Improving static analyses of C programs with conditional predicates

Sandrine Blazy, David Bühler, Boris Yakobowski

Improving Static Analyses of C Programs with Conditional Predicates[☆]

Sandrine Blazy[a], David Bühler[b], Boris Yakobowski[b]

[a] *IRISA — University of Rennes*
[b] *CEA, LIST, Software Reliability Lab*

Abstract

Static code analysis is increasingly used to guarantee the absence of undesirable behaviors in industrial programs. Designing sound analyses is a continuing trade-off between precision and complexity. Notably, dataflow analyses often perform overly wide approximations when two control-flow paths meet, by merging states from each path.

This paper presents a generic abstract interpretation based framework to enhance the precision of such analyses on join points. It relies on *predicated* domains, that preserve and reuse information valid only inside some branches of the code. Our predicates are derived from conditional statements, and postpone the loss of information.

The work has been integrated into Frama-C, a C source code analysis platform. Experiments on real generated code show that our approach scales, and improves significantly the precision of the existing analyses of Frama-C.

Keywords: Static Analysis, Abstract Interpretation, Dataflow Analysis, Path Sensitivity

Publication history. This article is a revised and extended version of the paper "Improving Static Analyses of C Programs with Conditional Predicates" published in the FMICS 2014 conference proceedings.

1. Introduction

Formal program verification is an increasingly sought-after approach to guarantee the absence of undesirable behaviors in software. Static code analysis has already shown its industrial applicability to prove safety properties on critical or embedded code. Still, so as to remain tractable, these analyses involve sound

[☆]This work was partially funded by EU FP7 (project STANCE, grant 317753) and Agence Nationale de la Recherche (project VECOLIB, grant ANR-14-CE28-0018-03)

Email addresses: `sandrine.blazy@irisa.fr` (Sandrine Blazy), `david.buhler@cea.fr` (David Bühler), `boris.yakobowski@cea.fr` (Boris Yakobowski)

```
1    if (flag1)
2      { fd1 = open(path1);
3        if (fd1 == -1) exit(); }
4    [...] // code 1
5    if (flag2)
6      { fd2 = open(path2);
7        if (fd2 == -1) {
8          if (flag1) close(fd1);
9          exit();  } }
10   [...] // code 2
11   if (flag1) close(fd1);
12   if (flag2) close(fd2);
```

Figure 1: Example of interleaved conditionals

but incomplete approximations of a program behavior. This may lead to false alarms, when some required properties cannot be proved statically even though they always hold at runtime. Abstract interpretation [1, 2] is a well-known framework to over-approximate program executions through *abstractions* of the most precise mathematical characterization of the program. Designing such abstractions is a continuing trade-off between precision and efficiency.

Flow-sensitivity, which allows to infer static properties that depend on program points, is often considered as a prerequisite to obtain a precise program analysis. More aggressive analyses are *path-sensitive*: the analysis of a program statement depends on the control-flow path followed to reach this statement. Nevertheless, most analyses sacrifice full path-sensitivity and perform approximations when two control-flow paths meet. Those approximations may lead to a significant loss of precision, and may preclude inferring some interesting properties of the program.

Consider as an example the code fragment of Figure 1, which is a simplified version of a real-life program that opens and closes file descriptors. Proving that the three calls to the `close` function are correct, i.e. that the corresponding `fd` variable has been properly created following the calls to the `open` function, heavily relies on the possible values for the `flag1` and `flag2` variables. An analysis that does not keep track of the relation between `flag1` and `fd1` on the one hand, and `flag2` and `fd2` on the other hand, will not be able to prove that the program is correct.

In this paper, we define an analysis in which information about the conditionals that have been encountered so far is retained using boolean predicates. These predicates guard the values inferred about the program. Our analysis is parameterized by a pre-existing analysis domain, which we use to derive a new *predicated* analysis. More precisely, we propagate two kinds of information that are not present in the original domain: a context and an implication map.

1. A *context* is a boolean predicate synthesized from the guards of the conditionals that have been reached so far, and that is guaranteed to hold at the current program point. In our example, at the beginning of line 8, the context would be $\texttt{flag2} \wedge (\texttt{fd2} = -1)$.
2. An *implication map* is a set of facts from the original analysis domain,

guarded by boolean predicates. Each fact is guaranteed to hold when its guard holds. In this example, we suppose that the analysis domain keeps track of whether `open` returned a valid file descriptor, or -1 in case of error. Here are the implications we would like to infer after line 6:

$$\texttt{flag1} \mapsto \mathsf{valid\_fd}\,(\texttt{fd1}) \qquad \texttt{true} \mapsto \mathsf{valid\_fd}\,(\texttt{fd2}) \vee (\texttt{fd2} = -1)$$

The first implication results from the analysis of the conditionals at lines 1–3; it precisely models the information we need between `flag1` and `fd1`. The second implication is simply the postcondition of the `open` function, which holds unconditionally.

Based on abstract interpretation, our framework is generic: it enables mechanically augmenting any standard dataflow analysis with predicates, regardless of its specific properties. The results stated in this paper have been formally verified with Coq, an interactive proof management system.

We also integrated it into Frama-C, a modular platform dedicated to the analysis of C code [3]. Frama-C provides various sound analyses based on abstract interpretation, deductive verification or testing, implemented by a collection of plugins built around a common kernel. These plugins collaborate through logical properties expressed in ACSL, a C specification language [4, 5]. Among them, the Value Analysis plugin [6, 7] performs a forward dataflow analysis to compute an over-approximation of the possible values of variables at each program point. It aims at ensuring the absence of run-time errors in a given program. The domains of Value Analysis are rich, and building path-sensitive analyses on top of them requires significant efforts, especially to achieve scalability. This article shows that predicated analyses over more focused – hence simpler to implement – domains may significantly enhance the precision of the results of the Value Analysis plugin, while remaining scalable.

Contributions. This paper presents a domain-agnostic framework to track sets of disjunctive abstract states, each one being qualified with a predicate for which the state holds. The main novelties of this approach are that:

- the joins on the underlying abstract states are not postponed; instead new predicates preserve the information lost by these joins. We believe this approach, also used by [8], is a worthwhile alternative to full disjunctive domains, that are known to be very costly.
- the analysis does not maintain a strict partition of the abstract states, as the predicates we use are not mutually exclusive (in contrast to abstract domains based e.g. on BDDs);
- this design enables some optimisations that are crucial for scalability, as confirmed by our experimental results on an industrial, generated program;
- the Coq proof increases very significantly the confidence in our formalization.

- this analysis is particularly suited to code generated from synchronous languages, as they frequently contain patterns such as

```
1    int x;
2    if(a) x=1;
3    // ... x unchanged
4    if(a) b=x;
```

Overview. The remainder of this paper is organized as follows. First, Section 2 introduces the language our analysis operates over, simplified for the sake of illustration. Section 3 defines the predicated domains and their operations, and Section 4 explains how to build a predicated analysis over a standard dataflow analysis. The Coq proofs of the soundness of our analysis are outlined in Section 5. Section 6 exposes some related works. Section 7 describes two domains that we used to validate our framework, and Section 8 presents the experimental evaluation of our practical implementation. Finally, Section 9 draws some conclusions.

2. A Generic Abstract Interpretation Based Framework

Our static analysis is based on abstract interpretation [1, 2]. It is mostly independent of the target language, even though our implementation (Section 8) handles C programs. For the sake of brevity, we only present here a toy language. Abstract interpretation links a very precise, but generally undecidable, *concrete* semantics, to an *abstract* one – the abstract semantics being a sound approximation of the concrete one. This section first defines the syntax of our toy language, then its concrete and abstract semantics.

Syntax. Figure 2 presents the syntax of our language. Programs operate over a fixed, finite set of variables $\mathcal{V}$ whose values belong to an unspecified set $\mathbb{V}$. Expressions are either variables, constants, or the application of a binary operator $\star$ to expressions. We stratify expressions e in **exp** and conditions c in $\mathbb{C}$, the truth value of an element of $\mathbb{V}$ being given by a mapping T from $\mathbb{V}$ to booleans. Statements are either assignments such as $x := e$, or tests $c \triangleleft$ that halt execution when the condition does not hold. A program P is represented by its control-flow graph where nodes are integer-numbered program points and edges are labeled by statements. A control-flow graph is represented by a set of triples (source node, statement, destination node). By convention, the program starts at node 0. Encoding standard program constructs such as `if` or `for` in such graphs is immediate and not detailed in this paper. For clarity, we write our examples using a C-like syntax.

Concrete Semantics. A concrete state of the program at a node n of its control-flow graph is described by an environment $\rho \in \mathbb{V}^{\mathcal{V}}$ assigning a value to each variable. The semantics $[\![e]\!]_\rho$ (resp. $[\![c]\!]_\rho$) of an expression e (resp. a condition c) is its evaluation in the environment ρ, and implicitly depends on the evaluation of the operators $\star$.

$$
\begin{array}{llll}
e \in \mathbf{exp} ::= x & x \in \mathcal{V} & \mathtt{i} \in \mathbf{stmt} ::= x := e & \\
\quad | \quad v & v \in \mathbb{V} & \quad | \quad c \triangleleft & \\
\quad | \quad e \star e & & & \\
c, p \in \mathbb{C} ::= e \,|\, \neg c \,|\, c \wedge c \,|\, c \vee c \,|\, \mathtt{true} \,|\, \mathtt{false} & & P \in \mathbf{prog} \triangleq \mathcal{P}(\mathbb{N} \times \mathbf{stmt} \times \mathbb{N}) &
\end{array}
$$

Figure 2: Syntax of our language

Our concrete semantics maps each program node n to the set $\mathbb{S}\,(n)$ of all possible environments at this point; hence our semantics is a function in $\mathcal{P}\left(\mathbb{V}^{\mathcal{V}}\right)^{\mathbb{N}}$. The semantics $[\![\mathtt{i}]\!]$ of a statement $\mathtt{i}$ is a transfer function over a set of states, described by the first equalities of Figure 3a. After an assignment $x := e$, the variable x is bound (in the new states) to the value of the expression e. A test blocks execution, only allowing states in which the condition holds. The concrete semantics of the entire program P is then the smallest solution of the rightmost equations of Figure 3a.

Abstract Semantics. Abstract interpretation based analyses rely on an abstract domain $\mathcal{L}$, whose computable elements model a set of concrete states at a given program point. Such abstract domains must provide:

- a partial order $\sqsubseteq_{\mathcal{L}}$ over abstract states,
- a monotone *concretization* function $\gamma_{\mathcal{L}}$ from $\mathcal{L}$ to $\mathcal{P}\left(\mathbb{V}^{\mathcal{V}}\right)$, linking the abstract states to the concrete ones,
- greatest and smallest elements $\top_{\mathcal{L}}$ and $\bot_{\mathcal{L}}$, such that $\gamma_{\mathcal{L}}(\top_{\mathcal{L}}) = \mathbb{V}^{\mathcal{V}}$ and $\gamma_{\mathcal{L}}(\bot_{\mathcal{L}}) = \emptyset$,
- sound over-approximations join $\sqcup_{\mathcal{L}}$ and meet $\sqcap_{\mathcal{L}}$ of the union and intersection of concrete states,
- sound abstract transfer functions $[\![\mathtt{i}]\!]^{\sharp}_{\mathcal{L}}$ from $\mathcal{L}$ to $\mathcal{L}$ that over-approximate the concrete semantics.

The theorems for the soundness of the abstract semantics are stated in the leftmost column of Figure 3b. The rightmost column defines the abstract semantics, which is the least solution of the system of equations. The soundness properties ensure that any solution is a correct approximation of the concrete semantics. In practice, such systems are solved by iterative dataflow analysis [9, 10].

