Compiling with Abstract Interpretation (with appendices)

DORIAN LESBRE, Université Paris-Saclay, CEA, List, France
MATTHIEU LEMERRE, Université Paris-Saclay, CEA, List, France

Rewriting and static analyses are mutually beneficial techniques: program transformations change the intensional aspects of the program, and can thus improve analysis precision, while some efficient transformations are enabled by specific knowledge of some program invariants. Despite the strong interaction between these techniques, they are usually considered distinct. In this paper, we demonstrate that we can turn abstract interpreters into compilers, using a simple free algebra over the standard signature of abstract domains. Functor domains correspond to compiler passes, for which soundness is translated to a proof of forward simulation, and completeness to backward simulation. We achieve translation to SSA using an abstract domain with a non-standard SSA signature. Incorporating such an SSA translation to an abstract interpreter improves its precision; in particular we show that an SSA-based non-relational domain is always more precise than a standard non-relational domain for similar time and memory complexity. Moreover, such a domain allows recovering from precision losses that occur when analyzing low-level machine code instead of source code. These results help implement analyses or compilation passes where symbolic and semantic methods simultaneously refine each other, and improves precision when compared to doing the passes in sequence.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Formal software verification*; • **Theory of computation** → **Program analysis**; **Program verification**; **Abstraction**; *Equational logic and rewriting*.

Additional Key Words and Phrases: Compilers, Abstract Interpretation, Static Single Assignment (SSA).

1 INTRODUCTION

Syntactic transformations, also called symbolic methods [Miné 2006], are an essential tool to improve the precision of abstract domains. For instance, compiled code usually executes sequences of small instructions over temporary variables. Analyzing such code one instruction at a time leads to precision losses compared to source analysis because the analysis lacks context. Logozzo and Fähndrich [2008] call this the *limited code window* problem, and show that solving it requires the use of syntactic term manipulation. Moreover, when the compilation target is machine code, a precise analysis can only be obtained if it reconstructs simple conditions from the machine semantics (e.g. it is more precise to analyze $x > y$ than an instruction sequence involving a xor between the overflow and signed flag) [Balakrishnan and Reps 2010; Djoudi et al. 2016]. Outside of compiled code, many authors have used syntactic transformations to improve the precision of abstract domains at a low cost [Boillot and Feret 2023; Gange et al. 2016; Lemerre 2023; Miné 2006].

However, many such syntactic transformations benefit from a prior semantic analysis. For example, rewriting $x \mid 4$ into $x + 4$ (where $\mid$ means bitwise or, as in C) holds only if the second bit of x always has value 0. Common examples include dead code elimination or constant propagation, which requires a static analysis to identify constant boolean conditions or expressions. In general, the essence of compiler optimization is to symbolically rewrite the program using semantic invariants computed in a prior analysis pass [Cousot and Cousot 2002].

Thus, syntactic transformations both benefit from, and help improve, semantic analyses. Therefore, applying each in different passes raises the phase ordering issue, and the best precision is obtained by performing both simultaneously [Click and Cooper 1995; Cousot and Cousot 1979]. A notable application of such a simultaneous analysis is machine code analysis, which often fails to terminate due to excessive imprecision. Here, syntactic rewrites enabled by found invariants are useful to undo compiler transformations and recover a representation which is easier to analyze.

Authors' addresses: Dorian Lesbre, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, dorian.lesbre@cea.fr; Matthieu Lemerre, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, matthieu.lemerre@cea.fr.

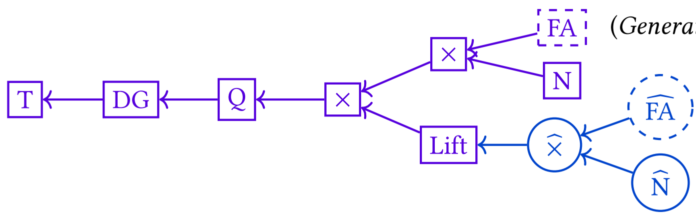


Fig. 1. Functor chain of our analysis: with the final domain on the left and base domains on the right. Arrows point from arguments to the functor that uses them. Square purple nodes are Imp domains, and circular blue nodes are SSA domains.

Abstract interpretation [Cousot and Cousot 1977] provides a generic method for combining analysis passes [Cousot and Cousot 1979] by encoding them as operations over an abstract domain with a common interface. The classical interface requires a join, inclusion, widening, and analysis of statements. Some syntactic transformations have been implemented as abstract domains under this interface (e.g. [Boillot and Feret 2023; Chang and Leino 2005; Gange et al. 2016; Gulwani and Necula 2004; Kildall 1973; Miné 2006]), but until recently, such domains could not produce recursive terms. This limited the syntactic transformations that an abstract interpreter could perform to local transformations, unlike the usual syntactic translation method used in compilation. This restriction was lifted in Lemerre [2023], where an abstract interpreter was used to perform a complex syntactic transformation, SSA translation, using simple abstract domains. The resulting algorithm is arguably simpler than the standard methods [Aycock and Horspool 2000; Brandis and Mössenböck 1994; Braun et al. 2013; Cytron et al. 1991; Sreedhar and Gao 1995].

Problem. While the work of Lemerre [2023] hinted that some compilation techniques could be performed by abstract interpretation, it left many questions unanswered, such as:

- Can compilation-by-abstract-interpretation generalize to transformations other than SSA translation, i.e. to other input or output languages? In particular, Lemerre [2023] does not perform any control-flow transformation other than dead-code elimination, and maintains a 1-to-1 correspondence between source and target locations.
- How can compilation-by-abstract-interpretation interact with semantic analyses in practice? Lemerre [2023] proposed using a regular reduced product [Cousot and Cousot 1979]. However, one might prefer other generic domain combinations [Cousot and Cousot 1979; Venet 1996], or a more specialized combination. This is especially true if one wants to perform the analysis on the SSA translation instead of the source program.
- What are the cost and precision advantages of compilation-by-abstract-interpretation (and in particular, SSA translation) when used to improve the precision of a static analysis? Do standard compilation techniques apply to this framework, such as rewriting terms to improve global value numbering? How would they impact precision? Lemerre [2023] only stated that his symbolic expression abstract domain has a low computational complexity.

Contributions. Our overall contribution is to provide answers to the above questions by presenting an abstract interpreter design. In this design, syntactic transformations, seen semantically as abstract domains, can be combined with semantic analyses so that both run simultaneously and help each other. We summarize this as “compiling with abstract interpretation”, which not only means performing the compilation using abstract interpretation, but also to simultaneously use the program transformations as a means to improve the abstract interpretation.

Figure 1 presents the overall design of our abstract interpreter as a collection of abstract domains that are all executed simultaneously. More specifically:

- Section 4 (FA domain) explains how we can generate imperative programs by abstract interpretation. We use a domain of free algebras over the classical abstract domain signature, dually interpreted as a set of states and as a transition system. The result of this free-algebra analysis is a program graph which is isomorphic to the source program graph;
- Section 5 (Q, DG, T, and $\times$ functors) shows that functor domains[1], commonly used in abstract interpretation to transform analyses, can be viewed as compiler transformation passes. The setting is related to the tagless-final staged interpreters of Carette et al. [2009]. We show in particular that these passes preserve semantics: functor soundness implies a forward simulation and completeness implies a backward simulation with the same relation;
- Section 6 (circular hatted blue nodes) shows why static analysis of SSA programs requires an abstract domain with a different signature (and free algebra $\widehat{\mathrm{FA}}$) than the usual signature for imperative programs;
- Section 7 (Lift functor) implements an SSA-translation compiler pass. This is achieved by a functor that lifts an abstract domain with SSA domain signature to an abstract domain with an imperative domain signature;
- Section 8 ($\widehat{\mathrm{N}}$ domain), presents a "non-relational" analysis for the SSA signature, based on the combination of symbolic expressions [Lemerre 2023] with single-value abstractions[2], such as intervals. We prove that lifting this non-relational SSA domain to an imperative domain is always more precise than the usual imperative analysis while incurring only a constant overhead. Moreover, it is the first known domain we know to have the strong relative completeness [Logozzo and Fähndrich 2008] property: it allows analyzing a compiled program with the same precision as the original (solving the limited code window problem);
- Section 9 evaluates our approach by describing Codex, a static analyzer, based on this technique. It can handle both C and machine code and has been successfully used on industrial code bases [Nicole et al. 2021, 2022]. The multiple simultaneous translations (both at the source and SSA level) improve the precision and simplify the design of the analyzer in practice. We also present a simplified analyzer, TAI, that closely corresponds to Figure 1. Using it, we compare SSA numerical analysis to a standard non-relational analysis in terms of performance and precision. Both appear in the open source software artifact accompanying this article [Lesbre and Lemerre 2024a].

This is the full version (with appendices) of Lesbre and Lemerre [2024b].

2 A SMALL EXAMPLE

The top of Figure 2 presents a small example program, both in C code and in Imp (the simple input language of our analysis, Section 3). It consists of a simple loop, with a branching path testing whether the loop invariant holds. Here, the `F` macro stands for any complex numeric operation, changing it to another expression should work just as well. The graph distinguishes between conditional edges and assignment edges.

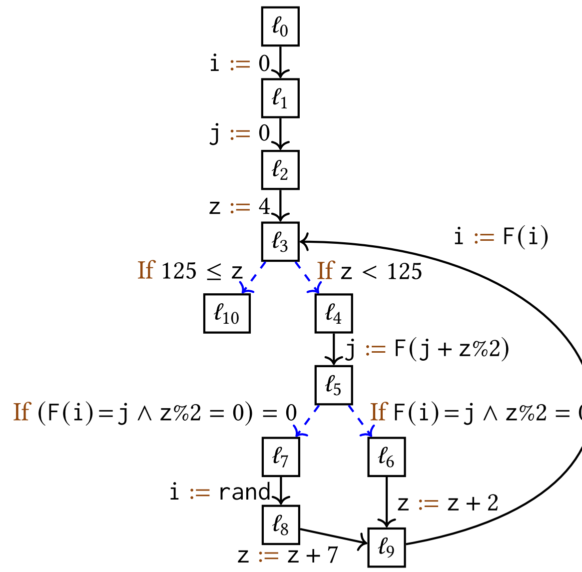
Proving the invariant, and optimizing the dead code of the else branch away is fairly involved. It is not optimized by modern compilers like GCC or LLVM. Doing so requires simultaneously performing numerical analysis (to learn that `z` is even), some syntactic transformations (to learn that `F(j + z%2)` is `F(j)`), optimistic global value numbering (to learn that $i = j$), and dead code elimination so that no analysis takes the else branch (which breaks all those properties). Performing all analyses in one pass is crucial, as no analysis is strong enough to prove the full invariant alone.

The rest of the figure displays the result of analyzing this program with various domains presented in this paper. Using the free algebra domain from Section 4 yields a renaming of the initial graph by

[1] also called cofibered domains [Venet 1996]

[2] called basis in Miné [2004] or partitioned lattice per variable in Rastello and Bouchez Tichadou [2022]

```c
#define F(x) ((x) * (x) + 1)
void example(int rand) {    // ℓ0
  int i = 0;                // ℓ1
  int j = 0;                // ℓ2
  int z = 4;                // ℓ3
  while(z < 125){           // ℓ4
    // Invariant: i = j ∧ z%2 = 0
    j = F(j + z % 2);       // ℓ5
    if(F(i) == j && z%2 == 0){//ℓ6
      z += 2;
    } else { // Dead code   // ℓ7
      i = rand;             // ℓ8
      z += 7;
    }                       // ℓ9
    i = F(i);
  }                         // ℓ10
}
```



(a) Example input program in C (left) and translated to Imp representation (right).

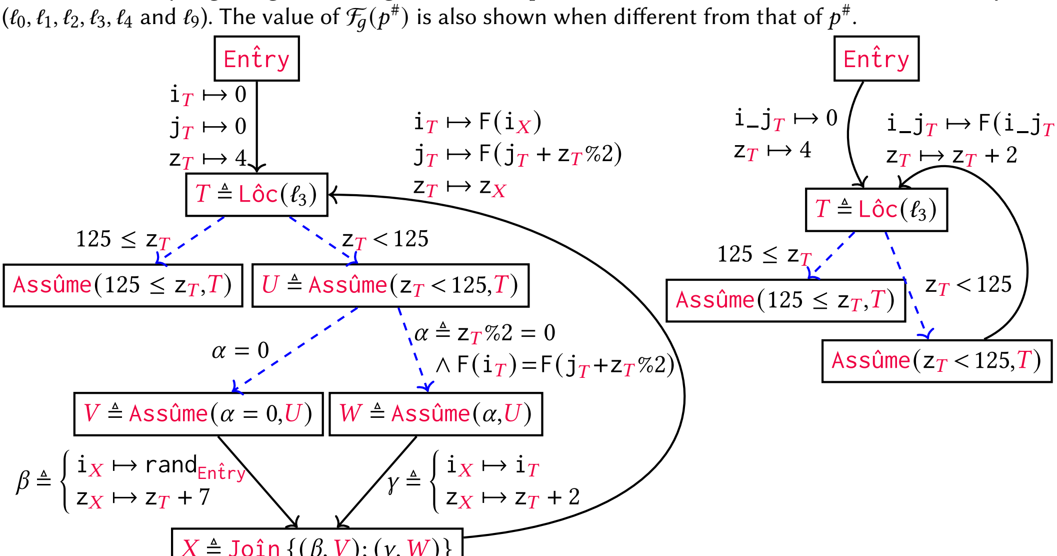
$$p^\#(\ell_0) \;=\; \texttt{Entry} \qquad\qquad p^\#(\ell_1) \;=\; \texttt{Apply}(\texttt{i} := 0,\ \texttt{Entry})$$

$$p^\#(\ell_2) \;=\; \texttt{Apply}(\texttt{j} := 0,\ \texttt{Apply}(\texttt{i} := 0,\ \texttt{Entry})) \qquad\qquad p^\#(\ell_4) \;=\; \texttt{Apply}(\text{If}\ \texttt{z} < 125,\ \texttt{Loc}(\ell_3))$$

$$p^\#(\ell_3) \;=\; \texttt{Loc}(\ell_3) \qquad \mathcal{F}_g(p^\#)(\ell_3) \;=\; \texttt{Join}\left\{\texttt{Apply}(\texttt{z} := 4,\ p^\#(\ell_2));\ \texttt{Apply}(\texttt{i} := \texttt{F}(\texttt{i}),\ p^\#(\ell_9))\right\}$$

$$p^\#(\ell_9) \;=\; \texttt{Join}\left\{\texttt{Apply}(\texttt{z} := \texttt{z} + 7,\ p^\#(\ell_8));\ \texttt{Apply}(\texttt{z} := \texttt{z} + 2,\ p^\#(\ell_6))\right\}$$

(b) Result of analyzing using the free algebra domain ($p^\# \triangleq \text{analyse}(\text{FA})$, Section 4) for a selection of points ($\ell_0, \ell_1, \ell_2, \ell_3, \ell_4$ and ℓ_9). The value of $\mathcal{F}_g(p^\#)$ is also shown when different from that of $p^\#$.



(c) Analysis with bare SSA translation (left: $\text{Lift}(\widehat{\text{FA}})$, Sections 6 and 7), and translation combined with numerical analysis and simple rewrites (right: $\text{Lift}(\hat{\text{Q}}(\widehat{\text{FA}}\widehat{\times}\widehat{\text{N}}))$). For legibility, we use T, U, V, W, X as short names for terms and α, β, γ as short names for edges

Fig. 2. Example input program and results of different analysis.

$$z \in \mathbb{Z} \text{ (Integers)} \qquad x \in \mathbb{X} \text{ (Variables)} \qquad \sigma \in \Sigma \triangleq \mathbb{X} \to \mathbb{Z} \text{ (Stores)} \qquad \ell \in \mathbb{L} \text{ (Locations)}$$

$$\mathbb{S} \triangleq \mathbb{L} \times \Sigma \qquad \diamond \in \{+; -; \times; /; \%; =; \neq; <; \leq; \land; \lor\} \qquad e \in \mathbb{E} \triangleq z \mid x \mid e \diamond e \mid e\,?\,e : e \text{ (Expressions)}$$

$$R \in \mathbb{R} \triangleq \mathrm{If}\ e \mid x := e \text{ (Syntactic relations)} \qquad \mathcal{G} \in \mathbb{G} \triangleq \mathbb{L} \times \mathbb{R} \times \mathbb{L} \to \{0; 1\} \text{ (Program graph)}$$

$$\mathcal{E}[\![\cdot]\!] \in \mathbb{E} \to \Sigma \to \mathbb{Z}_\perp \qquad \mathcal{R}[\![\cdot]\!] \in \mathbb{R} \to (\Sigma \times \Sigma \to \{0; 1\}) \qquad \to_{\mathcal{G}} \in \mathbb{S} \times \mathbb{S} \to \{0; 1\}$$

$$\mathcal{E}[\![z]\!](\sigma) \triangleq z \qquad \mathcal{E}[\![x]\!](\sigma) \triangleq \sigma(x) \qquad \mathcal{E}[\![e_{\text{cond}}\,?\,e_{\text{true}} : e_{\text{false}}]\!](\sigma) \triangleq \begin{cases} \perp & \text{if } \mathcal{E}[\![e_{\text{cond}}]\!](\sigma) = \perp \\ \mathcal{E}[\![e_{\text{true}}]\!](\sigma) & \text{if } \mathcal{E}[\![e_{\text{cond}}]\!](\sigma) \neq 0 \\ \mathcal{E}[\![e_{\text{false}}]\!](\sigma) & \text{if } \mathcal{E}[\![e_{\text{cond}}]\!](\sigma) = 0 \end{cases}$$

$$\mathcal{E}[\![e_\ell \diamond e_r]\!](\sigma) \triangleq \begin{cases} \perp & \text{if } (\diamond \in \{/; \%\} \land \mathcal{E}[\![e_r]\!](\sigma) = 0) \lor (\perp \in \{\mathcal{E}[\![e_l]\!](\sigma); \mathcal{E}[\![e_r]\!](\sigma)\}) \\ \mathcal{E}[\![e_\ell]\!](\sigma) \diamond \mathcal{E}[\![e_r]\!](\sigma) & \text{otherwise} \end{cases}$$

$$\mathcal{R}[\![\mathrm{If}\ e]\!](\sigma, \sigma') \triangleq \sigma = \sigma' \land \mathcal{E}[\![e]\!](\sigma) \notin \{0; \perp\} \qquad \mathcal{R}[\![x := e]\!](\sigma, \sigma') \triangleq \mathcal{E}[\![e]\!](\sigma) \neq \perp \land \sigma' = \sigma[x \mapsto \mathcal{E}[\![e]\!](\sigma)]$$

$$(\ell, \sigma) \to_{\mathcal{G}} (\ell', \sigma') \triangleq \exists R \in \mathbb{R}, \mathcal{G}(\ell, R, \ell') \land \mathcal{R}[\![R]\!](\sigma, \sigma')$$

Fig. 3. Imp syntax (top) and semantics (bottom).

Theorem 4.1, so we only show how a few select points are renamed in Figure 2b. Finally, Figure 2c shows the result of our SSA translations, both as a standalone analysis (left), and combined with other analysis that prove the invariant (right). The first one closely resembles the intermediate representation that a compiler would generate, although our SSA variant deviates slightly from typical SSA (ϕ functions replaced by join nodes). Note how binding edges only appear before joins.

3 NOTATIONS AND BACKGROUND

This section presents the background notions used in this paper, with their associated notations. Specifically, it describes common notations; introduces a small example language: Imp; and presents the signature of our abstract domains along with an example numeric domain.

3.1 Notations

We write $X_\perp \triangleq X \cup \{\perp\}$ for the set X with an extra element $\perp \notin X$. We use $\mathcal{P}(X) \triangleq \{Y \mid Y \subseteq X\}$ for the set of subsets of X and $\mathcal{P}_f(X) \triangleq \{Y \in \mathcal{P}(X) \mid Y \text{ finite}\}$ for the set of finite subsets of X.

