Flow insensitive relational static analysis

Solène Mirliaz, David Pichardie

Technical report

Flow insensitive relational static analysis

Domain: Static Analysis

Solène Mirliaz
David Pichardie
Celtique team – IRISA

Abstract: Static analysis of a program allows to predict the properties of its executions without actually executing the program. Abstract interpretation provides the mathematical theory to design such analysis. In particular, it helps design *relational* analyses, which keep track of the relations between variables. These analyses are costly because they usually require computations at every program points (they are flow-sensitive). To reduce these computations, we design a flow insensitive static analysis that can provide a relational invariant on the variables. This invariant is global, there is only one for the program analysed, but it must have the same precision as if we used a flow-sensitive one. A specific representation of the program, namely the *Static Single Information* (SSI) form, allows us to preserve precision thanks to the indexing of the variables. This report presents the concepts of abstract interpretation and the SSI form then details the designed analysis, ensuring its soundness at each step.

Contents

1 Introduction

It is mandatory to properly check the behaviour of a critical software before running it, when an error can have disastrous results, such as putting lives at risk. This is what we call program verification. Its purpose is to prevent any "bad" behaviour of a program. The guarantees provided should be stronger than tests, that can only explore part of all the possible executions. As Dijkstra said, if a test can prove the existence of a bug, it cannot prove its absence [9].

Automatic verification can take several forms, such as static analysis or model checking. The latter needs to have a model of the program and a specification that we want to satisfy. Model checking [3] refers to the tools and techniques to ensure that the model complies with the specification. Static analysis, on the other hand, performs the analysis directly on the code, not on a model. But this analysis is necessarily limited: it has been proved by Rice in 1951 that the properties of a program cannot be inferred in general from its code alone (unless the property is trivial, that is always true or false). Properties can be for example "the program terminates" or "the final value of X is twice the initial value of Y".

To ascertain whether such a property holds on a program requires explaining what the program actually does. This is what we call the *semantics* of the program. It is not possible to design an algorithm that automatically computes the complete semantics of any program. In other words, it is not possible to compute all possible executions of any program. Analyzers thus rely on an approximate *abstract semantics* of the program that is computable. We call the complete, uncomputable semantics the *concrete semantics*. The link between the two semantics must be properly expressed.

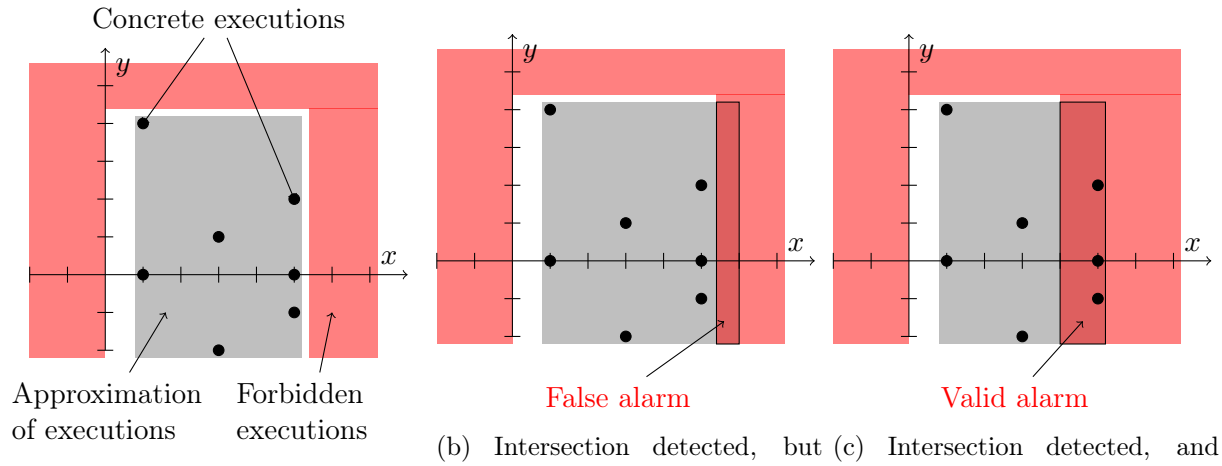
The *abstract semantics* can claim a wide variety of properties. Most common ones include asserting the safety of a program, asserting the absence of undefined behaviour or bugs. Because of the approximation, there can be false alarms (Figure 1): the analysis reports a property violation, while it cannot happen in the concrete semantics. A *precise* analysis will report few false alarms. However, for safety properties, we want a *sound* analysis, where no alarm is silent: if the analysis asserts that all properties are satisfied, then in the concrete semantics it must also be the case. This corresponds to an overapproximation of all possible execution paths.

Abstract interpretation is a framework used to design static analyses. It defines a generic formalization of such analyses, and provides proofs on the guarantees they claim. More precisely, it helps defining the abstract domain (what properties one can express) of the abstract semantics, and how the analysis is performed on the program (the algorithm to compute the properties).

The domains of abstract interpretation will usually approximate the state of the variables, that is for each variable or expression depending on them, approximates its value. The goal is to have *numerical invariants* of these variables. The domains chosen for the variables are often classified into two categories: relational and non-relational domains. A relational domain allows inferring comparisons between the variables of the program, for instance $X - Y < 2$, while a non-relational one only allows comparison to constants ($X < 2 \wedge Y = 3$).

Choosing the best abstraction for a given problem is not trivial. Typically, increasing the precision of an abstraction will lead in a higher memory and time consumption.

Besides the domain, the algorithm computing it calls for reflection too. Textbook abstract interpretation algorithms compute the abstract state at each program points. It can be done by induction on the syntax (denotational semantics) or by a system of equations to solve. Such *dense* computation is heavy in memory or computation time. We would like to perform *sparser* analyses, that would not have one abstract state per program point. The less abstract states are kept, the



(a) No intersection, the program is safe

(b) Intersection detected, but there is no concrete execution in the intersection.
It is a false alarm.

(c) Intersection detected, and there is concrete executions in the intersection.
The program is unsafe

Figure 1: The approximation of the executions must at least cover the concrete executions. In case of intersection between the forbidden executions and the approximation, an alarm is raised

sparser the analysis is. This sparsity should not however deteriorate the precision of the analysis. Less dense analyses that are sill precise were made using a specific representation of the program: the *Single Static Assignment* (SSA). This representation allows to have distinct versions of each variable. Thanks to that, one abstract state can represent the variable at several program point.

With this internship we want to reach the sparsest analysis with only one invariant for the whole program. Our idea is to use the *Single Static Information* (SSI) form that introduces even more distinct versions for each variable. It can be used to make non-relational sparse analysis, but there is no guarantee that it can help design a *relational sparse* analysis with a decent precision and efficiency (adding new versions for each variable has a cost). Designing the sparse relational analysis and ensuring its correctness is the main goal of this internship. Evaluating its precision and efficiency is an ongoing work.

Outline The main principles of abstract interpretation are introduced in Section 2.1. Section 2.2 presents the SSA and SSI forms and their interesting properties to build sparse analyses. Finally, Section 2.3 gives some examples of relational analyses taking advantage of intermediate representations and abstract interpretation. In Section 3 we develop the core of the internship: the building of a sparse, flow insensitive analysis. We first give the concrete semantics of the SSI form in Section 3.1. Then, in Section 3.2, we detail the abstract domain we used. Indeed, our will to build a global invariant imposes that we craft a specific abstract domain to maintain precision. The abstract semantics, which is the core of the analysis, is given in Section 3.3. The proof of correctness of this analysis is detailed in this section. A quick look at an OCaml prototype is provided at the end, in Section 3.4. Section 4 summarizes our approach to design a sparse, flow-insensitive analysis and the pros and cons of it.

Remark The Section 2 is a recap from the bibliographic study. We added the Definition 2.4 of a semilattice and the Theorem 2.2 of transfer fixpoint. In Section 2.1.2 we formalized the semantics of a program using the equational approach rather than the denotational semantics we used to present, as it is the approach we choose for our analysis. This Section also introduces partial functions for environments and handy operations on them, such as projection on a set of variables. Finally, the presentation of the intermediate representations has been developed, in particular the SSI form.

2 State of the art

2.1 Static Inference of Numeric Invariants by Abstract Interpretation

Abstract Interpretation is a domain introduced by Cousot and Cousot in 1977 [6]. This term gathers the concepts necessary to build an approximate static analysis. The correctness of this analysis is expressed with respect to the *concrete semantics*, which is not computable.

In this section, we introduce the notions required by abstract interpretation, such as partial order, lattices but also the notion of fixpoint. Then, we define more precisely the concept of *semantics* of a program. Finally, we present several abstract domains for numeric invariants among the most common ones.

2.1.1 Abstract Interpretation main principles

2.1.1.1 Order theory

Whether it is for the concrete or the abstract semantics, partial orders are useful to describe, compare and compute numerical invariants. We introduce it here as it was defined in [14].

Definition 2.1. [Partial Order, Poset] *A partial order $\sqsubseteq$ on a set X is a relation $\sqsubseteq \in X \times X$, that is reflexive ($\forall x \in X, x \sqsubseteq x$), anti-symmetric ($\forall x, y \in X, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$) and transitive ($\forall x, y, z \in X, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$)*

A partial order set, or poset, $(X, \sqsubseteq)$ is a set X paired with a partial order $\sqsubseteq$.

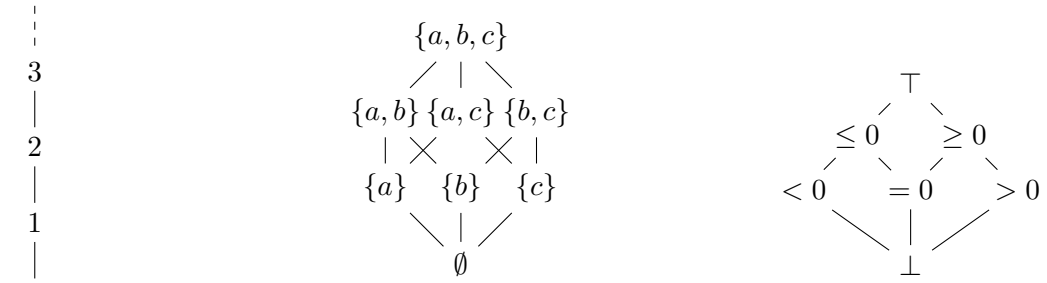
For instance, for any set of elements S, the set of its parts, $\mathcal{P}(S)$ with the inclusion partial order $\subseteq$ is a poset.

In a poset $(X, \sqsubseteq)$, an *upper bound* of two elements $a, b \in X$ is defined as any element $c \in X$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$. A *lower bound* is defined similarly by $c \sqsubseteq a$ and $c \sqsubseteq b$. If c is the smallest *upper bound*, it is called the *least upper bound*, or *join*, and is noted $c = a \sqcup b$. Similarly, we define the *greatest lower bound*, or *meet*, noted $c = a \sqcap b$.

In $(\mathcal{P}(S), \subseteq)$, the join is the union $\cup$ and the meet is the intersection $\cap$.

A convenient way of representing a poset is by using a Hasse diagram. In such a diagram, elements of X are nodes organized such that the greater elements are higher. Order relation are represented by edges between elements. The one obtained by reflexivity and transitivity are omitted. Figure 2 pictures several examples of Hasse diagrams.

In abstract interpretation, elements of a poset will often represent the properties about values of variables. For instance, consider the sign poset of Figure 2c. Let us suppose we want to know the sign of some variable x after a branching. Our variable is equal to zero in the first branch and strictly inferior to zero in the other one. At the join point, the variable will be associated to the element $(= 0) \sqcup (< 0) = (\leq 0)$ of the poset. Now suppose we have a condition `x < 0` to enter a



(a) Hasse diagram of $(\mathbb{N}, \leq)$.

(b) Hasse diagram of $(\mathcal{P}(X), \subseteq)$ where $X = \{a, b, c\}$.

(c) Hasse diagram of the sign poset.

Figure 2: Hasse diagrams of different posets.

branch. Then in the branch we will associate the value of x to the element $(\leq 0) \sqcap (< 0) = (< 0)$ in the poset.

Poset may not be enough in this case, as we want to be able to compute the join or meet of any two elements. A *lattice* is then preferred.

Definition 2.2. [Lattice] *A lattice $(X, \sqsubseteq, \sqcup, \sqcap)$ is a poset such that for every two elements of X, the least upper bound and the greatest lower bound are defined.*

The definition of join and meet can be extended to a set of elements $A \subseteq X$, and are then noted $\bigsqcup A$ and $\bigsqcap A$. Notice that this set A can be infinite.

If it exists, the *least element* $\bigsqcap X$ is denoted $\bot$, called *bottom*. The *greatest element* $\bigsqcup X$ is denoted $\top$, called *top*, if it exists.

Definition 2.3. [Complete lattice] *A complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a poset such that for any subset of X (even infinite ones), the least upper bound and the greatest lower bound are defined.*

Notice that $\bot = \bigsqcup \emptyset$. In a complete lattice both $\top$ and $\bot$ always exist.

Figures 2b and 2c both represent complete lattices, while Figure 2a is not as it does not have an upper element $\top$.

Complete lattices offer convenient properties for soundness but it is not always possible to define a join or a meet for any subset of our poset. Posets which can define them only on non-empty finite subset are called semilattices.

Definition 2.4. [Semilattice] *A join-semilattice is a poset were the join is defined for any non-empty finite subset. Dually, a meet-semilattice is a poset were the meet is defined for any non-empty finite subset.*

2.1.1.2 Fixpoints

As soon as a language includes loops, the analysis needs to build a loop invariant, holding before entering the loop and after each iteration, upon reentering the body. This invariant corresponds to a *fixpoint*.

For a function $f : X \mapsto X$, called an *operator*, a *fixpoint* is any point $x \in X$ such that $f(x) = x$. What the function f is exactly in the case of static analysis will be precised in the next subsection.

In poset, we can identify more elements related to fixpoints:

Definition 2.5. [Fixpoints] *Given a poset* $(X, \sqsubseteq)$ *and an operator* $f : X \to X$

- x *is a* postfixpoint *of* f *if* $f(x) \sqsubseteq x$
- $\text{lfp}(f) = \min\{x \in X | f(x) = x\}$*, if it exists, is the least fixpoint of* f*.*

In the semantics, the least fixpoint will represent the tightest invariant. Tarski's theorem ensures its existence in complete lattices and expresses it as the least postfixpoint.

Theorem 2.1. [Tarski's Theorem] *If* $f : X \to X$ *is a monotonic operator in a complete lattice* $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$*, then the set of fixpoints of* f *is a non-empty complete lattice. In particular,* $\text{lfp} f = \bigsqcap\{x \in X, f(x) \sqsubseteq x\}$ *exists.*

2.1.1.3 Approximations

In abstract interpretation we use the posets to rank information. For static analysis, the common information collected includes the values of variables or the possibles states of the system at some program points. Partial order are helpful to compare, to join and to restrict this information. Higher elements represent less precise information, as they represent the union of information. In the concrete world, the properties of variables are elements of some poset $(C, \leq)$. It is quite common to use $(\mathcal{P}(\mathbb{Z}), \subseteq)$ as a poset for concrete values: an element represents the exact set of values a variable can hold. Another poset $(A, \sqsubseteq)$ is used for the abstract representation. For instance, elements of A can be intervals.

These two posets must be linked at least through the *concretization* function, noted γ.

Definition 2.6. [Concretization] *Let* $(A, \sqsubseteq)$ *and* $(C, \leq)$ *be two posets. A concretization function* $\gamma : (A, \sqsubseteq) \to (C, \leq)$ *is a monotonic (i.e.* $\forall x, y \in A, x \sqsubseteq y \implies \gamma(x) \leq \gamma(y)$*) function associating to each abstract element of* A *a concrete element in* C*.*

For instance, let us consider a concrete poset for sets of integers: $(C, \leq) = (\mathcal{P}(\mathbb{Z}), \subseteq)$. We decide to use the intervals to represent the sets: $A = \{[a, b] | a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\}$. For the order, we choose the inclusion of intervals: $[a, b] \sqsubseteq [c, d]$ if and only if $a \geq c$ and $b \leq d$. $(A, \sqsubseteq, \sqcup, \sqcap)$ is the lattice of intervals. Then the concretization simply corresponds to all integers between the bounds $\gamma([a, b]) = \{x \in \mathbb{Z} | a \leq x \leq b\}$. It is important to understand that $[a, b]$ is merely a notation, not the actual set of integers between the bounds.

The abstract domain must overapproximates the concrete world, so that no possible execution can be ignored. This *overapproximation* property, which we referred as soundness, can be formalized.

Definition 2.7. [Soundness] $a \in A$ *is a sound approximation of* $c \in C$ *if and only if* $c \leq \gamma(a)$*. It is* exact *in case of equality.*

As we will see in the next subsection, fixpoints are essential for the analysis. But the approximation chosen may not satisfy the condition of Tarski's theorem of being a complete lattice. In our case, the abstract domain is only a meet-semilattice. Luckily, the Tarskian fixpoint approximation theorem, also referred as the *fixpoint transfer theorem*, asserts that the postfixpoints in the abstract world are sound approximations of the least fixpoints.

Theorem 2.2. [Fixpoint transfer] *Let* C *be a complete lattice and* A *a poset, with the concretization function* $\gamma : A \to C$ *being monotonic. Let* $F : C \to C$ *be a* monotonic *function in the concrete realm, and* $F^\sharp : A \to A$ *be a function in the abstract one. If* $X \in A$ *is a post-fixpoint of* $F^\sharp$ *, that is* $F^\sharp(X) \sqsubseteq X$*, then having* $F \circ \gamma \subseteq \gamma \circ F^\sharp$ *is enough to guarantee that* $\text{lfp}(F) \subseteq \gamma(X)$*.*

```
stmt ::= V ← expr                                  (assignment, V ∈ 𝕍)
       | assert(cond)                              (assertion)
       | stmt ; stmt                               (sequence)
       | if cond then stmt else stmt endif         (conditional)
       | while cond do stmt done                   (loop)
       | skip                                      (no op)
```

Figure 3: Syntax of the While language

Proof. Let us consider a post-fixpoint $X \in A$. γ is monotonic, that is $A \sqsubseteq B$ implies $\gamma(A) \subseteq \gamma(B)$. So by applying γ on the post-fixpoint relation $F^\sharp(X) \sqsubseteq X$, we get that $\gamma \circ F^\sharp(X) \subseteq \gamma(X)$. Also, let us suppose that $F \circ \gamma \subseteq \gamma \circ F^\sharp$. Then $F \circ \gamma(X) \subseteq \gamma \circ F^\sharp(X) \subseteq \gamma(X)$.

So $\gamma(X)$ is a post-fixpoint of F.