Proposition 1. *All behaviors of the concrete semantics are captured by the abstract one. That is,* $\forall n \in P, \mathbb{S}\,(n) \subseteq \gamma_{\mathcal{L}}\left(\mathbb{S}^{\sharp}_{\mathcal{L}}\,(n)\right)$

Entailment and equivalence of conditions. In the following, we will need to compare some conditions, in particular to decide whether one condition logically implies another. To do so, we choose a coarse interpretation, that treats the expressions present inside conditions as uninterpreted terms. Let Δ be the set $\mathbf{exp}^{\{\mathtt{true},\mathtt{false}\}}$ of functions from expressions to booleans. Given a valuation

(a) Concrete semantics

$$\begin{array}{l|l} [\![x := e]\!]\,(S) \triangleq \{\rho\,[x \mapsto [\![e]\!]_\rho] \mid \rho \in S\} & \mathbb{S}\,(0) \triangleq \mathbb{V}^{\mathcal{V}} \\ [\![c \triangleleft]\!]\,(S) \triangleq \{\rho \mid \rho \in S \wedge \mathsf{T}\,([\![c]\!]_\rho) = \texttt{true}\} & \mathbb{S}\,(n) \triangleq \bigcup_{(m,\mathtt{i},n)\in P} [\![\mathtt{i}]\!]\,(\mathbb{S}\,(m)) \end{array}$$

(b) Abstract semantics

$$\begin{array}{l|l} \gamma_{\mathcal{L}}(\top_{\mathcal{L}}) = \mathbb{V}^{\mathcal{V}} & \mathbb{S}^\sharp_{\mathcal{L}}\,(0) \triangleq \top_{\mathcal{L}} \\ \gamma_{\mathcal{L}}(l_1) \cup \gamma_{\mathcal{L}}(l_2) \subseteq \gamma_{\mathcal{L}}(l_1 \sqcup_{\mathcal{L}} l_2) & \mathbb{S}^\sharp_{\mathcal{L}}\,(n) \triangleq \bigsqcup_{\mathcal{L}} \left\{ [\![i]\!]^\sharp_{\mathcal{L}}\left(\mathbb{S}^\sharp_{\mathcal{L}}\,(m)\right) \mid (m,\mathtt{i},n) \in P \right\} \\ [\![\mathtt{i}]\!]\,(\gamma_{\mathcal{L}}(l)) \subseteq \gamma_{\mathcal{L}}([\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}\,(l)) & \end{array}$$

Figure 3: Concrete and abstract semantics

$\delta \in \Delta$, we lift it to a valuation on conditions in the obvious way, *e.g.* $\delta(c_1 \wedge c_2) = \delta(c_1) \wedge \delta(c_2)$ where in the r.h.s., the symbol $\wedge$ is the usual conjunction operator. We say that a condition c_1 *entails* another condition c_2, written $c_1 \vdash c_2$ when the evaluation of c_1 implies the evaluation of c_2 for all valuations. Similarly, we define the *equivalence* $\dashv\vdash$ of two conditions as their mutual entailment.

$$\begin{aligned} c_1 \vdash c_2 &\triangleq \forall \delta \in \Delta,\ \delta(c_1) \Rightarrow \delta(c_2) \\ c_1 \dashv\vdash c_2 &\triangleq \forall \delta \in \Delta,\ \delta(c_1) \Leftrightarrow \delta(c_2) \end{aligned}$$

For example, $((x > y) \wedge (z = 0)) \wedge (h = 2) \vdash (h = 2) \wedge (x > y)$ holds.

As a partial preorder, this entailment remains quite weak. Since it does not give a meaning to the operators $\star$ inside expressions, the relation between e.g. $c > 3$ and $c \geq 1$ is not captured, and $c > 3 \vdash c \geq 1$ does *not* hold. This is by design, so that implication and equivalence may be decided efficiently. The real entailment relation may be arbitrarily stronger: any decidable preorder compatible with $\mathsf{T}\,([\![c]\!]_\rho)$ is also suitable. We briefly discuss in Section 9 some possible extensions, in particular to give a meaning to expressions inside conditions.

3. The Predicated Domain

We show in this section how to augment a generic abstract domain with conditional predicates. We first define our predicated domain, equip it with a lattice structure, and then define operations suitable for an efficient analysis.

3.1. Predicated Elements

Our analysis builds a *predicated* domain on top of any abstract domain $\mathcal{L}$; we refer to $\mathcal{L}$ as the *underlying* domain. The information we propagate in this new domain is two-fold:

1. A mapping I from predicates in $\mathbb{C}$ to elements of $\mathcal{L}$, called a *map*. Maps stand for implications from guards to (abstract) values. Hence they contain *conditional* information: if I maps p to l, then l is a correct approximation of the state as soon as p holds.

2. A boolean predicate $c \in \mathbb{C}$, called the *context*, standing for a set of facts that we know to hold at the current program point. Contexts are used to preserve information when performing a join operation. In particular, the join defined in Section 3.3 uses the context to form new interesting guards.[1]

We use the syntax $\lambda p.l$ to denote the map from p to l. We write $\langle p \to l \rangle \in I$ to mean that I guards l by p, and $I(p)$ for l. We say that $\langle p \to l \rangle$ is *trivial* when $l = \top_{\mathcal{L}}$, as the value $\top_{\mathcal{L}}$ brings no information whatsoever. In order to have a decidable semantics, we restrict ourselves to *finite* maps in which all but a finite number of implications are trivial. This restriction is also important because we often perform seemingly infinite intersections $\bigsqcap_{p \in \mathbb{C}} I(p)$. In fact, those intersections always involve a finite number of guards p bound to a value different from $\top_{\mathcal{L}}$.

We also require the guard `false`, which corresponds to a contradiction, to be bound to $\bot_{\mathcal{L}}$. In the following, we only mention non-trivial guards, and omit the guard for `false`.[2]

We call a context and a map that satisfy these properties a *context-implication-map pair,* ranged over by Φ and abbreviated as *CI-pair*. We define $\mathcal{L}^{\texttt{pred}}$, the *predicated domain over* $\mathcal{L}$*,* as the set of such CI-pairs. CI-pairs will represent the abstract state of our predicated analysis.

The concretization of a CI-pair is defined as follows. We say that an implication $\langle p \to l \rangle$ *holds* in a concrete state ρ when, if p holds in the concrete state ρ, then ρ belongs to the concretization of l. Formally, $[\![p]\!]_\rho \Rightarrow \rho \in \gamma_{\mathcal{L}}(l)$. The concretization $\gamma_{\texttt{pred}}(c, I)$ of a CI-pair is the set of states wherein c is true and all implications of I hold.

$$\gamma_{\texttt{pred}}(c, I) \triangleq \{\rho \mid [\![c]\!]_\rho = \texttt{true} \wedge \forall p \in \mathbb{C},\ [\![p]\!]_\rho \Rightarrow \rho \in \gamma_{\mathcal{L}}(I(p))\}$$

Notice that the concretization is consistent with our convention for trivial implications, which hold by definition in any concrete state. Therefore, only the non-trivial implications impact the concretization of a CI-pair.

Rewriting Guards. For the sake of clarity, we use a special notation $\lambda_\sqcap$ to denote the application of a rewriting operator on the *guards* of a map. Given an operator O from guards to guards, applying it naively on an implication map I would lead to "collisions": distinct guards $p_1, \ldots, p_n$ may be rewritten by O into a single guard p. In this case, $O(I)$ should bind p to the meet of all the values previously mapped to $p_1, \ldots, p_n$, i.e. to $I(p_1) \sqcap_{\mathcal{L}} \ldots \sqcap_{\mathcal{L}} I(p_n)$. Our notation $\lambda_\sqcap$ makes implicit this meet. Formally, given $f : \mathbb{C}^n \to \mathbb{C}$ and $l : \mathbb{C}^n \to \mathcal{L}$:

$$\lambda^{\vec{x}}_\sqcap (f(\vec{x})) \,.\, l(\vec{x}) \quad \text{means} \quad \lambda p. \bigsqcap_{\vec{x} \in \mathbb{C}^n} \{l(\vec{x}) \mid p \dashv\vdash f(\vec{x})\}$$

[1] In our analysis, presented in Section 4, contexts are always derived from the guards of the test statements present in the program; see for example the discussion on the example of Figure 1.

[2] By a slight abuse of notation, we also omit the guard for `false` when defining maps through the notation $\lambda p.l$.

$f(\vec{x})$ should be seen as a *pattern*, that involves the variables bound by $\vec{x}$, but may also mention other variables bound elsewhere. For instance, to add by a conjunction a predicate c to the guard of each implication of a map I, we will write the new map as $I' = \lambda^p_{\sqcap}\,(p \wedge c)\,.\,I\,(p)$. Here, the notation stands for $\lambda p.\, \bigsqcap_{\mathcal{L}} \{I\,(q) \mid \forall q \in \mathbb{C},\ q \dashv\vdash p \wedge c\}$. For any predicate p such that $p \not\vdash c$, the new map binds the predicate $p \wedge c$ to the meet of both previous values $I\,(p)$ and $I\,(p \wedge c)$. On the contrary, p is now bound to $\top_{\mathcal{L}}$ (the meet of the empty set) since no predicate q verify $p \dashv\vdash q \wedge c$.

3.2. Predicated Lattice

Assuming that ($\mathcal{L}$,$\sqcup_{\mathcal{L}}$,$\sqcap_{\mathcal{L}}$) is a lattice, we can equip the set of CI-pairs with a derived lattice structure. For convenience, given a CI-pair $\Phi = (c, I)$, we use $\Phi(p)$ for $I(p)$. First and foremost, note that for a given CI-pair Φ and a predicate p, not only does $\Phi\,(p)$ approximates the concrete states whenever p holds, but so do all the $\mathcal{L}$-states bound in Φ to weaker guards p'. We can therefore define an even more precise abstraction of the states implied by p by gathering all the abstract states guarded by such a guard p', and over-approximating their intersection. We call this abstraction *consequence*.

Definition 1. Given $\Phi = (c, I)$, the *consequence* $\Phi \downarrow p$ of p in Φ is defined as:

$$\Phi \downarrow p \triangleq \prod_{p' \in \mathcal{L}} \{I(p') \mid p \wedge c \vdash p'\}$$

It is immediate that $\Phi \downarrow p \sqsubseteq_{\mathcal{L}} \Phi(p)$ indeed holds for all Φ and p. Also, guards that contradict the context have $\bot_{\mathcal{L}}$ as a consequence, since I maps `false` to $\bot_{\mathcal{L}}$.

Example 1. In the following examples, $\mathcal{L}$ is a basic interval domain. Consider a CI-pair Φ with the trivial context `true` and two non-trivial implications, $p \to x \in [2; 6]$ and $q \to x \in [1; 3]$. Then Φ also carries some information for $p \wedge q$, since $\Phi \downarrow (p \wedge q) = \{x \in [2; 3]\}$. Suppose now that the context of Φ is $p \wedge r$. Then $\Phi \downarrow \texttt{true} = \{x \in [2; 6]\}$, since $p \wedge r \vdash p$.

Using the consequence operator, we can define a preorder on CI-pairs, as well as join and meet operations. This will induce a lattice structure on the set of CI-pairs. A CI-pair $\Phi_1 = (c_1, I_1)$ is more precise than $\Phi_2 = (c_2, I_2)$, which we write $\Phi_1 \sqsubseteq^{\downarrow}_{\texttt{pred}} \Phi_2$, when c_1 is stronger than c_2 and all the consequences of Φ_1 are more precise than those of Φ_2. The join $\Phi_1 \sqcup^{\downarrow}_{\texttt{pred}} \Phi_2$ has a context equal to the disjunction of c_1 and c_2, and associates each predicate to the join of its consequences in Φ_1 and Φ_2. Conversely, the meet $\Phi_1 \sqcap^{\downarrow}_{\texttt{pred}} \Phi_2$ has a context equal to the conjunction of c_1 and c_2, and a map obtained by lifting $\sqcap_{\mathcal{L}}$ pointwise. Finally, $\sqsubseteq^{\downarrow}_{\texttt{pred}}$ establishes naturally an equivalence relation $\sim^{\downarrow}_{\texttt{pred}}$ on the set of CI-pairs.