Let $X \rightharpoonup Y$ be the set of partial functions from X to Y and $X \to Y$ the set of total functions from X to Y (their domain is exactly X). Functions are seen as sets of bindings $x \mapsto y$, replacing curly braces {} with brackets []. So $[z \mapsto z + 1 \mid z \in \mathbb{Z}]$ is the successor function, and $[0 \mapsto 1; 1 \mapsto 2]$ is a function defined only on 0 and 1. We use the short notation $[x \in X \mapsto f(x)]$ for $[x \mapsto f(x) \mid x \in X]$.

For a function f, we denote its domain by $\mathtt{dom}\, f$ and its image by $\mathtt{img}\, f$. We denote the image of x under f by $f(x)$. We use $f[g] \triangleq [v \mapsto f(v) \mid v \in \mathtt{dom}\, f \setminus \mathtt{dom}\, g] \cup g$ for the function f updated with all bindings of g. Often g will be a single binding $[x \mapsto y]$ which leads to the notation: $f\,[x \mapsto y]$.

We view relations R as multi-variable predicates $R \in X \times Y \to \{0; 1\}$ (also called indicative functions). We often write then as logic formulas using the usual logical operators ($=$, $\land$, $\lor$).

3.2 Imp syntax and semantics

We use a small imperative programming language named Imp, defined in Figure 3. Program expressions $e \in \mathbb{E}$ are composed of integers ($\mathbb{Z}$), variables ($\mathbb{X}$), binary operators ($\diamond$), and a ternary if-then-else operator. A program $\mathcal{G} \in \mathbb{G}$ is a directed graph, with *location* identifiers $\ell \in \mathbb{L}$ as vertices. Its edges are labelled by syntactic relations $R \in \mathbb{R}$, which are either guard conditions or single variable assignments. $\mathbb{L}$ is finite. This language supports loops (looping path in the graph), arbitrary gotos, but not function calls, as it has no memory and thus no call stack.

$\Sigma^\#$(set of abstract states)	$\gamma \in \Sigma^\#_\bot \to \mathcal{P}(\Sigma)$	
$entry \in \Sigma^\#$	$\Sigma \subseteq \gamma(entry)$	(EntrySound)
$apply \in \mathbb{R} \times \Sigma^\# \to \Sigma^\#_\bot$	$\left\{\sigma' \in \Sigma \,\middle|\, \exists \sigma \in \gamma(s^\#), \mathcal{R}[\![R]\!](\sigma, \sigma')\right\} \subseteq \gamma(apply(R, s^\#))$	(ApplySound)
$join \in \mathcal{P}_f(\Sigma^\#) \to \Sigma^\#$	$\bigcup_{s^\# \in S^\#} \gamma(s^\#) \subseteq \gamma(join(S^\#))$	(JoinSound)
$widen \in W \times \Sigma^\# \times \Sigma^\# \to \Sigma^\#$	$\gamma(s^\#) \cup \gamma(t^\#) \subseteq \gamma(widen(\ell, s^\#, t^\#))$	(WSound)

$$
\begin{array}{rcll}
\mathcal{F}_g & \in & (\mathbb{L} \rightharpoonup D.\Sigma^\#) \to (\mathbb{L} \rightharpoonup D.\Sigma^\#) & (D \text{ is an } \textsc{Imp} \text{ domain})\\
\mathcal{F}_g(p^\#) & \triangleq & \ell_0 \mapsto D.entry & \\
& | & \ell \mapsto D.join \left\{ D.apply(R, p^\#(\ell')) \,\middle|\, \begin{array}{r} \text{for } \ell' \in \mathbb{L}, R \in \mathbb{R} :\ \mathcal{G}(\ell', R, \ell) \wedge \ell' \in \operatorname{dom} p^\# \\ \wedge\, D.apply(R, p^\#(\ell')) \neq \bot \end{array} \right\} &
\end{array}
$$

$$
\begin{array}{rcll}
\nabla_W & \in & (\mathbb{L} \rightharpoonup D.\Sigma^\#) \to (\mathbb{L} \rightharpoonup D.\Sigma^\#) \to (\mathbb{L} \rightharpoonup D.\Sigma^\#) & \\
p^\# \nabla_W q^\# & \triangleq & \ell \mapsto D.widen(\ell, p^\#(\ell), q^\#(\ell)) & \text{if } \ell \in W \text{ (set of widening points)}\\
& | & \ell \mapsto q^\#(\ell) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{analyse(D)} & \in & \mathbb{L} \rightharpoonup D.\Sigma^\# \\
\texttt{analyse(D)} & \triangleq & \operatorname{lfp}\left[p^\# \in (\mathbb{L} \rightharpoonup D.\Sigma^\#) \mapsto p^\# \nabla_W \mathcal{F}_g(p^\#)\right]
\end{array}
$$

Fig. 4. Imp abstract domain signature (top left) properties (top right) and analysis (bottom).

Semantics. $\mathcal{E}[\![e]\!](\sigma)$ evaluates the expression e to an integer, using the *store* σ for variable values. Arithmetic operators are standard, using euclidean division and modulo. Divisions by 0 interrupt the program (i.e., return $\bot$). Comparison operators are defined to return 1 when true and 0 otherwise. Boolean operators are non-lazy, and consider any non-zero value as true.

$\mathcal{R}[\![\cdot]\!]$ transforms a syntactic relation $\mathbb{R}$ into a mathematical relation on *program states* (pairs of locations and stores). Guards do not change the store but block execution when the condition evaluates to 0 or $\bot$; assignments $x := e$ change the value of x to the evaluation of e in the input state, leaving the other variables unchanged.

The semantics of a program $\mathcal{G} \in \mathbb{G}$ is given as a transition relation $(\ell, \sigma) \to_{\mathcal{G}} (\ell', \sigma')$ between states. There is a transition between two states if there exists an edge between their locations in $\mathcal{G}$ such that the edge's relation is verified by the stores. This is not necessarily deterministic, as a state might have multiple valid successors. We write $\to^*_{\mathcal{G}}$ for the reflexive transitive closure of $\to_{\mathcal{G}}$. We assume that all outgoing edges from a location are labelled by different relations.[3]

Finally, programs have an initial location $\ell_0 \in \mathbb{L}$, which has no predecessors. We say that a state $(\ell, \sigma) \in \mathbb{S}$ is *reachable* when there is a $\sigma_0 \in \Sigma$ such that $(\ell_0, \sigma_0) \to^*_{\mathcal{G}} (\ell, \sigma)$.

3.3 Abstract interpretation of Imp

Abstract domains [Cousot and Cousot 1977] are algebraic structures whose signature is given at the top of Figure 4. They contain a set of abstract states $\Sigma^\#$ whose meaning is given by a concretization function γ, mapping an abstract state to a set of states. This function is only used in proofs and needs not to be computable. To lighten notations, we lift this concretization to $\Sigma^\#_\bot$ and assume that $\gamma(\bot) = \emptyset$ for all domains.

The domain operations (top left of Figure 4) must be computable. *entry* is the program entry point. $apply(R, s^\#)$ represents all the states that can be obtained from $s^\#$ after applying a relation R. *join* computes an over-approximation of finite union, and is used at merge points in the control flow. *widen* is a widening operation, used to ensure termination of the analysis. For the sake of

[3]This allows uniquely identifying a program path from the trace of applied relations. It is true on deterministic programs, but can also be enforced on non-deterministic ones (using rewrites $e \mapsto e \times 1$ if needed). This simplifies Theorem 4.1.

$$\mathrm{N}.\Sigma^\# \triangleq \mathbb{X} \to \mathbb{Z}^\# \qquad \mathrm{N}.\mathit{entry} \triangleq [x \in \mathbb{X} \mapsto (-\infty, +\infty)]$$

$$\mathrm{N}.\mathit{join}(\{\sigma_0^\#; \ldots; \sigma_n^\#\}) \triangleq \left[x \in \mathbb{X} \mapsto \sigma_0^\#(x) \sqcup_{\mathbb{Z}^\#} \ldots \sqcup_{\mathbb{Z}^\#} \sigma_n^\#(x)\right]$$

$$\mathrm{N}.\mathit{apply}(x := e,\ \sigma^\#) \triangleq \sigma^\#\left[x \mapsto \vec{\mathcal{E}}[\![e]\!](\sigma^\#)\right] \qquad \mathrm{N}.\mathit{apply}(\mathrm{If}\ e,\ \sigma^\#) \triangleq \sigma^\# \leftarrow e \neq 0$$

$$\mathrm{N}.\mathit{widen}(\_,\ \sigma_0^\#,\ \sigma_1^\#) \triangleq \left[x \in \mathbb{X} \mapsto \sigma_0^\#(x)\ \nabla_{\mathbb{Z}^\#}\ \sigma_1^\#(x)\right] \qquad \mathrm{N}.\gamma(\sigma^\#) \triangleq \left\{\sigma \,\middle|\, \forall x,\ \sigma(x) \in \gamma_{\mathbb{Z}^\#}(\sigma^\#(x))\right\}$$

Fig. 5. The numeric Imp abstract domain (N).

simplicity, we do not discuss much about termination here. We only require that widening chains, i.e. repeated applications of widening operations, eventually stabilize. Thus, we do not need an order relation in our domain signature, as we can use equality directly.[4] Another non-standard point is the need to pass a location (the widening point) as an argument to *widen*. This is required to ensure the convergence of abstract domains consisting in recursive terms [Lemerre 2023] by giving unique names to those terms. One can view domains implementing this signature as records whose fields are functions. We use the notation D.*apply* to denote the *apply* function of the domain D.

A domain is *sound* when its operations meet the soundness hypotheses given at the top right of Figure 4. It is *complete* when its operations meet the converse hypotheses, with set inclusion reversed.

Abstract interpretation of an Imp domain D is done via a standard dataflow analysis [Cousot and Cousot 1977], presented in the bottom of Figure 4. $\mathcal{F}_g$ joins at each point the *apply*s of the predecessors. It is undefined at points where the set in D.*join* is empty. We write W the *set of widening points*, i.e. points where widenings are performed. Any subset of $\mathbb{L}$ with at least one point on every cycle in the control-flow graph is a valid choice for W. Bourdoncle [1993] gives a method to compute a reasonably small W (set of loop heads). The final result of our analysis is given by `analyse`(D)[5]. It is a partial function mapping locations $\mathbb{L}$ to our domain state D.$\Sigma^\#$. It is undefined on locations determined unreachable. It is computed as least fixed-point (lfp) of the widening of $\mathcal{F}_g$.

The most precise domain we can define in this setting is the collecting semantics domain, denoted CS. The concretization γ of CS is the identity. It is not computable, but helps quantify the abstraction loss suffered by other domains.

$$\mathrm{CS}.\Sigma^\# \triangleq \mathcal{P}(\Sigma) \qquad \mathrm{CS}.\gamma(s^\#) \triangleq s^\# \qquad \mathrm{CS}.\mathit{entry} \triangleq \Sigma \qquad \mathrm{CS}.\mathit{join}(S^\#) \triangleq \textstyle\bigcup_{s^\# \in S^\#} s^\#$$

$$\mathrm{CS}.\mathit{apply}(R,\ s^\#) \triangleq \left\{\sigma \in \Sigma \,\middle|\, \exists\, \sigma' \in s^\#,\ \mathcal{R}[\![R]\!](\sigma', \sigma)\right\} \qquad \mathrm{CS}.\mathit{widen}(\_,\ s^\#,\ t^\#) \triangleq s^\# \cup t^\#$$

3.4 Example: non-relational numeric domain

A classical example domain is built on top of a single-value abstraction like intervals. They represent set of integers by pairs $[m : M] \in \mathbb{Z}^\# \triangleq \mathbb{Z} \cup \{-\infty\} \times \mathbb{Z} \cup \{+\infty\}$ with $m \leqslant M$, concretized by $\gamma_{\mathbb{Z}^\#}([m : M]) \triangleq \{z \in \mathbb{Z} \mid m \leqslant z \leqslant M\}$. We denote $\sqcup_{\mathbb{Z}^\#}$, $\sqcap_{\mathbb{Z}^\#}$, $\subseteq_{\mathbb{Z}^\#}$ and $\nabla_{\mathbb{Z}^\#}$ the usual join, meet, subset and widening operators on intervals [Cousot and Cousot 1977].

For each expression construct f of arity n we let $\vec{f} \in \mathbb{Z}^{\#n} \to \mathbb{Z}^\#$ be the associated *forward transfer function* (which yields an abstraction of f given abstractions of its arguments) and $\overleftarrow{f} \in \mathbb{Z}^{\#n} \times \mathbb{Z}^\# \to \mathbb{Z}^{\#n}$ the *backward transfer function* (which refines the abstractions of the arguments given an abstraction of the result of f).

Using these, we can define our first Imp domain: *the numeric domain*, denoted N. It is presented in Figure 5, where $\vec{\mathcal{E}}[\![\cdot]\!] \in \mathbb{E} \to (\mathbb{X} \to \mathbb{Z}^\#) \to \mathbb{Z}^\#$ evaluates the expression (similarly to $\mathcal{E}[\![\cdot]\!]$) in $\mathbb{Z}^\#$ using the forward transfer functions; and $\sigma^\# \leftarrow e \neq 0$ symbolizes refining $\sigma^\#$ using the backward transfer functions and the information that $e \neq 0$ (using an algorithm similar to the HC4 constraint propagation [Benhamou et al. 1999]). This numerical domain is sound.

[4]For more details on this, see Appendix B.

[5]The domain D is implicit in $\mathcal{F}_g$ and ∇_W, as it can be deduced from the analysis being considered, but explicit in `analyse`.

$$\mathtt{s}^\# \in \mathrm{FA}.\Sigma^\# \triangleq \mathtt{Entry} \mid \mathtt{Apply}(R,\ \mathtt{s}^\#) \mid \mathtt{Join}(\mathtt{S}^\#) \mid \mathtt{Loc}(\ell) \ \text{(Algebraic locations)}$$
$$\text{(where } R \in \mathbb{R} \text{ is a syntactic program relation, and } \mathtt{S}^\# \in \mathcal{P}_f(\mathrm{FA}.\Sigma^\#))$$

$$\begin{aligned} \mathrm{FA}.\mathit{entry} &\triangleq \mathtt{Entry} \\ \mathrm{FA}.\mathit{apply}(R,\ \mathtt{s}^\#) &\triangleq \mathtt{Apply}(R,\ \mathtt{s}^\#) \\ \mathrm{FA}.\mathit{widen}(\ell,\ \_,\ \_) &\triangleq \mathtt{Loc}(\ell) \end{aligned} \qquad \mathrm{FA}.\mathit{join}(\mathtt{S}^\#) \triangleq \begin{cases} \bot & \text{if } \mathtt{S}^\# \text{ is empty} \\ \mathtt{s}^\# & \text{if } \mathtt{S}^\# \text{ is a singleton } \{\mathtt{s}^\#\} \\ \mathtt{Join}(\mathtt{S}^\#) & \text{otherwise} \end{cases}$$

$$\mathrm{FA}.\gamma(\mathtt{Entry}) \triangleq \Sigma \qquad \mathrm{FA}.\gamma(\mathtt{Apply}(R,\ \mathtt{s}^\#)) \triangleq \left\{\sigma \in \Sigma \,\middle|\, \exists\, \sigma' \in \mathrm{FA}.\gamma(\mathtt{s}^\#),\ \mathcal{R}[\![R]\!]\,(\sigma', \sigma)\right\}$$

$$\mathrm{FA}.\gamma(\mathtt{Join}(\mathtt{S}^\#)) \triangleq \bigcup_{\mathtt{s}^\# \in \mathtt{S}^\#} \mathrm{FA}.\gamma(\mathtt{s}^\#) \qquad \mathrm{FA}.\gamma(\mathtt{Loc}(\ell)) \triangleq \Sigma$$

$$\textsc{TApply}\ \frac{}{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathtt{Apply}(R,\ \mathtt{s}^\#)} \qquad \textsc{TJoin}\ \frac{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathtt{t}^\# \qquad \mathtt{t}^\# \in \mathtt{S}^\#}{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathtt{Join}(\mathtt{S}^\#)} \qquad \textsc{TSelf}\ \frac{\mathcal{F}_g(p^\#)(\ell) = \mathtt{Join}(\mathtt{S}^\#) \qquad \mathtt{Loc}(\ell) \in \mathtt{S}^\#}{\mathtt{Loc}(\ell) \overset{\text{If } 1}{\longmapsto}_\# \mathtt{Loc}(\ell)}$$

$$\textsc{TLoc}\ \frac{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathcal{F}_g(p^\#)(\ell)}{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathtt{Loc}(\ell)} \qquad \textsc{VBase}\ \frac{\mathtt{s}^\# \in \operatorname{img} p^\#}{V(\mathtt{s}^\#)} \qquad \textsc{VRec}\ \frac{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathtt{t}^\# \qquad V(\mathtt{t}^\#)}{V(\mathtt{s}^\#)} \qquad \textsc{GraphGen}\ \frac{\mathtt{s}^\# \overset{R}{\longmapsto}_\# \mathtt{t}^\# \qquad V(\mathtt{t}^\#)}{\mathcal{G}_{p^\#}(\mathtt{s}^\#,\ R,\ \mathtt{t}^\#)}$$

Fig. 6. The free algebra Imp abstract domain (FA) (top) and rules for generating Imp programs (bottom).

We use intervals here as they are well-known and easy to define, but this abstraction can very easily be switched to other single-value abstractions, such as congruence [Miné 2017], bitwise/tristate [Michel and Hentenryck 2012; Miné 2012; Vishwanathan et al. 2022], or any product of these abstractions. For our running example (Figure 2), intervals alone cannot prove that `z` is even, thus we need another abstraction (bitwise or congruence).

4 FREE ALGEBRA OF THE DOMAIN SIGNATURE

We now explain how a free algebra over the domain signature of Figure 4 can be used to exactly recover the source program as a standard abstract interpretation. This is achieved thanks to the dual interpretation of this abstract domain: the classical interpretation as a set of states, and the other as a new program graph whose vertices are elements of the free algebra, and whose edges are given by the `Apply` terms. In this section, the generated program is isomorphic to the source. We will add transformations in Section 5.

4.1 Definition

The *free algebra Imp domain*, denoted FA, is presented in Figure 6. Its elements, called *algebraic locations* are cyclic terms in the free algebra of the abstract domain signature (Figure 4). Thus, the domain operations: FA.*entry*, FA.*apply* and FA.*join* just create terms using the `Entry`[6], `Apply` and `Join` function symbols. The other constructor, `Loc`, represents widening points, which also corresponds to recursion variables in the control-flow graph viewed as a cyclic term graph [Ariola and Klop 1996], as they are both used to break cycles. Thus, the widening operation FA.*widen* simply returns the relevant `Loc`, using the original program location name as a way to give a deterministic name to recursion variables and allow the analysis to terminate [Lemerre 2023]. Figure 2b shows the value of $p^\#$ obtained from performing an analysis using this domain on a few select points.

[6]We denote the Imp domain interface in *lowercase purple italics*, its implementations in the same style but prefixed with the domain short name, and the terms of its free algebra in `Capitalized Orange Typewriter Font`.

4.2 Concretization as a set of states

These algebraic locations can be interpreted in different ways. The first, more usual one, is as sets of states in the collecting semantics. It is given by the concretization FA.γ (which matches with the definition of the collecting semantics domain). Notice how the definition of the concretization simply maps our free algebra constructors to the corresponding operation in the collecting semantics. For instance, for the apply operation we have FA.γ(`Apply`(R, `s`$^\#$)) = CS.*apply*(R, FA.γ(`s`$^\#$)).

All FA domain operations are sound. FA.*join*, FA.*apply* and FA.*entry* transfer functions are also complete: only FA.*widen* loses precision as `Loc` concretizes into the entire set of stores Σ. This concretization can be refined by unfolding the fixed point (replacing `Loc`(ℓ) with $\mathcal{F}_g(p^\#)(\ell)$). The new term will still contain `Loc`(ℓ) as subterm, which can once again be unfolded, and so on. For the most precise version, unfold until a fixed point is reached here.