By Tarski's theorem, the least fixpoint of F is equal to the meet of all post-fixpoints, which is equivalent to saying that $\mathrm{lfp}(F)$ is lower (in the lattice) than any post-fixpoint. Thus $\mathrm{lfp}(F) \subseteq \gamma(X)$. □

Finally, we do not have guarantees that this (post-)fixpoint is computable in a finite time, and even though, we would like to reach it fast. To accelerate convergence toward a fixpoint, Cousot and Cousot introduced the widening operator ∇ [6]. In this original definition, this binary operator must ensure that a sequence $s_0 = X_0$, $s_1 = s_0 \nabla X_1, \dots s_n = s_{n-1} \nabla X_n \dots$, where X_i are arbitrary values, is not strictly increasing. The operator must also be an overapproximation of its arguments. In practice and in more recent works, the definition of the widening operator has been simplified and the first condition is that any sequence $X_{n+1} = X_0 \nabla X_n$ will converge in finite time [14].

2.1.2 Concrete and abstract semantics of programs

The semantics of a program is the mathematical meaning associated to it. It can be the final value returned by the program, the values of its variables or even whether it terminates or not. Here we will illustrate two approaches to present semantics using a simple imperative language called the While language. Its syntax is presented in Figure 3. The atomic statements are the variable assignment, the assertion and the skip instructions. Statements are composed with the sequence, the conditional or the while loop. The syntax of expressions and conditionals is not detailed here. Arithmetic expressions include constants, variables and binary operations. Conditionals include boolean constants, arithmetic comparisons and logical conjunction and disjunction. This is a quite common language and we will not detail it more here.

Let us first start with the concrete semantics. It is the most precise representation of the program behavior. If we could compute it, it would correspond exactly to the actual executions of the program. Alas, this is not possible due to Rice's theorem. Nonetheless, it is possible to express it mathematically. Here, we are interested into numerical invariants, we want to know the possible values of each variable or expression of the program. What we call the *concrete semantic* of the program will thus be a map from each program point $\ell \in \mathcal{L}$ to a set of *memory states*. In Miné's tutorial [14], the memory states are total functions from the variables $\mathbb{V}$ to the domain of value $\mathbb{I}$ $(= \mathbb{R}, \mathbb{Z}, \mathbb{N}, \dots)$. But for our work, to be able to express that a variable is not defined, we manipulate partial functions instead.

Definition 2.8. [Partial function] *A partial function from A to B if a function from A' to B where A' is a subset of A. It is noted $f : A \nrightarrow B$. Its* domain of definition *A' is noted* $\mathrm{dom}(f) \subseteq A$.

From this point we will use the term *domain* for the domain of definition.

A partial function can be represented as its set of associations. For instance let f be a partial function such that $\mathrm{dom}(f) = \{a_1, a_2, \ldots, a_k\} \subseteq A$, and for all $i \in [1, k]$, $f(a_i) = b_i \in B$, then f can be noted $f = \{a_1 \mapsto b_1, \ldots a_k \mapsto b_k\}$. Given a function f, the function $g = f[a \mapsto b]$ is the updated version of f. It evaluates to b for a and for any $x \neq a$ in the domain of f, it evaluates to $f(x)$. We have that $\mathrm{dom}(g) = \mathrm{dom}(f) \cup \{a\}$, $g(a) = b$ and for any $c \in \mathrm{dom}(f), c \neq a \implies g(c) = f(c)$. In some cases, we will need to have a restricted version of a partial function where some variables have been forgotten.

Definition 2.9. [Restriction] *Let $f \in A \nrightarrow B$ be a partial function and $A' \subseteq \mathrm{dom}(f)$, the restricted partial function $f_{|_{A'}}$ is defined such that $\mathrm{dom}(f_{|_{A'}}) = A'$ and for all $a \in A'$, $f_{|_{A'}}(a) = f(a)$.*

Environment Let $\mathbb{V}$ be the set of variables in the program, $\mathbb{I}$ the set of values these variables can be assigned to. An *environment* is a partial function $s : \mathbb{V} \nrightarrow \mathbb{I}$. The domain of s is the set of defined variables in s. $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \nrightarrow \mathbb{I}$ is the set of environments.

The framework for abstract interpretation presented by Miné [14] and the one of Cousot [7] both use total functions to represent environment. In this mindset, a variable that has not been assigned a value yet can have any value.

Projection Concrete and abstract domains for numerical invariants can be provided with an existential projection operator, noted proj and $\mathrm{proj}^\sharp$. The existential projection of some value S onto a set of variables V consists in forgetting all the variables of $\mathbb{V}$ that do not belong to V. In the concrete domains, this means that any concrete environment s of the projection of S is equal to a concrete environment of S restricted to V:

$$\mathrm{proj}_V(S) = \left\{ s \middle| s' \in S, s_{|_V} = {s'}_{|_V} \right\}$$

The abstract projection should overapproximate the concrete one:

$$\mathrm{proj}_V \circ \gamma(S) \subseteq \gamma \circ \mathrm{proj}^\sharp_V(S)$$

It is *exact* in case of equality. We require that for any abstract value S and set of variables V:

$$S \sqsubseteq \mathrm{proj}^\sharp_V(S)$$

Invariant The concrete semantics computes an *invariant* $I : \mathcal{L} \to \mathcal{P}(\mathcal{E})$ of the program, it associates to each program point the set of environments possible at this point during the execution of the program. Notice that the domain $\mathcal{P}(\mathcal{E})$ can be organized as a complete lattice: $(\mathcal{P}(\mathcal{E}), \subseteq, \cup, \cap, \emptyset, \mathcal{E})$. There are two approaches to define this invariant: the denotational-style semantics and the equational-based semantics. While in the denotational-style we compute invariants inductively on the syntax of statements, an equational-based semantics will rely on a system of equations to determine the environments possible at each program points. We will only detail the latter. It is computed for the control-flow graph of the program.

2.1.2.1 Control-flow graph

We define the set of program points $\mathcal{L}$ where we will compute the invariant. The control-flow graph (CFG) of the program is a graph where each node is a program point $\ell \in \mathcal{L}$, and each edge corresponds to an *atomic instruction*.

Atomic instruction An atomic instruction is either an assignment $[x \leftarrow e]$ or an arithmetic condition $[e_1 \bowtie e_2]$ ($\bowtie \in \{=, <, \leq\}$). The set of variables used in an expression e is noted $\text{vars}(e)$. Sometimes, we do not specify any instruction on an edge, and instead put the identity function id. It is semantically equivalent to put an arithmetic condition always true such as $[0 = 0]$.

CFG The CFG can be build inductively on the syntax of a program.

$$\text{cfg}[\ ^{\ell_1}\ \textbf{skip}\ ^{\ell_2}] \stackrel{\text{def}}{=} \{(\ell_1, \text{id}, \ell_2)\} \tag{1}$$

$$\text{cfg}[\ ^{\ell_1} x \leftarrow e\ ^{\ell_2}] \stackrel{\text{def}}{=} \{(\ell_1, [x \leftarrow e], \ell_2)\} \tag{2}$$

$$\text{cfg}[\ ^{\ell_1}\ \textbf{assert}\ (c)\ ^{\ell_2}] \stackrel{\text{def}}{=} \{(\ell_1, \text{id}, \ell_2)\} \tag{3}$$

$$\text{cfg}[\ ^{\ell_1} s_1;\ ^{\ell_2} s_2\ ^{\ell_3}] \stackrel{\text{def}}{=} \text{cfg}[\ ^{\ell_1} s_1\ ^{\ell_2}] \cup \text{cfg}[\ ^{\ell_2} s_2\ ^{\ell_3}] \tag{4}$$

$$\begin{aligned}\text{cfg}[\ ^{\ell_1}\ \textbf{if}\ c\ \textbf{then}\ ^{\ell_2} s_1\ ^{\ell_3}\ \textbf{else}\ ^{\ell_4} s_2\ ^{\ell_5}\ \textbf{endif}\ ^{\ell_6}] &\stackrel{\text{def}}{=} \{(\ell_1, [c], \ell_2), (\ell_1, [\neg c], \ell_4), (\ell_3, \text{id}, \ell_6), (\ell_5, \text{id}, \ell_6)\} \\ &\quad \cup \text{cfg}[\ ^{\ell_2} s_1\ ^{\ell_3}] \cup \text{cfg}[\ ^{\ell_4} s_2\ ^{\ell_5}]\end{aligned} \tag{5}$$

$$\begin{aligned}\text{cfg}[\ ^{\ell_1}\ \textbf{while}\ ^{\ell_2} c\ \textbf{do}\ ^{\ell_3} body\ ^{\ell_4}\ \textbf{done}\ ^{\ell_5}] &\stackrel{\text{def}}{=} \{(\ell_1, \text{id}, \ell_2), (\ell_2, [c], \ell_3), (\ell_3, \text{id}, \ell_2), (\ell_2, [\neg c], \ell_5)\} \\ &\quad \cup \text{cfg}[\ ^{\ell_3} body\ ^{\ell_4}]\end{aligned} \tag{6}$$

Figure 4 pictures the CFG of a program in the While language.

2.1.2.2 Equational semantics

The invariant of each program point is built by applying the effect of the incoming edges onto the environments of predecessor program points [1]. The effect of the incoming is actually the semantics of the atomic instruction attached to the edge in the CFG. Let us define the semantics of expressions first, then the one of atomic expressions.

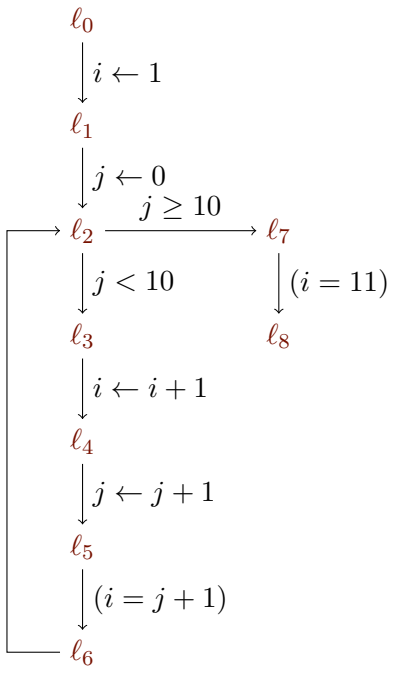
Semantics of expressions The semantics of an expression e is a function $\mathbb{E}[\![e]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{I})$ that outputs all possible evaluations of the expression given an environment. Such semantics allows undeterminism, for instance the expression $[0, 20]$ has the semantics $\{v \in \mathbb{I} | 0 \leq v \leq 20\}$ and $[-\infty, +\infty]$ has the semantics $\mathbb{I}$. The semantics is defined inductively on the syntax of the expression, with rules such as $\mathbb{E}[\![n]\!]s = \{n\}$ if n is a constant, $\mathbb{E}[\![x]\!]s = \{s(x)\}$ if $x \in \text{dom}(s)$, else $\mathbb{E}[\![x]\!]s = \emptyset$.

Let $\text{vars}(e)$ be the variables appearing in the expression e. Then if s is not defined over $\text{vars}(e)$, then the set outputted by the semantics is empty.

[1] Whether we take the predecessors or successors actually depends on the analysis performed. In this report we consider a forward analysis, where the predecessors are taken.

```
i := 1;
j := 0;
while (j < 10) do
  i := i + 1;
  j := j + 1;
  assert (i = j + 1)
done;
assert (i = 11)
```

(a) An example of program in the while language, with assertions



(b) The CFG of the program

Figure 4: The CFG of some program in the While language

Semantics of atomic instructions The semantics of assignment, $\mathbb{S}[\![\cdot]\!]$, and arithmetic condition, $\mathbb{C}[\![\cdot]\!]$ are both functions from a set of environments to another one. In the case of assignment, the semantics outputs the environments of the input updated with the new value for x. As for condition, it filters the environments in the input that satisfy the test.

$$\begin{aligned}
\mathbb{S}[\![x \leftarrow e]\!] : \quad & \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E}) \\
& S \to \{s[x \mapsto v] | s \in S, v \in \mathbb{E}[\![e]\!]s\} \\
\mathbb{C}[\![e_1 \bowtie e_2]\!] : \quad & \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E}) \\
& S \to \{s | s \in S, v_1 \in \mathbb{E}[\![e_1]\!]s, v_2 \in \mathbb{E}[\![e_2]\!]s, v_1 \bowtie v_2\}
\end{aligned}$$

Given some atomic instruction a, we note $[\![a]\!]$ its corresponding semantics, that is either $\mathbb{S}[\![a]\!]$ if a is an assignment or $\mathbb{C}[\![a]\!]$ if it is an arithmetic condition.

For example, let us start with the environment $\rho = \{x \mapsto 3\} \in \mathcal{E}$, that is the environment associating x with 3. Then the semantics of the expression $x + 2$ is defined inductively on the syntax:

$$\mathbb{E}[\![x + 2]\!]\rho = \{v_1 + v_2 | v_1 \in \mathbb{E}[\![x]\!]\rho, v_2 \in \mathbb{E}[\![2]\!]\rho\} = \{v_1 + 2 | v_1 \in \rho(x)\} = \{3 + 2\} = \{5\}$$

For a condition, such as $x + 2 > 0$, with R as the input set of environments.

$$\mathbb{C}[\![x + 2 > 0]\!]R = \{\rho | \rho \in R, v_1 \in \mathbb{E}[\![x + 2]\!]\rho, v_2 \in \mathbb{E}[\![0]\!]\rho, v_1 > v_2\} = \{\rho | \rho \in R, v \in \rho(x), v + 2 > 0\}$$

These semantics are monotonic functions.

Remark Notice that for all $s \in \mathbb{C}[\![e_1 \bowtie e_2]\!]$ then s is defined over the variables of e_1 and the ones of e_2.

Equational system For each program point, the set of environments possible is the union of the environments of the predecessors, where the semantics of the edge has been applied. The entry point is a special case: having no predecessors, its set of environments is composed of all partial functions where the arguments of the program are associated to any value. Let S_0 be the set of such states, $a_1, \ldots, a_n$ be the argument of the program.

$$S_0 \stackrel{\text{def}}{=} \bigcup_{i_1,\ldots,i_n \in \mathbb{I}^n} \{a_1 \mapsto i_1, \ldots, a_n \mapsto i_n\}$$

The equational system defines for each program point ℓ its set of memory state $\mathcal{X}_\ell$. The equational system of the program is thus:

$$\forall \ell \in \mathcal{L}, \mathcal{X}_\ell = \begin{cases} S_0 & \text{if } \ell \text{ is an entry point} \\ \bigcup_{(\ell',a,\ell) \in CFG} [\![a]\!](\mathcal{X}_{\ell'}) \end{cases}$$

Definition 2.10. *We say that a pair* (s, ℓ) *is a* reachable state *if* s *is in* $\mathcal{X}_\ell$.

Solution This system of equations always has a smallest solution with respect to the order $\subseteq$. If we consider the vector of invariants $\vec{\mathcal{X}} \stackrel{\text{def}}{=} (\mathcal{X}_{\ell_1}, \ldots, \mathcal{X}_{\ell_{|\mathcal{L}|}}) \in \mathcal{P}(\mathcal{E})^{|\mathcal{L}|}$, then the system of equations can be seen as the fixpoint of some function F such that $\vec{\mathcal{X}} = F(\vec{\mathcal{X}})$. $\mathcal{P}(\mathcal{E})^{|\mathcal{L}|}$ is organized on a complete lattice, and one can show that the semantic functions for atomic instructions are monotonic in $\mathcal{P}(\mathcal{E})$, which induces that F is monotonic in $\mathcal{P}(\mathcal{E})^{|\mathcal{L}|}$. Therefore, Tarski's fixpoint theorem ensures the existence of the least fixpoint for F.

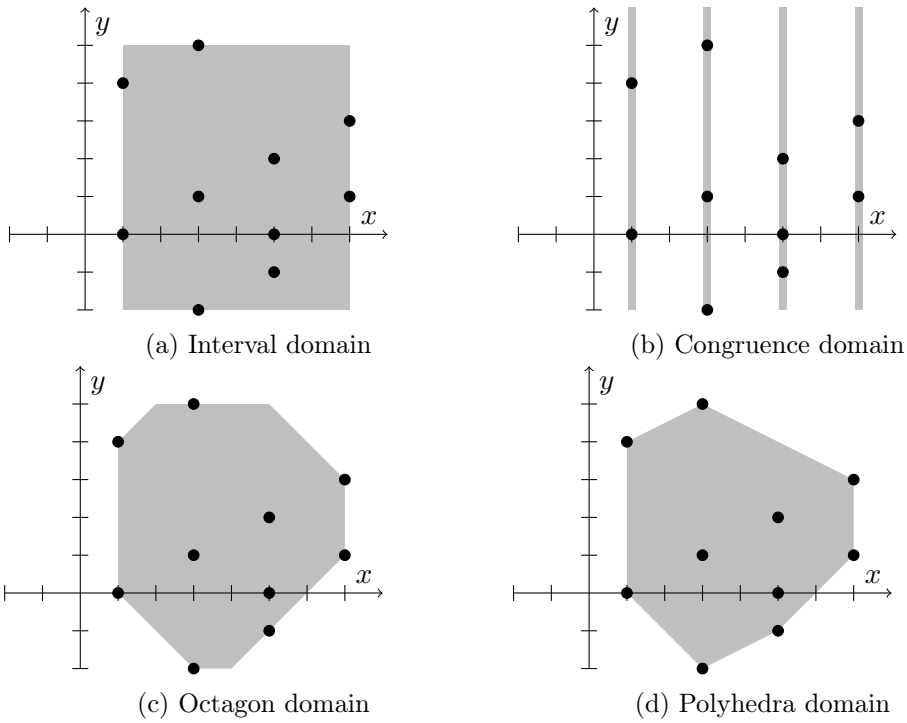
The equational semantics needs to keep in memory the environments at each program points, which can have a dreadful impact on memory consumption. On the other hand, the denotational semantics is defined inductively on the syntax. It does not memorize previous results, the environments at each program points. But in consequence it may spend time into re-exploring statements, for instance in the case of nested loops

Abstract semantics *Abstract semantics* also can be defined with one of these two styles. The difference is the domain: instead of a set of environments in $\mathcal{P}(\mathcal{E})$, semantics are defined for an abstract domain $\mathcal{D}$, usually organized on a complete lattice. When $\mathcal{D}$ is not organized on a complete lattice, we can still use the transfer theorem 2.2 to ensure the soundness of our analysis. It is possible to have a relaxed framework for such cases. All domains and semantics referring to the abstraction are traditionally noted with a $\sharp$: $\mathbb{S}^\sharp[\![\cdot]\!], \mathbb{C}^\sharp[\![\cdot]\!], \ldots$ We do not have guarantee on the monotonicity of the abstract semantics.

The system of equation for the abstract semantics can be solved with different algorithms, one of them being the chaotic iteration introduced by Cousot and Cousot in 1977.

2.1.3 Abstract domains for numeric invariants

Concrete numeric invariants are usually comprehensive representations of the program states. These invariants are possibly infinite sets of values or relations. They are not computable in the general



(a) Interval domain (b) Congruence domain

(c) Octagon domain (d) Polyhedra domain

Figure 5: Some examples of abstract domains. The dots represent concrete configurations. The gray zone is the abstract representation of the configurations, depending on the domain.

case nor representable in a machine. An *abstract* domain approximates sets of invariants, so that they are representable and computable. This section presents several classical abstract domains. Some of these domains are illustrated in Figure 5. We consider an analysis that keeps track of two variables x and y. Each dot on the figures corresponds to a concrete state of the program. The gray areas correspond to the sets of states approximated by the different domains.

