hence with IF (Section 10).

We first show that independence can be expressed in terms of constancy propagation.

Lemma 16. *Let $i, o \in V$. We have*

$$[o\#i] = \mathbf{V} \setminus \mathbf{i} \to \mathbf{o}.$$

Proof. Let $\delta \in \mathbf{V} \setminus \mathbf{i} \to \mathbf{o}$. We want to prove that $\delta \in [o\#i]$. Let hence σ_1, σ_2 be such that $\sigma_1|_{V\setminus i} = \sigma_2|_{V\setminus i}$. We have to prove that $\delta(\sigma_1)(o) = \delta(\sigma_2)(o)$. To that purpose, define $\overline{\delta}$ such that $\overline{\delta}(\sigma) = \sigma_1|_{V\setminus i}[i \mapsto \sigma(i)]$ for every $\sigma \in \Sigma$. We have $\overline{\delta} \in \mathbf{V} \setminus \mathbf{i}$ so that $\overline{\delta}; \delta \in \mathbf{o}$. This entails that $\delta(\sigma_1)(o) = \delta(\overline{\delta}(\sigma_1))(o) = \delta(\overline{\delta}(\sigma_2))(o) = \delta(\sigma_2)(o)$, as desired.

Let conversely $\delta \in [o\#i]$. We want to prove that $\delta \in \mathbf{V} \setminus \mathbf{i} \to \mathbf{o}$. Let hence $\overline{\delta} \in \mathbf{V} \setminus \mathbf{i}$. We have to prove that $\overline{\delta}; \delta \in \mathbf{o}$. To that purpose, let $\sigma_1, \sigma_2 \in \Sigma$. We have $\overline{\delta}(\sigma_1)|_{V\setminus i} = \overline{\delta}(\sigma_2)|_{V\setminus i}$. Hence $(\overline{\delta}; \delta)(\sigma_1)(o) = \delta(\overline{\delta}(\sigma_1))(o) = \delta(\overline{\delta}(\sigma_2))(o) = (\overline{\delta}; \delta)(\sigma_2)(o)$, as desired. □

This is enough to conclude that Independ and C → C are included in one another and hence coincide.

Lemma 17. *Let $[o_1\#i_1]\cdots[o_n\#i_n] \in$ Independ. We have*

$$[o_1\#i_1]\cdots[o_n\#i_n] = \bigcap_{j=1}^{n} \left(\mathbf{V} \setminus \mathbf{i_j} \to \mathbf{o_j}\right) \in \mathsf{C} \to \mathsf{C}$$

and hence Independ $\subseteq$ C → C.

Proof. By Lemma 16 and since C → C is a Moore family. □

Example 30. Consider $V = \{x, y\}$ and $[x\#x][x\#y][y\#x] \in$ Independ. By Lemma 17 we have

$$\begin{aligned}[x\#x][x\#y][y\#x] &= (\mathbf{y} \to \mathbf{x}) \cap (\mathbf{x} \to \mathbf{x}) \cap (\mathbf{y} \to \mathbf{y}) \\ \text{(Lemma 8)} &= (\varnothing \to \mathbf{x}) \cap (\mathbf{y} \to \mathbf{y}) \\ &= \mathbf{x} \cap (\mathbf{y} \to \mathbf{y}).\end{aligned}$$

By Example 27 we also conclude that $[x\#x][x\#y][y\#x] = \mathbf{x} \cap (\mathbf{y} \to \mathbf{y}) = y{\rightsquigarrow}y$. □

Lemma 18. *Let $\mathbf{w_1}\cdots\mathbf{w_n} \to \mathbf{v} \in \mathsf{C} \to \mathsf{C}$. We have*

$$\mathbf{w_1}\cdots\mathbf{w_n} \to \mathbf{v} = [v\#i_1]\cdots[v\#i_n]$$

where $\{i_1, \ldots, i_n\} = V \setminus \{w_1 \cdots w_n\}$. Moreover, we have C → C $\subseteq$ Independ.

Proof. By Lemma 8 we have

$$\begin{aligned}\mathbf{w_1}\cdots\mathbf{w_n} \to \mathbf{v} &= \bigcap_{w \in V\setminus\{w_1\cdots w_n\}} (\mathbf{V} \setminus \mathbf{w} \to \mathbf{v}) \\ \text{(Lemma 17)} &= [v\#i_1]\cdots[v\#i_n].\end{aligned}$$

Since $\mathbf{w_1}\cdots\mathbf{w_n} \to \mathbf{v_1}\cdots\mathbf{v_m} = \cap_{i=1,\ldots,m}(\mathbf{w_1}\cdots\mathbf{w_n} \to \mathbf{v_i})$ and since both C → C and Independ are Moore families, hence closed by intersection, the above proof that any arrow $\mathbf{w_1}\cdots\mathbf{w_n} \to \mathbf{v} \in$ C → C belongs to Independ lets us conclude that C → C $\subseteq$ Independ. □