4.3 Concretization as a program graph

We can also construct a new Imp program graph $\mathcal{G}_{p^\#} \in \text{FA}.\Sigma^\# \times \mathbb{R} \times \text{FA}.\Sigma^\# \longrightarrow \{0; 1\}$ from an abstract element $p^\# \in \mathbb{L} \rightharpoonup \text{FA}.\Sigma^\#$, or any analysis that uses the free algebra domain as a subdomain. To do so, we define an edge predicate $\mapsto_\# \in (\text{FA}.\Sigma^\# \times \mathbb{R} \times \text{FA}.\Sigma^\#) \longrightarrow \{0; 1\}$, and a vertex predicate $V \in \text{FA}.\Sigma^\# \longrightarrow \{0; 1\}$ by the rules at the bottom of Figure 6. Both depend on $p^\#$, not included it in their notation to keep them light. See Figure 7 for a small example graph built using these rules.

An `Apply`(R, `s`$^\#$) term represents a vertex (i.e., program location in the new graph) obtained by following an edge labelled by R coming from `s`$^\#$ (rule TApply). TJoin ensures that all the terms appearing in a `Join`(`S`$^\#$) term correspond to the same program location in the generated program graph, thus, for any edge going to `t`$^\#$ $\in$ `S`$^\#$, it adds an edge going to `Join`(`S`$^\#$). For the `Loc`(ℓ) case, we need to use the main transfer function $\mathcal{F}_g$ of our abstract interpretation to obtain the pre-state before widening (*join* of the *apply*s of the predecessors of ℓ). The rule TLoc then simply states that the transitions to `Loc` are the same as those to that pre-state (`Loc` is only introduced at a widening point as a renaming to avoid having recursive terms). Finally, TSelf ensures that immediate loops (ℓ being its own predecessor through a trivial relation) are preserved. Note that `Entry` has no predecessor, which corresponds to the assumption that ℓ_0 also has no predecessor.

The V predicate is used to limit our graph $\mathcal{G}_{p^\#}$ to terms that appear in the result of our analysis (`img`($p^\#$) by VBase) or their predecessors through $\mapsto_\#$ (VRec). It notably excludes the intermediate terms that appear in TJoin and TLoc. In practice, it means we are defining an equivalence relation on our states FA.$\Sigma^\#$, that relates `Join` and its contents, as well as `Loc` and its value before renaming; and choosing the `Join` and `Loc` terms as canonical representatives of their classes.

Finally, GraphGen defines our new graph $\mathcal{G}_{p^\#}$: its edges are elements of $\mapsto_\#$ that end in V. Using the free algebra domain on its own, this newly generated graph is isomorphic to the input (restricted to reachable locations). It is the same graph, whose vertices have been renamed by $p^\#$, as mentioned in the following theorem. This implies there is no abstraction loss when using this domain.

Theorem 4.1. *When $p^\# =$ `analyse`(FA), $\mathcal{G}_{p^\#}$ is isomorphic to $\mathcal{G}$ (restricted to reachable locations, i.e. locations ℓ such that there is a path from ℓ_0 to ℓ in $\mathcal{G}$) via $p^\#$:*

- *$p^\#$ is injective (restricted to reachable locations)*
- $\mathcal{G}_{p^\#} = \left\{(p^\#(\ell), R, p^\#(\ell')) \,\middle|\, \mathcal{G}(\ell, R, \ell') \wedge \ell \text{ reachable}\right\}$

Note that the proof (in Appendix A.2) uses the assumption that outgoing edges of each node are uniquely labelled. Without it, we lose injectivity of $p^\#$ as we merge identical children in $\mathcal{G}_{p^\#}$. $\mathcal{G}$ and $\mathcal{G}_{p^\#}$ would still be linked via a bisimulation between entry and widening points (where injectivity still holds), similar to that of Theorems 5.5 and 5.6.

5 TRANSFORMATION FUNCTORS AS COMPILER PASSES

A *functor* F is a function that creates a new Imp domain $F(D_1, \ldots, D_n)$ from a number of Imp domains $D_1, \ldots, D_n$ passed as arguments. Here we are interested in two specific kinds of functors: *transformation functors*, which only change the *apply* operation, and *product functors* which combine domains. We exhibit a criteria for soundness and completeness of such functors, and show how applying sound (resp. complete) functors to the free algebra domain lead to a forward (resp. backward) simulation between the widening points in the input and generated programs.

We say that an n-ary functor F is *sound* when for all sound domains $D_1, \ldots, D_n$, the domain $F(D_1, \ldots, D_n)$ is also sound. Similarly, we say that F is *complete* when for all complete domains $D_1, \ldots, D_n$, the domain $F(D_1, \ldots, D_n)$ is also complete. Note that functor soundness and completeness are compositional: if F and G are sound (or complete), then so is $D \mapsto F(G(D))$.

5.1 Transformation functors

Transformation functors are used to perform small, statement level transformations of our programs. Formally, a transformation functor F is any functor that (1) only modifies the *apply* operation of its argument D and (2) can only create values of type $D.\mathbb{\Sigma}^\sharp$ through the domain operations of D: specifically only $D.\mathit{apply}$ and $D.\mathit{join}$. This means that all domain components other than *apply* are equal to those of D; notably, $F(D).\mathbb{\Sigma}^\sharp = D.\mathbb{\Sigma}^\sharp$. Furthermore, $F(D).\mathit{apply}(R, s^\sharp)$ returns a combination of $D.\mathit{apply}$, $D.\mathit{join}$, and $s^\sharp$. The specific combination depends on R and $s^\sharp$. Using an opaque type, it is quite easy to enforce this constraint in code.

The state $s^\sharp$ can only be inspected through *sound queries*. We write $s^\sharp \vDash P$ when all elements abstracted by $s^\sharp \in D.\mathbb{\Sigma}^\sharp$ satisfy predicate $P \in \mathbb{\Sigma} \to \{0; 1\}$, i.e. $s^\sharp \vDash P$ implies $\forall \sigma \in D.\gamma(s^\sharp), P(\sigma)$. For instance $s^\sharp \vDash [\sigma \in \mathbb{\Sigma} \mapsto \sigma(x) = 0]$ means variable x is zero in all elements abstracted by $s^\sharp$. As the full function definition of P is a bit heavy, we abbreviate it as $s^\sharp \vDash \sigma(x) = 0$. Note that not having $s^\sharp \vDash P$ does not mean the property is false on $D.\gamma(s^\sharp)$.

Example 5.1 (Division guard functor). A simple example of a transformation is the *division guard* functor (DG), which adds an assertion "divisor is not 0" before every division:

$$\begin{aligned}
\texttt{safe}(\{e_1; \ldots; e_n\}, s^\sharp) &\triangleq D.\mathit{apply}(\mathrm{If}\ e_1 \neq 0, \ldots D.\mathit{apply}(\mathrm{If}\ e_n \neq 0, s^\sharp)\ldots) \\
DG(D).\mathit{apply}(\mathrm{If}\ e, s^\sharp) &\triangleq D.\mathit{apply}(\mathrm{If}\ e, \texttt{safe}(\texttt{divisors}(e), s^\sharp)) \\
DG(D).\mathit{apply}(x := e, s^\sharp) &\triangleq D.\mathit{apply}(x := e, \texttt{safe}(\texttt{divisors}(e), s^\sharp))
\end{aligned}$$

where $\texttt{divisors} \in \mathbb{E} \to \mathcal{P}_f(\mathbb{E})$ is the set of sub-expressions that appear to the right of a division or modulo operation in e. Furthermore, we extend *apply* so that $\mathit{apply}(\_, \bot) \triangleq \bot$. This way, $\texttt{safe} \in \mathcal{P}_f(\mathbb{E}) \times D.\mathbb{\Sigma}^\sharp \to D.\mathbb{\Sigma}^\sharp_\bot$ adds a guard for each expression in its argument set[7].

Example 5.2 (Ternary expression rewrite functor). A more complex example is the *ternary rewrite* functor (T), which replaces the ternary if-then-else in expressions by explicit jumps. The full definition is a bit technical, but for a single ternary expression, it can be defined as:

$$T(D).\mathit{apply}(\mathrm{If}\ (e_c\ ?\ e_t : e_e), s^\sharp) \triangleq D.\mathit{apply}\left(\mathrm{If}\ \mathtt{t}, D.\mathit{join}\begin{Bmatrix} D.\mathit{apply}(\mathtt{t} := e_t, D.\mathit{apply}(\mathrm{If}\ e_c, s^\sharp)); \\ D.\mathit{apply}(\mathtt{t} := e_e, D.\mathit{apply}(\mathrm{If}\ e_c = 0, s^\sharp)) \end{Bmatrix}\right)$$

where $\mathtt{t} \notin \mathbb{X}$ is a new, fresh variable. This example shows that transformation functors can change the type of program relations (in this case, we remove a construct and add a variable). The new variable requires changing the concretization as well: $T(D).\gamma(s^\sharp)$ is the same as $D.\gamma(s^\sharp)$ but removes $\mathtt{t}$ from the store.

[7] To avoid issues with nested divisions, the set passed to `safe` should be sorted in increasing size of terms.

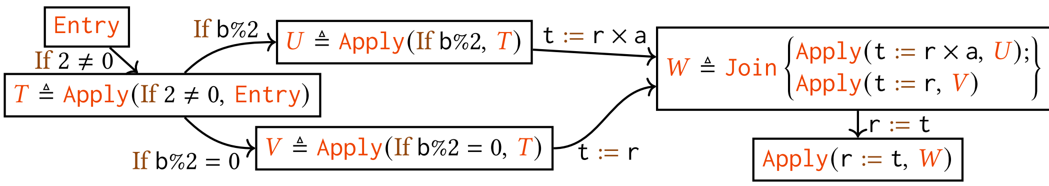


Fig. 7. Example compilation of the small program "r := b%2 ? r × a : r" through the division guard and ternary functors (analyse(DG(T(FA)))). We use T, U, V and W as short names for legibility.

Figure 7 shows the result of compiling a simple program (a single assignment) through the ternary rewrite (T) and division guard (DG) functors, applied to the free algebra (FA) domain of Section 4.

Example 5.3 (Query simplification functor). Both previous examples only perform syntactic transformations: their *apply* does not inspect the program state. A simple functor that does this is the *query simplification functor*:

$$\mathrm{Q}(\mathrm{D}).\mathit{apply}(x := e,\ s^\#) \triangleq \begin{cases} \bot & \text{if } s^\# \vDash \hat{\mathcal{E}}[\![e]\!](\sigma) = \bot \\ s^\# & \text{if } s^\# \vDash \sigma(x) = \hat{\mathcal{E}}[\![e]\!](\sigma) \\ \mathrm{D}.\mathit{apply}(x := z,\ s^\#) & \text{if } \exists z,\ s^\# \vDash \hat{\mathcal{E}}[\![e]\!](\sigma) = z \\ \mathrm{D}.\mathit{apply}(x := e,\ s^\#) & \text{otherwise} \end{cases}$$

$$\mathrm{Q}(\mathrm{D}).\mathit{apply}(\mathrm{If}\ e,\ (s^\#,\ \sigma^\#)) \triangleq \begin{cases} \bot & \text{if } s^\# \vDash \hat{\mathcal{E}}[\![e]\!](\sigma) \subseteq \{0; \bot\} \\ s^\# & \text{if } s^\# \vDash \hat{\mathcal{E}}[\![e]\!](\sigma) \neq 0 \\ \mathrm{D}.\mathit{apply}(\mathrm{If}\ e,\ s^\#) & \text{otherwise} \end{cases}$$

This simplifies an assignment to $\bot$ if one of the variables has no valid values; removes it if it doesn't change the value of x; simplifies it if the expression e has a constant value (thus performing constant propagation); and leaves it unchanged otherwise. For guards, it checks if the condition is false, in which case it returns $\bot$; or true, in which case it removes the guard. Q works well with the numerical domain, since queries using $\hat{\mathcal{E}}[\![\cdot]\!]$ can be computed using the forward evaluation function $\vec{\mathcal{E}}[\![\cdot]\!]$ (for example $\sigma^\# \vDash \hat{\mathcal{E}}[\![e]\!](\sigma) = 0$ is simply $\vec{\mathcal{E}}[\![e]\!](\sigma^\#) = [0:0]$)

This definition for transformation functors is quite restrictive. They only act on a single relation, and not on multiple statements. For example, they cannot simplify double assignments ($x := 0$ followed by $x := 1$) or change order of assignments. However, our SSA translation will perform these automatically by grouping assignments in blocks of bindings.

On the other hand, their simplicity allows proving some strong results. The following lemma shows that it suffices to prove soundness (respectively, completeness) of the functor applied to the collecting semantics domain (CS) to prove soundness (respectively, completeness) for any domain given as an argument.

Lemma 5.4 (Functor soundness and completeness). *A transformation functor F is sound if and only if F(CS) is sound. Similarly, F is complete if and only if F(CS) is complete.*

See proof in Appendix A.3. All three examples above are both sound and complete functors.

When proving soundness (or completeness) of a F(CS), one must take care to only use query soundness results ($s^\# \vDash P \implies P(s^\#)$), and not completness, as queries may be false even when P is always true. For instance, to prove soundness of a functor that looks like

$$\mathrm{F}(\mathrm{CS}).\mathit{apply}(R,\ s^\#) \triangleq \begin{cases} \ldots & \text{if } s^\# \vDash \mathtt{true} \\ \ldots & \text{otherwise} \end{cases}$$

we must prove soundness in the first branch (restricted to elements that satisfy P) and in the second branch. We cannot use the information "`true` is true on all elements of $s^\#$" to skip the proof of the second branch, as the query may return false.

5.2 Simulation theorems

Applying a transformation functor can be seen as a statement-level compiler pass. These passes can perform syntactic transformations (by inspecting the relation), semantic ones (through queries on the input states), or combine both. Since functor soundness and completeness are compositional, we can easily define each small transformation we want to perform as a functor, prove that these functors are sound (or complete) individually, and obtain the result for the whole chain.

Let us take a transformation functor F, we are interested the program generated by the free algebra domain under F. We write $p^\# = \texttt{analyse(F(FA))}$ the analysis result, and $\rightarrow_{\mathcal{G}_{p^\#}}$ the transition system associated with the new Imp program $\mathcal{G}_{p^\#}$ generated by the free algebra domain.

Theorem 5.5 (Sound functor forward simulation). *If F is a sound transformation functor, then for all reachable pairs (ℓ, σ) and (ℓ', σ') such that ℓ and ℓ' are the entrypoint or widening points:*

$$(\ell, \sigma) \rightarrow^+_{\mathcal{G}} (\ell', \sigma') \quad\Rightarrow\quad (p^\#(\ell), \sigma) \rightarrow^+_{\mathcal{G}_{p^\#}} (p^\#(\ell'), \sigma')$$

Theorem 5.6 (Complete functor backward simulation). *If F is a complete transformation functor, then for all entry or widening points ℓ, ℓ', and for all σ, σ':*

$$(p^\#(\ell), \sigma) \rightarrow^+_{\mathcal{G}_{p^\#}} (p^\#(\ell'), \sigma') \quad\Rightarrow\quad (\ell, \sigma) \rightarrow^+_{\mathcal{G}} (\ell', \sigma')$$

See proofs in Appendix A.3.

5.3 Product functors

Reduced domain products [Cousot and Cousot 1979] are a classical tool to combine domains. The basic product is a two argument functor. It returns a domain whose state is a pair of the states of its arguments; whose operations are the pairwise lifting the argument operation; and whose concretization is the intersection of their concretizations. We denote it with the infix $\times$.

Simply using this is equivalent to running the two analysis independently. For added benefit, add a query simplification functor Q on top of the product. Queries can then use information from both states to simplify the terms, and thus prove results that each individual domain could not.

A product between the free algebra domain and another domain can still generate a program graph. The rules TLoc and TSelf just need to be adapted a little as $\mathcal{F}_{\mathcal{G}}(p^\#)(\ell)$ is no longer a free algebra state, but a pair containing such a state, so we need to add a simple projection.

6 SSA SIGNATURE AND SSA FREE ALGEBRA

This section presents our SSA language, highlighting the differences to Imp. It then presents an abstract domain signature adapted to this language (similarly to the Imp signature in Figure 4) and a free algebra implementation of this signature (similar to the one from Section 4)

6.1 SSA syntax and semantics

We use the syntax and semantics of SSA (Figure 8) defined by Lemerre [2023] (corresponding to a high-level representation of the sea-of-nodes representation [Click and Paleczny 1995; Demange et al. 2018]), with small variations.

Syntax. An SSA expression $\hat{e} \in \hat{\mathbb{E}}$[8] is similar to a program expression $e \in \mathbb{E}$ but without the ternary if-then-else. SSA variables are composed of an identifier $i \in \mathbb{I}$ and a location $\hat{\ell} \in \hat{\mathbb{L}}$, which

[8]We use a hat ^ notation to differentiate SSA-specific objects from their Imp counterparts.

$i \in \mathbb{I}$ (Identifiers) $\quad \hat{\ell} \in \hat{\mathbb{L}}$ (Locations) $\quad i_{\hat{\ell}} \in \hat{\mathbb{X}} \triangleq \mathbb{I} \times \hat{\mathbb{L}}$ (Variables) $\quad \mathbb{\Gamma} \triangleq \hat{\mathbb{X}} \rightharpoonup \mathbb{Z}$ (Valuation)
$\hat{e} \in \hat{\mathbb{E}} \triangleq z \mid i_{\hat{\ell}} \mid \hat{e} \diamond \hat{e}$ (SSA expression) $\quad B \in \mathbb{B} \triangleq \hat{\mathbb{X}} \rightharpoonup \hat{\mathbb{E}}$ (Bindings) $\quad \hat{R} \in \hat{\mathbb{R}} \triangleq \hat{\mathtt{If}}\ \hat{e} \mid \mathtt{Bind}(B)$
$\hat{\mathbb{G}} \triangleq (\hat{\mathbb{L}} \times \hat{\mathbb{R}} \times \hat{\mathbb{L}}) \rightarrow \{0; 1\}$ (SSA graph) $\quad \hat{\mathbb{S}} \triangleq \hat{\mathbb{L}} \times \mathbb{\Gamma}$ (SSA state)

$$\mathrm{scope}_{\hat{\mathcal{G}}} \in \hat{\mathbb{E}} \to \mathcal{P}(\hat{\mathbb{L}}) \qquad \mathrm{scope}_{\hat{\mathcal{G}}}(i_{\hat{\ell}}) \triangleq \left\{ \hat{\ell}' \in \hat{\mathbb{L}} \,\middle|\, \hat{\ell} \text{ dominates } \hat{\ell}' \text{ in } \hat{\mathcal{G}} \right\}$$

$$\mathrm{unbind}_{\hat{\mathcal{G}}} \in \hat{\mathbb{L}} \times \mathbb{\Gamma} \to \mathbb{\Gamma} \qquad \mathrm{unbind}_{\hat{\mathcal{G}}}(\hat{\ell}, \Gamma) \triangleq \left[\hat{x} \in \mathrm{dom}\,\Gamma \mapsto \Gamma(v) \,\middle|\, \hat{\ell} \in \mathrm{scope}_{\hat{\mathcal{G}}}(\hat{x}) \right]$$

$$\mathrm{bind} \in \mathbb{B} \times \mathbb{\Gamma} \to \mathbb{\Gamma} \qquad \mathrm{bind}(B, \Gamma) \triangleq \left[\hat{x} \in \hat{\mathbb{X}} \mapsto \begin{cases} \hat{\mathcal{E}}[\![B(\hat{x})]\!](\Gamma) & \text{if } \hat{x} \in \mathrm{dom}\,B \\ \Gamma(\hat{x}) & \text{otherwise} \end{cases} \right]$$

$$\rightsquigarrow_{\hat{\mathcal{G}}} \in \hat{\mathbb{S}} \times \hat{\mathbb{S}} \qquad \begin{aligned} (\hat{\ell}, \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}} (\hat{\ell}', \Gamma') \triangleq\ & \exists\, \hat{e},\ \hat{\mathcal{G}}(\hat{\ell}, \hat{\mathtt{If}}\ \hat{e}, \hat{\ell}') \wedge\ \Gamma' = \Gamma \wedge \hat{\mathcal{E}}[\![\hat{e}]\!](\Gamma) \notin \{0; \bot\} \\ \vee\ & \exists\, B,\ \hat{\mathcal{G}}(\hat{\ell}, \mathtt{Bind}(B), \hat{\ell}') \wedge\ \Gamma' = \mathrm{unbind}_{\hat{\mathcal{G}}}(\hat{\ell}', \mathrm{bind}(B, \Gamma)) \end{aligned}$$

Fig. 8. SSA language syntax (top) and semantics (bottom).

determines its *scope*, i.e., the set of locations where the variable can appear. For simplicity in this paper, we choose $\mathbb{I} = \mathbb{X}$, i.e., the SSA variable $x_{\hat{\ell}}$ can be understood as the value of Imp variable x at location $\hat{\ell}$.