Definition 2. Let $\Phi_1 = (c_1, I_1)$ and $\Phi_2 = (c_2, I_2)$.

$$\Phi_1 \sqsubseteq^{\downarrow}_{\texttt{pred}} \Phi_2 \triangleq c_1 \vdash c_2 \wedge \forall p \in \mathbb{C},\ \Phi_1 \downarrow p \sqsubseteq_{\mathcal{L}} \Phi_2 \downarrow p$$
$$\Phi_1 \sim^{\downarrow}_{\texttt{pred}} \Phi_2 \triangleq c_1 \dashv\vdash c_2 \wedge \forall p \in \mathbb{C},\ \Phi_1 \downarrow p = \Phi_2 \downarrow p$$
$$\Phi_1 \sqcup^{\downarrow}_{\texttt{pred}} \Phi_2 \triangleq c_1 \vee c_2\ ,\ \lambda p.\ (\Phi_1 \downarrow p \sqcup_{\mathcal{L}} \Phi_2 \downarrow p)$$
$$\Phi_1 \sqcap^{\downarrow}_{\texttt{pred}} \Phi_2 \triangleq c_1 \wedge c_2\ ,\ \lambda p.\ (\Phi_1(p) \sqcap_{\mathcal{L}} \Phi_2(p))$$

We write $\mathcal{L}^{\texttt{pred}\sim}$ the set of CI-pairs quotiented by the relation $\sim^{\downarrow}_{\texttt{pred}}$. By construction, two CI-pairs that are equivalent w.r.t this relation contain exactly the same information. In fact, their concretizations are identical, as stated below:

Lemma 1. *Given two CI-pairs* Φ_1 *and* Φ_2*,* $\Phi_1 \sim \Phi_2$ *implies* $\gamma_{\texttt{pred}}(\Phi_1) = \gamma_{\texttt{pred}}(\Phi_2)$.

Equiped with the operators defined above, $\mathcal{L}^{\texttt{pred}\sim}$ is itself a lattice, as stated by the following three results.

Lemma 2. *The relation* $\sqsubseteq^{\downarrow}_{\texttt{pred}}$ *is a partial order on* $\mathcal{L}^{\texttt{pred}\sim}$.

We write sup (resp. inf) the least upper bound (resp. greatest lower bound) of two elements of $\mathcal{L}^{\texttt{pred}\sim}$. Then $\sqcup^{\downarrow}_{\texttt{pred}}$ and sup coincide, $\sqcap^{\downarrow}_{\texttt{pred}}$ and inf coincide, and $\sqsubseteq^{\downarrow}_{\texttt{pred}}$ induces a lattice structure over $\mathcal{L}^{\texttt{pred}\sim}$.

Lemma 3. $(\mathcal{L}^{\texttt{pred}\sim}, \sqsubseteq^{\downarrow}_{\texttt{pred}})$ *is a lattice, in which*

$$\Phi_1 \sqcup^{\downarrow}_{\texttt{pred}} \Phi_2 = \sup(\Phi_1, \Phi_2)$$
$$\Phi_1 \sqcap^{\downarrow}_{\texttt{pred}} \Phi_2 = \inf(\Phi_1, \Phi_2)$$

Finally, the predicated join and meet of CI-pairs are respectively over-approximations of the union and intersection of concrete states (with respect to the concretization function).

Lemma 4. $\sqcup^{\downarrow}_{\texttt{pred}}$ *and* $\sqcap^{\downarrow}_{\texttt{pred}}$ *are sound:*

$$\gamma_{\texttt{pred}}(\Phi_1) \cup \gamma_{\texttt{pred}}(\Phi_2) \subseteq \gamma_{\mathcal{L}}(\Phi_1 \sqcup^{\downarrow}_{\texttt{pred}} \Phi_2)$$
$$\gamma_{\texttt{pred}}(\Phi_1) \cap \gamma_{\texttt{pred}}(\Phi_2) \subseteq \gamma_{\mathcal{L}}(\Phi_1 \sqcap^{\downarrow}_{\texttt{pred}} \Phi_2)$$

We write $\top_{\texttt{pred}}$ and $\bot_{\texttt{pred}}$ for the most general and most restrictive CI-pairs, respectively. Both $\top_{\texttt{pred}}$ and $\bot_{\texttt{pred}}$ contain trivial implications only (except for `false`).

Definition 3. The greatest and least element of $(\mathcal{L}^{\texttt{pred}\sim}, \sqsubseteq^{\downarrow}_{\texttt{pred}})$ are respectively

$$\top_{\texttt{pred}} \triangleq (\texttt{true}, \lambda p. \top_{\mathcal{L}})$$
$$\bot_{\texttt{pred}} \triangleq (\texttt{false}, \lambda p. \top_{\mathcal{L}})$$

The definition of $\bot_{\texttt{pred}}$ might seem strange, as it would be tempting to bind all predicates to $\bot_{\mathcal{L}}$ instead. However, such a map would not be finite. Furthermore, since the context is `false`, the contents of the map are actually irrelevant. Indeed, given any map I and predicate p, $(\texttt{false}, I) \downarrow p = I(\texttt{false}) = \bot_{\mathcal{L}}$.

Computing joins. The definition we have given for $\sqcup^{\downarrow}_{\texttt{pred}}$ does not easily lend itself to an implementation. Indeed, our definition uses an universal quantification on all predicates, and the result of a join may contain an unbounded number of non-trivial implications. (Trivial implications do not contribute to the result of $\downarrow$, as their bound is $\top_{\mathcal{L}}$.) However, if the two inputs are finite CI-pairs, then there is a finite CI-pair in the equivalence class of the join.

Example 2. Consider two CI-pairs Φ_1 and Φ_2 with the same trivial context `true` and these respective non-trivial implications (the notation $[i]$ stands for the singleton interval $[i; i]$):

$$\begin{array}{l|l} p \to x \in [0] & \\ q \to x \in [1] & \quad r \to x \in [42] \end{array}$$

Then their join $\Phi_{\sqcup}$ has context `true`, and contains at least these implications:

$$p \wedge r \to x \in [0; 42] \qquad q \wedge r \to x \in [1; 42] \qquad p \wedge q \wedge r \to x \in [42]$$

All weaker or unrelated predicates are bound to $\top$, as either Φ_1 or Φ_2 (or both) has no information about them. The three implications above immediately arise from the definitions of $\sqcup^{\downarrow}_{\texttt{pred}}$. Finally, and this is the key to having a finite join, there exist maps representing $\Phi_{\sqcup}$ in which all the other (stronger) implications are trivial ones. Consider, for instance, a predicate s stronger than $p \wedge q \wedge r$. By definition of $\sqcup^{\downarrow}_{\texttt{pred}}$, it should be bound in $\Phi_{\sqcup}$ to $(\Phi_1 \downarrow s) \sqcup_{\mathcal{L}} (\Phi_2 \downarrow s)$, which is equal to $\{x \in [42]\}$. However, since $\Phi_{\sqcup}(p \wedge q \wedge r) \sqsubseteq_{\mathcal{L}} \Phi_{\sqcup} \downarrow s$ by definition, binding s to $\{x \in [42]\}$ is redundant, and it can instead be bound to $\top_{\mathcal{L}}$.

However, notice that we had to consider all combinations of conjunctions of predicates from Φ_1 and Φ_2 to compute $\Phi_{\sqcup}$. In the general case, computing a join is exponential in the number of implications present in its inputs. We let $|\Phi|$ be the number of non-trivial and non-`false` implications in the CI-pair Φ. There exist CI-pairs Φ_1 and Φ_2 such that $\Phi_1 \sqcup^{\downarrow}_{\texttt{pred}} \Phi_2$ requires at least $2^{|\Phi_1|+|\Phi_2|}$ implications to be represented.

3.3. A Weaker Join

The high complexity of the algebraic lattice structure of CI-pairs would be a serious hindrance to an efficient practical analysis. We construct instead a relaxed join operation. In essence, we define a *weak-join* $\sqcup_{\texttt{pred}}$ which is an upper bound of its arguments, but not the *least* [11]. Said otherwise, $\sqcup_{\texttt{pred}}$ is an over-approximation of $\sqcup^{\downarrow}_{\texttt{pred}}$.

Definition 4. Let $\Phi_1 = (c_1, I_1)$ and $\Phi_2 = (c_2, I_2)$ be two CI-pairs. The weak-join $\Phi_1 \sqcup_{\texttt{pred}} \Phi_2$ between them is defined as:

$$(c_1, I_1) \sqcup_{\texttt{pred}} (c_2, I_2) = (c_1 \vee c_2,\ \lambda p.\ (l_{\cup}(p) \sqcap_{\mathcal{L}} l_1(p) \sqcap_{\mathcal{L}} l_2(p)))$$

$$\text{where } \begin{cases} l_{\cup} = \lambda^{(p_1,p_2)}_{\sqcap}\, (p_1 \wedge p_2)\,.\, I_1(p_1) \sqcup_{\mathcal{L}} I_2(p_2) \\ l_1 = \lambda^{p_1}_{\sqcap}\, (\neg c_2 \wedge p_1)\,.\, I_1(p_1) \\ l_2 = \lambda^{p_2}_{\sqcap}\, (\neg c_1 \wedge p_2)\,.\, I_2(p_2) \end{cases}$$

```
1    x = 0;
2    y = 0;
3    v = 1;
4    if (c) {
5      x = v;
6    } else {
7      y = v;
8    }
9    w = 0;
10   if (c) {
11     c = 2;
12   }
```

line	Φ_{line}: state after the statement	
	context	implications
3	`true`	$\texttt{true} \mapsto v \in [1], x \in [0], y \in [0]$
5	c	$\texttt{true} \mapsto v \in [1], x \in [1], y \in [0]$
7	$\neg c$	$\texttt{true} \mapsto v \in [1], x \in [0], y \in [1]$
8	$c \vee \neg c$ $\equiv \texttt{true}$	$\texttt{true} \mapsto v \in [1], x \in [0,1], y \in [0,1]$ $c \mapsto v \in [1], x \in [1], y \in [0]$ $\neg c \mapsto v \in [1], x \in [0], y \in [1]$
9	`true`	$\texttt{true} \mapsto v \in [1], w \in [0], x \in [0,1], y \in [0,1]$ $c \mapsto v \in [1], w \in [0], x \in [1], y \in [0]$ $\neg c \mapsto v \in [1], w \in [0], x \in [0], y \in [1]$
10	c	$\texttt{true} \mapsto v \in [1], w \in [0], x \in [1], y \in [0]$
11	`true`	$\texttt{true} \mapsto v \in [1], w \in [0], x \in [1], y \in [0], c \in [2]$
12	`true`	$\texttt{true} \mapsto v \in [1], w \in [0], x \in [0,1], y \in [0,1]$ $c \mapsto v \in [1], w \in [0], x \in [1], y \in [0], c \in [2]$

Figure 4: Example of an analysis using a predicated interval analysis

The context of the weak-join remains the disjunction of the prior contexts. Within the implication map, the operator $l_\cup$ combines implications of the two previous maps: the $\mathcal{L}$-join of values present under guards p_1 and p_2 respectively in Φ_1 and Φ_2 is kept under the new guard $p_1 \wedge p_2$. Conversely, the operators l_1 and l_2 preserve the values only present in Φ_1 or Φ_2 respectively. A value l valid in Φ_1 under a guard p_1 may be present in the weak-join under a guard q, provided that the two following conditions hold. First, q must imply p_1, so that its consequence in Φ_1 is smaller than l. Second, q must contradict c_2, so that its consequence in Φ_2 is $\bot_{\mathcal{L}}$. Thus, the consequences of q in Φ_1 and Φ_2 are both included in the value l, which can be bound to q in the weak-join. We naturally choose $q = \neg c_2 \wedge p_1$. Symetrically, values present in Φ_2 are present under guards that negate c_1. Note that this additional information from Φ_i is useless if all guards $p \wedge \neg c_j$ contradict the new context, i.e. whenever $c_i \vdash c_j$.