Non-relational abstract domains refer to domains where relations (comparison) between variables are forgotten. This includes the sign domain, the constant and constant set domains, the interval domain and finally the congruence domain. In this last numeric domains, each variable is associated to an invariant.

Interval domain This is a very common domain, which associates to each variable an interval, with possibly infinite bounds, in $\{[a, b] | a \in \mathbb{I} \cup -\infty, b \in \mathbb{I} \cup +\infty, a \leq b\}$. Figure 5a illustrates the shape of the representation. In this domain, x is associated to the interval $[1, 7]$ and y to $[-2, 5]$. It is sometimes also referred as the *box* domain, given its shape in a 3-variables space.

Congruence domain This representation can be useful for loop analysis where the step increment is different from one. It associates to each variable the abstraction $a\mathbb{Z} + b$, meaning the variable is equal to a multiple of a plus b (a and b being constants). The shape is given on Figure 5b, with the same concrete states as before. Only the domain of x is pictured. It is $2\mathbb{Z} + 1$.

Non-relational domains might not be precise enough. Relations between variables can be useful to deduce if a test $x < y$ holds or not. Which relations are kept is up to choice, but the most

common domains are the octagon and polyhedra ones.

Octagon domain This domain keeps relations of the form $\pm x \pm y \leq c$ where c is a constant or $+\infty$. It besides adds an artificial variable v_0, always equal to zero, so that the octagon domain include intervals: $x \in [a, b]$ is equivalent to $x - v_0 \leq b$ and $v_0 - x \leq -a$. Figure 5c pictures the shape of the domain. For instance, the relation $-x - y \leq -1$ holds in this example. It defines the lower-left side of the octagon. In a two-variables space, the domain is a polygon with at most eight sides, hence the name octagon.

Polyhedra domain To extend the type of relations tracked, the polyhedra domain represents any affine inequalities between variables. Such inequalities are of the form $a_1x + a_2y + \cdots \geq b$ where $a_1, a_2, \ldots$ and b are constants in $\mathbb{I}$. Figure 5d pictures the shape of this new representation. It is the most precise seen here. This precision however comes with a cost in memory and computation time when update is needed.

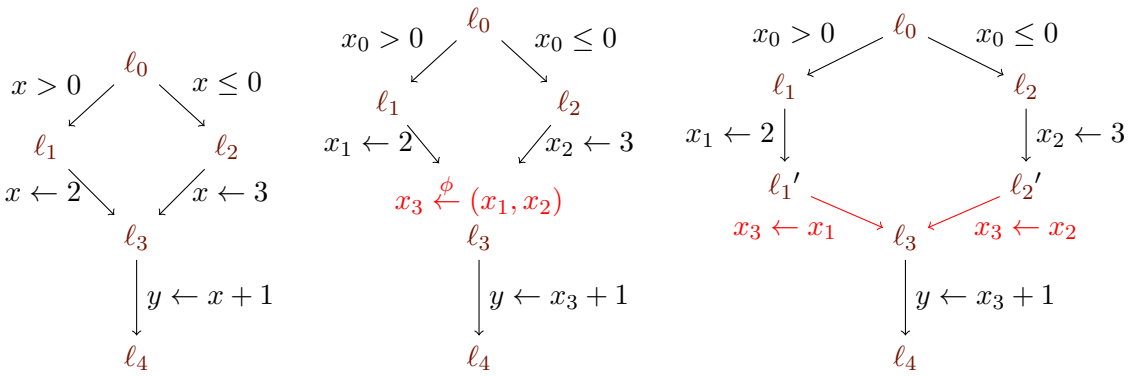
Choosing the most adapted domain clearly depends on the analysis performed. For instance, the Astrée analyzer [8], which aimed at synchronous programs in the realm of aviation, uses specific domains such as the non-relational Arithmetic-Geometric Progression abstract domain and the Clock domain. This analyzer actually combines several domains to achieve the precision wanted. For relations, it does not use the polyhedra domain but the octagon one. Although it is less precise, it is sufficient combined with other domains while being lighter in term of analysis runtime efficiency.

2.2 Intermediate Representations for numerical static analysis

An *intermediate representation* of a program is a representation of a program that respects useful properties for analysis or compilation purposes. This representation should keep the semantics of the original program. Besides the useful properties, the intermediate representation is also convenient to design one analysis that can perform on several languages, as long as there exists a front-end from the language to the intermediate form. For instance, the LLVM compiler uses the SSA form as an intermediate representation for its analyses, but has front-end for languages such as C, C++ or Java. In our case, we will design our analysis on the Static Single Information (SSI) form, as its properties allow flow-insensitivity in the case of non-relational analyses. We hope to be flow-insensitive for relational analyses thanks to this representation. Any language that can be transformed into this SSI form can thus be analysed.

But the SSI form is the results of an incremental construction of intermediate representation with the Static Single Assignment (SSA) form being a milestone. Thus we will first see the SSA form, then complete it with a new operator to get the SSI form.

These two representations targets a similar goal: make explicit in the code the def-use or the use-def chains. A def-use chain associates to each definition of a variable, the instruction that may use it. In other word, if a variable is defined at instruction ℓ and is used at instruction ℓ' then $\ell \to \ell'$ is in the def-use chain if there exists a path from ℓ to ℓ' such that no instruction on this path re-defines the variable. Conversely, $\ell \to \ell'$ is in the use-def chain if the variable used at ℓ may have been defined at ℓ'. (That is there is a path from ℓ' to ℓ such that the variable is not re-defined.) To do so, the analyses split the *live-range* of the variables. The live-range is the set of program points where the variable is alive (it is defined and may be used in a successor).



(a) Original program (b) Program in SSA form (c) Executable program

Figure 6: Example of representation in SSA form and then back to an executable one

2.2.1 Static Single Assignment form

The Static Single Assignment (SSA) form is a property for intermediate representations of programs proposed back in 1988 by Rosen et al. [17]. It has since become a standard for intermediate representation used by compilers, for instance LLVM [13] used it by default, and a middle-end using it was verified for CompCert [4]. The reason for this popularity is that a representation satisfying this property makes def-use chains explicit. In this form, one can know which instructions use a given declaration. Optimizations and analysis are usually simpler with this form. Besides, it is a relatively light property: the size of the SSA form is linear in the size of the original, non-SSA, program representation.

SSA form guarantees that a variable is assigned at only one program point. This is done by indexing the variable at each assignment of the original program. It does not mean the variable can be assigned only once during the *execution* of the program: the assignment can be in a loop. That is why it is called *static* single assignment form.

This property faces an issue when two branches of a program meet, for instance after a conditional statement. Consider the example illustrated in Figure 6. We represented the structured program below as its CFG. The ϕ-function is added to a node, that is to a program point.

The structured program on the right can be represented as a CFG. In that case, the condition will result in a diamond shaped graph. The edges on the left correspond to the *if* branch and the right-ones to the *else* branch. The two branches meet at the end of the **if**.

```
if (x > 0) then
  x ← 2
else
  x ← 3
endif
y ← x + 1
```

The variable x is assigned a value in both branch and we indexed each definition: $x_1 \leftarrow 2$ and $x_2 \leftarrow 3$. After the conditional, the original program computes $y \leftarrow x + 1$. Which x should we use here, x_1 or x_2 ? SSA form addresses this issue by introducing the ϕ-function, which selects the appropriate variable, depending on the branch the execution is coming from. After the conditional statement, when the two branches meet, the assignment $x_3 \overset{\phi}{\leftarrow} (x_1, x_2)$ is introduced. Then in the assignment for y, we use $x_3 : y \leftarrow x_3 + 1$. Notice that this ϕ-function is attached to the program point ℓ_3, it is not an instruction on an edge.

The ϕ-function solely exists in the intermediate representation. In the executable representation of the program, the ϕ-function will be replaced by the statements $x_3 \leftarrow x_1$ and $x_3 \leftarrow x_2$ respectively in each branch as seen in Figure 6c.

Inserting ϕ-function is the main challenge of the SSA-form, re-indexing the variables being trivial. Although ϕ-function could be put after each *join* point of the program (after conditionals and at each loop iteration in our While language), and for each variable, this is unnecessary and wasteful.

Given the graph of the program, the *path-convergence criterion* [2] states that a ϕ is needed for variable v at node n if two nodes have a definition of v and if there is a path from them to n. The paths should not meet anywhere before n, the final node of the path.

Determining these points requires to compute the dominance frontier. The best algorithm currently is the Lengauer-Tarjan's one, which time complexity is almost linear ($N \cdot \alpha(N)$ with N the size of the program and α the slowly growing inverse of the Ackermann function).

Each variable can insert ϕ-functions at the beginning of a node. The semantics of these different ϕ-functions must be applied in *parallel*.

Formally, the semantics of a parallel assignment of variables $a_1, \dots a_n$ to expressions $e_1, \dots, e_n$ consists in taking the values $v_1, \dots, v_n$ of $e_1, \dots, e_n$ and assign these values to $a_1, \dots a_n$.

$$\mathbb{S}[\![a_1 \leftarrow e_1 | \dots | a_n \leftarrow e_n]\!] S \stackrel{\text{def}}{=} \{s[a_1 \mapsto v_1] \dots [a_n \mapsto v_n] | s \in S, v_1 \in \mathbb{E}[\![e_1]\!]s, \dots, v_n \in \mathbb{E}[\![e_n]\!]\}$$

In this program, the set of environments at the end of the if branch should be the singleton $\{s\}$, with $s = \{x_1 \mapsto 2, y_1 \mapsto 4\}$. The semantics of the ϕ-function should assign in parallel the value for y_3 and x_3.

$$\begin{aligned}\mathbb{S}[\![x_3 \leftarrow x_1 | y_3 \leftarrow y_1]\!]\{s\} &= \{s[x_3 \mapsto v][y_3 \mapsto w] \\ &\quad | v \in \mathbb{E}[\![x_1]\!]s, w \in \mathbb{E}[\![y_1]\!]s\} \\ &= \{s[x_3 \mapsto 2][y_3 \mapsto 4]\}\end{aligned}$$

```
if (x > 0) then
  x1 ← 2
  y1 ← 4
else
  x2 ← 3
  y2 ← 6
endif
y3 ←φ (y1, y2)
x3 ←φ (x1, x2)
```

After the renaming, a copy propagation is often applied to reduce the number of variables. The consequence is that a block of ϕ-function can take this form:

$$y_1 \stackrel{\phi}{\leftarrow} (x_1, y_2)$$

$$x_1 \stackrel{\phi}{\leftarrow} (y_1, x_2)$$

In that case, the assignments must be made in parallel to preserve the semantics of the original program.

2.2.2 Static Single Information form

2.2.2.1 Introducing the σ-function

In the SSA form, the live-range is split at each definition and it is enough to infer the def-use chains. This is useful for forward analysis that infers information from the definition site (for instance constant propagation). But other analyses may infer information from the use sites, and

perform a backward analysis. When an intermediate representation needs to distinguish the usage of its variables, it needs an operator dual of the ϕ-function, called the σ-function. Placed before branches, it combines the information on usage from both branches.

The σ-function takes one variable as argument, and its semantics is to copy the value of this variable in several new variables. Each one of this new variables will be used in only one branch. The example below shows where the σ-function is added. Here, x_0 is copied into x_1, used in the then branch, and x_2, used in the else branch.

```
if (x > 0) then
  y1 ← x + 1
else
  y2 ← 2 × x
endif
y3 ←φ (y1, y2)
```

$\rightarrow$

```
(x1 , x2) ←σ x0
if (x > 0) then
  y1 ← x1 + 1        // x1 > 0
else
  y2 ← 2 × x2        // x2 ≤ 0
endif
y3 ←φ (y1, y2)
```

What is interesting is that we have two properties for the new variables: $x_1 > 0$ and $x_2 \leq 0$. Unlike the original code, we no longer need the flow-sensitivity to ensure that $y_1 > 1$ in the first branch, and that $y_2 \leq 0$ in the other.

With these two functions, ϕ and σ, various analyses do not need to track information at each program points, they can jump directly from and to program points where information is provided or necessary. From *dense* analyses, we get *sparse* ones. But the class of analyses that can benefit from it is limited.

2.2.2.2 Limitations

Given an analysis, it is possible to define whether an intermediate representation allows a sparse analysis or not. Both the analysis and the program representation need to satisfy the *Static Single Information* [16] property. Informally, the property is respected if the information on a variable is valid from its definition to its usage and only during this time, that is, during its live-range. Thus, transforming a program into a representation satisfying the property requires to *split* the live-range whenever the information changes (before condition, at a join, at assignment, or any other program point that modifies the information). Notice that what we call *information* depends on the analysis, and thus the SSI form of a program is defined with respect to an analysis. In our case, we deal with numerical analysis and will thus present the definition of the SSI form that satisfies the SSI property for this class of problems.

In this context, the SSI form is an extension of the SSA form that includes the σ-functions. Figure 7 pictures the SSI form of the example Figure 4. The SSI form satisfies the SSI property for analyses in the class of *Partitioned Variable Problems* (PVP) [16]. In this class of problems, the information at each program point can be partitioned between each variable: the information is collected per variable. If we consider non-relational numerical analyses, we are in the case of PVP. For instance a range analysis associates to each variable its range of value, independently of the other variables. In PVP, it is possible to define an equational system on the information of each variable rather than on each program point. This result in one invariant for the whole program, making the analysis sparse. While a dense analysis needs one invariant per program point, which has a significant memory cost, a sparse analysis can have one invariant, and potentially even update it in-place, which is both time and memory saving. Let N be the size of the program, V the number

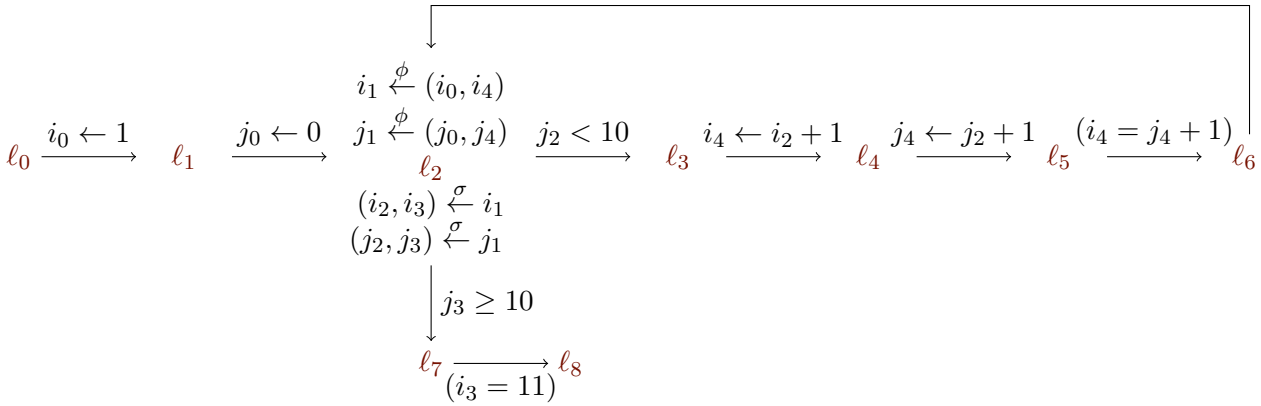


Figure 7: SSI form of the example

of variables. A dense analysis would need a memory space in $O(N \times V)$. The number of variables introduced by the SSA or SSI form is linear in the size of the program [2], so the invariant of a sparse analysis using this form will be in $O(N)$.

In the rest of this paper, we define a *sparse* relational analysis as an analysis able to compute one (relational) invariant for the whole program. This invariant can associate to any combination of variables (depending on the domain used to represent this invariant), the relation between them. The invariant can be seen as the set of constraints on the variables. For instance, in Figure 7, the variables i_2 and j_2 satisfy the constraint $i_2 = j_2 + 1$. Unfortunately, relational analyses are not PVP as the information deals with *pairs* (actually any tuple) of variables. This is a limitation stated in [16]. In the case of relational analyses, the information is collected for tuples of variables, thus if the information on one variable changes, then the information on all other changes too. The SSI form only reasons on the live-range of one variable to ensure the property of single information. In our case, we need to consider the live-range of combinations of variables, thus the live-range of variables would be split every time one of them acquire new information. This would mean that the SSI form would introduce about $V \times N$ variables, ruining the benefit of the sparse analysis. This limit of $O(V \times N)$ variables is our definition of the cost of a dense analysis, while $O(N)$ would be the ideal sparse analysis.

In practice, the analysis of a program does not need to infer all the relations between the variables. Most traditionally, an analyzer will look for assertions to prove in the code [11]. With this objective, not all relations are necessary and we can potentially decrease the amount of variables needed.

2.2.2.3 Construction and destruction

The construction of the SSI form of a program relies on the dominance tree, where nodes are the labels, and a label ℓ *dominates* another one ℓ' if every path from the entry points to ℓ' *must* pass by ℓ. Then ℓ is a parent of ℓ' in the dominance tree. The *dominance frontier* gives for each label ℓ, the sets of labels ℓ' that is *not* dominated by ℓ but that has a parent dominated by ℓ. If a variable is defined at ℓ, ℓ' will probably need a ϕ-function to join this definition from the one coming from another parent of ℓ'.

Construction The construction of the SSI form of a program is a three-steps algorithm [16], done for each variable.

The first step consists in computing the split live-range of the variable. It is split when the information on the variable may change and we need a new version of it. This step depends on the *splitting strategy*, which specifies which instructions change the information. In our case, it changes after each assignment, but also after a test where the variable appears, as we now know more precise information on it. Notice that for other PVP, the splitting strategy may be different. For instance let us consider a null pointer analysis in an object-oriented language. Then calling a method on an object brings information (if the program did not fail after this instruction, the object was not null).

The splitting strategy will induce other splitting points: at join and at branching, if two different information on a variable flow into each branch, then the live-range of the variable must be split at this join or branch. This steps needs the dominance frontier to determine these new splitting points. Once all splitting points for a variable have been computed, the ϕ- and σ-functions are added at branch and join. (In case of a splitting point that is not at a branch or join, then a simple copy is added.)

The second step consists in the renaming of the variable with indexes. It is a traversal of the dominance tree that stacks each new definition. The top of the stack is the name used for each use encountered during the traversal, as long as this definition dominates the use.

Finally, the third step removes dead code (copies not needed, σ copy that is not use in one branch, etc).

Destruction As traditional instruction sets do not provided the ϕ- and σ-functions, it is necessary to destruct the SSI form. This consists in replacing the ϕ and σ by copies and then proceed to a copy propagation to clean the code.

Cost Computing the dominance tree and the dominance frontier of a program from its CFG can be done efficiently with the Lengauer-Tarjan algorithm, as we saw for the SSA form [2]. Its time complexity is almost linear in the size N of the program ($N \cdot \alpha(N)$).