In an SSA graph $\hat{\mathcal{G}} \in \hat{\mathbb{G}}$, edges are either annotated by expressions ($\hat{\mathtt{If}}\ \hat{e}$), representing a condition, or bindings ($\mathtt{Bind}(B)$) mapping multiple variables to expressions. We denote by $\hat{\ell}_0$ the initial location where SSA programs start. We require that bindings edges are exactly those leading into a join node (node with multiple predecessors). Furthermore, an SSA variable $x_{\hat{\ell}}$ should only appear in the bindings leading into the join node at location $\hat{\ell}$ (location where it is bound), and it should appear in all the bindings leading into $\hat{\ell}$ (all bindings leading into a join node have the same domain). Thus, join nodes represent both program joins and what traditional SSA denotes by ϕ functions. If $\hat{\ell}$ has two predecessors $\hat{\mathcal{G}}(\hat{\ell}', \mathtt{Bind}([\hat{x} \mapsto \hat{e}]), \hat{\ell})$ and $\hat{\mathcal{G}}(\hat{\ell}'', \mathtt{Bind}([\hat{x} \mapsto \hat{e}']), \hat{\ell})$, then the scope of $\hat{x}$ is ℓ. In traditional SSA, $\hat{x}$ would be bound here to $\phi(\hat{e},\ \hat{e}')$. Two example SSA graphs are given in Figure 2c (where $\hat{\mathtt{If}}$ and $\mathtt{Bind}$ constructors have been left implicit).

Semantics. The interpretation of SSA expressions $\hat{\mathcal{E}}[\![\cdot]\!]$ is similar to that of Imp expressions $\mathcal{E}[\![\cdot]\!]$. The semantics of SSA is given as a transition system $(\hat{\mathbb{S}}, \rightsquigarrow)$ between SSA states. Given an SSA graph $\hat{\mathcal{G}}$, there is a transition $(\hat{\ell}, \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}} (\hat{\ell}', \Gamma')$ if there is an edge $\hat{\mathcal{G}}(\hat{\ell}, \hat{R}, \hat{\ell}')$ and the edge $\hat{R}$ is either a condition that evaluates to non-zero, or a binding, which is then evaluated and added to the environment.

We also have an unbinding operation that removes variables that are no longer in scope from the environment (where scope is defined using domination between locations). It is not necessary (SSA variables can be seen as assigned rather than bound [Schneider 2013]) but will help analyses, as it avoids maintaining information about useless variables. Unbinding only occurs at join nodes (whose incoming edges are annotated by bindings), since non-join nodes only have a single predecessor and thus only grow in scope.

6.2 SSA domain signature

The signature of SSA domains is given in Figure 9. It is similar to the previous signature of Figure 4 with a few key variations. The only relations applied on SSA are guards. To emphasize this, we rename *apply* to *assûme*. Since bindings only occur before joins, we place them directly in the signature of *joîn*. It no longer takes a set of states as argument, but a set states paired with their respective bindings. Once again, all bindings given to a join should have the same domain (define the same variables). For example, *joîn* $\left\{([\hat{x} \mapsto 3], \Gamma^{\#});\ ([\hat{x} \mapsto 5], \Gamma'^{\#})\right\}$ is the merging of two branches

$$\begin{array}{rclcrcl}
\Gamma^\# \in \mathbb{\Gamma}^\# & & \text{(set of abstract states)} & & \hat{join} & \in & \mathcal{P}_f(\mathbb{B} \times \mathbb{\Gamma}^\#) \to \mathbb{\Gamma}^\#_\bot \\
\hat{entry} & \in & \mathbb{\Gamma}^\# & & \hat{widen} & \in & W \times \mathbb{\Gamma}^\# \times \mathbb{\Gamma}^\# \to \mathbb{\Gamma}^\# \\
\hat{assume} & \in & \hat{\mathbb{E}} \times \mathbb{\Gamma}^\# \to \mathbb{\Gamma}^\#_\bot & & \hat{\gamma} & \in & \mathbb{\Gamma}^\# \to \mathcal{P}(\mathbb{\Gamma})
\end{array}$$

$$\begin{array}{rcl}
\hat{\mathcal{F}}_{\hat{g}} & \in & (\hat{\mathbb{L}} \rightharpoonup \mathrm{D}.\mathbb{\Gamma}^\#) \to (\hat{\mathbb{L}} \rightharpoonup \mathrm{D}.\mathbb{\Gamma}^\#) \\
\hat{\mathcal{F}}_{\hat{g}}(\hat{p}^\#) & \triangleq & \hat{\ell}_0 \mapsto \mathrm{D}.\hat{entry} \\
& | & \hat{\ell} \mapsto \mathrm{D}.\hat{assume}(\hat{e},\, \hat{p}^\#(\hat{\ell}')) \quad \text{if } \hat{\mathcal{G}}(\hat{\ell}',\, \hat{\mathtt{If}}\ \hat{e},\, \hat{\ell}) \land \hat{\ell}' \in \operatorname{dom} \hat{p}^\# \\
& | & \ell \mapsto \mathrm{D}.\hat{join}\left\{(B_k,\, \hat{p}^\#(\hat{\ell}_k)) \,\middle|\, (\hat{\ell}_k, B_k) \text{ such that } \hat{\mathcal{G}}(\hat{\ell}_k,\, \mathtt{Bind}(B_k),\, \hat{\ell}) \land \hat{\ell}_k \in \operatorname{dom} \hat{p}^\#\right\}
\end{array}$$

Fig. 9. SSA abstract domain signature (top) and abstract interpretation (bottom)

$$\begin{array}{rcl}
\Gamma^\# \in \widehat{\mathrm{FA}}.\mathbb{\Gamma}^\# & \triangleq & \hat{\mathtt{Entry}} \mid \hat{\mathtt{Assume}}(\hat{e},\, \Gamma^\#) \mid \hat{\mathtt{Join}}(\mathtt{C}^\#) \mid \hat{\mathtt{Loc}}(\ell) \text{ (SSA algebraic locations)} \\
& & \text{where } \hat{e} \in \hat{\mathbb{E}} \text{ and } \mathtt{C}^\# \in \mathcal{P}_f(\mathbb{B} \times \widehat{\mathrm{FA}}.\mathbb{\Gamma}^\#)
\end{array}$$

$$\widehat{\mathrm{FA}}.\hat{entry} \triangleq \hat{\mathtt{Entry}}$$

$$\widehat{\mathrm{FA}}.\hat{assume}(\hat{e},\, \Gamma^\#) \triangleq \hat{\mathtt{Assume}}(\hat{e},\, \Gamma^\#) \qquad \widehat{\mathrm{FA}}.\hat{join}(\mathtt{C}^\#) \triangleq \begin{cases} \bot & \text{if } \mathtt{C}^\# \text{ is empty} \\ \hat{\mathtt{Join}}(\mathtt{C}^\#) & \text{otherwise} \end{cases}$$

$$\widehat{\mathrm{FA}}.\hat{widen}(\ell,\, \_,\, \_) \triangleq \hat{\mathtt{Loc}}(\ell)$$

$$\widehat{\mathrm{FA}}.\hat{\gamma}(\hat{\mathtt{Entry}}) \triangleq \mathbb{\Gamma} \qquad \widehat{\mathrm{FA}}.\hat{\gamma}(\hat{\mathtt{Assume}}(\hat{e},\, \Gamma^\#)) \triangleq \left\{\Gamma \in \widehat{\mathrm{FA}}.\hat{\gamma}(\Gamma^\#) \,\middle|\, \hat{\mathcal{E}}[\![\hat{e}]\!](\Gamma) \neq 0\right\}$$

$$\widehat{\mathrm{FA}}.\hat{\gamma}(\hat{\mathtt{Join}}(\mathtt{C}^\#)) \triangleq \bigcup_{B,\Gamma^\# \in \mathtt{C}^\#} \left\{\mathrm{bind}(B,\, \Gamma) \,\middle|\, \Gamma \in \widehat{\mathrm{FA}}.\hat{\gamma}(\mathtt{C}^\#)\right\} \qquad \widehat{\mathrm{FA}}.\hat{\gamma}(\hat{\mathtt{Loc}}(\ell)) \triangleq \mathbb{\Gamma}$$

$$\text{TAssumeSSA}\ \frac{}{\Gamma^\# \xhookrightarrow{\hat{\mathtt{If}}\ \hat{e}}_\# \hat{\mathtt{Assume}}(\hat{e},\, \Gamma^\#)} \qquad \text{TJoinSSA}\ \frac{(B, \Gamma^\#) \in \mathtt{C}^\#}{\Gamma^\# \xhookrightarrow{\mathtt{Bind}(B)}_\# \hat{\mathtt{Join}}(\mathtt{C}^\#)} \qquad \text{TLocSSA}\ \frac{\Gamma^\# \xhookrightarrow{\hat{R}}_\# \hat{\mathcal{F}}_{\hat{g}}(\hat{p}^\#)(\hat{\ell})}{\Gamma^\# \xhookrightarrow{\hat{R}}_\# \hat{\mathtt{Loc}}(\hat{\ell})}$$

Fig. 10. The free algebra SSA abstract domain ($\widehat{\mathrm{FA}}$) (top) and rules for generating SSA programs (bottom).

$\Gamma^\#$ and $\Gamma'^\#$ with the additional information that $\hat{x}$ is 3 when coming from $\Gamma^\#$ and 5 when coming from $\Gamma'^\#$. This is how we represent what traditional SSA would denote with a ϕ function: $\hat{x} \mapsto \phi(3, 5)$. Note that this signature makes some constraints placed on our SSA programs explicit: it is clear that assume nodes have a single predecessor labelled by a guard, and join nodes have multiple predecessors labelled by bindings.

$\hat{\mathcal{F}}_{\hat{g}}$ is the transfer function used for the direct analysis of SSA programs. It is similar to $\mathcal{F}_g$, but explicitly separates treatment of guard edges (with $\hat{assume}$) and bindings before a join (with $\hat{join}$), whereas $\mathcal{F}_g$ performed both simultaneously using a *join* of *apply*s. The other components of our analysis (∇_W and `analyse`) are the same as in Figure 4.

The SSA domain also has soundness and correction hypothesis similar to those of the Imp domain in Figure 4, omitted here for the sake of brevity.

6.3 Free algebra of the SSA domain signature

Figure 10 presents the *free algebra SSA domain*, denoted $\widehat{\mathrm{FA}}$. Just like in Imp free algebra (Section 4), the domain operation simply create terms using the relevant function symbols.

This domain also presents a dual interpretation as sets of valuations (given by the concretization $\widehat{\mathrm{FA}}.\hat{\gamma}$) and as a program graph (given by the edge predicate $\hookrightarrow_\# \in (\widehat{\mathrm{FA}}.\mathbb{\Gamma}^\# \times \hat{\mathbb{R}} \times \widehat{\mathrm{FA}}.\mathbb{\Gamma}^\#) \to \{0; 1\}$). The rules for generating this graph are similar to those of the Imp free algebra. Note that contrary to TJoin, where a `Join` had the same predecessors as its elements, here the $\hat{\mathtt{Join}}$'s predecessors are

$$\mathrm{Lift}(\hat{\mathrm{D}}).\Sigma^\# \triangleq (\mathbb{X} \to \hat{\mathbb{E}}) \times \hat{\mathrm{D}}.\mathbb{\Gamma}^\# \qquad \mathrm{Lift}(\hat{\mathrm{D}}).\mathit{entry} \triangleq \left[x \in \mathbb{X} \mapsto x_{\hat{\mathrm{D}}.\hat{\mathit{entry}}}\right], \hat{\mathrm{D}}.\hat{\mathit{entry}}$$

$$\mathrm{Lift}(\hat{\mathrm{D}}).\mathit{apply}(x := e,\ (\sigma^\#, \Gamma^\#)) \triangleq \sigma^\# \left[x \mapsto \mathtt{subst}(e,\ \sigma^\#)\right], \Gamma^\#$$

$$\mathrm{Lift}(\hat{\mathrm{D}}).\mathit{apply}(\mathbf{If}\ e,\ (\sigma^\#, \Gamma^\#)) \triangleq \sigma^\#, \hat{\mathrm{D}}.\hat{\mathit{assume}}(\mathtt{subst}(e,\ \sigma^\#),\ \Gamma^\#)$$

$$\mathrm{Lift}(\hat{\mathrm{D}}).\mathit{join}\left\{(\sigma^\#_i, \Gamma^\#_i) \,\middle|\, i = 1..n\right\} \triangleq \begin{cases} \sigma^\#_1, \Gamma^\#_1 & \text{if } n = 1 \text{ (join of a singleton)} \\ \sigma^\#, \Gamma^\# & \text{if } \Gamma^\# \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$\text{where } \sigma^\# \triangleq \left[x \in \mathbb{X} \mapsto \begin{cases} e & \text{if } e = \sigma^\#_1(x) = \ldots = \sigma^\#_n(x) \text{ (equal in all branches)} \\ x_{\Gamma^\#} & \text{otherwise } (\exists\, i\ j,\ \sigma^\#_i(x) \neq \sigma^\#_j(x)) \end{cases}\right]$$

$$\text{and } \Gamma^\# \triangleq \hat{\mathrm{D}}.\hat{\mathit{join}}\left\{[x_{\Gamma^\#} \mapsto \sigma^\#_i(x) \mid x \text{ such that } \exists\, i\ j,\ \sigma^\#_i(x) \neq \sigma^\#_j(x)], \Gamma^\#_i \,\middle|\, i = 1..n\right\}$$

$$\mathrm{Lift}(\hat{\mathrm{D}}).\mathit{widen}(\ell, (\sigma^\#_\ell, \Gamma^\#_\ell), (\sigma^\#_r, \Gamma^\#_r)) \triangleq \left[x \in \mathbb{X} \mapsto \begin{cases} x_{\hat{\mathrm{D}}.\hat{\mathit{widen}}(\ell,\, \Gamma^\#_\ell,\, \Gamma^\#_r)} & \text{if } \sigma^\#_r(x) = x_{\Gamma^\#_r} \\ \sigma^\#_r(x) & \text{otherwise} \end{cases}\right], \hat{\mathrm{D}}.\hat{\mathit{widen}}(\ell, \Gamma^\#_\ell, \Gamma^\#_r)$$

$$\mathrm{Lift}(\hat{\mathrm{D}}).\gamma(\sigma^\#, \Gamma^\#) \triangleq \left\{\left[x \in \mathbb{X} \mapsto \hat{\mathcal{E}}[\![\sigma^\#(x)]\!](\Gamma)\right] \,\middle|\, \Gamma \in \hat{\mathrm{D}}.\hat{\gamma}(\Gamma^\#)\right\}$$

Fig. 11. The lift functor, lifting an SSA Domain $\hat{\mathrm{D}}$ into an Imp domain $\mathrm{Lift}(\hat{\mathrm{D}})$.

its elements. Instead of identifying each term in the joined set with the whole join, each term is the predecessor of the join, with the edge labelled by its bindings. This also means we no longer need a TSelf rule, as we can no longer collapse loops completely.

Like in Figure 6, generating the graph also requires a vertex predicate $\hat{V} \in \widehat{\mathrm{FA}}.\mathbb{\Gamma}^\# \to \{0; 1\}$ to filter the relevant nodes. It has the same rules as those of V, so they were omitted here. Figure 2c presents two example graphs generated from such free algebra terms.

Going further. We could easily show a version of Theorem 4.1 for direct analysis of the SSA free algebra domain, and define functors on SSA similarly to Section 5. However, apart from the product functor $\widehat{\times}$, we do not really need them as we are mostly interested in analyzing Imp programs, which can use Imp functors before reaching SSA domains through the SSA Lift functor (Section 7).

7 LIFTING SSA DOMAINS TO IMP DOMAINS

In this section, we present the *SSA Lift domain*, denoted Lift, a functor that lifts an SSA domain into an Imp domain. We then show that, when applied to the SSA free algebra domain, this functor is akin to compilation from Imp to SSA.

7.1 The SSA lift functor

The lift functor is detailed in Figure 11. Lift states are pairs of an abstract store, mapping from program variables to the SSA expressions they currently hold, and an SSA state $\hat{\mathrm{D}}.\mathbb{\Gamma}^\#$. The functor reuses the SSA states of the argument $\hat{\mathrm{D}}$ as SSA locations ($\hat{\mathbb{L}} = \hat{\mathrm{D}}.\mathbb{\Gamma}^\#$). The entrypoint $\mathrm{Lift}(\hat{\mathrm{D}}).\mathit{entry}$ contains a map from all program variables to initial SSA variables, paired with the SSA domain's entrypoint $\hat{\mathrm{D}}.\hat{\mathit{entry}}$.

Applying an assignment updates the store of the corresponding variable; and applying a guard updates the SSA state using $\hat{\mathrm{D}}.\hat{\mathit{assume}}$. Here the $\mathtt{subst} \in \mathbb{E} \times (\mathbb{X} \to \hat{\mathbb{E}}) \to \hat{\mathbb{E}}$ function substitutes all variables from a program expression $e \in \mathbb{E}$ by their value in $\sigma^\#$, which is an SSA expression. This only works if the constructs that appear in $\mathbb{E}$ are translatable to SSA expression constructs. Use transformation functors (Section 5) to simplify the language of program expressions if needed.

The $\hat{\mathrm{D}}.\mathit{join}\left\{(\sigma_i^\#, \Gamma_i^\#) \,\middle|\, i = 1..n\right\}$ function is a bit more complex. The new store $\sigma^\#$ maps x to the unique value if all argument stores evaluate x the same value, and to a new SSA variable otherwise (introducing a ϕ function). The new SSA state $\Gamma^\#$ is the $\hat{\mathrm{D}}.\widehat{join}$ of all locations $\Gamma_i^\#$ with the corresponding bindings for renamed variables. Note that this is a recursive definition, as SSA variables in the bindings are named $x_{\Gamma^\#}$ where $\Gamma^\#$ is the SSA state being defined. In practice, we break this mutual recursion through hash-consing [Filliâtre and Conchon 2006], each SSA state is given a unique numeric identifier and SSA variables only reference that identifier. Although not presented here, this join operation can easily be adapted to perform global value numbering [Lemerre 2023] by merging SSA variables which are equal in all branches. Performing GVN is required to optimize the dead code in Figure 2. Notice that by definition, the calls to $\hat{\mathrm{D}}.\widehat{join}$ respect the assumptions we made on our SSA form. All set elements bind the same variables, and, since those variables are named after the current location, they are bound nowhere else.

The widening simply calls $\hat{\mathrm{D}}.\widehat{widen}$ to determine the new SSA state, and renames any introduced variables in the store to match the new state. Note that this assumes both stores are fairly similar.