Example 31. Consider $V = \{x, y\}$ and $\mathbf{x} = \mathbf{O} \to \mathbf{x} \in \mathsf{C} \to \mathsf{C}$. By Lemma 18, by letting $\{i_1, i_2\} = V \setminus \varnothing = \{x, y\}$ we have $\mathbf{O} \to \mathbf{x} = [x\#x][x\#y]$. Consider now $\mathbf{y} \to \mathbf{y} \in \mathsf{C} \to \mathsf{C}$. By Lemma 18, by letting $\{i_1\} = V \setminus \{y\} = \{x\}$ we have $\mathbf{y} \to \mathbf{y} = [y\#x]$. We conclude that $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y}) = [x\#x][x\#y][y\#x]$ (this is the converse of Example 30). By Example 27 we also conclude that $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y}) = [x\#x][x\#y][y\#x] = y{\rightsquigarrow}y$. □

Theorem 7 (Indepen*d* *and* C → C *Coincide*). *We have*

$$\mathsf{Independ} = \mathsf{C} \to \mathsf{C}.$$

Proof. By Lemmas 17 and 18. □

As a consequence, one can pass from Fig. 2 to Fig. 3 by using Lemma 17 (as in Example 30) and from Fig. 3 to Fig. 2 by using Lemma 18 (as in Example 31).

Since Independ = C → C, we can extend to Independ the results already proved for C → C.

Proposition 9 (*Optimality, Condensing and Boolean Representation of* Independ). *The domain* Independ *is closed by linear refinement and is hence optimal and condensing. Its elements can be represented through Boolean formulas.*

Proof. By Theorems 7, 3 and 5. □

By Theorems 7 and 5 we conclude that the elements of $\mathsf{Independ}_{\{x,y\}}$, shown in Fig. 2, are each represented by the corresponding Boolean formula in Fig. 4.

Example 32. Consider $V = \{x, y\}$ and $[x\#x][x\#y][y\#x] \in \mathsf{Independ}_V$. In Example 30 we have seen that $[x\#x][x\#y][y\#x] = \mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$ and in Example 26 that $\mathbf{x} \cap (\mathbf{y} \to \mathbf{y})$, and hence also $[x\#x][x\#y][y\#x]$, is represented by the Boolean formula $\hat{x} \wedge (\check{y} \Rightarrow \hat{y})$. □

12. Discussion

We have used linear refinement [13] to reconstruct two existing domains: IF for information flow analysis (defined in [9] and implemented through Boolean formulas in [10]) and Independ for variable independence analysis (defined in [3]). This has allowed us to prove that they are optimal and condensing. Moreover, this gives them an efficient representation in terms of Boolean formulas. This result has been achieved by defining a new abstract domain C which expresses constancy of variables and by refining C into its linear refinement C → C. The latter has been shown to enjoy optimality, to be condensing and representable in terms of Boolean formulas. Finally, C → C has been shown to coincide with both IF and Independ. As a secondary result, this also proves that IF and Independ coincide.

The equivalence of IF and Independ does not entail that the analysis in [9] is equivalent to that in [3]. While the former is based on abstract compilation of the program into a Boolean formula [14], the latter uses a Hoare-like approach based on weakest preconditions and strongest postconditions. Those techniques are very different in the way they use the information expressed by the abstract domain. However, our result entails that whatever one can do with IF, one can also do it with Independ, and vice versa.

A recent work [15] presents formal properties of a family of flow-sensitive type systems for information flow analysis. The main result is that a *universal* type system exists and that it is the dual of a termination insensitive version of the Independ domain of [3]. However, there is no proof of optimality in the sense that we use the term in this paper: the abstract domain is closed by linear refinement and hence contains all possible dependences between abstract properties of the input (namely, constancy in the input) and abstract properties of the output. Moreover, [15] gives no proof of the condensing property. Our results in this paper allows one to implement an information flow analysis based on the efficient binary decision diagrams. This has been actually achieved in [10]. This is not the case for [15] which, as far as we know, has never been implemented and has only been defined for a simple `while` language. Furthermore, note that we consider termination sensitive information flows, so that a formal comparison with [15] is not easy.

The way we have reconstructed complex domains from simpler ones has many similarities with the reconstruction through linear refinement of abstract domains for groundness analysis of logic programs [19]. In particular, constancy is the imperative counterpart of groundness in logic programming. There, however, two iterations of linear refinement (only one here) are needed to reach an abstract domain which is closed w.r.t. further refinements. There might also be relations with strictness analysis of functional programs, which has also been proved to enjoy some optimality property [20]. There, optimality means that precision cannot be improved as long as constant symbols are abstracted away. It is enlightening to observe that the same abstraction is used in groundness analysis of logic programs, where all functor symbols are abstracted away. In information flow analysis, values are abstracted away, and only their constancy is observed. These similarities might not be casual. We are not saying however that such results from logic

and functional programming can be useful for information flow analysis of imperative programs. The languages and the abstract properties are very different. The similarity is only in the way we have reconstructed complex abstract domains from simpler ones, by using linear refinement.