Now let us detail the cost of the transformation for one variable. First, the splitting strategy gives the set of program points where information changes. Computing this strategy is linear in N, as it simply consists in examining each instruction (an edge of the CFG) and determine if its program points (the node defining this edge) are splitting points. Then this set of splitting points is extended with the dominance frontier of the predecessors or successors (depending on the SSI form built) of the splitting points. We extend this set to detect where ϕ- and σ-functions may be needed. This extension is done in one iteration on the set of splitting points, and given that the dominance frontier has already been computed, it has a complexity in $O(N)$. Once the set is complete, the ϕ- and σ-functions are inserted, which is also in $O(N)$ as it is a mere iteration on the extended set of splitting points.

Then the algorithm proceeds to the renaming. It is done through a traversal of the dominance tree, exactly like the SSA transformation, with a stack of definition to remember the last definition used in this path. All nodes of the dominance graph are transformed, and for each of them the successors are examined. Like for SSA, this step is almost linear in N [2]. Finally the dead code elimination is greatly simplified by the SSA-aspect of the SSI form: the uses of a definition can be extracted easily thanks to the re-indexing. This elimination can be performed in linear time with

respect to the size of the program.

Overall the construction of the SSI form for a variable is almost linear in the size of the program.

2.3 Relational analyses

While non-relational analysis are now easier to perform sparsely thanks to convenient intermediate representations, there is no satisfying fits-all representation of the program in the literature for general relational analyses. Yet, there exists several relational analyses with good performance. These analyses are highly valuable to detect bad behaviour of a program such as out of bound array accesses, or violation of assertions. With different objectives, different contexts or usages of the results, they are designed more efficiently and with better precision. However, they are also heterogeneous in their method and proofs, making it hard to reproduce the result on a different analysis. Abstract interpretation is a step to homogenize the results, giving a template that is easier to reproduce and adapt. Despite not using abstract interpretation, some of these analyses are relevant thanks to the representation of the program or of the information they use, as well as the optimized treatment they applied. This section pictures three examples of such relational analyzers, performing on the SSA form or a variation of it. They are sorted chronologically, and one can observe on one hand the development of intermediate representations of programs and on the other the representation of the information.

2.3.1 Global Value Numbering

Global Value Numbering (GVN) is a technique to detect when two computations in a program have equivalent results. The equivalence is then used for optimizations. For instance, if the result of the first computation is stored in a variable v, the second computation can be replaced by v, avoiding repeating it. This optimization is the Common Sub-expression Elimination (CSE). An other optimization consists in moving a computation made in two branches outside of it, shortening the code. For these reasons GVN is often found in optimizations of compilers. In a *local* value numbering, the scope of the computations is restricted to basic blocks, while it is extended to the whole control graph in a *global* one.

In 1988, Alpern et al. [1] proposed an algorithm for GVN. Their main contribution was to detect equivalence of expressions despite control structures such as conditionals. To work properly, the GVN needs a program respecting the SSA form, so that it can associate a unique value (or more exactly a symbolic expression) to each variable. For this expression to be unique, the variable must have only one static definition. Alpern et al.'s algorithm builds a *value graph* based on the SSA form of the program. It is a directed graph representing the symbolic computations of the program made at each assignment. There are two types of nodes, and in both cases, the order of the arguments is important, so the edges are ordered too.

Executable function It can be an operator such as +, or a constant (function of arity 0). The node representing the operator is linked to the arguments.

ϕ-function The transformation to SSA form creates ϕ-function at join points. In the case of $x = \phi(e_1, e_2)$, then the value graph of x will start with a ϕ-node. Each ϕ is labeled, to distinguish from the others. The ϕ-node is linked to each of its arguments, in our case, the value graph of e_1 and e_2.

Two variables are *equivalent* at some program point p if the last assignments of the variables before p correspond to *congruent* nodes in the value graph. *Congruence* of nodes is defined recursively: the nodes must have the same function symbol, applied to congruent arguments. In other words, for each leaving edge, taken in order, the destinations must be congruent. *Congruence* is a symmetric, reflexive and transitive relation, and the GVN algorithm must build its classes of equivalence. However, cycles can appear in the value graph. The current definition of the *congruence* is not enough. There are different sets of classes that can be solutions. Under the (strong) assumption that all variables are initially equivalent, the solution is the maximal fix point satisfying the relation. It is built starting with all expressions in the same set, which is separated when two expressions are proven to be non-equivalent. Finding such a fixpoint due to cycle in a graph is actually the same issue as the one faced when defining the semantics of loops in Section 2.1.2. Hopcroft's partitioning algorithm allows the construction of the classes in $O(E \log E)$ where E is the number of edges in the value graph.

As it is, this analysis does not take into account control structures. To do so, Alpern et al. observe that the results of assignments in two conditionals are equivalent if i) they are given the same values in each corresponding branches and ii) the predicates of the conditionals are congruent. To integrate this pattern in the value graph, they introduce a new function $\phi_{\texttt{if}}$.

At each conditional, for each variable assigned in the conditional, a $\phi_{\texttt{if}}$ node is added to the value graph with three arguments: the value given in the *then* branch, the value given in the *else* branch and finally the predicate. If the partitioning algorithm found two expressions with root $\phi_{\texttt{if}}$ to be congruent, it means the two expressions have congruent predicates and thus branch on the same direction. It also means that the values given are congruent.

Similarly, they introduce new functions for loops: $\phi_{\texttt{enter}}$ and $\phi_{\texttt{exit}}$. Here the conditions for two variables to be equivalent while assigned in a loop body are that i) they have the same initial value, ii) they are modified the same way and iii) loops are executed the same number of time.

$\phi_{\texttt{enter}}$ is inserted at the beginning of the loop, before evaluating the exit predicate. Its arguments are the variable coming in the loop and the one modified at the end of the loop body. This function is similar to the ϕ in the SSA form: it joins the value before entering the loop the first time with the one at the end of the loop body. If the partition algorithm find two congruent expressions with root $\phi_{\texttt{enter}}$, this means the variables have started with congruent values and are modified the same way: they remain congruent in the loop body.

$\phi_{\texttt{exit}}$ is inserted after the loop, when the predicate has stated its end. Its arguments are the predicate and the variable assigned before (the one assigned by a $\phi_{\texttt{enter}}$). If the partition algorithm finds two congruent expressions with root $\phi_{\texttt{exit}}$, this means the variables are congruent at each step of the loop (the first argument should be an expression with $\phi_{\texttt{enter}}$) and the loop should end at the same number of steps, as the exit predicates are congruent too. The end values should then be congruent too.

The proof of correctness provided by [1] proceed by contradiction to assert that congruent variables have the same value [2]. This proof does not follow the abstract interpretation principles. This analysis is efficient, but specialized to one class of problem. Our goal is to make a more general analysis, which would also have a proof more in the spirit of abstract interpretation theory.

[2] Actually, the variables have the same value if they are alive at the program point considered.

2.3.2 Elimination of array bounds checks

Access to an array is made by index, which can in times be out of the bounds of said array. While memory unsafe languages, such as C, do not check if an access to an array is valid, others, like Java, do. To do so, array accesses $A[i]$ are guarded by a bound check ($0 \leq i < A.\text{length}$?). In case the condition is violated, an exception is raised during execution precisely at the access instruction. But this systematic instrumentation of code is cumbersome for the execution (a lot of conditions to evaluate) and for optimizations (they cannot freely reorder the instructions as the exception must be raised at the moment precised by the semantics of the original program). Thus, removing as much as possible of the guards, while staying safe, is a major issue.

ABCD [5] is an example of such analysis. It relies on the e-SSA form, a variant of SSA, to perform a sparse analysis. The variant adds extra copies for variables after conditionals (and also after array accesses as, if it is passed, the index must be within bound). The π-function is in charge of the copies.

```
if (y > 0) then
  ( . . . )
else
  ( . . . )
```

```
if (y0 > 0) then
  y1 ← π(y0)
  ( . . . )
else
  y2 ← π(y0)
  ( . . . )
```

The behaviour of the π-function is really similar to the σ-function, despite its different placement: it copies a variable for each branch of a condition. Now, in the analysis it is possible to states that $y_1 > 0$ and $y_2 \leq 0$. In the branches, this information is propagated by the use of y_1 and y_2 instead of y_0. In this form, a property between two variables (for instance $i < A.\text{length}$) holds for all their common live-range (the set of program points where they are both alive). The analysis can be sparse, with only one invariant for the whole program. This form respects the SSI property for the elimination of array bounds checks. Yet, the analysis is relational: both the index of the access and the size of the array are variables.

ABCD proceeds in two steps to determine whether the indexes are within bounds. First, it checks the upper-bound ($i \leq A.\text{length}$?) then the lower-bound ($0 \leq i$?). The two analyses are similar. Relations tracked in the first case are of the form $v_i - v_j \leq c$ where c is a constant. This domain is more restricted than the octagon seen previously, it actually corresponds to the *zone domain* [14]. It is a convenient domain, where constraints can be represented on a graph. Vertices are variables, sizes of arrays and constants appearing in the program. An edge $v_i \to^c v_j$ denotes the constraint $v_j - v_i \leq c$. With a graph, it is not necessary to represent the constraints for all pair (v_i, v_j): it is assumed that $c = +\infty$ if there is no direct edge. The careful choice of this abstract domain for this application is a key-aspect of its efficiency. In our case, the abstract domain will be free to choose by a user of the analyzer, and thus we cannot assume its efficiency.

An important aspect of ABCD is that it is demand-driven. It will not compute the constraints for all array accesses, only the "hot" ones, the most executed ones. Thus, it will only ask to determine if $i < A.\text{length}$ for some A and i. With the graph representation, determining if $i < A.\text{length}$ is equivalent to find if $i - A.\text{length} \leq -1$, that is, no path from i to $A.\text{length}$ must weight less than -1. The algorithm answering must find the shortest path from i to $A.\text{length}$ and ensure that its weight is at least -1. In our case we will not consider all relations possible and the analysis will be demand driven too. Instead of an array bound check, we target assertions in the code that need

to be proven. The variables used at these assertions should respect the information property: any information on these variables should be valid during all their common live-range.

2.3.3 Path sensitive static analysis

PAGAI [11] is a path sensitive static analyzer that uses abstract interpretation principles. The analysis checks whether **assert** statements in the program always hold or not and computes invariants for loop-headers. The abstract domain is left to the choice of the user, with the polyhedra domain by default. The convergence accelerators are also free to choose. The tool is built upon the LLVM compiler infrastructure, and it takes as input the program in the LLVM bytecode, which is already in SSA form [13]. It then outputs invariants for a subset of program points that includes loop-headers. This tool is close to what we want to achieve: a configurable tool to perform static relational analysis. The analysis is only performed forward, although they claim that the techniques are not specific to it.

PAGAI benefits from the SSA form to perform a sparse analysis instead of a dense one. However, it is not as sparse as GVN or ABCD as it keeps track of invariants at several points in the program. We want to reach higher sparsity, as the cost in memory of several invariants can be prohibitive in some abstract domains. This means we need a better representation of the program: having several program points spares them the need of the SSI property.

Unfortunately, the formalism of PAGAI is not detailed in the article. Yet they present a noteworthy feature: quite surprisingly, the tool does not simply apply the least upper bound $\sqcup$ at join points of the program. Unlike classical abstract interpretation, it relies on a SMT solver to determine the paths possible and it only joins the outcome of these paths. Joining only what could be joined is an efficient way to keep precision. PAGAI benefits from the efficiency of modern SMT-solver to have a more precise result on the possible paths. This also means the expressiveness of SMT-solvers bounds the efficiency: for instance, congruence relationships are not dealt with properly. In that case, an impossible path may have to be taken into account. Whether the path will be considered or not depends on the SMT-solvers, not the abstract domain.

Other optimization brought by PAGAI includes removing variables from the abstract domain if it can be defined as a linear combination of other variables. This alleviates the polyhedra operations, that perform badly on high dimensions. This optimization actually concerns the abstract domain and we will not reproduce it here.

The goal of PAGAI is to ease comparison of different abstract domains, in terms of precision and computation times. Surprisingly, although the polyhedra domain should be more precise statically, it loses in precision because of its widening operators. The octagon and interval domains do not suffer such detrimental operators. Without surprise, the octagon domain gives more precise invariants than the interval one.

3 A sparse flow-insensitive relational static analysis

As we saw, the existing relational analyses are either very specific and achieve sparsity (GVN and ABCD) or they stay general but still need several invariants (PAGAI). We want to show that it is possible to have *one* invariant, that remains precise, for the whole program, using the SSI form. For the precision, we rely on the partition of the variables' lifetime, guaranteed by SSI. Thanks to that we were able to design a static analysis that is

Sparse and flow-insensitive There is only one invariant for the whole program.

Domain independent The abstract domain used to represent the invariant is left free to chose by the end-user. A few proofs of soundness requires additional conditions on the domain, in particular the projection should be exact. These conditions are satisfied by the interval, octagon and polyhedra domains presented before.

Relational It can keep track of the relations between variables.

Abstract interpretation-oriented It is described and proven using the abstract interpretation theory.

Our analysis must be able to give an abstract invariant for the program, such that it is a sound abstraction of the concrete semantic of the program. This correctness is evaluated with respect to the concrete semantics of the SSI form of the program.

Outline This section starts with the formal definition of the SSI form of a program as well as the concrete semantics we consider. As you will see, the invariant for such semantics needs an abstract domain able to express partial environments. We thus detail in the following subsection the overlay of our abstract domain that will allow this expressiveness. Then, the abstract semantics of the program can be defined, using the principles of abstract interpretation stated in the previous section. This abstract semantics consists in a single invariant that can be concretized as an overapproximation of the set of environments of the concrete semantics. Although the single invariant constraint imposes the join of information and a potential loss of precision, the properties of the SSI form actually ensures a satisfying level of precision. This is discussed in a dedicated subsection. Finally, this section ends with an overlook of the implementation in OCaml of a prototype of the analysis.

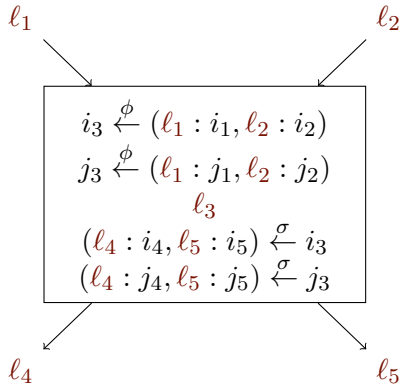
3.1 SSI form and concrete semantics

Our analysis is performed on the SSI form of the original program. Unlike the work presented in [16], where the algorithms are performed on a graph where the atomic instructions are the nodes of a graph, we build our SSI form from its CFG. This allows a definition of the semantics in the framework of abstract interpretation. This section does not detail the algorithms to build the form, as they are similar to the one in [16], but it presents the notations, as well as the concrete collecting semantics of the SSI form.

The SSI form of the program is built from its CFG. Besides the renaming of variables, this process attached ϕ- and σ-functions to labels. The SSI program is a graph with nodes representing the labels. These nodes can be annotated with the ϕ- and σ-function. The edges of the graph are annotated with the atomic instructions.

Notation The SSI intermediate representation of the program is a tuple (E, A, J, B) such that:

- E is the set of edges $(\ell_1 \to \ell_2)$ in the CFG.
- $A \in E \to$ Atom associates to each edge $\ell_1 \to \ell_2$ its atomic instruction given by the CFG.
- J (*join*) is a map from a label to the set of ϕ-functions defined at this label.
- B (*branch*) is the equivalent map for σ-functions.



In this example, the program point ℓ_3 has two predecessors (ℓ_1 and ℓ_2) and two successors (ℓ_4 and ℓ_5).
In the ϕ-functions we precise for each arguments the label associated. That is to say, i_3 is assigned i_1 when coming from ℓ_1. Similarly, we annotate the assigned variables of a σ-function with the label where the variable is defined. Here, i_4 is the version used for label ℓ_4.

Figure 8: The representation for a program in SSI form is the same as the CFG of the program, but we add the ϕ- and σ-functions to the nodes.

A ϕ-function $x \overset{\phi}{\leftarrow} (\ell_1 : x_1, \ldots, \ell_k : x_k)$ at ℓ is represented as a partial function from labels to the variable copied. $J(\ell)$ is thus the ϕ bloc at program point ℓ. It is a partial function that associates to each variable its ϕ-function.

$$J(\ell)(x) = \{\ell_1 \mapsto x_1, \ldots, \ell_k \mapsto x_k\} : \mathcal{L} \nrightarrow \mathbb{V}$$

$$J(\ell) : \mathbb{V} \nrightarrow (\mathcal{L} \nrightarrow \mathbb{V})$$

The labels $\ell_1, \ldots, \ell_k$ are exactly the source labels of the incoming edges of ℓ, noted $\text{In}(\ell)$. The term $J(\ell)(x)(\ell_1) = x_1$ can be read as "At ℓ, when coming from ℓ_1, the program assign x_1 to x".

A σ-function $(\ell_1 : x_1, \ldots, \ell_k : x_k) \overset{\sigma}{\leftarrow} x$ is represented the same way with $\ell_1, \ldots, \ell_k \in \text{Out}(\ell)$.

$$B(\ell)(x) = \{l_1 \mapsto x_1, \ldots, l_k \mapsto x_k\} : \mathcal{L} \nrightarrow \mathbb{V}$$

The term $B(\ell)(x)(\ell_1) = x_1$ can be read as "At ℓ, when going to ℓ_1, the program assign x to x_1".

From now on, it is the SSI form of the program that we call "the program" and we note $P = (E, A, J, B)$. The set of programs is noted $\mathbb{P}$.

In the rest of the section, we define the concrete semantic of a program in SSI form using this notation. It closely ressembles the concrete semantics given in Section 2.1, but adds the effect of ϕ- and σ-functions.

Single entry point We consider that the CFG has only *one* entry point ℓ_0 and that this entry point has no predecessors. Any CFG can be transformed to respect this condition, without loosing its semantics. All it needs is to add this program point, and add edges from it to the previous entry points, with the atomic instruction on these edges being the test $(0 = 0)$, for instance.

Concrete semantics of programs The concrete semantics of the program is given by the function $[\![\cdot]\!] : \mathbb{P} \to \mathcal{P}(\mathcal{L} \times \mathcal{E})$. It is defined as the least fixed point of a *global transfer function* F which applies the transfer function of each edge of the CFG, then joins the results.

$$[\![P]\!] \overset{\text{def}}{=} \text{lfp}(F)$$

We will first detail the transfer function of an edge then detail F.

Edge transfer function At a given program point ℓ, the possible set of environments depends on the predecessors. Let ℓ' be a predecessor of ℓ in the CFG. A state (ℓ, s) is reachable only if there exists an environment s' at ℓ' such that s is s' on which was applied (i) the potential σ-copies from ℓ' to ℓ, (ii) the atomic instruction from ℓ' to ℓ, and finally (iii) the ϕ-copies at ℓ, coming from ℓ'.