Finally, $\mathrm{Lift}(\hat{\mathrm{D}}).\gamma((\sigma^\#, \Gamma^\#))$ generates the set of represented stores, by using $\sigma^\#$ to map variable to SSA expressions, and then evaluating these expressions in a context given by $\hat{\mathrm{D}}.\hat{\gamma}(\Gamma^\#)$.

7.2 Compiling to SSA

We now consider running the analysis on the lift functor to the SSA free algebra domain $\widehat{\mathrm{FA}}$. We write $p^\# \triangleq \mathtt{analyse}(\mathrm{Lift}(\widehat{\mathrm{FA}}))$ the analysis result. Using it, we generate an SSA program $\hat{\mathcal{G}}_{p^\#}$ from the SSA free algebra. Just like for the functor products, this requires adapting the TLocSSA rule by adding a projection, since our states are not SSA free algebra states, but a pair (which includes an SSA free algebra state). With this setup, our analysis effectively compiles an Imp program to SSA form. We write $\rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}}$ the transition system associated with this new SSA program.

The following theorems show simulation results between the source and compiled programs. Since the source and target language are different, our simulation relation is no longer just equality:

$$\begin{array}{rcl} C & \in & \mathbb{S} \times \hat{\mathbb{S}} \to \{0;1\} \\ C((\ell,\, \sigma),\, (\Gamma^\#,\, \Gamma)) & \triangleq & \exists\, \sigma^\# \in \mathbb{X} \to \hat{\mathbb{E}},\ (\sigma^\#,\, \Gamma^\#) = p^\#(\ell) \wedge \sigma = \left[x \in \mathbb{X} \mapsto \hat{\mathcal{E}}[\![\sigma_0^\#(x)]\!](\Gamma)\right] \end{array}$$

The first part is compatibility between the Imp location ℓ and the SSA location $\Gamma^\#$ via $p^\#$, and the second part is compatibility between the Imp store σ and the SSA valuation Γ. Notice that with this relation, $\Gamma^\#$ is uniquely determined by ℓ, and σ is uniquely determined by Γ.

Theorem 7.1 (SSA compilation forward simulation). *For all reachable pairs $(\ell,\, \sigma)$ and $(\ell',\, \sigma')$ such that ℓ and ℓ' are entry or widening points, for all $\hat{s} \in \hat{\mathbb{S}}$ we have:*

$$(\ell,\, \sigma) \to_{\mathcal{G}}^{+} (\ell',\, \sigma') \ \wedge\ C((\ell,\, \sigma),\, \hat{s}) \quad\Rightarrow\quad \exists\, \hat{s}' \in \hat{\mathbb{S}},\ C((\ell',\, \sigma'),\, \hat{s}') \ \wedge\ \hat{s} \rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}}^{*} \hat{s}'$$

Furthermore, there exists an $\hat{s} \in \hat{\mathbb{S}}$ such that $C((\ell,\, \sigma),\, \hat{s})$ holds.
Finally, if $\hat{s} \rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}}^{*} \hat{s}'$ has length 0, then ℓ' is not a true loop head (it has a single reachable predecessor).

Theorem 7.2 (SSA compilation backward simulation). *For all SSA states $(\Gamma^\#,\, \Gamma)$ and $(\Gamma'^\#,\, \Gamma')$ where $\Gamma^\#$ and $\Gamma'^\#$ appear in $\mathtt{img}\, p^\#$ as images of widening or entry points, and for all $s' \in \mathbb{S}$ we have:*

$$(\Gamma^\#,\, \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}}^{+} (\Gamma'^\#,\, \Gamma') \ \wedge\ C(s',\, (\Gamma'^\#,\, \Gamma')) \quad\Rightarrow\quad \exists\, s \in \mathbb{S},\ C(s,\, (\Gamma^\#,\, \Gamma)) \ \wedge\ s \to_{\mathcal{G}}^{+} s'$$

Furthermore, there exists an $s' \in \mathbb{S}$ such that $C(s',\, (\Gamma'^\#,\, \Gamma'))$ holds.

See proofs in Appendix A.4.

$$\textsc{EvalReuse}\ \frac{\hat{e} \in \mathsf{dom}(\Gamma^\sharp)}{\Gamma^\sharp \vDash \hat{e} \Downarrow \Gamma^\sharp(\hat{e})} \qquad \textsc{EvalCst}\ \frac{}{\Gamma^\sharp \vDash z \Downarrow [z:z]} \qquad \textsc{EvalBinop}\ \frac{\hat{e}_1 \diamond \hat{e}_2 \notin \mathsf{dom}(\Gamma^\sharp) \quad \Gamma^\sharp \vDash \hat{e}_1 \Downarrow z_1^\# \quad \Gamma^\sharp \vDash \hat{e}_2 \Downarrow z_1^\#}{\Gamma^\sharp \vDash \hat{e}_1 \diamond \hat{e}_2 \Downarrow \vec{\diamond}(z_1^\#, z_2^\#)} \qquad \textsc{EvalVar}\ \frac{\hat{x} \notin \mathsf{dom}(\Gamma^\sharp)}{\Gamma^\sharp \vDash \hat{x} \Downarrow [-\infty:+\infty]}$$

$$\textsc{ReduceBwd}\ \frac{\Gamma^\sharp \vDash \hat{e}_1 \diamond \hat{e}_2 \Downarrow z^\# \quad \Gamma^\sharp \vDash \hat{e}_1 \Downarrow z_1^\# \quad \Gamma^\sharp \vDash \hat{e}_2 \Downarrow z_2^\# \quad (z_1'^\sharp, z_2'^\sharp) = \overleftarrow{\diamond}(z_1^\sharp, z_2^\sharp, z^\sharp) \quad (z_i'^\sharp \sqcap_{\mathbb{Z}^\#} z_i^\sharp) \subset_{\mathbb{Z}^\#} z_i^\sharp}{\Gamma^\sharp \Rightarrow \Gamma^\sharp\left[\hat{e}_i \mapsto z_i^\sharp \sqcap_{\mathbb{Z}^\#} z_i'^\sharp\right]} \qquad \textsc{ReduceFwd}\ \frac{\Gamma^\sharp \vDash \hat{e} \Downarrow z^\#}{\Gamma^\sharp \Rightarrow \Gamma^\sharp\left[\hat{e} \mapsto z^\#\right]} \qquad \textsc{ReduceBot}\ \frac{\Gamma^\sharp \vDash \hat{e} \Downarrow \bot_{\mathbb{Z}^\#}}{\Gamma^\sharp \Rightarrow \bot_{\widehat{\mathbb{N}}}}$$

$$\Gamma^\sharp \in \widehat{\mathbb{N}}.\mathbb{\Gamma}^\# \triangleq \hat{\mathbb{E}} \rightharpoonup \mathbb{Z}^\# \qquad \widehat{\mathbb{N}}.\widehat{entry} \triangleq \emptyset$$

$$\widehat{\mathbb{N}}.\hat{\gamma}(\Gamma^\sharp) \triangleq \left\{\Gamma \in \hat{\mathbb{X}} \rightharpoonup \mathbb{Z} \,\middle|\, \forall \hat{e} \mapsto z^\# \in \Gamma^\sharp,\ \hat{\mathcal{E}}[\![\hat{e}]\!](\Gamma) \in \gamma_{\mathbb{Z}^\#}(z^\#)\right\} \qquad \textsc{Assume}\ \frac{\Gamma^\sharp \vDash \hat{e} \Downarrow z^\# \quad \Gamma^\sharp\left[\hat{e} \mapsto z^\sharp \sqcap_{\mathbb{Z}^\#} (\neg 0)\right] \Rightarrow \Gamma'^\sharp}{\widehat{\mathbb{N}}.\widehat{assume}(\hat{e},\ \Gamma^\sharp) \triangleq \Gamma'^\sharp}$$

$$\mathtt{Nbind}(B,\ \Gamma^\sharp) \triangleq \Gamma^\sharp \uplus \left[\hat{x} \mapsto z^\# \,\middle|\, \hat{x} \mapsto \hat{e} \in B \land \Gamma^\sharp \vDash \hat{e} \Downarrow z^\#\right]$$

$$\textsc{Join}\ \frac{\mathtt{Nbind}(B_i,\ \Gamma_i^\sharp) \Rightarrow \Gamma_i'^\sharp \qquad i \in 1..n}{\widehat{\mathbb{N}}.\widehat{join}\left\{(B_i, \Gamma_i^\sharp) \mid i \in 1..n\right\} \triangleq \left[\hat{e} \mapsto z^\# \,\middle|\, \hat{e} \in \bigcap_i \mathsf{dom}(\Gamma_i'^\sharp) \land z^\# = \bigsqcup_{\mathbb{Z}^\#\, i} \Gamma_i'^\sharp(\hat{e})\right]}$$

$$\widehat{\mathbb{N}}.\widehat{widen}(\_,\ \Gamma^\sharp,\ \Gamma'^\sharp) \triangleq \left[\hat{e} \mapsto \Gamma^\sharp(\hat{e})\ \nabla_{\mathbb{Z}^\#}\ \Gamma'^\sharp(\hat{e}) \,\middle|\, \hat{e} \in \mathsf{dom}\,\Gamma^\sharp \cap \mathsf{dom}\,\Gamma'^\sharp\right]$$

Fig. 12. Evaluation rules for $\Downarrow$ (top), constraint propagation/reduction rules (middle), SSA numeric domain $\widehat{\mathbb{N}}$ (bottom).

These theorems statements are designed to be easily composable: if we have three widening point $(\ell,\ \sigma) \to_{\mathcal{G}}^{+} (\ell',\ \sigma') \to_{\mathcal{G}}^{+} (\ell'',\ \sigma'')$, we can use Theorem 7.1 twice and compose the results.

8 SSA BASED NUMERICAL ANALYSIS

In this section, we implement a numerical abstract domain $\widehat{\mathbb{N}}$ (similar to that of Section 3.4), but using the SSA domain signature. Using SSA form here allows storing information about expressions, and not just about variables, which improves precision. This is possible because variables are bound and not assigned, and thus their values, and the values of expressions that use them, never change. We illustrate the precision improvement through various examples, and prove that using $\mathrm{Lift}(\widehat{\mathbb{N}})$ is always more precise than $\mathbb{N}$.

8.1 The SSA numeric domain

The *SSA numeric domain* is presented at the bottom Figure 12. Its states are mappings from SSA symbolic expressions to a numerical single-value abstraction. The concretization of such an element $\Gamma^\sharp$ is the set of valuations which, when used to evaluate an expression $\hat{e}$ of $\Gamma^\sharp$, yield an integer in $\Gamma^\sharp(\hat{e})$. The entry point is the empty mapping. The domain operations require defining a forward evaluation judgement $\Gamma^\sharp \vDash \hat{e} \Downarrow z^\# \in (\widehat{\mathbb{N}}.\mathbb{\Gamma}^\# \times \hat{\mathbb{E}} \times \mathbb{Z}^\#) \to \{0; 1\}$ and a reduction operator $\Rightarrow \in (\Gamma^\sharp \times \Gamma^\sharp) \to \{0; 1\}$.

The judgement $\Gamma^\sharp \vDash \hat{e} \Downarrow z^\#$ intuitively means that we can deduce $\hat{e} \in \gamma_{\mathbb{Z}^\#}(z^\#)$ from $\Gamma^\sharp$. Formally, it is defined through the induction rules given at the top of Figure 12. They proceed by recursively evaluating the expression $\hat{e}$ (rule EvalBinop) until a constant (EvalCst), variable (EvalVar), or remembered expression (EvalReuse) is found in $\Gamma^\sharp$. Binary expressions are evaluated through forward transfer functions, remembered expression return their values and unremembered variables return the top element. The following lemma proves our judgement captures the intended meaning:

Lemma 8.1. *If* $\Gamma^\sharp \vDash \hat{e} \Downarrow z^\#$, *then* $\forall\, \Gamma \in \widehat{N}.\hat{\gamma}(\Gamma^\sharp)$, $\hat{\mathcal{E}}[\![\hat{e}]\!](\Gamma) \in \gamma_{\mathbb{Z}^\#}(z^\#)$.

Proof. By induction on expressions, and soundness of $\vec{\diamond}$. □

The judgement $\Gamma^\sharp \Rightarrow \Gamma'^\sharp$ is a reduction operator [Granger 1992], it implies that $\Gamma^\sharp$ and $\Gamma'^\sharp$ represent the same abstraction ($\widehat{N}.\hat{\gamma}(\Gamma^\sharp) = \widehat{N}.\hat{\gamma}(\Gamma'^\sharp)$) but $\Gamma'^\sharp$ is smaller. Its induction rules are given in the middle of Figure 12. ReduceBwd propagates constraints [Benhamou et al. 1999] of the form $\hat{e} \in \gamma_{\mathbb{Z}^\#}(z^\#)$ between the symbolic expressions. Thus, it learns from conditions [Granger 1992] (appearing e.g. in `if` statements). The result is saved when the precision has improved, which allows further evaluations with $\Rightarrow$ to also improve. ReduceFwd saves the result of evaluation. This will make the result of future joins more precise. Finally, ReduceBot quickly propagates the information that the current state is bottom (some constraint is unsatisfiable).

The domain operations are given as rules instead of functions as they depend on how many reductions ($\Rightarrow$) we wish to perform before returning the result. Performing more reductions will be more precise but also reduce performance.

$\widehat{N}.\textit{ass\^{u}me}(\hat{e},\ \Gamma^\sharp)$ propagates constraints, adding the information that the guard must be true (denoted as $\neg 0$[9]). `Nbind`$(B,\ \Gamma^\sharp)$ updates the abstract SSA state $\Gamma^\sharp$ by mapping $\hat{x}$ to the result of the evaluation of $\hat{e}$ for all bindings $\hat{x} \mapsto \hat{e}$ that appear in the bindings B. The last operation, $\widehat{N}.\textit{jo\^{i}n}$, applies the bindings, and then performs an intersection of the maps (only keeping expressions that are present in all branches, including freshly bound variables). We can perform an intersection because we know nothing important about a symbolic expression that is not present in every branch (in many cases, they will go out of scope and be unbounded).

8.2 Combination of SSA-based analysis and online SSA translation

We can use this SSA state abstraction with the translation of Section 7 to analyze an SSA program while it is being computed from the source program. This analysis combines our SSA abstract state $\Gamma^\sharp$ with an abstract store $\sigma^\# \in \mathbb{X} \to \hat{\mathbb{E}}$.

This combination is more precise than the standard non-relational numerical analysis performed by N (that constrains program variables instead of program values), i.e. we can abstract our combination to a standard numerical analysis.

$$\begin{array}{rcl} \alpha & \in & \mathrm{Lift}(\widehat{N}).\Sigma^\# \to N.\Sigma^\# \\ \alpha(\sigma^\sharp, \Gamma^\sharp) & \triangleq & \left[x \in \mathbb{X} \mapsto z^\# \,\middle|\, \Gamma^\sharp \vDash \sigma^\sharp(x) \Downarrow z^\#\right] \end{array}$$

Intuitively, this abstraction forgets the relations between the variables that are given by the symbolic expressions, and just sees them as opaque identifiers. In particular, we can prove that our analysis operations are monotonic (provided a suitable strategy for applying $\Rightarrow$, such as maximally applying $\Rightarrow$ when evaluation is limited to the symbolic expressions that appear in the abstract store), and thus that our combination is always more precise than the non-relational abstract domain for any succession of operations.

We can also provide specific examples where our analysis improves over a non-relational domain:

- **Reduction on related variables:** in `y := x+1; z := y*y; if(2 <= y <= 5) ...`, the $\mathrm{Lift}(\widehat{N})$ domain can prove that $4 \le z \le 25$ and $1 \le x \le 4$ while N cannot. This is very useful when analyzing machine code, which often places related variables in separate registers (and flags).

[9]Interval or congruence cannot represent this accurately, but a specialized $0/{\neq}\,0$ domain could be used here, or we could use a specific rule for when $\hat{e}$ is a comparison operator, since its value is either 0 or 1.

- **Propagation across statements:** in `c := x < 7; if(c) ...`, the SSA-based domain can prove $x < 7$ after the if, while the IMP numeric domain cannot. Our domain thus avoids the limitation that reduction over a condition [Granger 1992] is limited to the current statement.
- **Remembering previously known facts:** the Lift($\widehat{\mathrm{N}}$) domain can store information that the interval single-value abstraction cannot remember. For instance, it can prove both `if(x!=0) assert(x!=0)` and `if(x*x == 4) assert(x*x==4)`, both of which cannot be proven from a simple interval abstraction of `x`.
- **Benefiting from global value numbering:** We could improve precision further by adding global value numbering [Alpern et al. 1988; Rosen et al. 1988] in our Lift domain [Lemerre 2023]. We could then prove that both `i` and `j` are equal to 7 after running: `i := j := 0; while(i < 7) {i++; j++}`.

In general, the domain Lift($\widehat{\mathrm{N}}$) does not lose precision when analyzing a source which has computations split across multiple statements and variables, as is often the case with machine code. This is the strong relative completeness property, defined by Logozzo and Fähndrich [2008].

LEMMA 8.2 (STRONG RELATIVE COMPLETENESS). *Let $C \in \mathbb{G} \to \mathbb{G}$ be the transformation that flattens IMP program expression by writing all sub-expressions to new temporary variables, and π a projection that strips those newly introduced variables, then*

$$\mathtt{analyse}_{\mathcal{G}}(Lift(\widehat{N})) = \pi(\mathtt{analyse}_{C(\mathcal{G})}(Lift(\widehat{N})))$$

See proof in Appendix A.5.

9 EVALUATION

9.1 Evaluating using TAI

While our study is mostly theoretical, some of our claims can be validated through a practical implementation. We are interested in the following research questions: how does SSA-based numerical analysis (Lift($\widehat{\mathrm{N}}$)) compare to standard numerical analysis (N), both in terms of precision and complexity? What overhead is introduced by using free algebra domains and the SSA lift domain?

To answer these, we have written a small abstract interpreter named TAI in OCaml, following the definitions of this paper. It implements all domains and functors presented in this paper, and allows combining them freely. We have run TAI on some C programs generated using CSMITH [Yang et al. 2011] (limited to the constructs that IMP can represent: integer variables and non-recursive function calls only). We then recorded the average analysis time over 100 passes. This is only the time of the fixed-point calculation in `analyse`, it does not include parsing time or computation of the set W.

To validate precision, we run the analysis using N and Lift($\widehat{\mathrm{N}}$) in parallel. We compare the precision using two metrics: the first compares the abstractions of all variables in all locations, the second compares the abstractions of expressions that appear on outgoing edges in all locations.

Our results are presented in Table 1. We found that adding FA domains barely increases cost. It even improves it in some case as we used hash-consing to have fast equality on the free-algebra, thus skipping the slower numerical equality in our fixed-point computation. The SSA numeric domain increases cost by a reasonable factor. The free algebra alone is quite fast, the SSA free algebra is very slow mostly because of the Lift functor. Lift($\widehat{\mathrm{N}}\hat{\times}\widehat{\mathrm{FA}}$) is faster than Lift($\widehat{\mathrm{FA}}$) since queries on the numerical domain greatly reduce the number of nodes being considered.

In terms of precision, the SSA numeric abstraction is always equal or more precise than the standard one. The first metric's specific counts mean little, as variables often have the same value in multiple locations, so a single improvement can be counted multiple times. The second metric does not have this issue and shows that 5 to 10% of outgoing edges have a strict precision improvement in most programs. Note that these metrics say nothing of how big that improvement is.