Example 3. Let us continue Example 2. Operator $l_\cup$ creates only the first two implications stemming from the "full" join operator, thus avoiding the potential blow-up of processing all the combinations of conjunctions of predicates from Φ_1 and Φ_2. Also, the operators l_1 and l_2 do nothing here, as the negation of the contexts of the CI-pairs is `false`. Notice that the loss of precision regarding $p \wedge q \wedge r$ is irrecoverable: knowing $p \wedge r \to x \in [0; 42]$ and $q \wedge r \to x \in [1; 42]$, we can only deduce that $p \wedge q \wedge r \to x \in [1; 42]$. This is strictly less precise than the value $x \in [42]$ obtained with the "strong" join operator.

This is actually a general property of $\sqcup_{\texttt{pred}}$. The implications "missing" in the weak join always contain a conjunction of several guards from the same map. These are for example the guards of the form $(\bigwedge_i p_i) \wedge (\bigwedge_j p_j)$ with $|i| > 1$ or $|j| > 1$, where the p_i come from one map, and the p_j from the other.

$$
\begin{aligned}
\mathsf{lift}\,(\mathtt{i},(c,I)) &\triangleq \left(c, \lambda p.\, [\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}\,(I\,(p))\right)\\
\mathsf{kill}\,(x,(c,I)) &\triangleq \left(\mathsf{kill}^+_{\mathbb{C}}\,(x,c)\ ,\ \lambda^p_\sqcap\left(\mathsf{kill}^-_{\mathbb{C}}\,(x,p)\right).\,I\,(p)\right)\\
\mathsf{assume}\,(e,(c,I)) &\triangleq \left(c\wedge e\ ,\ \lambda^p_\sqcap\,(p\,[e\leftarrow \mathtt{true}]).\,I\,(p)\right)\\
[\![x := e]\!]^\sharp_{\mathtt{pred}}\,(\Phi) &\triangleq \mathsf{lift}\,(x := e, \mathsf{kill}\,(x,\Phi))\\
[\![c\,\triangleleft]\!]^\sharp_{\mathtt{pred}}\,(\Phi) &\triangleq \mathsf{lift}\,(c\,\triangleleft, \mathsf{assume}\,(c,\Phi))
\end{aligned}
$$

Figure 5: Definition of the abstract semantics $[\![.]\!]^\sharp_{\mathtt{pred}}$

Example 4. Consider now Figure 4, that introduces the result of a predicated analysis with the interval domain on the sample code on its left. We write Φ_i for the state at the end of line i, its context and non-trivial implications being shown in the two rightmost columns. We have $\Phi_8 = \Phi_5 \sqcup_{\mathtt{pred}} \Phi_7$ by definition. The value implied by `true` in Φ_8 comes from the operator $l_\cup$, and is equal to $I_5(\mathtt{true}) \sqcup_{\mathcal{L}} I_7(\mathtt{true})$. Conversely, the value implied by c comes from the operator l_1, which negates the context of Φ_7; furthermore, the value is exactly $\Phi_5(\mathtt{true})$. Note that the intervals inferred in Φ_5 and Φ_7 are entirely retained, guarded by the negations of the converse contexts; no information is actually lost.

The following result states that our weak-join $\sqcup_{\mathtt{pred}}$ is weaker than $\sqcup^{\downarrow}_{\mathtt{pred}}$. Since $\sqcup^{\downarrow}_{\mathtt{pred}}$ is the least upper bound, $\sqcup_{\mathtt{pred}}$ is an upper bound, hence correct.

Lemma 5. *: If Φ_1 and Φ_2 are two CI-pairs, then*

$$\left(\Phi_1 \sqcup^{\downarrow}_{\mathtt{pred}} \Phi_2\right) \sqsubseteq^{\downarrow}_{\mathtt{pred}} \left(\Phi_1 \sqcup_{\mathtt{pred}} \Phi_2\right)$$

Efficient implementation of the weak-join operation. Consider the definition of $\Phi_1 \sqcup_{\mathtt{pred}} \Phi_2$. The operators l_1 and l_2 are linear on the size of the corresponding map. On the other hand, $l_\cup$ requires $|\Phi_1| \times |\Phi_2|$ operations. Thus, the total complexity is *a priori* in $|\Phi_1| \times |\Phi_2|$.

We can however refine this bound. Any implication $\langle p \to l\rangle$ present in both Φ_1 and Φ_2 will exist in the join. Thus, any implication of the form $\langle p\wedge p' \to l \sqcup_{\mathcal{L}} l'\rangle$ is redundant with $\langle p \to l\rangle$ and does not need to be considered. An optimized implementation of the weak-join should thus consider only the subparts of the maps that are distinct. The practical complexity is now in $|\Phi_1^{\mathrm{diff}}| \times |\Phi_2^{\mathrm{diff}}|$, where $|\Phi_i^{\mathrm{diff}}|$ is the map that contains the implications of Φ_i not present in the other map. This is of particular interest when performing a dataflow analysis: the two maps at a join point share all implications collected before the control-flow split and not modified in-between.

4. A Predicated Analysis

This section presents the transfer functions used for a dataflow analysis on predicated domains, explains how to avoid the computation of redundant

guarded values, and details some strategies to decrease the practical complexity of our analysis.

4.1. The Abstract Transfer Functions

We first define an operator called deps from expressions to sets of variables $\mathcal{P}(\mathcal{V})$, that will be useful when computing the memory footprint of an expression or instruction. $\mathsf{deps}\,(e)$ is the set of variables on which the evaluation of e depends. On our toy language, this is the set of variables syntactically present in e. However, in a language with pointers, $\mathsf{deps}\,(e)$ usually depends on the current program point.

Figure 5 defines our abstract semantics for statements in $\mathcal{L}^{\mathtt{pred}}$. The gist of the analysis is to apply the transfer functions of $\mathcal{L}$ to each of its elements in the map, which is carried out by the lift function. However, to remain sound, we also need to modify predicates (either in the context or in a guard) whose truth values are possibly modified by a statement. Following standard dataflow terminology, we define a kill operator, that removes within predicates the expressions that depend on a certain variable x. This operator is used for an assignment such as $x := e$, as this instruction modifies the value of x. It relies on two $\mathsf{kill}_{\mathbb{C}}$ functions on predicates, whose action depend on whether the predicate occurs in a positive or a negative position. $\mathsf{kill}^{+}_{\mathbb{C}}\,(x, p)$ (resp. $\mathsf{kill}^{-}_{\mathbb{C}}\,(x, p)$) replaces by `true` (resp. `false`) the sub-expressions of p that depend on x, alternating with the other operator when they encounter the operator $\neg$.[3]

While kill and lift used in conjunction are sufficient to define a sound abstract semantics for $\mathcal{L}^{\mathtt{pred}}$, they never use the existing implications or enrich the context. The join operation retains some of the specific information of each branch (by creating new implications), but only when the branches have different non-`true` contexts. Thus, we define an operator assume that enriches the context by a new expression $e \in \mathbb{C}$, supposed to be satisfied, and replaces by `true` the occurrences of e in the guards of the map. As a side-effect, the value under a guard implied by e gets merged with the value under the guard `true`, refining it. This assume operator is extended in Figure 6 to predicates in disjunctive normal form: assuming a disjunction amounts to joining the assume of each conjunctive clauses, which are themselves the meet of the assume of the literals. Assuming the negation of an expression consists in replacing it by `false` in the guards.

Within our abstract semantics $[\![.]\!]^{\sharp}_{\mathtt{pred}}$, it is natural to use assume after a test $c\triangleleft$, where the predicate c holds by definition. This is exactly what we did in the examples of Section 3, to keep track of which branch of a conditional we were in.

Example 5. After line 4 in Figure 4, in the branches of the conditional, the operator assume enriches the context according to the condition. After the conditional, the context reverts to `true` due to the join between Φ_5 and Φ_7. At line 10, on a conditional with the same condition c, the assume operator

[3] Those operators are formally defined as function `kill_pred` in the Coq proofs.

$$\begin{aligned}
\mathsf{assume}\,(\neg e,(c,I)) &\triangleq \left(c\wedge\neg e\,,\ \lambda^{p}_{\sqcap}\,(p\,[e\leftarrow \mathtt{false}])\,.\,I\,(p)\right)\\
\mathsf{assume}\,(p_1\wedge p_2,\Phi) &\triangleq \mathsf{assume}\,(p_1,\Phi)\,\sqcap^{\downarrow}_{\mathtt{pred}}\,\mathsf{assume}\,(p_2,\Phi)\\
\mathsf{assume}\,(p_1\vee p_2,\Phi) &\triangleq \mathsf{assume}\,(p_1,\Phi)\,\sqcup^{\downarrow}_{\mathtt{pred}}\,\mathsf{assume}\,(p_2,\Phi)
\end{aligned}$$

Figure 6: Extended assume to predicates in disjunctive normal form.

maps the `true` guard to $I_9(\mathtt{true})\sqcap_{\mathcal{L}} I_9(c)$, as c is now `true`. We have re-learnt the information known about x and y at line 5. Notice that assume removes the guards that are redundant or incompatible with the context, keeping only the facts relevant at the current program point. On line 11, c is overwritten. Hence, the context c is reset to `true` by the kill operator. Finally, upon exiting the conditional, we lose the information $\neg c \mapsto v \in [1], w \in [0], x \in [0], y \in [1]$ coming from the "else" branch, as negating the context `true` results in an implication that never holds. But the information coming from the "then" branch is preserved under the guard $\neg\neg c$, equivalent to c.

As an optimization not presented in Figure 5, it is sometimes useful to *skip* the application of assume. Typically, if a preliminary analysis has detected that no part of condition c will never be tested again, there is no point in tracking it. Conversely, since any application of $\mathsf{assume}(p,\cdot)$ is sound – provided p actually holds – it is sometimes useful to use assume after some well-suited assignments. Consider for example $b := p$ where b is a boolean variable and p a predicate not dependent on b, a common pattern in generated code. We may assume $(b\vee\neg p)\wedge(\neg b\vee p)$ after such a statement. Then, on a test $b\,?$, the analysis will be able to re-learn p. Using the assume function more or less aggressively can be seen as a trade-off between precision and complexity — in particular because contexts are used by our weak-join operation to create new implications.

Soundness of the Analysis. The analysis we have defined above correctly approximates the concrete semantics of the program.

Lemma 6. *Our predicated analysis over $\mathcal{L}^{\mathtt{pred}}$ is sound.*

$$\begin{aligned}
\gamma_{\mathtt{pred}}\,(\Phi_1)\cup\gamma_{\mathtt{pred}}\,(\Phi_2) &\subseteq \gamma_{\mathtt{pred}}\left(\Phi_1\sqcup_{\mathtt{pred}}\Phi_2\right)\\
[\![\mathtt{i}]\!]\left(\gamma_{\mathtt{pred}}\,(\Phi)\right) &\subseteq \gamma_{\mathtt{pred}}\left([\![\mathtt{i}]\!]^{\sharp}_{\mathtt{pred}}\,(\Phi)\right)
\end{aligned}$$

Moreover, we can state a stronger result, that links, at a program point n, the abstract semantics of $\mathbb{S}^{\sharp}_{\mathcal{L}}$ with its equivalent $\mathbb{S}^{\sharp}_{\mathtt{pred}}$ for $\mathcal{L}_{\mathtt{pred}}$.

Lemma 7. *If the underlying transfer functions are monotonic, our predicated analysis is as precise as the non-predicated one.*

$$\begin{array}{ll}
\textit{if} & \forall l,l'\in\mathcal{L},\quad l\sqsubseteq_{\mathcal{L}} l'\Rightarrow [\![\mathtt{i}]\!]^{\sharp}_{\mathcal{L}}\,(l)\sqsubseteq_{\mathcal{L}}[\![\mathtt{i}]\!]^{\sharp}_{\mathcal{L}}\,(l')\\
\textit{then} & \forall n\in P,\quad \textit{given}\ (c_n,I_n)=\mathbb{S}^{\sharp}_{\mathtt{pred}}\,(n)\,,\ I_n(\mathtt{true})\sqsubseteq_{\mathcal{L}}\mathbb{S}^{\sharp}_{\mathcal{L}}\,(n)
\end{array}$$

Of course, the predicated analysis can be more precise. As an example, on line 10 of the program of Figure 4, the non-predicated analysis would have inferred the value $I_9(\texttt{true})$. Our own result – namely $I_{10}(\texttt{true})$ – is much more precise.