For every edge $(\ell' \to \ell) \in E$, we define a transfer function $T_{\ell' \to \ell} : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$. This transfer function is the composition of the transfer functions for step (i), (ii) and (iii), respectively T_σ, T_f and T_ϕ:

$$T_{\ell' \to \ell} \stackrel{\text{def}}{=} T_\phi \circ T_f \circ T_\sigma$$

The functions T_σ and T_ϕ are based on the same principle: collect all copies for the edge ℓ' to ℓ and get the semantics functions of the corresponding parallel assignments.

Let $\text{dom}(B(\ell')) = \{x_1, \ldots, x_n\}$ be the variables for which there is a σ-copy at ℓ'. These are the variables that will be copied. For every x_i, let $x_i' = B(\ell')(x_i)(\ell)$. They are the new copies for ℓ. Then

$$T_\sigma \stackrel{\text{def}}{=} \mathbb{S}[\![x_1' \leftarrow x_1 | \ldots | x_n' \leftarrow x_n]\!]$$

As for ϕ-functions, let $\text{dom}(J(\ell)) = \{y_1, \ldots, y_m\}$ be the variables resulting from a ϕ-copy and $y_i' = J(\ell)(y_i)(\ell')$ be the ones copied when coming from ℓ'. Then

$$T_\phi \stackrel{\text{def}}{=} \mathbb{S}[\![y_1 \leftarrow y_1' | \ldots | y_m \leftarrow y_m']\!]$$

In case where there is no ϕ of σ then these parallel assignments becomes identity functions in the semantics. Finally, if $A(\ell' \to \ell)$, the atomic instruction associated to the edge, is an assign $x \leftarrow v$, then $T_f \stackrel{\text{def}}{=} \mathbb{S}[\![x \leftarrow v]\!]$, else, it is an arithmetic condition $e_1 \bowtie e_2$, then $T_f \stackrel{\text{def}}{=} \mathbb{C}[\![e_1 \bowtie e_2]\!]$.

Example Figure 9 gives an example of tranfer function for an edge. As for the example of Figure 7, let us consider the edge $\ell_1 \to \ell_2$. There is no σ at ℓ_1, so $T_\sigma = \text{id}$.

$$T_{\ell_1 \to \ell_2} = T_\phi \circ T_f \circ T_\sigma = \mathbb{S}[\![i_1 \leftarrow i_0 | j_1 \leftarrow j_0]\!] \circ \mathbb{S}[\![j_0 \leftarrow 0]\!]$$

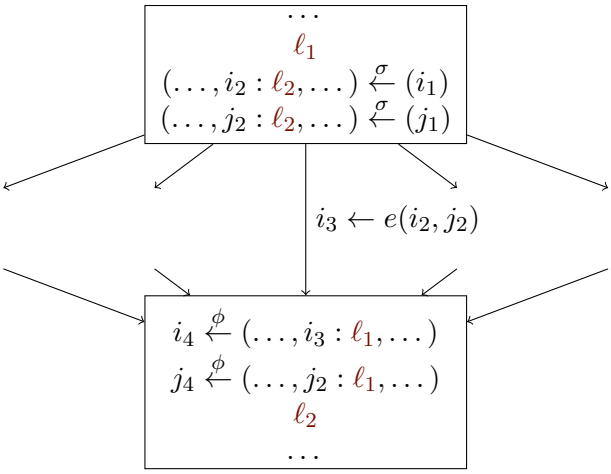
Let us consider the edge $\ell_2 \to \ell_3$. There is no ϕ at ℓ_3 so $T_\phi = \text{id}$.

$$T_{\ell_2 \to \ell_3} = T_\phi \circ T_f \circ T_\sigma = \mathbb{C}[\![j_2 < 10]\!] \circ \mathbb{S}[\![i_2 \leftarrow i_1 | j_2 \leftarrow j_1]\!]$$

Global transfer function Let $S \in \mathcal{P}(\mathcal{L} \times \mathcal{E})$ represents the current invariant of the program. Each of its elements is a pair of a label ℓ with an environment s, such that (ℓ, s) is a reachable state. Several environments can be associated to the same program point. $\mathcal{P}(\mathcal{L} \times \mathcal{E})$ is actually isomorphic to $\mathcal{L} \to \mathcal{P}(\mathcal{E})$ and we will use either definition at our convenience. More precisely, for a given label $\ell \in \mathcal{L}$ we note $S_\ell = \{s \in \mathcal{E} | (\ell, s) \in S\}$ the set of environments attached to ℓ in the invariant S.

The transfer function $F : (\mathcal{L} \to \mathcal{P}(\mathcal{E})) \to (\mathcal{L} \to \mathcal{P}(\mathcal{E}))$ transforms an invariant S by applying the transfer function of each edge of the CFG and joining the resulting sets of states. We note $S' \stackrel{\text{def}}{=} F(S)$ the result (and we note $F(S)(\ell) = S'_\ell$). S'_ℓ is the set of environments attached to ℓ in the invariant S'.

Let us define what should be S'_ℓ for each ℓ. Recall that the set of environments at the entry point ℓ_0 was defined as S_0. Then, $S'_{\ell_0} \stackrel{\text{def}}{=} S_0$. Let $T_{\ell' \to \ell}$ be the transfer function of an edge from ℓ'



$$T_\sigma = \mathbb{S}[\![i_2 \leftarrow i_1 | j_2 \leftarrow j_1]\!]$$
$$T_f = \mathbb{S}[\![i_3 \leftarrow e(i_2, j_2)]\!]$$
$$T_\phi = \mathbb{S}[\![i_4 \leftarrow i_3 | j_4 \leftarrow j_2]\!]$$
$$T_{\ell_1 \to \ell_2} = T_\phi \circ T_f \circ T_\sigma$$

Figure 9: Transfer function for some edge $\ell_1 \to \ell_2$

to ℓ. The new environments at ℓ are the union of all environments $S_{\ell'}$ at the predecessors composed with the transfer functions of the edge $\ell' \to \ell$.

$$S'_\ell \overset{\text{def}}{=} \bigcup_{(\ell' \to \ell) \in E} T_{\ell' \to \ell}(S_{\ell'})$$

From the invariant S we can build the invariant at a next step S'. Let S^k be the S at iteration k. Then in the end, the semantics of the SSI program is the union of all these S^k_ℓ once the fixpoint has been reached.

$$[\![P]\!] = \text{lfp} \left(\bigcup_{\ell \in \mathcal{L}} S^k_\ell \right)$$

Domain coherence As a result from this union, we have a set of partial functions that may not have the same domain (some variables have been defined on an edge and not on the others). In our example, the joining point ℓ_2 is associated with the union of the environments from ℓ_1 and the ones from ℓ_6. The environments from ℓ_1, after applying the transfer function from ℓ_1 to ℓ_2, have as a domain $\{i_0, j_0, i_1, j_1\}$. On the other hand, the environments of ℓ_6 have as a domain $\{i_0, j_0, i_1, j_1, i_2, j_2, i_4, j_4\}$. So the environments at ℓ_2 can have different domains.

3.2 Abstract domain

In the state of the art [12], abstract domains represent environments where the numerical information of each variable is tracked. These abstract environments concretize into *total* functions from variables to values. However, we want to be able to represent partial functions. That is why we build an overlay around some abstract domain $\mathcal{D}$, which can be any numerical abstract domains introduced previously. This section describes this overlay. We want to use the framework of abstract interpretation described by Miné [14]. To be able to use its tool, we provide useful properties for our overlay, such as the monotony of its concretization function. We first define the requirements of the original abstract domain $\mathcal{D}$.

Domain for environments The overlay is an abstract domain parametrized by another abstract domain that abstracts the environments. The domain for environments can be any relational or non-relational domain that allow expressing properties on the value of the variables. Still, it must satisfies a couple of properties. The set of abstract environments is noted $\mathcal{D}$, and it must be provided with a complete lattice $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. Also, there must exist a monotone concretization function from elements of the abstract domain to a set of environments $\gamma_{\mathcal{D}} : \mathcal{D} \to \mathcal{P}(\mathcal{E})$. Finally, the abstract domain must be provided with sound abstractions of assignments and arithmetic conditions. The library Apron, which can manipulate abstract domains for environments provides these properties on its domains [12].

$$\mathbb{S}^\sharp[\![x \leftarrow e]\!] : \mathcal{D} \to \mathcal{D}$$
$$\mathbb{S}^\sharp[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!] : \mathcal{D} \to \mathcal{D}$$
$$\mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!] : \mathcal{D} \to \mathcal{D}$$

such that

$$\forall X \in \mathcal{D} : \mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}}(X) \subseteq \gamma_{\mathcal{D}} \circ \mathbb{S}^\sharp[\![x \leftarrow e]\!](X)$$
$$\forall X \in \mathcal{D} : \mathbb{S}[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!] \circ \gamma_{\mathcal{D}}(X) \subseteq \gamma_{\mathcal{D}} \circ \mathbb{S}^\sharp[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!](X)$$
$$\forall X \in \mathcal{D} : \mathbb{C}[\![e_1 \bowtie e_2]\!] \circ \gamma_{\mathcal{D}}(X) \subseteq \gamma_{\mathcal{D}} \circ \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!](X)$$

Domain for partial functions In the concrete environments, we were able to state that some variables are not defined, using partial functions. This is not possible in an abstract environment of $\mathcal{D}$. A variable is either unconstrained (it can have any value) or bound by some constraints, but it always exists. Still, to increase precision, it is important to know which variables can be defined, and which cannot. We define an overlay of the abstract domain. The elements of this overlay are denoted by a superscript $+$. The new set of abstract environments for our abstract domain pairs an element of $\mathcal{D}$ with a set of variables: $\mathcal{D}^+ \stackrel{\text{def}}{=} ((\mathcal{D} \setminus \{\bot\}) \times \mathcal{P}(\mathbb{V})) \cup \{\bot^+\}$. The bottom element $\bot^+$ represents the empty set of environments. Informally, an abstract environment (X, V) represents the set of concrete environments included in $\gamma_{\mathcal{D}}(X)$ and defined over a set of variables included in V. Abstract states such as $(\bot, V)$ are not allowed, as they would all concretized in an empty set of environments (under the assumption that $\gamma(\bot) = \emptyset$). To avoid the redundancy, we only use $\bot^+$.

Example Let $X \in \mathcal{D}$ be an abstract environment. In the following examples, we will note $X \stackrel{\text{def}}{=} \{i = 0, j = i + 1\}$ to state that $X \stackrel{\text{def}}{=} \{s | s \in \gamma_{\mathcal{D}}(X), i = 0, j = i + 1\}$ Let $V = \{i, j\}$, then $(X, V) \in \mathcal{D}^+$. We did not precise it formally, but the variables constrained in X must be in V for (X, V) to be an element of $\mathcal{D}^+$. In other word, the projection of X on any subset $V' \subseteq V$ should be equal to X. This rule is preserved by the operators we will introduce and the analysis.

Projection In our framework of abstract interpretation ([14]), we need to have a partial order of the elements of $\mathcal{D}^+$. Given the interpretation of abstract environments, we decided to define the order using projections. In Mine's tutorial [14], the abstract projection we presented in Section 2.1 is rather called a non-deterministic assignment and is defined by:

$$\mathbb{V} \setminus V = \{x_1, \ldots, x_n\}, \text{proj}_V^\sharp(X) = \mathbb{S}^\sharp[\![x_1 \leftarrow [-\infty, +\infty] | \ldots | x_n \leftarrow [-\infty, +\infty]]\!]X$$

We will use this definition from now on.

Polyhedra and octagon abstract domain both have an exact non-deterministic assignment [14][15]. Remark that with this definition, for any $X \in \mathcal{D}$, $X \sqsubseteq \text{proj}_V^\sharp(X)$.

Example The projection of X from the previous example onto $U = \{j\}$ is defined by the set of constraints $\{j = 1\}$. For any $c \in \mathbb{I}$, the environment $f = \{j \mapsto 1, i \mapsto c\}$ is in the concretization of $\text{proj}_U^\sharp(X)$ (we forgot the value of i). There exists $f' \in \gamma_{\mathcal{D}}(X)$, such that $f'|_U = f|_U$, for instance $f' = \{j \mapsto 1, i \mapsto 0\}$.

Partial order Informally, all environments represented by (A, V) are also represented by (B, W) if (i) W at least contains V and if (ii) when we restrict B to the variables V, it contains at least A. For (ii), we say that A is included in the existential projection of B on V.

$$(A, V) \sqsubseteq^+ (B, W) \overset{\text{def}}{\Leftrightarrow} V \subseteq W \wedge A \sqsubseteq \text{proj}_V^\sharp(B)$$

Example Let us take $(X, V) \in \mathcal{D}^+$ from the previous example. Let $(Y, W) \in \mathcal{D}^+$ such that $Y \overset{\text{def}}{=} \{i = k, j = i+1, 0 \leq k\}$ and $W = V \cup \{k\}$. Compared to X, we added a positive variable k and specified it is equal to i. Then $(X, V) \sqsubseteq^+ (Y, W)$ as $V \subseteq W$ and $\text{proj}_V^\sharp(Y) = \{0 \leq i, j = i + 1\} \sqsupseteq X$. We consider that (Y, W) is less precise even if it has a new variable k, as it makes us loose the constraint $i = 0$.

Now consider $Z \overset{\text{def}}{=} \{i = 1, j = i + 1\}$ then there is no telling that $(X, V) \sqsubseteq^+ (Z, V)$ or $(Z, V) \sqsubseteq^+ (X, V)$ because neither $X \sqsubseteq Z$ nor $Z \sqsubseteq X$.

Concretization function The concretization function gives all possible partial functions represented by an abstract value $X^+ \in \mathcal{D}^+$. If $X^+ = (X, V)$, then the concretization is all partial functions defined over a subset of V that coincide with a (total) function of the concretization of X.

$$\begin{aligned}
&\gamma_{\mathcal{D}^+} : \mathcal{D}^+ \to \mathcal{P}(\mathcal{E}) \\
&\gamma_{\mathcal{D}^+}(X, V) \overset{\text{def}}{=} \left\{ s|_W \middle| s \in \gamma_{\mathcal{D}}(X), W \subseteq V \right\} \\
&\gamma_{\mathcal{D}^+}(\bot^+) \overset{\text{def}}{=} \emptyset
\end{aligned}$$

Example Let us take (X, V) from the previous example. Let us assume that $V = \{i, j\}$ and that the concretization of X in $\mathcal{D}$ is

$$\gamma_{\mathcal{D}}(X) = \left\{ \{i \mapsto v, j \mapsto v + 1\} \middle| v \in \mathbb{I}, v \geq 0 \right\}$$

Then the concretization of (X, V) in $\mathcal{D}^+$ is

$$\begin{aligned}
\gamma_{\mathcal{D}^+}(X, V) = &\left\{ \{i \mapsto v, j \mapsto v + 1\} \middle| v \in \mathbb{I}, v \geq 0 \right\} && W = V \\
&\cup \left\{ \{i \mapsto v\} \middle| v \in \mathbb{I}, v \geq 0 \right\} && W = \{i\} \subseteq V \\
&\cup \left\{ \{j \mapsto v + 1\} \middle| v \in \mathbb{I}, v \geq 0 \right\} && W = \{j\} \subseteq V \\
&\cup \left\{ \{\} \right\} && W = \emptyset \subseteq V
\end{aligned}$$

Monotony We want our partial order to be organized such that higher elements in $\mathcal{D}^+$ represent larger elements in the concrete world $\mathcal{P}(\mathcal{E})$. These abstract elements represents more program behaviours. This condition is respected if the concretization function is monotonic with respect to our partial order.

Theorem 3.1. [Monotony] *If the projection is exact, that is* $\gamma_{\mathcal{D}} \circ \text{proj}^\sharp_V(X) = \text{proj}_V \circ \gamma_{\mathcal{D}}(X)$ *for any* X *and* V*, then the concretization function is monotonic:*

$$X^+ \sqsubseteq^+ Y^+ \implies \gamma_{\mathcal{D}^+}(X^+) \subseteq \gamma_{\mathcal{D}^+}(Y^+)$$

Proof. If X^+ or Y^+ is $\bot^+$ then the implication trivially holds. Else, let us define $X^+ = (X, V)$ and $Y^+ = (Y, U)$. Let $s_{|_W} \in \gamma_{\mathcal{D}^+}(X, V)$ with $W \subseteq V$ and $s \in \gamma_{\mathcal{D}}(X)$. Then $s_{|_W} \in \gamma_{\mathcal{D}^+}(Y, U)$, as $W \subseteq V \subseteq U$ and by monotony of $\gamma_{\mathcal{D}}$, $s \in \gamma_{\mathcal{D}}(X) \subseteq \gamma_{\mathcal{D}} \circ \text{proj}^\sharp_V(Y) = \text{proj}_V \circ \gamma_{\mathcal{D}}(Y)$. So there exists $s' \in \gamma_{\mathcal{D}}(Y)$ such that $s_{|_V} = s'_{|_V}$. As $W \subseteq V$, necessarily, $s_{|_W} = s'_{|_W}$ and so we find a $s' \in \gamma_{\mathcal{D}}(Y)$ that guarantees $s_{|_W} \in \gamma_{\mathcal{D}^+}(Y, U)$. □

The exactness of the projection is a restrictive condition for our abstract domain $\mathcal{D}$. In the case of the polyhedron domain, the condition holds as the Fourier-Motzkin elimination provides an exact abstraction of the projection. It is also exact in the octagon domain.

Following the methodology of abstract interpretation (as proposed by Miné [14]), to give an abstract semantics mostly consists in finding the good abstraction of the concrete world and its operations. We need to find a sound abstraction for the concrete lattice $(\mathcal{P}(\mathcal{E}), \subseteq, \cup, \cap, \emptyset, \mathcal{E})$. We already have the abstraction for $\mathcal{P}(\mathcal{E})$, $\subseteq$, and $\emptyset$.

Semilattice $(\mathcal{D}^+, \sqsubseteq^+)$ is a poset, but we cannot define a join $\sqcup^+$ necessary to make it a lattice. We will instead simply define an abstract union $\cup^\sharp$. However, we have a meet-semilattice $(\mathcal{D}^+, \sqsubseteq^+, \sqcap^+, \bot^+, \top^+)$. The top is defined as $\top^+ = (\top, \mathbb{V})$.

Meet The meet, which abstracts the intersection of concrete states, can be defined as

$$(A, V) \sqcap^+ (B, W) \overset{\text{def}}{=} \begin{cases} \bot^+ & \text{If } C = \bot \\ (C, V \cap W) & \text{Otherwise} \end{cases} \quad \text{with } C = \text{proj}^\sharp_{V\cap W}(A) \sqcap \text{proj}^\sharp_{V\cap W}(B)$$

$$X^+ \sqcap^+ \bot^+ = \bot^+ \sqcap^+ X^+ \overset{\text{def}}{=} \bot^+$$

Proof. Let us check that it is indeed the greatest lower bound by using the definition of our order $\sqsubseteq^+$. First, let us check that it is a lower bound. We do have the inclusion of the set of variables $V \cap W$ into V and W. Then we need to check the inclusion of C in the projection of A and B onto the set of variables $V \cap W$. This is guaranteed by the definition of C which is exactly the intersection of the projection, thus included in $\text{proj}^\sharp_{V\cap W}(A)$ and $\text{proj}^\sharp_{V\cap W}(B)$. Hence, it is a lower bound.