File	LOC	N	Lift($\widehat{\text{N}}$)	N×FA	Lift($\widehat{\text{N}}$)×FA	FA	Lift($\widehat{\text{FA}}$)	Lift($\widehat{\text{N}}\hat{\times}\widehat{\text{FA}}$)
c00.c	237	57	130 (2.3)	66 (1.16)	125 (2.2)	6 (0.11)	130 (2.3)	136 (2.39)
c02.c	393	87	86 (0.99)	103 (1.18)	100 (1.14)	17 (0.2)	334 (3.82)	81 (0.93)
c04.c	304	13	39 (3.09)	11 (0.9)	41 (3.25)	3 (0.25)	45 (3.54)	40 (3.15)
c07.c	397	12	25 (2.09)	12 (1.05)	27 (2.27)	9 (0.8)	131 (11.1)	27 (2.28)
c18.c	292	84	193 (2.3)	93 (1.11)	207 (2.47)	8 (0.1)	234 (2.79)	180 (2.15)
c23.c	3174	50	348 (7.02)	52 (1.05)	357 (7.2)	90 (1.82)	20.7s (418)	346 (6.98)
c24.c	11076	6.2s	20.4s (3.3)	5.3s (0.86)	19.4s (3.14)	2s (0.33)	>10min	18.6s (3.01)
c29.c	2347	140	276 (1.98)	119 (0.85)	262 (1.88)	99 (0.71)	15.1s (108)	588 (4.21)
c30.c	1178	200	355 (1.77)	189 (0.95)	396 (1.98)	70 (0.35)	8.8s (44.2)	1361 (6.8)

Table 1. Execution time (in milliseconds) of our the analysis of each domain, along with the ratio (time for this domain/time for N). All domains were passed through the query simplification functor Q, which not mentioned in the header. LOC indicates lines of code in each file, as counted by `cloc`.

9.2 Practical experience

The compiling-with-abstract-interpretation method presented in this paper has also been implemented as part of a generic static analysis library named Codex. It is an abstract interpreter that supports not only the whole of C, but also a number of machine code formats (x86, ARM, AMD, RISC-V), notably used in [Nicole et al. 2021, 2022]. This library is a collection of abstract domains which lifts SSA-based numerical abstractions to standard analysis abstractions, whose interface correspond to either the C or machine code language. Most of the code is generic (and related to the memory abstractions, that are not covered in the present paper); the C-specific frontend requires only 3KLOC, and the binary-specific one 4KLOC (excluding the parsers that come from external components).

We have proved that the Lift($\widehat{\text{N}}$) domain is always more precise than the direct numerical analysis, and that it solves the small-code window problem. In practice, this domain is key to the precision of machine code analysis, but is also very often useful when analyzing C. An important feature of the SSA lift is that it is very easy to rewrite SSA expressions to improve precision, which is often needed when analyzing machine code [Djoudi et al. 2016].

One of the main applications of the free algebra domain is that we can automatically produce a simplified program that corresponds to all the traces leading to a remaining alarm or unproved assertion. This SSA program can easily be converted to Constrained Horn Clauses for verification by a goal-oriented software model checker like Spacer [Gurfinkel 2022] to remove these remaining alarms. We found that the simplifications performed by the abstract interpreter are key to help Spacer solve the formula (especially memory reasoning, which is a weak point of SMT solvers). Note, for instance that the nature of the terms in our SSA free algebra domain implies that the generated program is automatically sliced [Weiser 1984] for free.

10 RELATED WORK

Abstract interpretation for compilation, and compiling for abstract interpretation. Using static analyses to perform program transformations is the quintessential job of a compiler; we refer to Cousot and Cousot [2002] for a formal treatment of this subject. Studying how program transformations can affect the precision of an analysis has been comparatively less studied. It is known that functionally equivalent but intensionally different programs may yield different results when analyzed, and thus that program transformation may affect the precision of an analysis [Bruni et al. 2020; Giacobazzi et al. 2015].

One particular instance is the loss of precision induced when analyzing a compiled code compared to its source version. Logozzo and Fähndrich [2008] explains that compiled code analysis is less precise because instructions have a smaller code window: typical low-level instructions are three-address code "$r_i \leftarrow r_j \oplus r_k$" or conditional jumps on the value of a flag register "$\texttt{if}(z)\ \texttt{goto}\ l$", while instructions in source programs can view arbitrary large expressions with statements of the form "$x := e$" or "$\texttt{if}(e)$". They then establish notions of strong completeness, asking whether an analysis can be as precise on the source and binary executable. We prove that our SSA translation and SSA-based non-relational domain is always more precise than the standard non-relational abstract domain, and furthermore fulfills the strong relative completeness property, thus allowing byte-compiled code to be analyzed as precisely as source programs using this domain.

It is common to perform a preprocessing transformation to enhance the precision of static analyses. For instance, Djoudi et al. [2016] undoes compiler transformations to recover high-level conditions from sequences of machine code instructions to help their static analyzer. But often, these program transformations are performed online, during the analysis, so that the transformation can benefit from the invariants computed by the analysis. For instance, Miné [2006] linearizes expression and substitutes variables with their assigned expression; Boillot and Feret [2023] transform modular arithmetic to standard arithmetic when possible. In particular, the dynamic expression rewriting domains in MOPSA [Journault et al. 2019], used to simplify the language handled by the lower layers of the analysis, are very similar to the transformation functors of Section 5. Symbolic domains have also been used for the numerical properties that they can infer (e.g. to detect equalities [Chang and Leino 2005; Kildall 1973; Lemerre 2023]), or as part of an abstract domain (for instance, Gange et al. [2016] propagate non-relational values on terms instead of variables, similarly to our SSA-lift on SSA non-relational domain combination $\mathrm{Lift}(\widehat{\mathrm{N}})$).

Intertwining transformation and analysis. The traditional compiler design as a sequence of passes allows transformations and analyses to help each other. For instance, analyses may help perform register promotion, which will help analyses with a basic representation of memory. These improvements can be done in a fixed-point until maximum precision is reached. However, this will not be as precise as doing all the analyses and transformations simultaneously [Click and Cooper 1995], and transformations are often grouped to gain precision; sparse conditional constant propagation [Wegman and Zadeck 1991] is a prime example of this. Abstract interpretation provides systematic methods to combine analyses [Cousot and Cousot 1979], such as reduced products, which allows implementing these combinations while maintaining a modular code base. In practice, reduced products are implemented by having each analysis communicate through common abstractions (called communication channels in Astrée [Cousot et al. 2006]). Another method for combining analyses is the exchange of program transformations [Lerner et al. 2002]; a consequence of our work is that this can be viewed as using the free algebra abstract domain (FA) as a communication channel between domains. When the shared program fragment is sea-of-node SSA [Click and Paleczny 1995], as in Rompf [2012], then the communication channel is the free SSA algebra abstract domain.

Interpreters and compiler as (co)algebras on the program expressions. The idea of using an algebra signature over program expressions that can correspond to concrete or abstract semantics has been proposed in the context of structured programs. Our main contribution in this area is to use the standard abstract domain signature to generate programs.

A very inspiring work in this area are the *tagless-final interpreters* of Carette et al. [2009] and Kiselyov [2010]. In this work, the same language signature (for the PCF functional programming language) is used to implement both a concrete interpreter, a compiler, partial evaluation/constant propagation and transformation-passes functors that transform the program to compilation-passing style, a form which is equivalent to SSA [Kelsey 1995]. Our work differs in that our analysis signature corresponds to the abstract semantics rather than the concrete one (i.e. our work could be described as tagless-final abstract interpretation), and targets unstructured imperative programs rather than higher-order functional programs.

It is generally desirable that the structure of an abstract interpreter or compiler mimics (or is derived from) the structure of the concrete interpreter (see e.g. [Bodin et al. 2019; Roşu and Serbanuta 2010]). This enables building the abstract interpreter by composing abstractions of the different concepts of the concrete language [Darais et al. 2017, 2015; Keidel and Erdweg 2019; Sergey et al. 2013]. While in this paper we have composed abstraction using product and functor domains, it would be interesting to combine compiling-with-abstract interpretation with other compositional design of abstract interpreters to produce modular single-pass compilers.

Formal verification of compiler passes. The correctness theorem on our functor domains used as compiler passes is stuttering bisimulation between widening points of the program, which is unusual in the field of semantic-preserving compilers [Appel 2014; Leroy 2009]. There seems to be some benefits to these theorems. Firstly, as you cannot perform, neither in the source nor target language, an infinite number of steps without encountering a widening point, this ensures that the bisimulation between both traces remains synchronized. Secondly, it also allows including infinite traces in the correctness argument. Finally, bisimulation proofs handle non-deterministic program semantics (like the ones used in the paper), unlike the common technique of proving only forward simulation assuming that the target language is deterministic [Leroy 2009].

11 CONCLUSION

Our contributions can be summarized using the following key messages: abstract interpreters can be transformed into compilers by using a free algebra computing terms over the abstract domain signature. Different languages have different abstract signatures, that can be non-standard, like the SSA abstract domain. Functor domains can be seen as compiler passes that all run simultaneously, rather than sequentially. Combining symbolic and semantic analyses can significantly improve precision, both in theory and in practice, as exemplified by our SSA-based non-relational abstract domain.

In future work, it would be interesting to see if lifting functor domains to compiler passes can be an effective method to design compilers, as our practical experience with this method is limited to the compilation of a program to horn clauses and SMT formulas. Note that even if our source and target languages encodes conditional jumps as non-deterministism and guards, it is possible to re-encode the result using standard target languages like LLVM [Lemerre 2023]. An important issue is that abstract domains are usually sound but incomplete, which in the case of a functor used as a compiler pass, means that the pass adds behaviors that are not present in the source program. We believe however that many program transformations and corresponding functors are both sound and complete. It is also possible that behavioral refinement [Dockins 2012], which translates program that go wrong to executable traces, could also fit our framework; however, passes performing choice refinement, i.e. removing determinism from the source language, does not seem to be expressible as abstract interpretation functors. Another interesting direction would be to see if lifting semantic soundness and completeness proofs on functor domains (as done by Jourdan et al. [2015]) to compiler passes could be an effective method to formally implement and verify these passes. Finally, the current analysis is limited to forward analysis. Performing any backwards analysis (like liveness) must be done in a separate stage. It would be interesting to see if this restriction can lifted.

ACKNOWLEDGEMENTS

This research was supported in part by the Agence Nationale de la Recherche (ANR) grant agreement ANR-22-CE39-0014-03 (EMASS project).

DATA-AVAILABILITY STATEMENT

The software that supports Section 9 is available on Zenodo DOI 10.5281/zenodo.10895582 [Lesbre and Lemerre 2024a]. The Codex analyzer is available on www.codex.top [Lemerre et al. 2024].

REFERENCES

Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. 1988. Detecting equality of variables in programs, In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. *ACM-SIGACT Symposium on Principles of Programming Languages*, 1–11. https://doi.org/10.1145/73560.73561

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers.* Cambridge University Press.

Zena M. Ariola and Jan Willem Klop. 1996. Equational Term Graph Rewriting. *Fundamenta Informaticae* 26 (12 1996), 207–240. https://doi.org/10.3233/fi-1996-263401

John Aycock and R. Nigel Horspool. 2000. Simple Generation of Static Single-Assignment Form. In *9th International Conference on Compiler Construction – CC 2000 (LNCS, Vol. 1781)*. Springer, 110–124. https://doi.org/10.1007/3-540-46423-9_8

Gogul Balakrishnan and Thomas W. Reps. 2010. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6 (2010), 23:1–23:84. https://doi.org/10.1145/1749608.1749612

Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. 1999. Revising hull and box consistency, In Logic Programming: Proceedings of the 1999 International Conference on Logic Programming. *International Conference on Logic Programming*, 230. https://doi.org/10.7551/mitpress/4304.003.0024

Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

Jérôme Boillot and Jérôme Feret. 2023. Symbolic Transformation of Expressions in Modular Arithmetic. (10 2023), 84–113. https://doi.org/10.1007/978-3-031-44245-2_6

François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings, In Formal Methods in Programming and Their Applications, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.). *Formal Methods in Programming and Their Applications*, 128–141. https://doi.org/10.1007/bfb0039704

Marc M. Brandis and Hanspeter Mössenböck. 1994. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1684–1698. https://doi.org/10.1145/197320.197331

Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *22nd International Conference on Compiler Construction (CC 2013)*. https://doi.org/10.1007/978-3-642-37051-9_6

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. 2020. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.* 4, POPL (2020), 28:1–28:28. https://doi.org/10.1145/3371096

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509–543.

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. *Abstract Interpretation with Alien Expressions and Heap Structures*. Springer, 147–163. https://doi.org/10.1007/978-3-540-30579-8_11

Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (1995), 181–196. https://doi.org/10.1145/201059.201061

Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. *ACM Sigplan Notices* 30, 3 (3 1995), 35–49. https://doi.org/10.1145/202529.202534

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints *(POPL 1977)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *6th ACM Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL 1979)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot and Radhia Cousot. 2002. Systematic design of program transformation frameworks by abstract interpretation. In *29th Symposium on Principles of Programming Languages (POPL 2002)*, John Launchbury and John C. Mitchell (Eds.). ACM, 178–190. https://doi.org/10.1145/503272.503290

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Revised Selected Papers from the 11th Asian Computing Science Conference on Advances in Computer Science - Secure Software and Related Issues – ASIAN 2006 (Lecture Notes in Computer Science, Vol. 4435)*. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

David Darais, Matthew Might, and David Van Horn. 2015. Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA , USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 552–571. https://doi.org/10.1145/2814270.2814308

Delphine Demange, Yon Fernández de Retana, and David Pichardie. 2018. Semantic reasoning about the sea of nodes, In Proceedings of the 27th International Conference on Compiler Construction, Christophe Dubach and Jingling Xue (Eds.). *International Conference on Compiler Construction*, 163–173. https://doi.org/10.1145/3178372.3179503

Adel Djoudi, Sébastien Bardin, and Éric Goubault. 2016. Recovering High-Level Conditions from Binary Programs. In *: 21st International Symposium on Formal Methods (FM 2016)*. 235–253. https://doi.org/10.1007/978-3-319-48989-6_15

Robert W Dockins. 2012. *Operational refinement for compiler correctness.* Ph. D. Dissertation. Princeton University.

Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-Safe Modular Hash-Consing. *ML Workshop* (9 2006), 12–19. https://doi.org/10.1145/1159876.1159880

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Sø ndergaard, and Peter J. Stuckey. 2016. An Abstract Domain of Uninterpreted Functions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2016)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 85–103. https://doi.org/10.1007/978-3-662-49122-5_4

Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India , January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 261–273. https://doi.org/10.1145/2676726.2676987

Philippe Granger. 1992. Improving the results of static analyses of programs by local decreasing iterations. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, Springer Berlin Heidelberg, 68–79. https://doi.org/10.1007/3-540-56287-7_95

Sumit Gulwani and George C. Necula. 2004. A Polynomial-Time Algorithm for Global Value Numbering. In *Static Analysis Symposium (SAS 2004)*, Roberto Giacobazzi (Ed.). Springer, 212–227. https://doi.org/10.1007/978-3-540-27864-1_17

Arie Gurfinkel. 2022. Program Verification with Constrained Horn Clauses (Invited Paper). In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 19–29. https://doi.org/10.1007/978-3-031-13185-1_2

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified C static analyzer. *ACM SIGPLAN Notices* 50, 1 (2015), 247–259. https://doi.org/10.1145/2676726.2676966

Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2019. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *11th International Conference on Verified Software Theories, Tools, and Experiments - Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12031)*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-030-41600-3_1

Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, Michael D. Ernst (Ed.). ACM, 13–23. https://doi.org/10.1145/202529.202532

Gary A Kildall. 1973. A unified approach to global program optimization. In *1st annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages (POPL 1973)*. https://doi.org/10.1145/512927.512945

Oleg Kiselyov. 2010. Typed Tagless Final Interpreters. In *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures (Lecture Notes in Computer Science, Vol. 7470)*, Jeremy Gibbons (Ed.). Springer, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 7, Article 65 (1 2023), 30 pages. https://doi.org/10.1145/3571258

Matthieu Lemerre, Julien Simonnet, Olivier Nicole, Dorian Lesbre, Iker Canut, Corentin Gendreau, and Guillaume Girol. 2024. The Codex semantic library. https://github.com/codex-semantics-library/codex. Version 1.0-beta.

Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, John Launchbury and John C. Mitchell (Eds.).

ACM, 270–282. https://doi.org/10.1145/503272.503298

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Dorian Lesbre and Matthieu Lemerre. 2024a. Compiling with Abstract Interpetation: Artifact. https://doi.org/10.5281/zenodo.10895582

Dorian Lesbre and Matthieu Lemerre. 2024b. Compiling with Abstract Interpetation *(PLDI 2024)*. Article 162. https://doi.org/10.1145/3656392

Francesco Logozzo and Manuel Fähndrich. 2008. *On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis*. Springer, Berlin, Heidelberg, 197–212. https://doi.org/10.1007/978-3-540-78791-4_14

Laurent D. Michel and Pascal Van Hentenryck. 2012. Constraint Satisfaction over Bit-Vectors. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada , October 8-12, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7514)*, Michela Milano (Ed.). Springer, 527–543. https://doi.org/10.1007/978-3-642-33558-7_39

A. Miné. 2004. *Weakly relational numerical abstract domains*. Ph. D. Dissertation. École Polytechnique. http://www.di.ens.fr/~mine/these/these-color.pdf.

Antoine Miné. 2006. Symbolic methods to enhance the precision of numerical abstract domains. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*. Springer, 348–363. https://doi.org/10.1007/11609773_23

Antoine Miné. 2012. Abstract domains for bit-level machine integer and floating-point operations. In *WING'12 - 4th International Workshop on invariant Generation*. Manchester, United Kingdom, 16. https://doi.org/10.29007/b63g

Antoine Miné. 2017. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages* 4, 3-4 (2 2017), 120–372. https://doi.org/10.1561/2500000034

Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. 2021. No Crash, No Exploit: Automated Verification of Embedded Kernels. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*. 27–39. https://doi.org/10.1109/RTAS52030.2021.00011

Olivier Nicole, Matthieu Lemerre, and Xavier Rival. 2022. Lightweight Shape Analysis Based on Physical Types. In *23rd International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2022 (Lecture Notes in Computer Science, Vol. 13182)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 219–241. https://doi.org/10.1007/978-3-030-94583-1_11

Fabrice Rastello and Florent Bouchez Tichadou (Eds.). 2022. *SSA-based Compiler Design*. Springer.

Tiark Rompf. 2012. *Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming*. Ph. D. Dissertation. École Polytechnique Fédérale de Lausanne.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations, In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. *ACM-SIGACT Symposium on Principles of Programming Languages*, 12–27. https://doi.org/10.1145/73560.73562

Grigore Roşu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79 (2010), 397–434. https://api.semanticscholar.org/CorpusID:13756844

Sigurd Schneider. 2013. Semantics of an intermediate language for program transformation. *preparation. Master's Thesis. Universität des Saarlandes* (2013).