4.2. Improving the Analysis: Avoiding Redundant Values

Amongst the value guarded in the implications of a map I, the value under the `true` guard plays a special role. $I\,(\texttt{true})$ always holds by definition, and represents the broadest, less precise knowledge we have on the state. All other values can be used to refine this value, under some hypothesis. Indeed, whenever a predicate p is satisfied, the meet between $I\,(\texttt{true})$ and $I\,(p)$ is a correct abstraction of the state, more precise than $I(\texttt{true})$.

Based on this reasoning, all information carried by $I\,(\texttt{true})$ can be removed from the other values without any loss of precision. Furthermore, the guarded values may be seen as complementing $I\,(\texttt{true})$, and can be handled differently. In particular, the transfer functions of the underlying domain may be expensive – even more so if they are precise. Applying them under each guard is likely to be costly, and may uselessly duplicate some information in each implication of the map.

Reducing the size of the guarded values, as well as the cost of treating them, is essential to decrease the practical complexity of the predicated analysis. For this purpose, we require two additional features from the underlying domain $\mathcal{L}$.

1. A transfer function $[\![\texttt{i}, p]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ over statements `i`, parameterized by the predicate p that guards the processed value. This way, the analysis can be more precise on the `true` guard only and avoid the duplication of new information. Thus, $[\![\texttt{i}, \texttt{true}]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ may be defined as $[\![\texttt{i}]\!]^\sharp_{\mathcal{L}}$, while $[\![\texttt{i}, \cdot]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ applied to a non-`true` guard should be defined as a very imprecise operation, that only guarantees the soundness of the analysis on $\mathcal{L}$. Formally, we only require $[\![\texttt{i}, \cdot]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ to be an over-approximation of $[\![\texttt{i}]\!]^\sharp_{\mathcal{L}}$. The lift operator is then redefined as

$$\mathsf{lift}\,(\texttt{i}, (c, I)) \triangleq \left(c, \lambda p.\, [\![\texttt{i}, p]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}\,(I\,(p))\right)$$

2. A difference operation $\setminus_{\mathcal{L}}$ that discards information already contained in another element of $\mathcal{L}$, that we use to simplify implication maps. Ideally, $a \setminus_{\mathcal{L}} b$ should be as large as possible (w.r.t. $\sqsubseteq_{\mathcal{L}}$), while retaining all the information of a not already present in b. To be sound, we require $a \sqsubseteq_{\mathcal{L}} a \setminus_{\mathcal{L}} b$. We define an operator reduce, that simplifies each implication w.r.t. the value mapped in the `true` guard. It can be used at any time, but it is most useful whenever the shape of the map has changed significantly and redundancies may have been introduced (i.e. after a join or an assumption).

$$\mathsf{reduce}\,(I) \triangleq \lambda p.\, I\,(p) \setminus_{\mathcal{L}} I\,(\texttt{true})$$

```
3     ...
4     if (c) {
5       x = v;
6     } else {
7       y = v;
8     }
9     w = 0;
10    ...
```

line	context	implications after the statement
8	$c \vee \neg c \equiv \texttt{true}$	$\texttt{true} \mapsto v \in [1]\,; x \in [0,1]\,; y \in [0,1]$ $c \mapsto x \in [1]\,; y \in [0]$ $\neg c \mapsto x \in [0]\,; y \in [1]$
9	$\texttt{true}$	$\texttt{true} \mapsto v \in [1]\,; w \in [0]\,; x \in [0,1]\,; y \in [0,1]$ $c \mapsto x \in [1]\,; y \in [0]$ $\neg c \mapsto x \in [0]\,; y \in [1]$

Figure 7: Analysis of Figure 4 with factorization.

These two operators may discard a lot of information; ideally, they would just keep the values that the non-predicated analysis fails to compute. In fact, provided that $\setminus_{\mathcal{L}}$ is such that no information is irrecoverably lost by its application, then reduce actually preserves the information contained in the map: only its actual *contents* are altered.

Lemma 8. *Suppose that* $(a \setminus_{\mathcal{L}} b) \sqcap_{\mathcal{L}} b = a$ *holds for all* $a, b \in \mathcal{L}$*. Then we have* $(c, I) \sim^{\downarrow}_{\texttt{pred}} (c, \mathit{reduce}\,(I))$.

Example 6. Let us come back to the example of Figure 4. The join and the lift function duplicate the value of variables v and w at line 8 and 9 respectively. Here, our previous analysis kept more information within implications than needed. The benefit of the improvements described above are shown in Figure 7. When joining the values coming from lines 5 and 7, the reduce operator removes under the guards c and $\neg c$ the information about v, which is already present under the weaker guard `true`. In parallel, after line 9, the modified lift operator does not apply the full interval analysis to the values guarded by c and $\neg c$. Instead, we use a simpler abstraction, that only removes information about variables that are overwritten. This way, the information about w is no longer duplicated.

Finally, the new transfer functions $[\![\texttt{i}, \cdot]\!]^{\sharp}_{\mathcal{L}\times\mathbb{C}}$ should always have access to the special value $I\,(\texttt{true})$. Thus, the transfer functions may rely on $I\,(\texttt{true})$ for the parts of the processed value that have been removed by the difference operator.

Application to standard domains. A non-relational domain, such as the interval domain, is usually an environment that maps variables (or more complex memory locations) to abstract numeric values. For such a domain, the difference operation can be implemented pointwise, by dropping the bindings already included in the reference state, namely $I\,(\texttt{true})$.

For all predicates p, a sound transfer function $[\![\texttt{i}, p]\!]^{\sharp}_{\mathcal{L}\times\mathbb{C}}$ has to remove the numeric values bound to the variables that are possibly modified by the statement `i`. But they should avoid creating new bindings in a state guarded by a predicate $p \not\vdash \texttt{true}$. A worthwhile trade-off could be to add more information

to a guarded state only when the statement involves variables present in this particular state. Indeed, thanks to the application of reduce, if such a state contains a mapping for a given variable, then its numeric value is more precise than the one bound in the state $I\,(\texttt{true})$. Thus, the evaluation of the statement may really be more precise in the guarded state. Note that this last refinement is not implemented in the domains used for our experimental evaluation.

These two optimizations have another advantage. Indeed, they decrease the size of the guarded values and postpone their alteration until it becomes necessary. Thus, the implications collected before a split of the control-flow graph are more likely not to be modified in the branches. This maximises the shared subparts of the CI-pairs propagated through parallel branches. Since our efficient join only considers the distinct subparts of maps to create new implications, maximizing the shared subparts prevents an unnecessary increase in the size of the predicated maps.

For a relational domain, the principles of those improvements would be the same, but the difference operation may be more difficult to implement. However, it does not have to be complete: it is always sound to retain some redundancy.

4.3. Propagating Unreachable States

Whenever the lifted transfer function of the underlying domain returns the abstract state $\bot_{\mathcal{L}}$ for a value kept under a guard p, there is by definition no concrete state where p holds. We can thus refine the CI-pair by assuming the predicate $\neg p$. The former and latter maps are not equivalent, as their context differ. However, their concretization is exactly the same, as stated by the result below.

Lemma 9. *Given* Φ *and* p *such that either* $\Phi(p) = \bot_{\mathcal{L}}$ *or* $\Phi \downarrow p = \bot_{\mathcal{L}}$*, then* $\gamma_{\texttt{pred}}\,(\Phi) = \gamma_{\texttt{pred}}\,(\mathit{assume}\,(\neg p, \Phi))$

Through this mechanism, information can flow from the underlying domain to the predicated one, by means of the contrapositive of the collected implications.

This also means we could embed the context of our abstract states directly in the implication map. A domain mathematically isomorphic to ours is obtained simply by mapping the negation of the context to bottom. However, we chose a formalisation that keeps separate the context and the implication map. For the sake of clarity firstly, as these two components play very different roles in the analysis. Secondly, this design provides more leeway in the implementation, in particular to select finely the predicates of the context.

4.4. Convergence of the Analysis

Throughout the analysis of a given program, all guards of non trivial implications present in a map are derived from the conditionals of the program, so their number remains finite. In practice, this number can be high; we discuss a possible way of limiting it in Section 8. The predicated analysis essentially amounts to performing the underlying analysis over the values under each guard

(except for the assume operations, which allow us to be more precise). Thus, if the underlying domain provides (or requires) a widening operator to effectively compute the fixpoint, then it can (and should) be lifted as well. Finally, if the underlying transfer functions are monotonic, so are the predicated ones, which ensures the termination of our analysis.

5. A Verified Soundness Proof

The lattice structure of the predicated domain, the relation between the join and the weak-join and the soundness of the analysis have been formalized and proven in Coq [12], an interactive theorem prover. The proofs scripts are mechanically checked by the Coq kernel, ensuring their correctness. This increases very significantly the confidence in our formalization. In particular, the soundness of the weak-join operator, and of its optimized version, was a non-trivial result.

This section gives a brief outline of this development. It can be skipped by readers unfamiliar with Coq. A correspondance between the notations of this article and the Coq ones is available at the beginning of the script.

5.1. Prerequisites

Our Coq development is parameterized by the following elements, that are kept abstract.

Expressions and environments. We require three sets, standing for numeric values `Value`, variables `Var`, and expressions `Exp`. Concrete states are environments in `Env` := `Var` → `Value`. Given an environment, the evaluation function `eval_expr` assesses the value of an expression. The `update` function models the assignment of a single variable in an environment, and the `deps` function verifies that if an expression `exp` does not depend on a variable `var`, then no update of `var` can affect the evaluation of `exp`.

Lattice. We require a lattice $(\mathtt{L}, \sqsubseteq, \sqcup, \sqcap)$ – the underlying abstract domain – plus the correctness of its operations. The Coq development requires a *complete* lattice. Indeed, the join and meet are infinitary, and have type $(\mathtt{L} \to \mathtt{Prop}) \to \mathtt{L}$; the first argument denotes the set of elements of `L` that are being joined or met.[4] This was done to simplifiy the proofs, as using a binary join or meet would have required to reason on the order in which the operations are performed.

Analysis over $\mathcal{L}$. We require a monotonic concretization function `concr` from `L` to `Env`, and monotonic transfer functions `assign` and `assume` with the properties of Section 2.

[4] Readers unfamiliar with Coq can simply see `Prop` as the set of booleans.

5.2. Lattice Structure

We define the predicates as an inductive structure over the expressions, and extend the evaluation on expressions to predicates. The entailment $\vdash$ and equivalence $\dashv\vdash$ between predicates are defined using this evaluation. We then show that the constructors `LAnd`, `LOr` and `LNot` for predicates are *morphisms* of the relations induced by $\vdash$ and $\dashv\vdash$, and register those lemmas in the type classes mechanism of Coq [13]. Then, given $p \dashv\vdash q$, we can prove $(p \wedge r) \dashv\vdash (q \wedge r)$ directly by a rewriting step. This mechanism is used extensively throughout the proofs.

CI-pairs are records of a predicate (the context) and a function from predicates to `L` (the map). To simplify some definitions, we do not require `false` to be always bound to $\bot_{\mathcal{L}}$. Instead, we prove that all abstract operations preserve this property, called `false_bottom` in the development.

The intermediate function `in_map ci P` gathers the values of `L` bound to guards of `ci` that satisfy the proposition (on predicates) `P`. The definition of the consequence `ci` $\downarrow p$ is then:

$$\sqcap\ (\texttt{in\_map ci}\ \ (\texttt{fun p}' \Rightarrow \texttt{p} \wedge (\texttt{context ci}) \vdash \texttt{p}'))$$

The inclusion `CI_incl`, equivalence `CI_equiv`, join `CI_join` and meet `CI_meet` of CI-pairs are then defined as specified in Section 3, together with the proofs of their correctness: `CI_incl` is an order relation, `CI_equiv` an equivalence relation, `CI_join` the least upper bound and `CI_meet` the greatest lower bound of two CI-pairs. These proofs heavily rely on intermediate lemmas about consequences.