It is also the greatest. First, the set of variables cannot be augmented or there would be variables that are not elements of V and W. Second, let us suppose that there exists $C' \sqsupseteq C$ such that C' is in the projection of A and B onto $V \cap W$: $C' \sqsubseteq \text{proj}^\sharp_{V\cap W}(A)$ and $C' \sqsubseteq \text{proj}^\sharp_{V\cap W}(B)$. Then necessarily, by definition of the meet $\sqcap$, $C' \sqsubseteq \text{proj}^\sharp_{V\cap W}(A) \sqcap \text{proj}^\sharp_{V\cap W}(B) = C$. So C is the greatest lower bound. □

Abstract union We need to define an abstraction of the concrete union of sets of environments. However, it is not possible to define a join. A counter example can be found using a geometric approach.

Geometric approach Environments can be represented as points in a geometric space where the dimensions are the variables. An abstract environment overapproximates a set of environments, that is a set of points in the geometric view. All the elements of $\mathcal{D}$ can be represented as a set of points in the space defined by the variables $\mathbb{V}$. However, elements of $\mathcal{D}^+$ are defined over different sets of variables. We represent them in their respective subspace. For instance, let us consider that the variables are $\mathbb{V} = \{a, b, c\}$ and that we have $A^+ = (A, \{a, c\})$ such that the environments in the concretization of A satisfies $a = c$. A^+ is a line in the plane defined by $\{a, c\}$. But A is an element of $\mathcal{D}$ defined over the variables $\mathbb{V}$. A has no constraint on b. A is thus a plane in the 3D-space defined by $\mathbb{V}$. This is illustrated on Figure 10.

For the partial order $A^+ \sqsubseteq B^+$, the two objects A^+ and B^+ may not be defined in the same geometric space: V_A can be different of V_B, but we still want to compare the abstract values. To do so we compare A with the projection of B onto the variables of V_A (remember that $V_A \subseteq V_B$ by the definition of the partial order). It is an *existential* projection: all variables not in V_A are unconstrained by the projection. Let us consider A^+ as in the previous example and $C_2^+ = (C_2, \mathbb{V})$ from the Figure 10, defined with the environments in its concretization satisfying $a = c = b$. Let us check that $A^+ \sqsubseteq^+ C_2^+$. First, $V_A \subseteq \mathbb{V}$. Second, the projection of C_2 onto V_A consists in forgetting the constraint on b. The projection of C_2 should thus be the plane in $\mathbb{V}$ defined by $a = c$, which is exactly A and thus confirm the order $A \sqsubseteq C_2$, and then $A^+ \sqsubseteq^+ C_2^+$.

To demonstrate that a join cannot be defined, we need the following lemma that guarantees that the least element that is greater than two other elements is necessarily defined over the union of their variables.

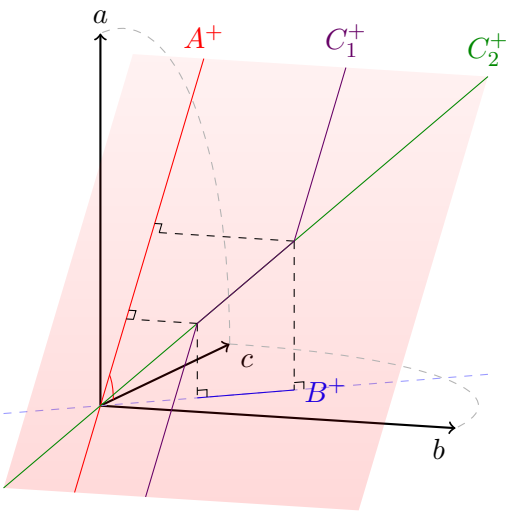
Lemma 3.1. [Minimal variable size] *Let (A, V_A), (B, V_B) and (C, V_C) be three elements of our abstract domain $\mathcal{D}^+$. Then if the latter is greater than the first two, $(A, V_A) \sqsubseteq^+ (C, V_C)$ and $(B, V_B) \sqsubseteq^+ (C, V_C)$, then we can define an intermediate element in the poset that is greater than the first two but is less than (C, V_C): $(A, V_A) \sqsubseteq^+ (C, V_A \cup V_B)$, $(B, V_B) \sqsubseteq^+ (C, V_A \cup V_B)$ and $(C, V_A \cup V_B) \sqsubseteq^+ (C, V_C)$.*

Proof. By definition of the order, V_C must include V_A and V_B. Also, by definition of the projection, for any set of variables V, $C \sqsubseteq \text{proj}_V^\sharp(C)$. In particular, it holds when $V = V_A \cup V_B$. Thus, $(C, V_A \cup V_B) \sqsubseteq^+ (C, V_C)$.

As for the first order, $A \sqsubseteq \text{proj}_{V_A}^\sharp(C)$ by definition of the partial order and $V_A \subseteq V_A \cup V_B$ so $(A, V_A) \sqsubseteq^+ (C, V_A \cup V_B)$ trivially holds. The same applies to (B, V_B). □

This lemma ensures that the least upper bound of two elements must be defined over the union of their variables, or a lesser element could be deduced from it.

Counter example to join For this counter example we use the geometric approach to show that there exists at least one case where for two abstract environments (A, V_A) and (B, V_B), there does not exist a least element (C, V_C) such that $(A, V_A) \sqsubseteq^+ (C, V_C)$ and $(B, V_B) \sqsubseteq^+ (C, V_C)$. This means that we cannot define a join $\sqcup^+$ for our domain: it would not be defined for any A^+ and B^+. Let A be defined over $V_A = \{a, c\}$ with as a constraint $(a = c)$, and B defined over $V_B = \{b, c\}$



The different lines are defined by the following constraints.

- A^+, in the plane (a,c), by $a = c$
- B^+, in the plane (b,c), by $b = c \in [4, 6]$
- C_1^+, in the space (a,b,c), by $a = c$ and if $c \leq 4$, $b = 4$, if $c \in [4, 6]$, $c = b$ and if $c \geq 6$, $b = 6$
- C_2^+, in the space (a,b,c), by $a = c = b$

As for the elements of $\mathcal{D}$, all represented in the space (a, b, c), the projection of C_1 and C_2 are both the plane defined by $a = c$, that is exactly A. C_1 and C_2 both includes A in their projections on $V_A = \{a, c\}$ and B in their projection on $V_B = \{b, c\}$. Yet neither $C_1 \subseteq C_2$ nor $C_2 \subseteq C_1$. Besides, they are minimal: removing a part of C_1 or C_2 would break the inclusion of A in their projection.

Figure 10: Example of impossible least upper bound with three variables

with as a constraint $(4 \leq b = c \leq 6)$. These two elements are represented on the Figure 10. B^+ is represented as a segment, in the plane $\{b, c\}$. We want to find (C, V_C) that would be the least upper bound of (A, V_A) and (B, V_B). First, we must have $V_C = V_A \cup V_B = \{a, b, c\}$ to guarantee the order and to be a least element (by Lemma 3.1). So C^+ will be a 3D object in the space $\{a, b, c\}$. Then, the projection of C on $\{a, c\}$ must contain the line A, and its projection on $\{b, c\}$ must contain the segment B. But a unique minimal C cannot be found. Let us consider two candidates C_1 and C_2. C_1 is defined by the constraints $a = c$ and is segmented for the constraints on b: if $c \leq 4$, then $b = 4$, if $4 \leq c \leq 6$, then $b = c$ and finally if $c \geq 6$ then $b = 6$. Its projection on $\{a, c\}$ is exactly A. Its projection on $\{b, c\}$ contains the segment B. It is a valid upper bound of both (A, V_A) and (B, V_B). Now, C_2 is defined by the constraint $a = c = b$, that is it is a line in the 3D space. Its projection on $\{a, c\}$ and $\{b, c\}$ respectively are both lines that contains A and B respectively. So (C_2, V_C) is also a valid upper bound for (A, V_A) and (B, V_B). But there is no telling whether C_1 is greater than C_2 or the contrary: none of this 3D object contains the other. Also, removing any part of C_1 or C_2 would break the order $A \sqsubseteq \text{proj}^\sharp_{V_A}(C_i)$. They are minimal elements but not unique. This prevents us from defining a join.

Abstract union We define our abstraction of the union, noted $\cup^\sharp$, as the meet of two information. On the one hand we combine the information on the common variables $(V_A \cap V_B)$ and on the other we combine the information from the other variables.

$$(A, V_A) \cup^\sharp (B, V_B) \overset{\text{def}}{=} \left(\left(\text{proj}^\sharp_{V_A \cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A \cap V_B}(B) \right) \sqcap \text{proj}^\sharp_{V_A \setminus V_B}(A) \sqcap \text{proj}^\sharp_{V_B \setminus V_A}(B), V_A \cup V_B \right)$$

And

$$X^+ \cup^\sharp \bot^+ \stackrel{\text{def}}{=} \bot^+ \cup^\sharp X^+ \stackrel{\text{def}}{=} X^+$$

In practice this operator will be applied on values such that $V_A \subseteq V_B$, so the projection on $V_A \backslash V_B = \emptyset$ will be $\top$.

Example Let $X \stackrel{\text{def}}{=} \{i = 0, j = i+1\}$, $Y \stackrel{\text{def}}{=} \{i = 1, j = i+1, k = i\}$, $V = \{i, j\}$, $W = \{i, j, k\}$. First, we do not have $(X, V) \sqsubseteq^+ (Y, W)$ as the information $i = 0$ is not included in Y. As for the abstract union of these two elements, let us detail the result of each projection.

$$\begin{aligned}
\text{proj}^\sharp_{V\cap W}(X) &= \text{proj}^\sharp_V(X) = X \\
\text{proj}^\sharp_{V\cap W}(Y) &= \{i = 1, j = i + 1\} \\
\text{proj}^\sharp_{V\backslash W}(X) &= \text{proj}^\sharp_\emptyset(X) = \top \\
\text{proj}^\sharp_{W\backslash V}(Y) &= \{k = 1\}
\end{aligned}$$

And so

$$\begin{aligned}
(X, V) \sqcup^+ (Y, W) &= \big((X \sqcup \{i = 1, j = i + 1\}) \sqcap \top \sqcap \{k = 1\}, W\big) \\
&= \big(\{0 \leq i \leq 1, j = i + 1\} \sqcap \{k = 1\}, W\big) \\
&= \big(\{0 \leq i \leq 1, j = i + 1, k = 1\}, W\big)
\end{aligned}$$

In the end, the link between k and i has been lost.

This abstract union must be proven sound. This can be proved easily once we proved that the abstract union is an upper bound of each of its operands.

Lemma 3.2. [Abstract union is upper bound] *For any* $A^+, B^+ \in \mathcal{D}^+$,

$$A^+ \sqsubseteq^+ A^+ \cup^\sharp B^+ \quad \text{and} \quad B^+ \sqsubseteq^+ A^+ \cup^\sharp B^+$$

Proof. Let us prove the order $A^+ \sqsubseteq^+ A^+ \cup^\sharp B^+$, the proof is symmetrical for B. Let $A^+ = (A, V_A)$ and $B^+ = (B, V_B)$. First, the set of variables V_A is included in the set of variables of the abstract union $(V_A \cup V_B)$. Then, let us consider the different terms of the meet.

$$\begin{aligned}
A &\sqsubseteq \text{proj}^\sharp_{V_A\cap V_B}(A) \\
&\sqsubseteq \text{proj}^\sharp_{V_A\cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A\cap V_B}(B) \\
\text{and } A &\sqsubseteq \text{proj}^\sharp_{V_A\backslash V_B}(A)
\end{aligned}$$

This give us that A is less than the meet of two of the operands.

$$\begin{aligned}
A &\sqsubseteq \Big(\text{proj}^\sharp_{V_A\cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A\cap V_B}(B)\Big) \sqcap \text{proj}^\sharp_{V_A\backslash V_B}(A) \\
&\sqsubseteq \text{proj}^\sharp_{V_A} \left(\Big(\text{proj}^\sharp_{V_A\cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A\cap V_B}(B)\Big) \sqcap \text{proj}^\sharp_{V_A\backslash V_B}(A)\right)
\end{aligned}$$

This proves a first part: $A^+ \sqsubseteq^+ \left(\left(\text{proj}^\sharp_{V_A \cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A \cap V_B}(B)\right) \sqcap \text{proj}^\sharp_{V_A \setminus V_B}(A), V_A \cup V_B\right)$. We still have to prove that the remaining of the meet is an upper bound of A^+. It corresponds to the projection of B onto the variables V_B not in V_A. The projection on V_A of this projection is necessarily $\top$.

$$A \sqsubseteq \text{proj}^\sharp_{V_A} \circ \text{proj}^\sharp_{V_B \setminus V_A}(B) = \text{proj}^\sharp_{V_A \cap (V_B \setminus V_A)}(B) = \text{proj}^\sharp_{\emptyset}(B) = \top$$

Thus, $A^+ \sqsubseteq^+ (\text{proj}^\sharp_{V_B \setminus V_A}(B), V_A \cup V_B)$. As we proved that A^+ is less than all operands, we do have that it is less than the meet.

$$\begin{aligned} A^+ &\sqsubseteq^+ \left(\left(\text{proj}^\sharp_{V_A \cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A \cap V_B}(B)\right) \sqcap \text{proj}^\sharp_{V_A \setminus V_B}(A), V_A \cup V_B\right) \sqcap^+ \left(\text{proj}^\sharp_{V_B \setminus V_A}(B), V_A \cup V_B\right) \\ &= \left(\left(\text{proj}^\sharp_{V_A \cap V_B}(A) \sqcup \text{proj}^\sharp_{V_A \cap V_B}(B)\right) \sqcap \text{proj}^\sharp_{V_B \setminus V_A}(B) \sqcap \text{proj}^\sharp_{V_A \setminus V_B}(A), V_A \cup V_B\right) \\ &= A^+ \cup^\sharp B^+ \end{aligned}$$

In the case where $V_A \subseteq V_B$, the abstract union is indeed an upper bound. □

Theorem 3.2. [Soundness of abstract union] *The abstract union $\cup^\sharp$ is sound with respect to the concretization function $\gamma_{\mathcal{D}^+}$: for any A^+ and $B^+ \in \mathcal{D}^+$,*

$$\gamma_{\mathcal{D}^+}(A^+) \cup \gamma_{\mathcal{D}^+}(B^+) \subseteq \gamma_{\mathcal{D}^+}\left(A^+ \cup^\sharp B^+\right)$$

Proof. This is simply the result of applying the monotonicity of $\gamma_{\mathcal{D}^+}$ onto the result of the previous lemma. □

Abstract atomic instructions To be able to use the theory of abstract interpretation we still need to define the abstract assignments and conditions in our domain. For assignments and conditions, let $(X, V) \in \mathcal{D}^+$ be an abstract environment.

$$\mathbb{S}^\sharp[\![x \leftarrow e]\!]^+(X, V) = \begin{cases} (\mathbb{S}^\sharp[\![x \leftarrow e]\!]X, V \cup \{x\}) & \text{If } \text{vars}(e) \subseteq V \\ \bot^+ & \text{Otherwise} \end{cases}$$

$$\mathbb{S}^\sharp[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!]^+(X, V) = \begin{cases} (\mathbb{S}^\sharp[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!]X, V \cup \{x_1, \ldots, x_n\}) \\ \qquad \text{If } \text{vars}(e_1) \cup \cdots \cup \text{vars}(e_n) \subseteq V \\ \bot^+ \qquad \text{Otherwise} \end{cases}$$

$$\mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]^+(X, V) = \begin{cases} (\mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]X, V) & \text{If } \text{vars}(e_1) \cup \text{vars}(e_2) \subseteq V \\ \bot^+ & \text{Otherwise} \end{cases}$$

$$\mathbb{S}^\sharp[\![x \leftarrow e]\!]^+\bot^+ = \bot^+ \quad \text{and} \quad \mathbb{S}^\sharp[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!]^+\bot^+ = \bot^+ \quad \text{and} \quad \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]^+\bot^+ = \bot^+$$

Theorem 3.3. [Soundness of the new operators] *The operators of the new abstract domain are sound with respect to the concrete semantics.*

$$\forall X^+ \in \mathcal{D}^+ : \mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}^+}(X^+) \subseteq \gamma_{\mathcal{D}^+} \circ \mathbb{S}^\sharp[\![x \leftarrow e]\!]^+(X^+)$$

$$\forall X^+ \in \mathcal{D}^+ : \mathbb{S}[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!] \circ \gamma_{\mathcal{D}^+}(X^+) \subseteq \gamma_{\mathcal{D}^+} \circ \mathbb{S}^\sharp[\![x_1 \leftarrow e_1 | \ldots | x_n \leftarrow e_n]\!]^+(X^+)$$

$$\forall X^+ \in \mathcal{D}^+ : \mathbb{C}[\![e_1 \bowtie e_2]\!] \circ \gamma_{\mathcal{D}^+}(X^+) \subseteq \gamma_{\mathcal{D}^+} \circ \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]^+(X^+)$$

Proof. In the case where $X^+ = \bot^+$, all these inclusions trivially hold.

Let us prove the soundness of our new assign for $x \leftarrow e$. Let $(X, V) \in \mathcal{D}^+$ be an abstract environment. We first develop each side of the relation to prove (the concrete environments will be noted s in the left side, t is the right one).

$$\begin{aligned}\mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}^+}(X, V) &= \mathbb{S}[\![x \leftarrow e]\!](\{s|_W | s \in \gamma_{\mathcal{D}}(X), W \subseteq V\}) \\ &= \left\{ s' = s|_W[x \mapsto v] \middle| s \in \gamma_{\mathcal{D}}(X), W \subseteq V, v \in \mathbb{E}[\![e]\!]s|_W \right\}\end{aligned}$$

If $\text{vars}(e) \not\subseteq V$, then there is no $v \in \mathbb{E}[\![e]\!]s|_W$ and the term becomes

$$\mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}^+}(X, V) = \emptyset = \gamma_{\mathcal{D}^+}(\bot^+) = \gamma_{\mathcal{D}^+} \circ \mathbb{S}^\sharp[\![x \leftarrow e]\!]^+(X, V)$$

Else, $\text{vars}(e) \subseteq V$. We develop the right-hand term, defining an auxiliary term T.

$$\begin{aligned}\gamma_{\mathcal{D}^+} \circ \mathbb{S}^\sharp[\![x \leftarrow e]\!]^+(X, V) &= \gamma_{\mathcal{D}^+}(\mathbb{S}^\sharp[\![x \leftarrow e]\!](X), V \cup \{x\}) \\ &= \left\{ t'|_U \middle| t' \in \gamma_{\mathcal{D}} \circ \mathbb{S}^\sharp[\![x \leftarrow e]\!](X), U \subseteq V \cup \{x\} \right\} \\ &\supseteq \left\{ t'|_U \middle| t' \in \mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}}(X), U \subseteq V \cup \{x\} \right\} \\ &\supseteq \left\{ t'|_U \middle| t \in \gamma_{\mathcal{D}}(X), t' = t[x \mapsto u], u \in \mathbb{E}[\![e]\!]t, U \subseteq V \cup \{x\} \right\} \overset{\text{def}}{=} T\end{aligned}$$

Let us show that for all $s' \in \mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}^+}(X, V) \overset{\text{def}}{=} S$, then $s' \in T$. We have to find the corresponding t, U and $u \in \mathbb{E}[\![e]\!]t$. $s' \in S$ implies the existence of an environment s and a value v. Let $t = s \in \gamma_{\mathcal{D}}(X)$, $u = v$, $U = W \cup \{x\}$ and $t' = t[x \mapsto u]$. The equality $s' = \left(s|_W\right)[x \mapsto v] = (s[x \mapsto v])|_{W \cup \{x\}} = t'|_U$ is valid on the whole domain $W \cup \{x\}$.