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic abstract interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 399–410. https://doi.org/10.1145/2491956.2491979

Vugranam C. Sreedhar and Guang R. Gao. 1995. A Linear Time Algorithm for Placing phi-nodes. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*, Ron K. Cytron and Peter Lee (Eds.). 62–73. https://doi.org/10.1145/199448.199464

Arnaud Venet. 1996. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Proceedings of the Third International Symposium on Static Analysis (SAS '96)*. Springer-Verlag, London, UK, 366–382. https://doi.org/10.1007/3-540-61739-6_53

Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 254–265. https://doi.org/10.1109/CGO53902.2022.9741267

Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (4 1991), 181–210. https://doi.org/10.1145/103135.103136

Mark D. Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, :SERIES: PLDI 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

$$\forall\, \ell\ (s^\#_n),\quad \text{the sequence } t^\#_{n+1} \triangleq \mathrm{D}.\mathit{widen}(\ell,\ t^\#_n,\ s^\#_n) \text{ stabilizes in finite time} \qquad (\textsc{WidenValid})$$

$$\mathrm{D}.\gamma(\mathrm{D}.\mathit{entry}) \subseteq \mathrm{CS}.\mathit{entry} \qquad (\textsc{EntryComplete})$$

$$\forall\, R\, s^\#,\quad \mathrm{D}.\gamma(\mathrm{D}.\mathit{apply}(R,\ s^\#)) \subseteq \mathrm{CS}.\mathit{apply}(R,\ \mathrm{D}.\gamma(s^\#)) \qquad (\textsc{ApplyComplete})$$

$$\forall\, S^\#,\quad \mathrm{D}.\gamma(\mathrm{D}.\mathit{join}(S^\#)) \subseteq \mathrm{CS}.\mathit{join}\left\{\mathrm{D}.\gamma(s^\#) \,\middle|\, s^\# \in S^\#\right\} \qquad (\textsc{JoinComplete})$$

$$\forall\, \ell\, s^\#\, t^\#,\quad \mathrm{D}.\gamma(\mathrm{D}.\mathit{widen}(\ell,\ s^\#,\ t^\#)) \subseteq \mathrm{D}.\gamma(t^\#) \qquad (\textsc{WidenComplete})$$

Fig. 13. Extra hypotheses on an abstract domain D

A PROOFS

A.1 Proofs: notations and background

Figure 13 formally presents hypotheses on a domain D that were only described in the paper. Widening should always lead to convergence: we say a sequence $(s^\#_n)$ *stabilizes* when it is constant after a certain time: $\exists\, m,\ \forall\, n,\ n \geqslant m \implies s^\#_m = s^\#_n$[10]. A domain's soundness hypotheses were presented in the top right of Figure 4, here we also state their complementary completeness hypotheses.

For abstract interpretation, we want domains to be sound (so that our analysis says something meaningful about all program behaviors). For compilation, we want domains to be complete (so that the compiled program behaviors are included in the source behaviors).

Let us take a domain D.

Lemma A.1 (Termination). *Assuming D satisfies* WSound *and* WidenValid, `analyse`(D) *reaches a fixed point in a finite number of steps.*

Proof. There are a finite number of widening points in W, which each stabilize in finite time. By WidenValid. Thus, they are all constant after the maximum of their stabilization times. After that, there are no loops in the program, so the remaining points also stabilize in finite time (bounded by longest path). □

In the rest of this section, we write $p^\#_\infty \triangleq \texttt{analyse(D)}$, and $p^\#_n$ the n-th iteration of the fixed point computation in `analyse`. These verify $p^\#_\infty = p^\#_\infty \nabla_W \mathcal{F}_g(p^\#_\infty)$ and $p^\#_0 = [\ell_0 \mapsto \mathrm{D}.\mathit{entry}]$ and $p^\#_{n+1} = p^\#_n \nabla_W \mathcal{F}_g(p^\#_n)$.

Lemma A.2 (Soundness). *If D is sound, then for all reachable pairs (σ, ℓ), we have $\sigma \in \mathrm{D}.\gamma(p^\#_\infty(\ell))$.*

Proof. By induction on the shortest path $(\sigma_0, \ell_0) \longrightarrow^*_{\mathcal{G}} (\sigma, \ell)$ that makes (σ, ℓ) reachable.

If that path is empty then $\sigma = \sigma_0$ and $\ell = \ell_0$. Therefore, we have $p^\#_\infty(\ell) = \mathrm{D}.\mathit{entry}$ and, by EntrySound, $\Sigma \subseteq \mathrm{D}.\gamma(p^\#_\infty(\ell))$.

For the recursive case we have $(\sigma_0, \ell_0) \longrightarrow^*_{\mathcal{G}} (\sigma_n, \ell_n) \longrightarrow_{\mathcal{G}} (\sigma, \ell)$ and $\sigma_n \in \mathrm{D}.\gamma(p^\#_\infty(\ell_n))$. We write $R_0, \ldots, R_n$ the respective edge relations on this path. We can show that $\sigma \in \mathrm{D}.\gamma(\mathcal{F}_g(p^\#_\infty)(\ell))$. Consider $\mathcal{F}_g(p^\#_\infty)(\ell)$, it is the $\mathrm{D}.\mathit{join}$ of a set. That set contains $\mathrm{D}.\mathit{apply}(R_0,\ p^\#_\infty(\ell_n))$ since

- $\mathrm{D}.\gamma(p^\#_\infty(\ell_n))$ is non-empty by induction, so $p^\#_\infty(\ell_n) \neq \bot$
- $\mathcal{G}(\ell_n, R_n, \ell)$ holds by $(\sigma_n, \ell_n) \longrightarrow_{\mathcal{G}} (\sigma, \ell)$.

[10]Often in abstract interpretation, we only want to reach a post fixed point instead of a true fixed point. See Appendix B for details on how to do so with this framework.

- the D.*apply* is also not $\bot$ else this would contradict the soundness of D.*apply*, (we know D.γ(D.*apply*(R_n, $p^\#_\infty(\ell_n)$)) should contain σ as $(\sigma_n, \ell_n) \to_{\mathcal{G}} (\sigma, \ell)$ implies $\mathcal{R}[\![R_n]\!]\,(\sigma_n, \sigma)$, then use ApplySound).

The join set thus contains D.*apply*(R_n, $p^\#_\infty(\ell_n)$), and the soundness of D.*join* and D.*apply* suffice to show $\sigma \in$ D.$\gamma(\mathcal{F}_g(p^\#_\infty)(\ell))$.

If ℓ is not a widening point, then we have $p^\#_\infty(\ell) = \mathcal{F}_g(p^\#_\infty)(\ell)$ and the result is shown. In the other case, $p^\#_\infty(\ell) =$ D.*widen*(ℓ, $p^\#_\infty(\ell)$, $\mathcal{F}_g(p^\#_\infty)(\ell)$) so the soundness of D.*widen* is sufficient to conclude. □

Lemma A.3 (completeness). *If D is complete, then for all pairs (σ, ℓ), we have $\sigma \in D.\gamma(p^\#_\infty(\ell)) \Rightarrow (\sigma, \ell)$ reachable.*

Proof. We show the property holds for all $p^\#_n$ by induction on n.

It is true at $n = 0$ since $p^\#_0 = [\ell_0 \mapsto$ D.*entry*] (by EntryComplete).

Suppose it is true of $p^\#_n$, then since $p^\#_{n+1} = p^\#_n \nabla_W \mathcal{F}_g(p^\#_n)$ and since ∇_W and $\mathcal{F}_g$ only use complete operations, it remains true. □

A.2 Proofs: Free algebra of the domain signature

Proof of Theorem 4.1:

Theorem 4.1. *When $p^\# =$ `analyse`(FA), $\mathcal{G}_{p^\#}$ is isomorphic to $\mathcal{G}$ (restricted to reachable locations, i.e. locations ℓ such that there is a path from ℓ_0 to ℓ in $\mathcal{G}$) via $p^\#$:*

- *$p^\#$ is injective (restricted to reachable locations)*
- $\mathcal{G}_{p^\#} = \left\{(p^\#(\ell), R, p^\#(\ell')) \,\middle|\, \mathcal{G}(\ell, R, \ell') \wedge \ell \text{ reachable}\right\}$

Proof. For the injectivity of $p^\#$, we reason by structural induction on $\mathtt{s}^\#$, showing the following property: "for all ℓ and ℓ' such that $p^\#(\ell) = p^\#(\ell') = \mathtt{s}^\#$, we have $\ell = \ell'$". This is shown largely by inspecting the fixpoint equality $p^\# = p^\# \nabla_W \mathcal{F}_g(p^\#)$ and unfolding $\mathcal{F}_g$ and ∇_W.

- Case $\mathtt{s}^\# = \mathtt{Entry}$, the only operation that introduces such a term is the FA.*entry* in $\mathcal{F}_g$, which is only ever applied at ℓ_0, so $\ell = \ell' = \ell_0$.
- Case $\mathtt{s}^\# = \mathtt{Loc}(\ell'')$, the only operation that introduces a `Loc` is FA.*widen*(, „ w)hich is only ever applied at the point it is called, so $\ell = \ell' = \ell''$.
- Case $\mathtt{s}^\# = \mathtt{Apply}(R, \mathtt{t}^\#)$: by unfolding $\mathcal{F}_g$, we know $\mathtt{t}^\#$ must be in $\mathtt{img}\,p^\#$, so we can apply the induction hypothesis there. Conclude using the requirement that outgoing edges of Imp nodes are uniquely labelled.
- The `Join` case is similar to the `Apply` case, just pick any element of the non-empty set: it must be an `Apply` by inversion of $\mathcal{F}_g$, so we can reuse the same reasoning.

Now for the second point.

Consider an edge $\mathcal{G}(\ell, R, \ell')$, assuming ℓ is reachable. Since the analysis is sound, this implies FA.$\gamma(p^\#(\ell)) \neq \emptyset$ and therefore $p^\#(\ell) \neq \bot$.

We can show that $p^\#(\ell) \overset{R}{\longmapsto}_\# \mathcal{F}_g(p^\#)(\ell')$ holds. By definition of $\mathcal{F}_g$, the right-hand side is a FA.*join* of FA.*apply* of the predecessors of ℓ', and ℓ is such a predecessor. The join is thus not empty, so it is either the single element `Apply`(R, $p^\#(\ell)$) or the `Join` of a set containing it. Conclude by TApply and TJoin.

Since $p^\# = p^\# \nabla_W \mathcal{F}_g(p^\#)$, we have two cases, either we didn't widen at ℓ' ($p^\#(\ell') = \mathcal{F}_g(p^\#)(\ell')$), and thus $p^\#(\ell) \overset{R}{\longmapsto}_\# p^\#(\ell')$ holds directly, or we performed a widening, and the same holds via TLoc. This result is sufficient to show that $\mathcal{G}_{p^\#}(p^\#(\ell), R, p^\#(\ell'))$ holds using VBase and GraphGen.

Consider now an edge $\mathcal{G}_{p^\#}(\mathtt{s}^\#, R, \mathtt{t}^\#)$. By definition of $\mathcal{G}_{p^\#}$, $V(\mathtt{t}^\#)$ holds, so by case disjunction:

- VBase case: there is some ℓ' such that $\mathtt{t}^\# = p^\#(\ell')$. We have $\mathtt{s}^\# \xmapsto{R}_\# \mathcal{F}_g(p^\#)(\ell')$ since either ℓ' isn't a widening point (and so $p^\#(\ell') = \mathcal{F}_g(p^\#)(\ell')$) or it is, in which case we use TLoc. $\mathcal{F}_g(p^\#)(\ell')$ isn't empty since it has an incoming transition. So it is either a `Join` of `Apply` of predecessors of ℓ' or a single such `Apply`. Either way (optionally using TJoin), we can show that there exists a predecessor ℓ such that $\mathtt{s}^\# \xmapsto{R}_\# \mathtt{Apply}(R,\ p^\#(\ell))$. Inverting TApply allows us to conclude $\mathtt{s}^\# = p^\#(\ell)$ and that $\mathcal{G}(\ell, R, \ell')$.
- VRec case: in the previous case, we have shown that if $\mathtt{s}^\# \xmapsto{R}_\# p^\#(\ell')$ then there exists ℓ such that $\mathtt{s}^\# = p^\#(\ell)$. Thus by immediate recursion, $V(\mathtt{t}^\#)$ implies $\mathtt{t}^\# \in \mathtt{img}\, p^\#$, so we can always use the base case. □

Notice that in this proof, TSelf cannot occur since we never simplify edges, and VRec is unused as all points that would be added by it are already true by VBase. These extra constructs are only useful when dealing with transformation functors (Section 5) which can add extra intermediate states (which require VRec to be included) or remove no-op relations (which may require TSelf to avoid deleting loops).

A.3 Proofs: Transformation functors as compiler passes

Proof of Lemma 5.4:

Lemma 5.4 (Functor soundness and completeness). *A transformation functor F is sound if and only if F(CS) is sound. Similarly, F is complete if and only if F(CS) is complete.*

Proof. We start by the soundness proof.

The direct implication is trivial, since CS is sound.

For the reciprocal: F(CS) sound $\Rightarrow$ ($\forall$D, D sound $\Rightarrow$ F(D) sound), let us take a sound domain D. We only have to show ApplySound in F(D), since all other operations are equal to those of D.

Let us take $R \in \mathbb{R}$ and $s^\# \in \mathrm{D}.\Sigma^\#$. We know F(D).*apply*($R$, $s^\#$) is a composition of D.*apply*, D.*join*, and $s^\#$. (That composition may depend on R, or on $s^\#$ through queries). We write it as D.Composition($s^\#$).

- By soundness of D (and immediate induction on Composition): CS.Composition(D.$\gamma(s^\#)$) $\subseteq$ D.γ(D.Composition($s^\#$)).
- We can show $\left\{\sigma' \in \Sigma \,\middle|\, \exists \sigma \in \gamma(s^\#), \mathcal{R}[\![R]\!](\sigma, \sigma')\right\} \subseteq$ CS.Composition(D.$\gamma(s^\#)$) This is trivial if there are no queries (Composition only depends on R). Since any query that holds on $s^\#$ will also hold on $\gamma(s^\#)$. (as the collecting semantics domain is the most precise domain).
- Thus: F(CS).*apply*(R, D.$\gamma(s^\#)$) $\subseteq$ F(D).*apply*(R, $s^\#$)
- Conclude by soundness of F(CS).

The completeness proof is similar: it only inverts the set inclusions. □

Proof of Theorem 5.5: The proof requires modifying the graph generation rules a bit. Specifically, we need to replace TJoin by two new rules:

$$\textsc{TJoinElide}\quad \frac{\mathtt{s}^\# \xmapsto{R}_\# \mathtt{t}^\# \qquad \mathtt{t}^\# \in \mathtt{S}^\# \qquad \mathtt{t}^\# \notin \mathtt{img}\, p^\#}{\mathtt{s}^\# \xmapsto{R}_\# \mathtt{Join}(\mathtt{S}^\#)} \qquad\qquad \textsc{TJoinNoElide}\quad \frac{\mathtt{t}^\# \in \mathtt{S}^\# \qquad \mathtt{t}^\# \in \mathtt{img}\, p^\#}{\mathtt{t}^\# \xmapsto{\text{If } 1}_\# \mathtt{Join}(\mathtt{S}^\#)}$$

The problem with TJoin is that, when using transformation functors, an element $\mathtt{t}^\#$ that is both in $\mathtt{img}\, p^\#$ and in a join, since we can now remove trivial applies. This would break our graph generation, duplicating all edges to $\mathtt{t}^\#$ (they will also go into the join), and making $\mathtt{t}^\#$ a local sink.

One solution to this is to remove $\mathtt{t}^\#$ from the new graph, but then we may lose the simulation (as no point equivalent to $\mathtt{t}^\#$ exists in the new graph, there is only the join, which has more parents).

The solution chosen here is to only carry over to the join the edges that don't lead into the `img` $p^\#$ (TJoinElide), and add an explicit edge for the others (TJoinNoElide). Note that with these rules, Theorem 4.1 still holds (as TJoinNoElide is never used when all applys introduce terms). We chose to only present TJoin in the main paper in an attempt at keeping the presentation simple.

In order to prove Theorem 5.5, we start by proving a stronger, more technical simulation result:

Lemma A.4. *If F is a sound transformation functor, then for all reachable pairs (ℓ, σ) and (ℓ', σ') such that $(\ell, \sigma) \to_{\mathcal{G}} (\ell', \sigma')$, we have $(p^\#(\ell), \sigma) \to^*_{\mathcal{G}_{p^\#}} (p^\#(\ell'), \sigma')$.*

Proof. Consider a transition $(\ell, \sigma) \to_{\mathcal{G}} (\ell', \sigma')$. Let R be the associated transition. By definition of $\to_{\mathcal{G}}$, we know that $\mathcal{R}[\![R]\!](\sigma, \sigma')$ holds.

Since FA is sound, F(FA) is also sound. Thus, by Lemma A.2, $\sigma \in \mathrm{F(FA)}.\gamma(p^\#(\ell))$.

Since F is a transformation functor, $\mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell))$ is a combination of $\mathrm{FA}.\mathit{apply}$, $\mathrm{FA}.\mathit{join}$ and $p^\#(\ell)$, That is to say, it is a free algebra term made of `Apply`, `Join` and $p^\#(\ell)$. Looking at the rules for our edge predicate, we can show that such a term represents a DAG between a unique source ($p^\#(\ell)$) and a unique sink (the full term).

By soundness of F, $\mathrm{FA}.\gamma(\mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell)))$ must contain σ'. Thus, looking at the concretization of the DAG term, we can show by induction on the DAG term, that there must exist a path with intermediate states $\mathtt{s}^\#_1..\mathtt{s}^\#_{n-1}$ and relations $R_1..R_n$ such that $p^\#(\ell) \xmapsto{R_1}_\# \mathtt{s}^\#_1 \xmapsto{\cdots}_\# \ldots \xmapsto{\cdots}_\# \mathtt{s}^\#_{n-1} \xmapsto{R_n}_\# \mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell))$. and $(\mathcal{R}[\![R_1]\!]\,;..;\mathcal{R}[\![R_n]\!])(\sigma, \sigma')$ (where $R_1;R_2$ is relation composition: $(R_1;R_2)(a, b) \triangleq \exists\, c,\ R_1(a, c) \wedge R_2(c, b)$)

- if that term is just $p^\#(\ell)$, then our path is a single vertex. Since the apply was removed, we know by soundness that R is no-op on $p^\#(\ell)$, thus $\sigma = \sigma'$ (the empty relation composition is equality).
- if that term is $\mathtt{Apply}(R_1,\ \mathtt{s}^\#_1)$ we know there exists $\sigma'' \in \mathrm{FA}.\gamma(\mathtt{s}^\#_1)$ such that $\mathcal{R}[\![R_1]\!]\ (\sigma'', \sigma')$ (by unfolding of $\sigma' \in \mathrm{FA}.\gamma(\mathtt{Apply}(R_1,\ \mathtt{s}^\#_1))$).
  Apply the induction hypothesis on σ'' to get a path from $p^\#(\ell)$ to $\mathtt{s}^\#_1$, then extend that path by one step using TApply.
- if that term is a join, we know the concretization of a join is the union of the concretization of its elements, so we can apply the induction hypothesis directly on the relevant element, then use TJoinElide or TJoinNoElide to turn the last path step into a step which leads into the full join instead of the single element.
  Notice that we can only use TJoinElide if the path yielded by the induction hypothesis has non-zero length. However, if the path has length 0 then the element is $p^\#(\ell)$, so TJoinNoElide. This is the proof step that fails when only using TJoin, as it also needs a non-zero length path.

That last step can be turned into $\mathtt{s}^\#_{n-1} \xmapsto{R_n}_\# p^\#(\ell')$ via TJoinElide or TJoinNoElide (if $\mathcal{F}_g(p^\#)(\ell')$ is a join) and TLoc (if ℓ' is a widening point).

All elements of the path satisfy V : the last point by VBase and all previous ones by VRec.

This is enough to show $(p^\#(\ell), \sigma) \to^*_{\mathcal{G}_{p^\#}} (p^\#(\ell'), \sigma')$ since $\mathcal{G}_{p^\#}$ we have shown the edge and vertex predicates hold along the given path. ◻

Theorem 5.5 (Sound functor forward simulation). *If F is a sound transformation functor, then for all reachable pairs (ℓ, σ) and (ℓ', σ') such that ℓ and ℓ' are the entrypoint or widening points:*

$$(\ell, \sigma) \to^+_{\mathcal{G}} (\ell', \sigma') \quad \Rightarrow \quad (p^\#(\ell), \sigma) \to^+_{\mathcal{G}_{p^\#}} (p^\#(\ell'), \sigma')$$

Proof. Apply Lemma A.4 as many times as the initial path is long to get a path $(p^\#(\ell),\,\sigma) \to^*_{\mathcal{G}_{p^\#}} (p^\#(\ell'),\,\sigma')$. If the obtained path has non-zero length, then the result is shown. If its length is 0, then $p^\#(\ell) = p^\#(\ell')$ and $\sigma = \sigma'$. It is easy to show $p^\#$ is injective when limited to entry or widening points as such terms are constructed with either `Entry` or `Loc`. Thus $\ell = \ell'$. This case is why we have the TSelf rule, it gives us a special edge to capture the fact that ℓ is a direct predecessor of itself. □

Proof of Theorem 5.6: In order to prove Theorem 5.6, we start by proving a stronger, more technical simulation result:

Lemma A.5. *If F is complete, then for all pairs $(\ell,\,\sigma)$ and $(\ell',\,\sigma')$ such that ℓ is ℓ_0 or a widening point and $(p^\#(\ell),\,\sigma) \to^*_{\mathcal{G}_{p^\#}} (p^\#(\ell'),\,\sigma')$, then we have $(\ell,\,\sigma) \to^*_{\mathcal{G}} (\ell',\,\sigma')$.*

Furthermore, the right path has length 0 only if the left path also has length 0.