5.3. Weak-Join

Proving the correctness of the weak-join – and of its optimization – is the more involved part of the development.

The weak-join of CI-pairs is defined as the meet of three CI-pairs, which correspond to the three operators $l_\cup$, l_1 and l_2 of Definition 4. The first one is created by the function `CI_conj_join`, and the latter two are symmetrically created by the same function `CI_neg_join`. Given Φ_1 and Φ_2, their weak-join is the meet of `CI_conj_join` Φ_1 Φ_2, `CI_neg_join` Φ_1 Φ_2 and `CI_neg_join` Φ_2 Φ_1. We prove that the results of `CI_conj_join` and `CI_neg_join` are greater (less precise) than the original join. Therefore, so is their meet, and the weak-join is indeed greater than the join.

We then validate the final optimization of the weak-join. `CI_shared ci1 ci2` contains only implications that belong to both `ci1` and `ci2`; other predicates are bound to $\top$. Conversely, `CI_diff ci1 ci2` contains only the elements of `ci1` mapped to different values in `ci2`; other predicates are bound to $\top$. The efficient weak-join is the meet between the shared CI-pair and the previous weak-join of the rests. The last lemma asserts the equality between the former and the efficient weak-join.

5.4. Analysis

The concretization `CI_concr` links CI-pairs to concrete environments. We prove that this concretization is monotonic, and consistent with the join and meet operations. The `deps` function is also extended to predicates. We then define a function `kill_pred` that implements the two operators $\mathsf{kill}^+_{\mathbb{C}}$ and $\mathsf{kill}^-_{\mathbb{C}}$, defined respectively as `weaken_pred` and `strengthen_pred`.

Then we define the transfer function `CI_assign` and `CI_assume` such as specified in Section 4.1. Finally, we ensure the soundness of their definition:

- If the environment `env` is a concretization of `ci`, and if the expression `exp` evaluates to `val` in `env`, then `update var val env` is a concretization of `CI_assign var exp ci`;
- If the environment `env` is a concretization of `ci`, and if the expression `exp` evaluates to a positive value in `env`, then `env` is a concretization of `CI_assume exp ci`.

The soundness of `CI_assign` does not depend on the exact definitions of the functions `weaken_pred` and `strengthen_pred` on predicates. We actually prove that more involved operators $\mathsf{kill}_{\mathbb{C}}$ are also sound. For example, we could use two operators that invert assignments of the form $x := x + k$ in the guards, instead of removing all the occurrences of x.

Those generalized operators are called `weaken` and `strengthen` in the formalization, and take as additional argument the expression to which the variable is being assigned. Let us recall that the operator $\mathsf{T}(\cdot)$ injects values of $\mathbb{V}$ into booleans. The properties that must be satisfied by `weaken` and `strengthen` are as follows:

$$\forall c \in \mathbb{C}, \rho \in \mathbb{V}^{\mathcal{V}}, x \in \mathcal{V}, e \in \mathbf{exp}:$$
$$\begin{cases} \mathsf{T}\left([\![c]\!]_\rho\right) \Rightarrow \mathsf{T}\left([\![\texttt{weaken}\,(c,x,e)]\!]_{\rho[x\mapsto[\![e]\!]_\rho]}\right) \\ \mathsf{T}\left([\![\texttt{strenghten}\,(c,x,e)]\!]_{\rho[x\mapsto[\![e]\!]_\rho]}\right) \Rightarrow \mathsf{T}\left([\![c]\!]_\rho\right) \end{cases}$$

Given the assignment $x := x+3$, taking for `weaken` the function that replaces occurrences of x by $x - 3$ is obviously correct here. The drawback of this approach is that a potentially infinite number of new predicates may be created, which might lead to a non-terminating analysis. Performing widening steps on the context and the guards might be required.

6. Related Work

Convex numeric domains, such as intervals, polyhedra, octagons and linear equalities, are widely used in abstract interpretation. Their convexity enables scalable analysis but impedes the representation of disjunctive invariants, causing overly wide imprecisions. Therefore, a large body of work has been devoted to remedy this shortcoming. *Disjunctive completion* [14, 15] of abstract domains avoids the computation of joins by propagating multiple abstract states in parallel along the analysis. One downside is that the code may need to be fully

analyzed for each separate state, whereas our framework strives to minimize the unnecessary computations by getting rid of redundancy. On the other side, disjunctive completion can be used to unroll loop symbolically, something our approach does not handle. However, as widening is notably hard to perform properly on disjunctive sets [16], most of these analyses operate on a beforehand bounded number of disjunct states. Then, some join must eventually be performed, and a distance between abstract states [17, 18] can be used to first rejoin the most related states. This is linked to our difference operator, since nearby states (according to this distance metrics) should have small differences.

Additional information can be attached to the disjunct components. In practice, such disjunctive domains are often stored by binary decision-diagram (BDD) [19] where the nodes contain some predicates and the leaves are numeric abstract states.

Boolean partitioning [20] distinguishes the numerical values of small sets of variables with respect to the truth values of some boolean variables. Such a partitioning may include several boolean trees working on different sets of variables, chosen by heuristics. By comparison, our predicated domain is built on entire states of the underlying domain. Indeed, we do not restrict the variables that may appear in our abstract states. This could be an interesting extension to further improve scalability. Under a guard p, we could choose to keep information only on the variables that are read or written inside an `if` whose condition involves p. The *Binary Decision Tree Abstract Domain*, proposed by Chen and Cousot [21], uses the conditionals of the program as nodes for the tree, as in our analysis. In their work, the shape of the tree is mostly static, making the join operation simpler to implement. The function transfer for assignments preserves all nodes, unlike in our approach where some guards are weakened or removed. On the other hand, the whole tree must be rebuilt, which may be very costly.

Trace partitioning [22, 23] associates each component with some set of execution paths, and involves heuristics on the control-flow to choose a partition of the traces that guides the disjunction. Also, traces should be merged when it is no longer useful to keep them separate; syntactic criteria are used to detect such merge points. *Property simulation* [24] avoids the cost of full path-sensitivity for proving a single fixed property: it groups the abstractions of execution states wherein the given property has the same state. The *boxes* domain [25] implements a specific disjunctive refinement of intervals with decision diagrams extended over linear arithmetic, while our framework is parametrized by the underlying domain under consideration. Closer to our approach, although focusing on termination proof through backward analysis, [26] designs a decision tree abstract domain from linear constraints to generic values. Some effort is also made to maintain a canonical representation of the trees. However, unlike our setup, the ordering and the join are point-to-point operations relying on unification of trees. Also, widening must be used, as the height of the trees height is not bounded a priori.

While binary decision trees make choices on the truth values of boolean predicates, the *Segmented Decision Tree Abstract Domain* [27] can express proper-

ties depending on the range of values of arithmetic variables. Each node is a disjunction over exclusive value intervals for a variable, specified by a symbolic segmentation, and the number of possible choices is not bounded *a priori*. Since the number of segments may grow indefinetely, the widening operator must act on the shape of the tree.

As disjunctive completions, all these domains make a strict partition of their abstract states. Each component of the disjunction is the only abstraction of the concrete states for some cases, and the join between them are postponed as much as possible. Therefore, the analysis can not be relaxed on some disjunct without losing precision for the cases it represents. In contrast, our framework performs the join immediately, but preserves the lost information in abstract values guarded in implications. Then, these separate values provide some additional information to the join, but are not intended to be interpreted alone. This design allows the optimizations proposed in Section 4.2, where the treatment of these ancillary values is lightened. This would be impossible to implement on a disjunctive domain without leading to arbitrary precision loss.

Predicate abstraction [28] would infer a single fact along all execution paths, but this fact may be arbitrarily complex and thus can express disjunctive properties. CEGAR [29] improves predicate abstraction by refining the invariants inferred using counter-examples. Different approaches have been proposed to combine numeric domains with predicates [30, 31, 32]. In particular, Fischer et al. show how to build analyses that propagate a map from a determinate set of predicates to the numerical elements of any existing dataflow analysis [32]. As usual in counter-examples based techniques, the predicates are incrementally found by successive refinement iterations of the analysis, that prune out unverified invariants. Still, finding the proper predicate may be arbitrarily complex, resulting in hard to predict analysis times. Also, the refinement phase requires decidable theories and powerful decision procedures to find the counter-examples from which the predicate is deduced. We instead chose to limit ourselves to uninterpreted predicates relating the conditionals present in the program, for simplicity and predictability. Furthermore, predicate abstraction is mostly goal-driven, and used by model-checkers to prove that a certain property is valid. Our predicated framework uses predicates to postpone the loss of precision inherent to joins in abstract interpretation, but it is not goal-driven. Instead, the same analysis will be done for e.g. all the potential runtime errors of the program (which is why the analysis needs to be run only once). In particular, improving an unsufficiently precise analysis requires designing more fine-grained analysis domains. This contrasts with the automatic refinement available with predicate abstraction.

Otherwise, Mihaila and Simon present in [8] another way to synthesize predicates by observing losses in abstract domain. They propagate a single numeric state augmented with sets of implications between predicates, specifically generated by the numerical abstract domain at join points. For the domain of intervals, the join between the two states where $x \in [0; 5]$ and $x \in [10; 15]$ would typically produce the implication $x > 5 \Rightarrow x > 10$. At conditionals in the control-flow graph, implications are fed to the underlying domain to recover the

numeric loss. The transfer functions follow the same general considerations than ours, but the predicates stem from the numeric domain and are not restricted to the conditionals of the program. Thus they also heed to avoid generating redundant implications, although this makes the full recovery more intricate than in our construction.

Finally, combining abstract domains is a standard way to enhance the abilities of static analysis based on abstract interpretation. These analyses were introduced in the founding papers [14] and widely studied since [33]. In particular, our predicated domain is a *reduced product* between contexts and maps, and the maps can be seen as instances of a *reduced cardinal power* [14], where the base is the chosen underlying abstract domain and the exponent is the set of conditional predicates.

7. Applications

This section describes the two abstract domains on which we have instantiated a predicated analysis in the Frama-C platform. Of course, our framework could also be applied to other domains, e.g. intervals or the "valid file descriptors" domain used for Figure 1.

7.1. A First Abstract Domain: Initialized Variables

Our first domain retains, at each program point, the set of variables that were properly initialized. This domain can be used to prove that no unitialized variables are read at execution time. We used it successfully on generated C programs (Section 8). In this kind of code, variable initialization may happen inside conditionals, and far away from the points where the variable is used.

In the abstract semantics of this domain, we introduce a new default value $\varnothing$ in $\mathbb{V}$, to which all variables are equal at program entry (i.e. $\mathbb{S}\,(0) \triangleq \{\lambda x.\varnothing\}$).

$$\begin{aligned} \gamma_{\mathsf{init}}\,(V) &= \{\rho \mid \forall x \in V, \rho\,(x) \neq \varnothing\} \\ [\![x := e]\!]^\sharp_{\mathsf{init}}\,(V) &= \begin{cases} V \cup \{x\} & \text{if } \mathsf{deps}\,(e) \subseteq V \\ V \setminus \{x\} & \text{otherwise} \end{cases} \\ [\![c \triangleleft]\!]^\sharp_{\mathsf{init}}\,(V) &= V \end{aligned} \qquad \begin{aligned} \mathsf{deps}\,(x := e) &\triangleq \mathsf{deps}\,(e) \\ \mathsf{deps}\,(c \triangleleft) &\triangleq \mathsf{deps}\,(c) \end{aligned}$$

The execution of a statement is correct when all the involved variables are initialized. We extend deps to instructions: $\mathsf{deps}\,(\mathtt{i})$ denotes the set of variables the statement $\mathtt{i}$ depends on. Then, a program P is correct according to this initialized semantics when $\forall\,(n, \mathtt{i}, m) \in P, \mathsf{deps}\,(\mathtt{i}) \subseteq \mathbb{S}^\sharp_{\mathsf{init}}\,(n)$.