Now, we must show that $t'|_U$ is in T. First, we do have that $t \in \gamma_{\mathcal{D}}(X)$ as $s = t$ and s is an element of the concretization. Then, we must show that $u \in \mathbb{E}[\![e]\!]t$. The equality $s = t$ implies that $u = v \in \mathbb{E}[\![e]\!]s = \mathbb{E}[\![e]\!]t$. Finally, $U \subseteq V \cup \{x\}$ as $U = W \cup \{x\} \subseteq V \cup \{x\}$. This concludes the proof that $s' \in T$, and so

$$\mathbb{S}[\![x \leftarrow e]\!] \circ \gamma_{\mathcal{D}^+}(X, V) \subseteq \gamma_{\mathcal{D}^+} \circ \mathbb{S}^\sharp[\![x \leftarrow e]\!]^+(X, V)$$

The proof for the parallel assignment can be done likewise.

The proof for the arithmetic condition is similar. We used the fact that if an expression e can be computed on a environment s restricted over the set of variables W, that is $\mathbb{E}[\![e]\!]s|_W \neq \emptyset$, then extending the domain will yield the same set: $\mathbb{E}[\![e]\!]s|_W = \mathbb{E}[\![e]\!]s$. If $\text{vars}(e_1)$ and $\text{vars}(e_2) \subseteq V$ then

$$\begin{aligned}\mathbb{C}[\![e_1 \bowtie e_2]\!] \circ \gamma_{\mathcal{D}^+}(X, V) &= \mathbb{C}[\![e_1 \bowtie e_2]\!](\{s|_W | s \in \gamma_{\mathcal{D}}(X), W \subseteq V\}) \\ &= \left\{ s|_W \middle| \begin{array}{l} s \in \gamma_{\mathcal{D}}(X), W \subseteq V \\ v_1 \in \mathbb{E}[\![e_1]\!]s|_W, v_2 \in \mathbb{E}[\![e_2]\!]s|_W, v_1 \bowtie v_2 \end{array} \right\} \\ &= \{s|_W | s \in \mathbb{C}[\![e_1 \bowtie e_2]\!] \circ \gamma_{\mathcal{D}}(X), W \subseteq V\} \\ &\subseteq \{s|_W | s \in \gamma_{\mathcal{D}} \circ \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!](X), W \subseteq V\} \\ &= \gamma_{\mathcal{D}^+}(\mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!](X), V) \\ &= \gamma_{\mathcal{D}^+} \circ \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]^+(X, V)\end{aligned}$$

Else, if V is too small, then the second line will be equal to the empty set as no W and no v_1 or v_2 can be found. The empty set is equal to the concretization of $\gamma_{\mathcal{D}^+} \circ \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]^+(X, V)$ in the case where V does not contain the variables of e_1 and e_2. □

From global to local invariant We saw that in traditional flow-sensitive analysis, the semantics of the program is computed with a function F applied on an element S of $\mathcal{D}^{|\mathcal{L}|}$. F itself applies transfer functions on each abstract environment S_ℓ of $\mathcal{D}$. The soundness of the abstract semantics with respect to the concrete one is proven with the concretization function $\gamma_{\mathcal{D}}$ from abstract environments in $\mathcal{D}$ to set of concrete environments in $\mathcal{P}(\mathcal{E})$. Then, we link these different sets to their labels, obtaining the set of states $\mathcal{P}(\mathcal{L} \times \mathcal{E})$. In our case, we won't have one abstract environment per label, so we do not have an element of $\mathcal{D}^{|\mathcal{L}|}$ for our invariant, but instead a single environment in our overlay $\mathcal{D}^+$ for all labels. So we will still have a concretization function $\gamma_{\mathcal{D}^+}$ from $\mathcal{D}^+$ to $\mathcal{P}(\mathcal{E})$ but then we have to define how to get the set of states $\mathcal{P}(\mathcal{L} \times \mathcal{E})$.

We decided to leave this job to a concretization function $\gamma : \mathcal{D}^+ \to \mathcal{P}(\mathcal{L} \times \mathcal{E})$. Let us consider a definition where every label is associated to any environment from the concretization.

$$\gamma(X^+) = \{(\ell, s) | \ell \in \mathcal{L}, s \in \gamma_{\mathcal{D}^+}(X^+)\}$$

This concretization function is monotone in X^+, as $\gamma_{\mathcal{D}^+}$ is also monotone in X^+.

Associating the complete concretization to each program point is a significant overapproximation. We can actually be more precise by considering $s \in \gamma_{\mathcal{D}^+}(X^+)$ for label ℓ only if the variables of s, its domain, may all be defined at ℓ. Consider for instance the program of Figure 7. At program point ℓ_1, only i_1 is defined. There is no need to consider the partial function $\{i_1 \mapsto 1, j_1 \mapsto 0\}$ as j_1 cannot be defined. We did not present this more precise γ for the sake of concision.

3.3 Abstract semantics

In this Section, we define the abstract semantics computed by the analysis. We prove that it is sound, with respect to the concrete semantics given to the SSI form of the program.

Abstract semantics soundness Let $F^\sharp$ be the abstract transfer function of the program. Any post-fixpoint $X^+ \in \mathcal{D}^+$ of $F^\sharp$ is a sound approximation of the concrete semantics if $[\![P]\!] \subseteq \gamma(X^+)$. Where $[\![P]\!]$ is the concrete semantics of P given in the previous section, that is to say the least fixed point of the concrete transfer function F ($[\![P]\!] = \text{lfp}(F)$). We saw with the fixpoint transfer theorem that this condition on post-fixpoint is a consequence of $F \circ \gamma \subseteq \gamma \circ F^\sharp$, provided that F and γ are monotonic (Theorem 2.2). We already proved the monotonicity of F and γ. Let us define $F^\sharp$, prove that it is monotonic and that it respects the condition of the fixpoint transfer theorem.

Transfer function In the concrete semantics, we defined for each edge $\ell \to \ell' \in E$, a transfer function $T_{\ell \to \ell'} = T_\phi \circ T_f \circ T_\sigma$, where T_ϕ and T_σ are parallel assignments transfer functions, and T_f corresponds either to the semantics of an assignment or an arithmetic condition, depending on the edge $\ell \to \ell'$. We defined the abstract semantics of an edge similarly. Let us suppose that,

$$T_\sigma = \mathbb{S}[\![x'_1 \leftarrow x_1 | \dots | x'_n \leftarrow x_n]\!]$$

$$T_\phi = \mathbb{S}[\![y_1 \leftarrow y'_1 | \dots | y_m \leftarrow y'_m]\!]$$

Then

$$T^\sharp_{\ell\to\ell'} \stackrel{\text{def}}{=} T^\sharp_\phi \circ T^\sharp_f \circ T^\sharp_\sigma$$

$$T^\sharp_\sigma \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![x'_1 \leftarrow x_1 | \dots | x'_n \leftarrow x_n]\!]$$

$$T^\sharp_\phi \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![y_1 \leftarrow y'_1 | \dots | y_m \leftarrow y'_m]\!]$$

Also, if $T_f = \mathbb{S}[\![x \leftarrow e]\!]$ then $T^\sharp_f \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![x \leftarrow e]\!]$. Else, if $T_f = \mathbb{C}[\![e_1 \bowtie e_2]\!]$ then $T^\sharp_f \stackrel{\text{def}}{=} \mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!]$. As the abstract semantics of assignment and arithmetic condition are sound, one can show that their composition is sound too. Thus $T^\sharp_{\ell\to\ell'}$ is a sound approximation of $T_{\ell\to\ell'}$: for any $X^+ \in \mathcal{D}^+$,

$$T_{\ell\to\ell'} \circ \gamma_{\mathcal{D}^+}(X^+) \subseteq \gamma_{\mathcal{D}^+} \circ T^\sharp_{\ell\to\ell'}(X^+)$$

$F^\sharp$ as a composition Let us define a new $F^\sharp$ as a composition of transfer functions $F^\sharp_{\ell\to\ell'}$ each associated to one edge (the order of the composition does not matter for soundness, only for efficiency of the fixpoint iteration). We also define $F^\sharp_0$, the transfer function that will add the abstract environment for the entry point, that we note X^+_0 ($S_0 \subseteq \gamma_{\mathcal{D}^+}(X^+_0)$).

$$F^\sharp \stackrel{\text{def}}{=} F^\sharp_0 \circ \bigcirc_{(\ell\to\ell')\in E} F^\sharp_{\ell\to\ell'}$$

Let $F^\sharp_{\ell\to\ell'}(X^+) \stackrel{\text{def}}{=} X^+ \cup^\sharp T^\sharp_{\ell\to\ell'}(X^+)$ and $F^\sharp_0(X^+) \stackrel{\text{def}}{=} X^+_0 \cup^\sharp X^+$. We will make all our proofs using the facts that

$$X^+ \sqsubseteq^+ F^\sharp_{\ell\to\ell'}(X^+) \quad \text{and} \quad T^\sharp_{\ell\to\ell'}(X^+) \sqsubseteq^+ F^\sharp_{\ell\to\ell'}(X^+)$$
$$X^+ \sqsubseteq^+ F^\sharp_0(X^+) \quad \text{and} \quad X^+_0 \sqsubseteq^+ F^\sharp_0(X^+)$$

We need to ensure that this $F^\sharp$ is sound with respect to F and γ. To do so, we will use a small lemma, that ensures that $F^\sharp$ is monotonic.

Lemma 3.3. [Monotony] *$F^\sharp$ is monotonic, that is for any X^+ and Y^+ in $\mathcal{D}^+$,*

$$X^+ \sqsubseteq^+ Y^+ \implies F^\sharp(X^+) \sqsubseteq^+ F^\sharp(Y^+)$$

Proof. For each edge e, $F^\sharp_e$ is monotonic so their composition is monotonic too. Finally, $F^\sharp_0$ is monotonic, so $F^\sharp$ is monotonic. □

Theorem 3.4. [Soundness] *$F^\sharp$ respects the condition of the fixpoint transfer Theorem 2.2 with respect to F and γ, that is to say $F \circ \gamma \subseteq \gamma \circ F^\sharp$.*

Proof. Let $X^+ \in \mathcal{D}^+$ and $S \stackrel{\text{def}}{=} \gamma(X^+) = \{(\ell, s) | \ell \in \mathcal{L}, s \in \gamma_{\mathcal{D}^+}(X^+)\}$. The proof consists in proving the two inclusions:

$$F \circ \gamma = F(S) \subseteq \gamma(X^+_0) \cup \bigcup_{(\ell_1\to\ell_2)\in E} \gamma \circ T^\sharp_{\ell_1\to\ell_2}(X^+) \subseteq \gamma \circ F^\sharp(X^+) = \gamma \circ F^\sharp_0 \circ \bigcirc_{e\in E} F^\sharp_e(X^+)$$

Let us show the first one, $F \circ \gamma$ is included in the union of $\gamma \circ X^+_0$ and the union of $\gamma \circ T^\sharp_{\ell_1\to\ell_2}$ over the edges $\ell_1 \to \ell_2 \in E$. First, at line 3, we unpair in each state the label ℓ from the environment

s. As any label can be associated with any environment, we did overapproximate the set of states. At line 4 we no longer give only the set of environments S_{ℓ_1} to the transfer function $T_{\ell_1\to\ell_2}$, but instead we give *all* the environments computed, for all labels. This overapproximation is possible as the $T_{\ell_1\to\ell_2}$ are monotonic. We then exploit the soundness of $T^\sharp_{\ell_1\to\ell_2}$ and X^+_O at line 5.

$$
\begin{aligned}
F\circ\gamma(X^+) &= F(S) && (1)\\
&= \left\{(\ell',s)\middle|(\ell\to\ell')\in E, s\in T_{\ell\to\ell'}(S_{\ell'})\right\}\cup\{(\ell_0,s)|s\in S_0\} && (2)\\
&\subseteq \left\{(\ell',s)\middle|(\ell_1\to\ell_2)\in E, s\in T_{\ell_1\to\ell_2}(S_{\ell_1})\right\}\cup\left\{(\ell',s)\middle|\ell'\in\mathcal{L}, s\in S_0\right\} && (3)\\
&\subseteq \left\{(\ell',s)\middle|(\ell_1\to\ell_2)\in E, s\in T_{\ell_1\to\ell_2}(\underbrace{\bigcup_{\ell\in\mathcal{L}} S_\ell}_{\gamma_{\mathcal{D}^+}(X^+)})\right\}\cup\left\{(\ell',s)\middle|\ell'\in\mathcal{L}, s\in\gamma_{\mathcal{D}^+}(X^+_0)\right\} && (*)\\
& && (4)\\
&\subseteq \left\{(\ell',s)\middle|(\ell_1\to\ell_2)\in E, s\in\gamma_{\mathcal{D}^+}\circ T^\sharp_{\ell_1\to\ell_2}(X^+)\right\}\cup\gamma(X^+_0) && (5)\\
&= \gamma(X^+_0)\cup\bigcup_{(\ell_1\to\ell_2)\in E}\gamma\circ T^\sharp_{\ell_1\to\ell_2}(X^+) && (6)
\end{aligned}
$$

This final result is included in $\gamma\circ F^\sharp$. First, let us show by induction on the number of edges that the right term is a subset of $\gamma\circ\bigcirc_{e\in E}F^\sharp_e(X^+)$.

- If $E=\emptyset$, then the union is an empty set. The inclusion is trivial.
- If there is only one edge $e\in E$, then we can conclude by simply applying the monotonicity of γ
$$\gamma\circ T^\sharp_e(X^+)\subseteq\gamma\circ F^\sharp_e(X^+)$$
- Else, let us suppose that $E=E'\cup\{e\}$ and that we have proved the inclusion on E': $\bigcup_{e'\in E'}\gamma\circ T^\sharp_{e'}(X^+)\subseteq\gamma\circ\bigcirc_{e'\in E'}F^\sharp_{e'}(X^+)$. Also, we have $X^+\sqsubseteq^+\bigcirc_{e'\in E'}F^\sharp_{e'}(X^+)$. Then

$$
\begin{aligned}
\bigcup_{e'\in E}\gamma\circ T^\sharp_{e'}(X^+) &= \gamma\circ T^\sharp_e(X^+)\cup\bigcup_{e'\in E'}\gamma\circ T^\sharp_{e'}(X^+) && (1)\\
&\subseteq \gamma\circ T^\sharp_e(X^+)\cup\gamma\circ\underbrace{\textstyle\bigcirc_{e'\in E'}F^\sharp_{e'}(X^+)}_{\stackrel{\text{def}}{=}C(X^+)} && (2)\\
&= \gamma\circ T^\sharp_e(X^+)\cup\gamma\circ C(X^+) && (3)\\
&\subseteq \gamma\circ T^\sharp_e\circ C(X^+)\cup\gamma\circ C(X^+) && (4)\\
&\subseteq \gamma\circ F^\sharp_e\circ C(X^+) = \gamma\circ F^\sharp_e\circ\textstyle\bigcirc_{e'\in E'}F^\sharp_{e'}(X^+) && (5)\\
&= \gamma\circ\textstyle\bigcirc_{e'\in E}F^\sharp_{e'}(X^+) && (6)
\end{aligned}
$$

We use the monotonicity of $T^\sharp_e$ to get line 4 from the previous one. We then use the hypotheses on $F^\sharp_e$ and the soundness of $\cup$ to get the next line.

Thus the inclusion holds for any set of edges E. In the end,

$$\begin{aligned} F \circ \gamma(X^+) &\subseteq \gamma(X_0^+) \cup \bigcup_{(\ell_1 \to \ell_2) \in E} \gamma \circ T^\sharp_{\ell_1 \to \ell_2}(X^+) \\ &\subseteq \gamma(X_0^+) \cup \bigcirc_{e' \in E} F^\sharp_{e'}(X^+) \\ &\subseteq \gamma \circ F^\sharp_0 \circ \bigcirc_{e' \in E} F^\sharp_{e'}(X^+) \\ &= \gamma \circ F^\sharp(X^+) \end{aligned}$$

So $F^\sharp$ satisfies the condition of the fixpoint transfer theorem. □

Precision In the beginning of this last proof, the line $(*)$ raises an issue for precision. From this line, we no longer say that s is an element of $S'_{\ell_1} = T_{\ell_1 \to \ell_2}(S_{\ell_1})$. This set of environments corresponds to the application of the transfer function of an edge $\ell_1 \to \ell_2$ on the environments at ℓ_1. Now we apply this transfer function on *all* environments. Especially, we apply the transfer function even if the set of environments at ℓ_1 could be empty (the branch is never taken). From this point, we loose all information on the control flow.

Example Consider that the current abstract environment is (X, V) with $X \overset{\text{def}}{=} \{j_2 \in [0,4], i_2 = j_2 + 1, j_4 \in [0,4], i_4 = j_4 + 1\}$, $V = \{i_2, j_2, i_4, j_4\}$ and we are examining the program of Figure 7 [3]. We need to apply the transfer function of the edges $\ell_3 \to \ell_4$ and $\ell_4 \to \ell_5$. With the first one, the invariant forget that $i_4 = j_4 + 1$ as we assign it a new value. But we also loose the relation between i_2 and i_4, as we make an abstract union with (X, V) which does not have such relation.

$$\begin{aligned} F^\sharp_{\ell_3 \to \ell_4}(X, V) &= \mathbb{S}^\sharp[\![i_4 \leftarrow i_2 + 1]\!]^+(X, V) \cup^\sharp (X, V) \\ &= (\{j_2 \in [0,4], i_2 = j_2 + 1, j_4 \in [0,4], i_4 = i_2 + 1\}, V \cup \{i_4\}) \cup^\sharp (X, V) \\ &= (\{j_2 \in [0,4], i_2 = j_2 + 1, j_4 \in [0,4], i_4 \in [1,6]\}, V) \\ &= (X', V) \end{aligned}$$

Then we apply the transfer function of the other edge.

$$\begin{aligned} F^\sharp_{\ell_4 \to \ell_5}(X', V) &= \mathbb{S}^\sharp[\![j_4 \leftarrow j_2 + 1]\!]^+(X', V) \cup^\sharp (X', V) \\ &= (\{j_2 \in [0,4], i_2 = j_2 + 1, j_4 = j_2 + 1, i_4 \in [1,6], \}, V \cup \{j_4\}) \cup^\sharp (X', V) \\ &= (\{j_2 \in [0,4], i_2 = j_2 + 1, j_4 \in [0,5], i_4 \in [1,6]\}, V) \end{aligned}$$

There we see that the relation between i_4 and j_4 has been lost.