Our notion of constancy for a variable v coincides with the definition of *agreement* for v among different runs given in [1]. The relation between constancy propagation and independence is hence already recognised in [1], although no proof of optimality or condensing is provided there. Moreover, we also link constancy to information flows and we use the elegant technique of linear refinement to build independency from constancy.

We are confident that our work can be generalised to *declassified* forms of non-interference (or, equivalently, to declassified forms of information flows or of variable independence), such as *abstract non-interference* [11]. One should consider a declassified form of constancy as the basic domain to refine. Declassified constancy means that a variable is *always* bound to a given *abstract* value, as specified by the declassification criterion, rather than to a given *concrete* value. This amounts to modifying Definition 3 by using an abstract notion of equality between $\delta(\sigma_1)(v)$ and $\delta(\sigma_2)(v)$. For instance, this equality might only observe the parity or the sign of concrete values. The generalisation of our work to abstract non-interference is however completely out of the scope of this paper.

References

[1] T. Amtoft, S. Bandhakavi, A. Banerjee, A logic for information flow in object-oriented programs, in: Proc. of 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06, Charleston, South Carolina, USA, January 2006, pp. 91–102.
[2] T. Amtoft, A. Banerjee, Information flow analysis in logical form, in: R. Giacobazzi (Ed.), Proc. of Static Analysis Symposium, SAS'04, in: Lecture Notes in Computer Science, vol. 3148, Springer-Verlag, 2004, pp. 100–115.
[3] T. Amtoft, A. Banerjee, A logic for information flow analysis with an application to forward slicing of simple imperative programs, Science of Computer Programming 64 (1) (2007) 3–28.
[4] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.
[5] D. Clark, C. Hankin, S. Hunt, Information flow for algol-like languages, Computer Languages and Security 28 (1) (2002) 3–28. April.
[6] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'77, 1977, pp. 238–252.
[7] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proc. of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'79, 1979, pp. 269–282.
[8] D. Denning, P. Denning, Certification of programs for secure information flow, Communications of the ACM 20 (7) (1977) 504–513.
[9] S. Genaim, R. Giacobazzi, I. Mastroeni, Modeling secure information flow with Boolean functions, in: P. Ryan (Ed.), ACM SIGPLAN and GI FoMSESS Workshop on Issues in the Theory of Security, April 2004, pp. 55–66.
[10] S. Genaim, F. Spoto, Information flow analysis for Java Bytecode, in: R. Cousot (Ed.), Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'05, in: Lecture Notes in Computer Science, vol. 3385, Paris, France, January 2005, pp. 346–362.
[11] R. Giacobazzi, I. Mastroeni, Abstract non-interference: Parameterizing non-interference by abstract interpretation, in: Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'04, Venice, Italy, January 2004, pp. 186–197.
[12] R. Giacobazzi, F. Ranzato, F. Scozzari, Making abstract domains condensing, ACM Transactions on Computational Logic (ACM-TOCL) 6 (1) (2005) 33–60.
[13] R. Giacobazzi, F. Scozzari, A logical model for relational abstract domains, ACM Transactions on Programming Languages and Systems (ACM TOPLAS) 20 (5) (1998) 1067–1109.
[14] M. Hermenegildo, W. Warren, S.K. Debray, Global flow analysis as a practical compilation tool, Journal of Logic Programming 13 (2–3) (1992) 349–366.
[15] S. Hunt, D. Sands, On flow-sensitive security types, in: Proc. of 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06, Charleston, South Carolina, USA, January 2006, pp. 79–90.
[16] P. Ørbæk, J. Palsberg, Trust in the λ-calculus, Journal of Functional Programming 7 (6) (1997) 557–591.
[17] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE Journal on Selected Areas in Communications 21 (1) (2003) 5–19.
[18] A. Sabelfeld, D. Sands, A PER model of secure information flow in sequential programs, Higher-Order and Symbolic Computation 14 (1) (2001) 59–91.
[19] S. Scozzari, Logical optimality of groundness analysis, Theoretical Computer Science 277 (1–2) (2002) 149–184.
[20] M.C. Sekar, P. Mishra, I.V. Ramakrishnan, On the power and limitation of strictness analysis based on abstract interpretation, in: Proc. of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'91, Orlando, Florida, January 1991, pp. 37–48.
[21] F. Spoto, Information flow is linear refinement of constancy, in: Proc. of the International Colloquium on Theoretical Aspects of Computing, ICTAC'05, October 2005, Hanoi, Vietnam, in: Lecture Notes in Computer Science, vol. 3722, pp. 351–365.
[22] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security 4 (2–3) (1996) 167–187.
[23] G. Winskel, The Formal Semantics of Programming Languages, The MIT Press, 1993.