7.2. A Second Abstract Domain: Herbrand Equalities

Our experiments also relied on a symbolic domain tracking Herbrand equalities between C expressions. It aims at enhancing the precision of Frama-C's existing Value Analysis plugin, whose abstract domains are non-relational. Our intentions are also somewhat similar to those of Miné [34], in particular abstracting over temporary variables resulting from code normalization. Our equality

$$
\begin{aligned}
[\![c \lhd, p]\!]^\sharp_{\mathsf{eq}\times\mathbb{C}}(E) &\triangleq \begin{cases} E \cup \{e_1 = e_2\} & \text{if } p \equiv \texttt{true} \text{ and } c = (e_1 = e_2) \\ E & \text{otherwise} \end{cases} \\
[\![x := e, p]\!]^\sharp_{\mathsf{eq}\times\mathbb{C}}(E) &\triangleq \begin{cases} \mathsf{kill}_{\mathsf{eq}}(x, E) \cup \{x = e\} & \text{if } p \equiv \texttt{true} \text{ and } x \notin \mathsf{deps}(e) \\ \mathsf{kill}_{\mathsf{eq}}(x, E) & \text{otherwise} \end{cases} \\
\mathsf{kill}_{\mathsf{eq}}(v, E) &\triangleq \{(a = b) \in E \mid v \notin \mathsf{deps}(a) \wedge v \notin \mathsf{deps}(b)\} \\
E \setminus_{\mathsf{eq}} F &\triangleq \{(a = b) \in E \mid (a = b) \notin F\} \\
\gamma_{\mathsf{eq}}(E) &\triangleq \{\rho \mid (a = b) \in E \Rightarrow [\![a]\!]_\rho = [\![b]\!]_\rho\}
\end{aligned}
$$

Figure 8: Abstract semantics for the equality domain

domain boils down to retaining equalities stemming from assignments or equality conditions. Its formal definition is presented in Figure 8, where the set E of equalities increases on tests involving an equality, and on assignments that do not refer to the variable being modified. To be sound, the transfer function on assignments must also remove equalities that involve the overwritten variable, through the $\mathsf{kill}_{\mathsf{eq}}$ operator. Following Section 4.2, we present simplified transfer functions, for which only the `true` guard is enriched, and in which the operator $\setminus_{\mathsf{eq}}$ can be used to remove redundant equalities.

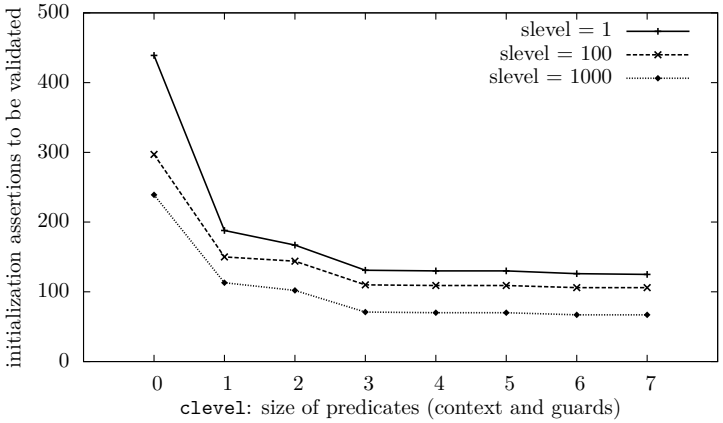
This domain lends itself to a natural extension of our analysis, namely the strengthening of the context and the guards by backward-propagating information from $\mathcal{L}$. For example, we can quotient all the predicates of the CI-pairs by the congruence relation induced by the equalities stored in the map. Furthermore, when applying the operator kill (following an assignment, say to x) on an expression e that involves x, we may instead substitute e by another equal expression. This is more precise than removing the occurrences of x in the guards and the context.

8. Experimental Results

We have integrated our predicated analyses framework as a new plugin of the Frama-C platform. This plugin complements the results of the Value Analysis (abbreviated as VA). We used it with the two domains presented in the previous section; the obtained results are presented in this section.

8.1. Alarms detectable by our Analysis

At each program point where it cannot guarantee the absence of run-time error, VA emits as *alarm, i.e.* an ACSL assertion that excludes the failure case. These alarms may correspond to real bugs, if the statement can give rise to an error at execution time, or may be due to a lack of precision. To limit imprecisions caused by junctions in the control-flow graph, VA implements an instance of trace partitioning [7], and propagates multiple abstract states coming from different branches separately. As dissociating every feasible execution path leads to intractable analyses, the maximum number of parallel states maintained



slevel	assertions to be validated	initialization assertions	remaining assertions/clevel					
			1	2	3	4	5	6
1	632	439	188	167	131	130	130	126
100	488	297	150	144	110	109	109	106
1000	430	239	113	102	71	70	70	67
2000	409	218	100	96	65	64	64	61

time of the Value Analysis

slevel	time
1	1.4s
100	6.7s
1000	175s
2000	400s

time of the predicated analysis

clevel	time
1	0.7s
2	0.73s
3	0.8s
4	1.0s
5	1.9s
6	10s
7	85s

Figure 9: Experimental results

by VA is limited by a parameter called slevel. Once this threshold is reached, all new states are joined and propagated without trace partitioning. Still, high slevel values may lead to high analysis time.

Using a predicated analysis over a simple domain to prove some of the ACSL assertions emitted by VA can avoid this blow up. By construction, our plugin mainly improves VA's results on successive test statements with identical conditions[5]. Although such pattern is relatively unusual in idiomatic C code, it is much more frequent in generated programs, for which our method is well adapted.

8.2. Current Implementation

Our predicated analyses plugin has been designed to be modular. It is parameterized by the underlying abstract domain, and builds a dataflow analysis with predicates for this domain. It was thus easy to instantiate it with the two domains presented in Section 7. The implementation is available at `http://yakobowski.org/predicated.html`.

This plugin runs after the Value Analysis plugin, which it mainly uses to get aliasing information on pointers. This information is needed to ensure the soundness of the deps operator. For convenience, we also chose to reuse some data structures of VA. In particular, the maps from predicates to abstract values are patricia trees with hash-consing. Finally, all the predicates are normalized into a disjunctive normal form. This way, a CI-pair never manages different equivalent predicates.

Generated programs can include a very large number of nested conditional branches and loops, leading to overly wide contexts in our own analysis. To avoid a complexity explosion, we limit the number of literals in the predicates used in contexts and guards (thereby decreasing the precision of our results), according to a parameter clevel. The join removes any implications whose guard exceeds this limit; in the context, we try to keep the most recent predicates.

8.3. Results on Variables Initialization

We tested our plugin on a C program of 5800 lines generated by the industrial environment SCADE, devoted to real-time software. As often with such codes, multiple conditionals are heavily used — typically to test automata states or clocks. In fact, this program has an extremely high ratio of conditionals w.r.t. the total number of statements: 819 out of 2830. Furthermore, many conditionals are complex, with multiple conjunctions and disjunctions. If the operators && and || are desugared into multiple `if`, the resulting program has 9576 statements, and 3428 conditionals. Thus, this program is a very good benchmark for an analysis.

Our results are presented in Figure 9. We first applied VA, which emitted various assertions it could not validate (column 2). As expected, a higher slevel results in fewer alarms. Between 55% and 70% of those are assertions requiring

[5] Modulo conjunction, disjunction and negation, but only over uninterpreted expressions.

variables to be properly initialized (column 3), which are those the domain of Section 7.1 understands. We then ran our predicated analysis, instantiated by this domain, with different limits for the size of predicates. For the values of clevel we used, the number of initialization assertions still unproven after the predicated analysis are shown in the five corresponding columns. Hence, lower figures are better. The analysis time for the VA and for the predicated analysis are also given, according to their parameter; they are independent from each other. Remember that VA must be run (at least with slevel 1) before our analysis can run. So the two timings should be added to compare the total analysis time.

While VA produces significantly less alarms with a higher slevel, its analysis time also increases drastically. This is unsurprising, as fully partitioning for k successive conditionals may require as much as 2^k distinct states. On the other hand, our plugin is effective to quickly validate numerous assertions left unproven by VA, even with strongly limited predicates. The precision of our analysis increases rapidly with the clevel parameter, while the analysis time remains reasonable. More generally, it turns out that small contexts are sufficient to retain most of the relevant information: fewer assertions remain to be validated with clevel = 1 (2.1s) and slevel = 1 than with clevel = 0 and slevel = 2000 (400s). Intuitively, even inside deeply nested conditionals (which generates complex contexts), the more recent guards are often the more useful. In general, our results show that it is much more cost efficient to increase the clevel parameter than the slevel parameter. Those results are extremely encouraging, given the difference in maturity between our plugin and VA. Indeed, the abstract domains of VA have been considerably optimized for speed for many years.

8.4. Validation of the Optimizations

The relevance of the improvements developed in Section 4.2 have been validated through some experiments on the same code as above. We compared the efficiency of the optimized predicated analysis (named *Opt* in the results) with two modified versions of our framework:

- one without any difference operation, where each relevant value is kept unreduced in the join (*Diffless*);
- one in which the original transfer function of the underlying abstract domain is applied to each abstract value in the implications (*OrigTF*).

We used as underlying domains the two domains presented in Section 7. For each configuration and for different size limits for predicates, the Figure 10 shows the analysis time and some measure of the amount of information propagated during the analysis. With the domain of initialized variables, we give the average and the maximum numbers of implications kept during the analysis at each program point. For the equality domain, we give the average and the maximum numbers of equalities (in implications) propagated by the analysis at each program point. A timeout denotes an analysis time that exceeds 10 minutes.

	Initialization			Equalities			
clevel	Opt	Diffless	OrigTF	Opt	Diffless	OrigTF	
1	$0.7s$ 10 22	$0.7s$ 10 22	$0.9s$ 10 22	$2.2s$ 187 304	$11s$ 1751 3895	$28s$ 1786 4320	time average max
2	$0.73s$ 25 44	$0.74s$ 26 45	$2.9s$ 97 302	$3.7s$ 391 680	$58s$ 6983 18653	timeout	time average max
3	$0.8s$ 89 157	$1.0s$ 101 172	$30s$ 777 2732	$8.4s$ 1015 2269	$240s$ 23616 73518	timeout	time average max
4	$1.1s$ 279 552	$1.6s$ 382 601	timeout	$26s$ 3195 9258	timeout	timeout	time average max
5	$1.9s$ 847 1132	$4.8s$ 1255 1764	timeout	$160s$ 9004 46386	timeout	timeout	time average max

Opt : Optimized analysis as described in the article
Diffless : Analysis without the difference operation
OrigTF : Analysis with the original transfer functions applied in each implication

Figure 10: Effectiveness of the optimizations

The introduction of the difference operation has little impact on the performance analysis for the initialization domain. In contrast, it greatly improves the equality analysis, as it removes many equalities from the implications, and all the operations on equality sets depend on their size. Not surprisingly, the benefits of the difference operation depend on the underlying abstract domain, as its aim is to reduce the size of the abstract values.

The application of a lighter transfer function for the implications speeds up substantially the analysis for both domains. Not only this new function is itself faster than the original one, but it also leads to a large diminution of superfluous implications. Indeed, the original transfer function alters systematically all implications, and thus the shared subparts of two CI-pairs are minimized after a disjunction. With the lighter transfer function, most of the implications collected before a split in the control-flow are propagated in the branches without any change, and are kept unchanged in the join. The combination of these implications is then avoided, thanks to the optimization of the join presented at the end of Section 4.