Because we have one invariant for the whole program, we cannot prevent the application of the transfer function on all environments. To still retain some information, we need to delay this broad application as much as possible. Instead of applying the transfer function *per edge*, we want to apply them *per bloc*. A bloc is a path with no ingoing or outgoing edges in the intermediate program points, for instance, in Figure 4a, $\ell_2 \to \ell_3 \to \ell_4 \to \ell_5 \to \ell_6 \to \ell_2$ is a block, but not

[3] In practice i_1 and j_1 should also be in V in this example, but it has no influence on the precision lost we explain here.

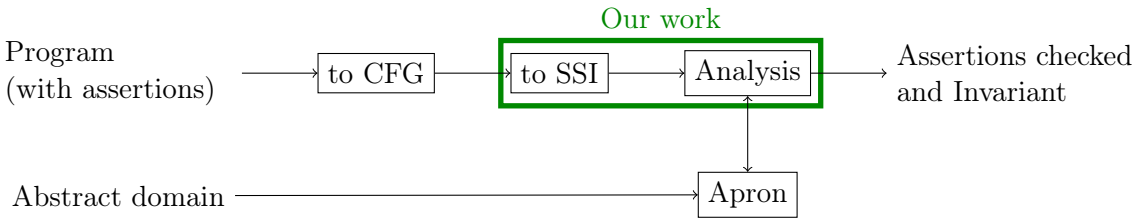


Figure 11: Workload of the prototype

The transformation to SSI is not our main contribution, as it is based on an already existing algorithm [16]. However we choose a different formalism, based on the CFG, to conform to the state of the art in abstract interpretation.

$\ell_0 \to \ell_1 \to \ell_2 \to \ell_7 \to \ell_8$. What we mean by "apply per block" is that if in the CFG, there is a bloc from ℓ_1 to ℓ_2, then the composition of all transfer functions along this path can be applied on the abstract environment.

$$F^\sharp_{\ell_6\to\ell_2} \circ \cdots \circ F^\sharp_{\ell_2\to\ell_3}(X^+) \overset{\text{Becomes}}{\Longrightarrow} X^+ \cup^\sharp T^\sharp_{\ell_6\to\ell_2} \circ \cdots \circ T^\sharp_{\ell_2\to\ell_3}(X^+)$$

Thus, if an arithmetic condition is not satisfied along the path, resulting in a bottom abstract environment, none of the next transfer functions will be applied.

Example In the case of our example program, we build the paths $p_1 \overset{\text{def}}{=} \ell_0 \to \ell_1 \to \ell_2$, $p_2 \overset{\text{def}}{=} \ell_2 \to \ell_3 \to \ell_4 \to \ell_5 \to \ell_6 \to \ell_2$ and $p_3 \overset{\text{def}}{=} \ell_2 \to \ell_7 \to \ell_8$. The path p_1 has the transfer function $\mathbb{S}^\sharp[\![i_1 \leftarrow i_0 | j_1 \leftarrow j_0]\!] \circ \mathbb{S}^\sharp[\![j_0 \leftarrow 0]\!] \circ \mathbb{S}^\sharp[\![i_0 \leftarrow 1]\!]$. The path p_2 has the transfer function $\mathbb{S}^\sharp[\![i_1 \leftarrow i_4 | j_1 \leftarrow j_4]\!] \circ \mathrm{id} \circ \mathbb{S}^\sharp[\![j_4 \leftarrow j_2 + 1]\!] \circ \mathbb{S}^\sharp[\![i_4 \leftarrow i_2 + 1]\!] \circ \mathbb{C}^\sharp[\![j_2 < 10]\!] \circ \mathbb{S}^\sharp[\![i_2 \leftarrow i_1 | j_2 \leftarrow j_1]\!]$ By composition, this last transfer function only requires i_1 and j_1 to be defined in the abstract value. Also, if j_1 is never strictly less than 10, then the condition $\mathbb{C}^\sharp[\![j_2 < 10]\!]$ will always result in a $\bot^+$ value. Then none of the following semantic functions will define the variables i_2, j_2, i_4, j_4 which will not be defined in the final invariant.

3.4 Implementation

The analysis presented before, which consists in finding the least fixpoint of some abstract transfer function, was implemented in OCaml. It takes as input a program in While language with assertions. The analysis has an option to choose between an octagon or polyhedra abstract domain. The output is the global invariant of the program (expressed as a value of the abstract domain). The analysis is iterative and does not implement acceleration techniques. If the fixpoint is indeed found, then each assertion of the original program is checked, and whether it passes or not is printed.

This analysis is based on a existing parser for the While language which outputed the CFG of the input program. The algorithm to transform this CFG into an SSI form will not be detailed here as an efficient transformation can be found in [2] (efficient computation of the dominance tree necessary) and [16] (actual transformation). The main difference between our algorithm and the one cited is that we perform the algorithm on a CFG, so the nodes are program points, while in the cited one the nodes are instructions. In our implementation, the output is a value of type `ssiPrgm`. This structure contains information on the program such as its variables, the CFG, or also the $\phi-$

and $\sigma-$ nodes attached to each label. (Arrays are often used to represent a total map with labels as key.)

```
module VarMap : Map.S with type key = Syntax.var
module LabelMap : Map.S with type key = Syntax.label

type copyNode = (Syntax.var LabelMap.t) VarMap.t  /* 𝕍 ⇸ (ℒ ⇸ 𝕍) */

type ssiPrgm = {
  nbLabels: int;            /* The labels of the CFG */
  root: Syntax.label;       /* The entry point */
  vars: Syntax.var array;   /* The original variables of the program */
  args: Syntax.var array;   /* The arguments of the program */
    /* The edges of the CFG, with the atomic instruction attached */
  instrs: (Syntax.label * Cfg.instr * Syntax.label) array;
  phis:   copyNode array;   /* φ− functions per label: ℒ ⇸ (𝕍 ⇸ (ℒ ⇸ 𝕍)) */
  sigmas: copyNode array;   /* Idem for σ− functions */
    /* The dominance tree of the labels, as a parent-children relation */
  parent: Syntax.label array;
  children: (Syntax.label list) array;
    /* Successors and predecessors for each label */
  succ: (Syntax.label list) array;
  pred: (Syntax.label list) array;
 }
```

For abstract domains we used the Apron library [12]. The Apron module is a OCaml interface to the C implementation of the library. It offers a common interface for various abstract domains (such as interval, octagon, polyhedra, etc). Its architecture separates the manager of the abstract domain from the user-interface that can for example ask the creation of a new abstract value, join or meet existing ones, perform assignment or restrict an abstract value to satisfy a linear constraint. Thanks to this library, we were able to write an analysis using the user-interface, and then test it for different abstract domains.

As for the analysis itself, it is composed of two main steps: the extraction of the transfer functions and the iteration until a fixpoint is found. The first step begins with the extraction of the blocs from the SSI form. Then for each of these blocs $\ell_1 \to \cdots \to \ell_n$ we compute its transfer function $T^\sharp_{\ell_1\to\cdots\to\ell_n}(X^+) \stackrel{\text{def}}{=} T^\sharp_{\ell_{n-1}\to\ell_n} \circ \cdots \circ T^\sharp_{\ell_1\to\ell_2}(X^+)$. After that, the analysis starts from S_0^+ (that is the abstract state where only the arguments are defined, without any constraint on them), and iterates until a fixpoint is reached. Let $F^\sharp_{\ell_1\to\cdots\to\ell_n}(X^+) \stackrel{\text{def}}{=} X^+ \cup^\sharp T^\sharp_{\ell_1\to\cdots\to\ell_n}$. At each step, the analysis applies all $F^\sharp_{\ell_1\to\cdots\to\ell_n}$ in composition. A verbose option allows to observe the change on the abstract value each time a transfer function is applied. It also highlights the transfer function that can be applied or not (due to our abstract semantics in $\mathcal{D}^+$ that only apply the transfer function of $\mathcal{D}$ if all the necessary variables are present).

The analysis, which take the SSI program as input and displays the invariant and the summary of assertions, corresponds to about 500 lines of OCaml code. The transformation to SSI, which do not include a cleanup phase (to propagate copy and remove useless variables), corresponds to the same amount.

The prototype currently does not handle convergence acceleration, in the form of a widening operator for instance. Its iterations are bounded by an arbitrary number. (If the fixpoint is not reached in this time, the resulting invariant is not correct.)

If the fixpoint is found, each assertion in the original program is evaluated with respect to this invariant to check whether they can be verified by the analysis. As invariant can be complex to read this also significantly help measure precision of the analysis. Notice that the assertion are merely conditions with only one successor and can be extracted *after* the conversion to SSI. This means that the variables used in it are correctly indexed with the accessible version of the variable, and there is no other transformation to perform. Apron already provides the necessary function to test whether an abstract value satisfies or not a constraint.

Example Let us consider the program from Figure 4a.

First, the program is put in SSI form. Then the Blocs are created. The assertion points are extracted from the CFG. A condition in the CFG is an assertion if it has only one successor. (That is why the condition $\ell_5 \to \ell_2$, is considered to be an assertion.)

```
> ./analyse -invaroct 16 example/ex.c
Transforms made (3 blocs)
-------------------------
Bloc 0 [|j_3 <- j_1|i_3 <- i_1|]  /* Parallel assignements */
       (\/ [|j_3 -_i,n 10 >= 0|]) /* Condition. Dijunction of conjunctions */
       {\/ [|11 -_i,n i_3 = 0|]}  /* Assertion. Idem */

Bloc 1 [|j_2 <- j_1|i_2 <- i_1|]
       (\/ [|10 -_i,n j_2 > 0|])
       [|j_4 <- j_2 +_i,n 1|]
       [|i_4 <- i_2 +_i,n 1|]
       {\/ [|j_4 +_i,n 1 -_i,n i_4 = 0|]}
       {\/ [|0 -_i,n 0 = 0|]}
       [|j_1 <- j_4|i_1 <- i_4|]

Bloc 2 [|j_0 <- 0|]
       [|i_0 <- 1|]
       [|j_1 <- j_0|i_1 <- i_0|]

-------------------------
```

The number of iteration has been here fixed to 16, and the abstract domain used is the octagon one. It takes 14 iterations for the analysis to reach a fixpoint without any convergence acceleration.

```
Start computation (maxIteration = 16)

Fix point reached

Stop at iteration 14
Final invariant:
[|
  i_0 -1. >=0; - i_0 +1. >=0; ...
|]
```

The invariant in the octagon domain is very verbose and we elude it here.

```
Variables: [|0> i:int; 1> i_0:int; 2> i_1:int; 3> i_2:int; 4> i_3:int;
             5> i_4:int; 6> j:int; 7> j_0:int; 8> j_1:int; 9> j_2:int;
```

```
              10> j_3:int; 11> j_4:int|]

Checking the assertion points
  l7 : Assert {\/ [|11 -_i,n i_3 = 0|]} is satisfied
  l5 : Assert {\/ [|0 -_i,n 0 = 0|]} is satisfied
  l4 : Assert {\/ [|j_4 +_i,n 1 -_i,n i_4 = 0|]} is satisfied
```

In this case, the fixpoint is reached and so the assertion points are checked.

For the polyhedra domain, the fixpoint is reached in the same amount of steps and the assertions are satisfied, but the invariant is concise.

```
> ./analyse -invarpolka 16 example/ex2.c
...
Fix point reached

Stop at iteration 14
Final invariant:
 [|
-i_1+j_1+1=0; -i_2+j_4=0; -i_2+j_2+1=0; -i_2+i_4-1=0; j_3-10=0; j_0=0;
i_3-11=0; i_0-1=0; -i_1+i_2+1>=0; -i_2+10>=0; i_2-1>=0; i_1-1>=0|]
```

In this invariant, one can observe that the relation $i = j+1$ is preserved for all matching versions (same index) of i and j.

We then also tested the implementation on a variant of the program. This variant has an argument c, which we constraint to be between 1 and 9.

```
if (c > 0 and c < 10) then
  i = 1; j = 0;
  while (j <= c) do
    i = i + 1;
    j = j + 1;
  done;
  assert (i >= c + 2);          sat
  assert (i = j + 1);           sat
  if (5 < i) then
    assert (4 < j);             sat
  else
    assert (4 >= j);            sat
  endif
else skip endif
```

With the octagon domain the fixpoint is reached in 13 iterations. However the polyhedra domain needs 22 iterations. This echoes with Astrée [8] using the more time efficient octagon domain rather than the polyhedra one. Loosing in precision by making broader approximations accelerates the convergence.

Efficiency One global invariant was the solution to avoid multiple memory-consuming invariants. However in practice we do not have one invariant at any time. Indeed, we define the abstract

semantics of path transfer function as a union of the previous invariant and the one on which the transfer functions of the edges have been applied. Thus we need two abstract values to be able to compute the abstract union. Apron does provide an in-place union of assignment: the *fold* operator [10] (also called *weak-update*). Folding the variable w in v on the abstract value X is the same as making the union of X and $\mathbb{S}^\sharp[\![v \leftarrow w]\!]X$ and then remove w from the possible variables:

$$\gamma_{\mathcal{D}} \circ \text{fold}_{(w,v)}(X) \stackrel{\text{def}}{=} \{s[v \leftarrow w]_{|_{\mathbb{V}\setminus\{w\}}} | s \in \gamma_{\mathcal{D}}(X)\} \cup \{s_{|_{\mathbb{V}\setminus\{w\}}} | s \in \gamma_{\mathcal{D}}(X)\}$$

Unfortunately this operator can only perform a weak-update for one variable at a time. We saw in the previous section that applying the union between each assignment makes us lost all precision. The fold is thus not a solution. Thus for now, the prototype keeps in memory two invariants.

4 Conclusion

The goal of this internship was to design a static analysis that combines efficiency – the analysis must be sparse with only one invariant for the whole program – with precision – it must keep properties between several variables, not just properties on single variables.

Static analyses allows to check the correct behaviour of a program without executing it, only by looking at its source code. Besides safety verification, it is also used to perform sound optimizations. For instance ABCD [5] removes unnecessary bound checks and GVN [17] removes redundant computations, both optimizations being safe with respect to the semantics of the program. These analyses are efficient, but their techniques are not easily reusable for other analyses. They are designed for specific purposes. The theory of Abstract interpretation is a framework of mathematical concepts and theorems that allow the design of analyses in a more generic way. It also provides generic proof of the soundness of such analyses, unlike the specific proofs of ABCD or GVN. The principles of this theory are versatile, the precision of the analysis (dependent of the abstract domain) or the semantics of the language can be easily changed. That is why we chose Abstract interpretation to design our analysis.

We target numerical analyses, that is analyses tracking the values of variables. Depending on the abstract domain chosen, the expressivity available for these properties varies, and can allow more precise conclusions. Numerical domains are usually classified into two categories: relational or non-relational. In a relational domain, one can express properties between several variables, for instance "x is less than y", while it is not possible in a non-relationnal one.

State-of-the-art analyses for non-relational properties are quite efficient as they benefit from representations of the program that helped them by giving more information, in a more practical way. A popular example of such representation is the SSA form [13]. It makes explicit the def-use chains, and gives for each program point, for each variable used at this program point, the last definition of the variable, using indexing of variables to ensure the unicity of this definition. Unfortunately, such representation of the program is not enough to perform efficient relational static analyses.

Our goal was to design a *sparse* static relational analysis based on Abstract Interpretation and on a convenient representation of the program, the SSI form [16]. We define an analysis as *sparse* if it computes one invariant for the whole program, instead of one per program point. For instance ABCD is sparse. Abstract Interpretation theory designs analyses with one invariant per program point, but sparser relational analysis are possible, as shown with the tool PAGAI [11]. This analyzer

computes invariant per assertion points (chosen by the end user) and at the loop head. We want to reach higher sparsity.

To compute a unique invariant, while keeping precision, we used or designed several elements for the analysis. First, we needed to have several versions of the variables, to distinguish in the invariant several moments of their lives. More concretely, the live-range of the variables is split when the information about them changes. The SSI form guarantees this property. Then, we needed to define an overlay of abstract domains that can represent partial functions. This is new in Abstract Interpretation, where abstract domains represent total functions (both in theoretical explanations [14] or concrete implementation of abstract domains [12]). In this report, we showed that such overlay does not allow an abstract domain organized on a complete lattice, which is more convenient but not mandatory for Abstract interpretation. Despite the weaker property on our overlay, we were still able to design a sparse analysis that uses it, and we were able to prove its soundness with respect to the concrete semantics of the program.

In addition, we implement a prototype of this analysis in OCaml, using the Apron library [12] to manage abstract domains, and an existing parser to extract the CFG of a program. This prototype helped evaluate the precision and efficiency of the program on concrete cases. The concrete benefits of such analysis should now be studied on real-life examples. Its efficiency is a balance between the number of variables introduced by the SSI form (at worst $O(N)$ for each variable, with N the size of the program) and the number of invariants removed with the sparsity policy (N, that is one per program point). The abstract domain plays an important role here, as the linear increase of number of variable can be a polynomial increase of time cost depending on the domain. For instance, the polyhedra domain is known [11] to perform badly with more variables. A solution, implemented in PAGAI [11], is to remove variables when they are equal to an affine combination of the other variables. Such solution is domain-specific and was thus not studied here.

In conclusion, this works proves the possibility of designing a sparse relational static analysis. It should now be compared with existing analyzer such as PAGAI, to determine its benefits or drawbacks on efficiency on concrete cases. This a future work that requires a more clever implementation than the current prototype. This prototype still lack a widening operator to guarantee convergence of the analysis. Also, the precision of the analysis should be compared to a flow-sensitive one. Although some intuition was given during the proofs of correctness, we need now to prove the importance of the SSI form to keep precision.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, 1988. ISBN 0-89791-252-7. doi: 10.1145/73560.73561. URL `http://doi.acm.org/10.1145/73560.73561`.

[2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. ISBN 0-521-58388-8.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[4] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2014.

[5] R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI 2000*, pages 321–333. ACM, 2000.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL 1979*, pages 269–282. ACM Press, 1979.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. LNCS #3444, page 21. Springer, 2005. doi: 10.1007/b107380. URL `https://hal.archives-ouvertes.fr/hal-00084293`.

[9] Edsger W. Dijkstra. On the reliability of programs. circulated privately. URL `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF`.

[10] D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and S. Sagiv. Numeric domains with summarized dimensions. In *TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2004.

[11] J. Henry, D. Monniaux, and M. Moy. PAGAI: A path sensitive static analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

[12] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV 2009*, volume 5643, pages 661–667. Springer, 2009.

[13] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[14] A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017.

[15] Antoine Miné. *Weakly relational numerical abstract domains*. PhD thesis, Ecole Polytechnique X, 2004.

[16] F. Pereira and F. Rastello. Static Single Information form. Chapter 11 in the SSA-book, 2018. URL `http://ssabook.gforge.inria.fr/latest/book.pdf`.

[17] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.