8.5. Experiments on Examples from the Literature

We have successfully applied our predicated analyses, instantiated with the domain of equalities, to various codes of the literature [32, 17, 8, 35] — starting with the motivating example introduced in Figure 1. To analyze it, we simply

```
1     if (i >= 0 && i < 10)
2         x = 1;
3     else
4         x = 0;
5     [...]
6     if (x==1)
7         /*@ assert  0 <= i < 10;  */
8         a[i] = 42;
```

```
p = &x;                                  1
while (n > 0) {                          2
    /*@ assert p != 0; */               3
    x = *p + x;                          4
    n--;                                 5
    if (!(n > 0))                        6
        p = 0;                           7
}                                        8
```

Figure 11: Examples of disjunctions in the literature

modeled the `open` function as a random assignment to 1 or -1 (corresponding to a successful or failed call, respectively), and replaced the calls to `close` by an assertion requiring the file descriptor to be 1. The implications gathered along the analysis link `flag1` and `flag2` to the value of `fd1` and `fd2`, and the `close` assertions are directly proved.

Two other interesting examples are presented in Figure 11. The properties required for the program to be correct are given as ACSL assertions. The left one requires the variable i to be within the bounds of the array a at line 8, which is effectively ensured by the condition $x == 1$. This pattern — storing a predicate within a boolean, which is tested later — is actually quite frequent. Disjunctive domains handle this code naturally, since they propagate two complete separate states after the disjunction, while our analysis is guided by the meaning of the implications. At line 6, we have no implication of the form $\langle x = c \to \_ \rangle$; however, we have $\langle \neg (0 \leq i < 10) \to x = 0 \rangle$, which becomes $\langle \neg (0 \leq i < 10) \to \bot \rangle$ at line 7. The predicate $0 \leq i < 10$ can then be added to the context, as stated in Section 4.3. This is sufficient to validate the assertion.

Finally, disjunctive domains may distinguish loop iterations, by propagating one abstract state for each iteration. Without specific predicates able to label each iteration, our framework cannot offer the same expressiveness. Still, we can sometimes convey relevant information through a loop. The example on the right of Figure 11 shows a loop in which a pointer p is dereferenced and then freed (set to 0) in its last iteration. Our predicated analyses infer $\langle \neg (n > 0) \to p = 0 \rangle$ and $\langle n > 0 \to p = \&x \rangle$, thus ensuring the validity of dereferencing p in the loop, where the context is $n > 0$.

8.6. Scalability on Non-Generated Programs

In order to verify the scalability of our approach, we also tested our plugin on various codes available in the regression tests of the Verasco static analyzer [36] and Frama-C. The first 6 examples are available in the `test` directory of Verasco; the next 2 tests are available in the `tests` directory of Frama-C. `indus` is a proprietary industrial code. Unlike for the program of Section 8.3 (that we henceforth refer to as `scade`), the goal was not to validate some assertions. In fact, in some of the examples, no alarms are emitted by the Value Analysis (VA). In some others, the alarms are out-of-the scope for the kind of assertions handled by the domains on which we instantiated our predicated analyses (Section 7). Instead, we are only interested in scalability. Thus, we only present the analysis

Examples	LOC	Value Analysis	Pred. Initialization		Pred. Equalities	
			0 clevel	∞ clevel	0 clevel	∞ clevel
`almabench`	360	$2.2s$	$0.21s$	$0.21s$	$0.21s$	$0.22s$
`arc4-sb`	160	$0.26s$	$0.19s$	$0.19s$	$0.19s$	$0.19s$
`auth`	160	$0.45s$	$0.24s$	$0.26s$	$0.25s$	$0.26s$
`fft`	190	$0.21s$	$0.19s$	$0.19s$	$0.20s$	$0.21s$
`nbody`	180	$0.57s$	$0.19s$	$0.19s$	$0.20s$	$0.20s$
`smult`	350	$6.2s$	$0.40s$	$0.40s$	$0.88s$	$0.90s$
`adpcm`	610	$0.34s$	$0.22s$	$0.22s$	$0.24s$	$0.25s$
`idct`	610	$1.8s$	$0.22s$	$0.22s$	$0.25s$	$0.29s$
`indus`	19000	$5.9s$	$0.49s$	$0.50s$	$0.64s$	$1.2s$

Figure 12: Cost of predicated analyses relative to the Value Analysis (VA)

times, for both domains, that can be compared with the time taken by VA itself. The results are given in Figure 12.

All examples are hand-written code, in which the structure of the conditionals is far simpler than in `scade`. We compared two configurations of the analysis. First, a dumbed-down version of the analysis with 0 `clevel`, in which the implication map is limited to `true`. Second, an analysis in which all guards are kept. Regarding the impact of `clevel`, we observed very limited differences between the two configurations. The only really meaningful difference occurs with the equality domain, on `indus`. Those results show that the limit on the number of predicates that can be tracked was only necessary for `scade`, in which (1) the number of conditionals was atypical; (2) all functions were inlined by the code generator. (Our plugin performs its analyses in a function-modular way, and the implications of a callee are not propagated back in the caller. Thus, it is sensitive to the size of functions.)

Finally, we compared the analysis time for our plugin with VA. A direct comparison is not easy: unlike our plugin, VA computes an interprocedural fixpoint (using an agressive caching mechanism to avoid re-analyzing functions). Also, the analysis domains are quite different. That being said, in many examples (including the biggest ones), the analysis times for our predicated analysis are negligible relative to the time spent in VA.

9. Conclusion

This work provides a generic framework to enhance the precision of standard dataflow analyses. This framework constructs a derived predicated analysis able to mitigate information loss at junction points of the control-flow graph,

by retaining the conditional values about each branch. Our analysis strives to minimize redundant information processing due to these disjunctions. Experimental tests led through the static analysis platform Frama-C on generated C code showed that a predicated analysis over simple domains can significantly improve the results of prior analyses.

Future works. The literals of our predicates are expressions that we currently consider as uninterpreted. In order to improve our analysis, we intend to give some meaning to the operators in these expressions and to extend the logical implication between guards accordingly. In particular, we could handle successive conditions on distinct but related expressions, such as $(x \geq 0) \lhd$ and $(x \geq 2) \lhd$. The difficulty lies in finding an equational theory for $\vdash$ that would be expressive enough, but not too costly.

Another interesting extension would be to preserve more information when encountering invertible assignments such as $x := x + 1$. Currently, all information about x is lost in the context and the guards afterwards. This requires some care to avoid producing an infinite number of new predicates, which would endanger the convergence of the analysis.

Moreover, prior syntactic analyses or heuristics could help to select relevant predicates for the contexts, which would no longer be extended at each test statement. This would avoid maintaining implication guards that will never be useful again later in the program. Likewise, we could use heuristics to define variable packing strategies, in order to limit the abstract states themselves. The boolean partitioning used in Astrée [20] keeps information only for some syntactically well-chosen variables. We could do the same, by keeping in the state under a guard p only the variables that are related to p in the program.

Finally, it would be worthwhile to apply our predicated analysis over more complex abstract domains.

Acknowledgements

We would like to thank the anonymous referees for their insightful comments on the present article.

References

[1] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: ACM Symposium on Principles of Programming Languages (POPL), ACM, 1977, pp. 238–252.

[2] P. Cousot, R. Cousot, Abstract interpretation frameworks, J. Log. Comput. 2 (4) (1992) 511–547.

[3] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-C - a software analysis perspective, in: SEFM, Vol. 7504 of LNCS, 2012, pp. 233–247.

[4] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, ACSL: ANSI/ISO C Specification Language, Version 1.8 (2014).
URL http://frama-c.com/download/acsl-implementation-Neon-20140301.pdf

[5] L. Correnson, J. Signoles, Combining analyses for C program verification, in: FMICS, 2012, pp. 108–130.

[6] P. Cuoq, V. Prevosto, B. Yakobowski, Frama-C's value analysis plug-in.
URL http://frama-c.com/download/value-analysis-Neon-20140301.pdf

[7] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-c: A software analysis perspective, Formal Asp. Comput. 27 (3) (2015) 573–609. doi:10.1007/s00165-014-0326-7.
URL http://dx.doi.org/10.1007/s00165-014-0326-7

[8] B. Mihaila, A. Simon, Synthesizing predicates from abstract domain losses, in: NASA Formal Methods - 6th International Symposium, 2014, pp. 328–342.

[9] F. Nielson, H. R. Nielson, C. Hankin, Principles of program analysis, Springer, 2005.

[10] F. Bourdoncle, Efficient chaotic iteration strategies with widenings, in: Formal Methods in Programming and their Applications, Springer, 1993, pp. 128–141.

[11] S. Sankaranarayanan, M. Colón, H. B. Sipma, Z. Manna, Efficient strongly relational polyhedral analysis, in: Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI, 2006, pp. 111–125.

[12] The Coq development team, The Coq proof assistant reference manual, LogiCal Project, version 8.4pl6 (2015).
URL http://coq.inria.fr

[13] M. Sozeau, N. Oury, First-class type classes, in: Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings, 2008, pp. 278–293.

[14] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL), 1979, pp. 269–282.

[15] R. Giacobazzi, F. Ranzato, Optimal domains for disjunctive abstract intepretation, Sci. Comput. Program. 32 (1-3) (1998) 177–210.

[16] R. Bagnara, P. M. Hill, E. Zaffanella, Widening operators for powerset domains, in: Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI, 2004, pp. 135–148.

[17] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, A. Gupta, Static analysis in disjunctive numerical domains, in: Static Analysis, 13th International Symposium, SAS, 2006, pp. 3–17.

[18] C. Popeea, W. Chin, Inferring disjunctive postconditions, in: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, 2006, pp. 331–345.

[19] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Computers 35 (8) (1986) 677–691.

[20] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, Static analysis and verification of aerospace software by abstract interpretation, in: Proc. of AIAA Infotech@Aerospace, no. AIAA-2010-3385, 2010, p. 38.

[21] J. Chen, P. Cousot, A binary decision tree abstract domain functor, in: Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings, 2015, pp. 36–53.

[22] M. Handjieva, S. Tzolovski, Refining static analyses by trace-based partitioning using control flow, in: Static Analysis, 5th International Symposium, SAS, 1998, pp. 200–214.

[23] L. Mauborgne, X. Rival, Trace partitioning in abstract interpretation based static analyzers, in: Programming Languages and Systems, 14th European Symposium on Programming, ESOP, 2005, pp. 5–20.

[24] M. Das, S. Lerner, M. Seigle, ESP: path-sensitive program verification in polynomial time, in: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2002, 2002, pp. 57–68.

[25] A. Gurfinkel, S. Chaki, Boxes: A symbolic abstract domain of boxes, in: Static Analysis - 17th International Symposium, SAS, 2010, pp. 287–303.

[26] C. Urban, A. Miné, A decision tree abstract domain for proving conditional termination, in: Static Analysis - 21st International Symposium, SAS, 2014, pp. 302–318.

[27] P. Cousot, R. Cousot, L. Mauborgne, A scalable segmented decision tree abstract domain, in: Time for Verification, Essays in Memory of Amir Pnueli, 2010, pp. 72–95.

[28] S. Graf, H. Saïdi, Verifying invariants using theorem proving, in: CAV, Vol. 1102 of LNCS, 1996, pp. 196–207.

[29] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: Computer Aided Verification, 12th International Conference, CAV, 2000, pp. 154–169.

[30] A. Gurfinkel, S. Chaki, Combining predicate and numeric abstraction for software model checking, STTT 12 (6) (2010) 409–427.

[31] D. Beyer, T. A. Henzinger, G. Théoduloz, Program analysis with dynamic precision adjustment, in: 23rd IEEE/ACM International Conference on Automated Software Engineering ASE, 2008, pp. 29–38.

[32] J. Fischer, R. Jhala, R. Majumdar, Joining dataflow with predicates, in: ESEC/SIGSOFT FSE, 2005, pp. 227–236.

[33] A. Cortesi, G. Costantini, P. Ferrara, A survey on product operators in abstract interpretation, in: Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, 2013, pp. 325–336.

[34] A. Miné, Symbolic methods to enhance the precision of numerical abstract domains, in: Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI, Vol. 3855 of LNCS, 2006, pp. 348–363.

[35] M. Heizmann, J. Hoenicke, A. Podelski, Software model checking for people who love automata, in: Computer Aided Verification - 25th International Conference, CAV 2013, 2013, pp. 36–52.

[36] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie, A formally-verified C static analyzer, in: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL), 2015, pp. 247–259.