Proof sketch. Proceed by strong induction on the path $(p^\#(\ell),\,\sigma) \to^*_{\mathcal{G}_{p^\#}} (p^\#(\ell'),\,\sigma')$. Using $p^\# = p^\# \nabla_W \mathcal{F}_g(p^\#)$, and unfolding $\mathcal{F}_g$, we can obtain a direct predecessor ℓ'' of ℓ that must be on this path. We can use the induction hypothesis to get from ℓ to ℓ'', followed by completeness to get from ℓ'' to ℓ'.

There are some subtle cases when $p^\#(\ell'')$ is equal to $p^\#(\ell)$ (or $p^\#(\ell')$), but $\ell'' \neq \ell$ or $\ell'' \neq \ell'$ (i.e. when successive nodes of $\mathcal{G}$ have been merged into one). We have still made progress in those cases since we cannot go through a location twice (as that would imply a loop in the source program, which would lead to a new widening point). □

Proof. Consider a transition path $(p^\#(\ell),\,\sigma) \to^*_{\mathcal{G}_{p^\#}} (p^\#(\ell'),\,\sigma')$, with $(\ell,\,\sigma)$ reachable and ℓ the entrypoint or a widening point. Unfolding $\to_{\mathcal{G}_{p^\#}}$, we obtain a path $p^\#(\ell) \xmapsto{R_1}_\# \mathsf{s}^\#_1 \xmapsto{\cdots}_\# \ldots \xmapsto{\cdots}_\# \mathsf{s}^\#_{n-1} \xmapsto{R_n}_\# p^\#(\ell')$. We show the result for when no widening point appears on this path. If one does, split the path where it appears, and apply the result to each segment, then compose the resulting paths.

We proceed by strong induction on this path.

If the path is empty we have $p^\#(\ell) = p^\#(\ell')$ and $\sigma = \sigma'$. We distinguish two cases:

- If $\ell = \ell'$ the result is immediate (empty path)
- Else $\ell \neq \ell'$.
  ℓ' is not a widening point. This is because otherwise $p^\#(\ell') = \mathtt{Loc}(\ell')$ and since ℓ is either the entrypoint (`Entry`) or a widening ($\mathtt{Loc}(\ell)$), we would get $\ell = \ell'$ from $p^\#(\ell) = p^\#(\ell')$.
  Since ℓ' is not a widening point, we know that $p^\#(\ell') = \mathcal{F}_g(p^\#)(\ell')$. We also know $p^\#(\ell')$ is either `Entry` or $\mathtt{Loc}(\ell)$ since it is equal to $p^\#(\ell)$.
  Unfolding $\mathcal{F}_g$, we can see that ℓ' must have a unique (non $\bot$) predecessor ℓ'' through a relation R (since it isn't a `Join`) and $\mathrm{F(FA)}.\mathit{apply}(R,\,p^\#(\ell'')) = \mathtt{Loc}(\ell)$.
  As a transformation functor, F(FA).*apply* can't introduce the `Loc`, so it must be its argument: $p^\#(\ell'') = p^\#(\ell)$. Thus, we have an empty path, between argument and F(FA).*apply*. Completeness therefore implies that $\mathcal{R}[\![R]\!]$ contains the empty relation composition, equality.
  Therefore, for all σ' we have $(\ell'',\,\sigma') \to_{\mathcal{G}} (\ell',\,\sigma')$, meaning it is sufficient to show we have a path ending in $(\ell'',\,\sigma')$ since that path can then be prolonged to reach ℓ'.
  We can then repeat the disjunction on $\ell'' = \ell$ using the same reasoning. This terminates because (1) there are a finite number of program locations (2) we cannot go through the same location twice (else there would be a loop, which would lead to a widening point). So at some point, we will get to ℓ.

For the recursive case, consider the last step $\mathsf{s}^\#_{n-1} \xmapsto{R_n}_\# p^\#(\ell')$. We would like to show $\mathsf{s}^\#_{n-1} \xmapsto{R_n}_\# \mathcal{F}_g(p^\#)(\ell')$, as this allows reasoning about the pre-state.

- This is immediate if ℓ' is not a widening.
- If $p^\#(\ell')$ is $\mathsf{Loc}(\ell')$ and the last edge predicate comes from TLoc, the result is true since it is the premise of TLoc.
- Else $p^\#(\ell')$ is $\mathsf{Loc}(\ell')$ and the last edge predicate comes from TSelf. This implies the path has length one; its relation is If 1 (so $\sigma = \sigma'$); the start and end match $\ell = \ell'$; and $\mathcal{F}_g(p^\#)(\ell)$ is a $\mathsf{Join}(\mathsf{S}^\#)$ with $p^\#(\ell) \in \mathsf{S}^\#$.

  In that case, since $\mathcal{F}_g(p^\#)$ is a join that contains $p^\#(\ell)$, there is a looping path from ℓ to itself in $\mathcal{G}$. Furthermore, since the relation on this path have been erased by the functor, completeness implies they are all trivial. Thus, this circular path in $\mathcal{G}$ proves the lemma in this case.

For all remaining cases, we have $\mathsf{s}^\#_{n-1} \xmapsto{R_n}_\# \mathcal{F}_g(p^\#)(\ell')$. The right term is not $\bot$ or Entry (as it has an incoming edge), thus by unfolding $\mathcal{F}_g$ and F(FA).*join*, it is either a Join of an F(FA).*apply* of the predecessors, or directly an F(FA).*apply* of the unique predecessor. Either way (use TJoinElide or TJoinNoElide in the first case), we know there exists a ℓ'', predecessor of ℓ' through a relation R such that $\mathsf{s}^\#_{n-1} \xmapsto{R_n}_\# \mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell''))$

Since the path contains no Locs (as there are no widening points on it), it implies that $p^\#(\ell)$ is a subterm of $\mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell''))$ (It is easy to show that, when only using TApply, TJoinElide and TJoinNoElide, we have $\mathsf{s}^\# \xmapsto{R}_\# \mathsf{t}^\# \Rightarrow \mathsf{s}^\#$ subterm of $\mathsf{t}^\#$). Furthermore, by definition of transformation functors, $p^\#(\ell'')$ is a source subterm of $\mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell''))$.

$p^\#(\ell)$ cannot be a strict super-term of $p^\#(\ell'')$ since it is the entrypoint or a widening point (it has no subterms and cannot be introduced by F(FA).*apply*), thus it is either $p^\#(\ell'')$ or a subterm of it.

- Case $p^\#(\ell) = p^\#(\ell'')$
  - If $\ell = \ell''$, then the chain simplifies to

    $$p^\#(\ell) \xmapsto{R_1}_\# \mathsf{s}^\#_1 \xmapsto{\cdots}_\# \ldots \xmapsto{\cdots}_\# \mathsf{s}^\#_{n-1} \xmapsto{R_n}_\# \mathrm{F(FA)}.\mathit{apply}(R,\ p^\#(\ell))$$

    Using the completeness yields $\mathcal{R}[\![R]\!]\ (\sigma,\ \sigma')$.
  - The case $\ell'' \neq \ell$ is handled similarly to the initialization: ℓ'' isn't a widening, so it has a single parent ℓ''' through a relation R', and $\mathcal{R}[\![R']\!]$ contains equality, so showing the result ending in ℓ''' is sufficient. Repeat the disjunction on ℓ'''. This terminates because there is a finite number of locations, and we cannot loop.
- If $p^\#(\ell)$ is a strict subterm of $p^\#(\ell'')$ then either $p^\#(\ell'')$ appears on the path, or it is equal to the path's last term $p^\#(\ell')$. In the first case, use the recursion hypothesis on the first segment to obtain a path from ℓ to ℓ'', followed by completeness for the (non-empty) path from ℓ'' to ℓ'.

  The second case is the third time we run into the 0-step problem: $p^\#(\ell'') = p^\#(\ell')$ and $\ell'' \neq \ell'$. It is solved in the same way as the first two occurrences: we can show that $(\ell'', \sigma') \rightarrow_{\mathcal{G}} (\ell', \sigma')$ by completion; and that ℓ'' has a parent that satisfies the same hypotheses, and that we will eventually take a non-zero step along our path since there are a finite number of locations, and we cannot loop.

In all cases, we have taken a step in $\mathcal{G}$, so we know the new path will not have length 0. □

Theorem 5.6 (Complete functor backward simulation). *If F is a complete transformation functor, then for all entry or widening points ℓ, ℓ', and for all σ, σ':*

$$(p^\#(\ell),\ \sigma) \rightarrow^+_{\mathcal{G}_{p^\#}} (p^\#(\ell'),\ \sigma') \quad \Rightarrow \quad (\ell,\ \sigma) \rightarrow^+_{\mathcal{G}} (\ell',\ \sigma')$$

Proof. Apply Lemma A.5 as many times as the initial path is long to get a path $(\ell,\ \sigma) \rightarrow^*_{\mathcal{G}} (\ell',\ \sigma')$. Since the initial path is not empty, we also know this new path is not empty. □

A.4 Proofs: Lifting SSA domains to Imp domains

Proof of Theorem 7.1: we start by proving the following technical lemma.

Lemma A.6 (SSA forward simulation). *For all reachable pairs $(\ell,\ \sigma)$ and $(\ell',\ \sigma')$ such that $(\ell,\ \sigma) \rightarrow_{\mathcal{G}} (\ell',\ \sigma')$:*

- *For all $(\Gamma^\#,\ \Gamma)$ such that $C((\ell,\ \sigma),\ (\Gamma^\#,\ \Gamma))$, there exists $(\Gamma'^\#,\ \Gamma')$ such that we have both $C((\ell',\ \sigma'),\ (\Gamma'^\#,\ \Gamma'))$ and $(\Gamma^\#,\ \Gamma) \rightsquigarrow^*_{\hat{\mathcal{G}}_{p^\#}} (\Gamma'^\#,\ \Gamma')$.*
- *There exists $(\Gamma^\#,\ \Gamma)$ such that $C((\ell,\ \sigma),\ (\Gamma^\#,\ \Gamma))$*
- *The $(\Gamma^\#,\ \Gamma) \rightsquigarrow^*_{\hat{\mathcal{G}}_{p^\#}} (\Gamma'^\#,\ \Gamma')$ is of length 0 only when ℓ' has a single predecessor.*

Proof sketch. For existence, $\Gamma^\#$ is uniquely determined by $p^\#(\ell)$ and Γ exists by soundness and definition of $\mathrm{Lift}(\widehat{\mathrm{FA}}).\gamma$. For the other point, proceed by disjunction on $\mathcal{F}_g(p^\#)(\ell')$, which is either the same SSA term $\Gamma^\#$ as in $p^\#(\ell)$, an $\mathtt{Ass\hat{u}me}(e,\ \Gamma^\#)$, or $\mathtt{Jo\hat{i}n}$ containing either $\Gamma^\#$ or $\mathtt{Ass\hat{u}me}(e,\ \Gamma^\#)$. These case yield paths of lengths 0, 1, 1 and 2 respectively. □

Proof. Take $(\ell,\ \sigma)$ and $(\ell',\ \sigma')$ reachable such that $(\ell,\ \sigma) \rightarrow_{\mathcal{G}} (\ell',\ \sigma')$ via a relation R. Since both are reachable, their values through $p^\#$ are not $\bot$ by soundness. Let us denote $(\sigma^\#_0, \Gamma^\#) \triangleq p^\#(\ell)$ and $(\sigma^\#_1, \Gamma'^\#) \triangleq p^\#(\ell')$. Soundness also implies $\sigma \in \mathrm{Lift}(\widehat{\mathrm{FA}}).\gamma((\sigma^\#_0, \Gamma^\#))$ so, by unfolding $\mathrm{Lift}(\widehat{\mathrm{FA}}).\gamma$, we know there exists Γ such that $\forall\, x,\ \hat{\mathcal{E}}[\![\sigma^\#_0(x)]\!](\Gamma) = \sigma(x)$. This proves the second point. For the first, take any Γ that satisfies this property.

We know $\ell' \neq \ell_0$ since it has a predecessor ℓ. Furthermore, if ℓ' is a widening point, we can use TLocSSA to obtain a new $\Gamma'^\#$ with the same transitions which is equal to the one in $\mathcal{F}_g(p^\#)(\ell')$. In the remaining cases, we have $\Gamma'^\#$ equal to the one in $\mathcal{F}_g(p^\#)(\ell')$.

The term built by $\mathcal{F}_g$ is a $\mathrm{Lift}(\widehat{\mathrm{FA}}).$*join* of $\mathrm{Lift}(\widehat{\mathrm{FA}}).$*apply* of the predecessors (which include ℓ). Looking how these map to our SSA operations, we notice four cases: *join* may or may not introduce a $\mathtt{Jo\hat{i}n}$, and *apply* may or may not introduce an $\mathtt{Ass\hat{u}me}$.

- If R is a guard "If e", then an $\mathtt{Ass\hat{u}me}$ is introduced; since the guard holds on (σ, σ'), we also know $\sigma = \sigma'$ and $\mathcal{E}[\![e]\!](\sigma) \neq 0$. Therefore, we also have $\hat{\mathcal{E}}[\![\mathtt{subst}(e,\ \sigma^\#_0)]\!](\Gamma) \neq 0$, which implies that $(\Gamma^\#,\ \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}} (\mathtt{Ass\hat{u}me}(\mathtt{subst}(e,\ \sigma^\#_0),\ \Gamma^\#),\ \Gamma)$.

  If no join is introduced, then we know $\mathtt{Ass\hat{u}me}(\mathtt{subst}(e,\ \sigma^\#_0),\ \Gamma^\#) = \Gamma'^\#$ and $\sigma^\#_0 = \sigma^\#_1$. We set $\Gamma' \triangleq \Gamma$. We have a path of length 1, and the property on Γ' is immediate given the one on Γ.

  If a join is introduced, let B be the set of bindings for the $\Gamma^\#$ branch. We define $\Gamma' \triangleq \mathrm{unbind}_{\hat{\mathcal{G}}_{p^\#}}(\Gamma'^\#, \mathrm{bind}(B, \Gamma))$ and we have $(\Gamma^\#,\ \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}} (\mathtt{Ass\hat{u}me}(e,\ \Gamma^\#),\ \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}_{p^\#}} (\Gamma'^\#,\ \Gamma')$. This is a path of length 2.

  To show the condition on Γ', take a variable x:

  - If x isn't in $\mathrm{dom}\, B$, then its values are equal in all join branches. This means $\sigma^\#_1(x) = \sigma^\#_0(x)$. It also implies that that expression was in scope in all branches and therefore is still in scope at the join. So its evaluation is neither modified by $\mathrm{unbind}_{\hat{\mathcal{G}}_{p^\#}}$ (in scope), nor by $\mathrm{bind}(B, \Gamma)$ (not in $\mathrm{dom}\, B$).

    Thus, $\hat{\mathcal{E}}[\![\sigma^\#_1(x)]\!](\Gamma') = \hat{\mathcal{E}}[\![\sigma^\#_0(x)]\!](\mathrm{unbind}_{\hat{\mathcal{G}}_{p^\#}}(\Gamma'^\#, \mathrm{bind}(B, \Gamma))) = \hat{\mathcal{E}}[\![\sigma^\#_0(x)]\!](\Gamma) = \sigma(x) = \sigma'(x)$

- Otherwise, it is bound to a new variable $x_{\Gamma'^{\#}}$. Thus, it is in scope of $\Gamma'^{\#}$ and not removed by $\mathrm{unbind}_{\hat{\mathcal{G}}_{p^{\#}}}$. Furthermore, B must bind $x_{\Gamma'^{\#}}$ to $\sigma_0^{\#}(x)$, so evaluating it also yields the same value.

- If R is an assignment "$x := e$", no `Assûme` is introduced; and $\sigma' = \sigma\,[x \mapsto \mathcal{E}[\![e]\!](\sigma)]$.
If no join is introduced then we know that $\Gamma'^{\#} = \Gamma^{\#}$ and $\sigma_1^{\#} = \sigma_0^{\#}\left[x \mapsto \mathtt{subst}(e,\ \sigma_0^{\#})\right]$ hold. We set $\Gamma' \triangleq \Gamma$. We have a path of length 0. All that needs to be shown is the property on Γ', which is true for all variables except x trivially, and true on x by compatibility between `subst` and $\mathcal{E}[\![\cdot]\!]$. This is the only case where the path has length 0, and it implies ℓ' only has a single predecessor (no join). Thus, it proves the third point.
If we introduce a join, let B be the set of bindings for the $\Gamma^{\#}$ branch. We define $\Gamma' \triangleq \mathrm{unbind}_{\hat{\mathcal{G}}_{p^{\#}}}(\Gamma'^{\#}, \mathrm{bind}(B, \Gamma))$ and we have $(\Gamma^{\#}, \Gamma) \rightsquigarrow_{\hat{\mathcal{G}}_{p^{\#}}} (\Gamma'^{\#}, \Gamma')$. This is a length 1 path. Showing the condition on Γ' is the same as in the previous case with a join. □

Theorem 7.1 (SSA compilation forward simulation). *For all reachable pairs $(\ell,\ \sigma)$ and $(\ell',\ \sigma')$ such that ℓ and ℓ' are entry or widening points, for all $\hat{s} \in \hat{\mathbb{S}}$ we have:*

$$(\ell,\ \sigma) \rightarrow^{+}_{\mathcal{G}} (\ell',\ \sigma') \ \wedge\ C((\ell,\ \sigma),\ \hat{s}) \ \Rightarrow\ \exists\, \hat{s}' \in \hat{\mathbb{S}},\ \ C((\ell',\ \sigma'),\ \hat{s}') \ \wedge\ \hat{s} \rightsquigarrow^{*}_{\hat{\mathcal{G}}_{p^{\#}}} \hat{s}'$$

Furthermore, there exists an $\hat{s} \in \hat{\mathbb{S}}$ such that $C((\ell,\ \sigma),\ \hat{s})$ holds.
Finally, if $\hat{s} \rightsquigarrow^{*}_{\hat{\mathcal{G}}_{p^{\#}}} \hat{s}'$ has length 0, then ℓ' is not a true loop head (it has a single reachable predecessor).

The condition on ℓ' when paths have length 0 means it is either an extra point added to W (not a loop head in the initial program), or it is the head of a loop that is syntactically never taken (e.g. `while (...) {...; break; }`). When combining SSA translation with transformation functors, it can also be the head of a loop that our analysis proves to be broken before completing the first iteration (e.g. `while (c)` when `c` = 0, or `while (...) {... if (c) break; }` when `c` ≠ 0).

Proof. We apply Lemma A.6 as many times as the input path is long, and then compose the resulting paths. □

Proof of Theorem 7.2: we start by proving the following technical lemma.

Lemma A.7 (SSA backward simulation). *For all paths $(\Gamma^{\#},\ \Gamma) \rightsquigarrow^{*}_{\hat{\mathcal{G}}_{p^{\#}}} (\Gamma'^{\#},\ \Gamma')$ such that $\Gamma^{\#}$ and $\Gamma'^{\#}$ appear in $\mathtt{img}\, p^{\#}$:*

- *For all $(\ell',\ \sigma')$ such that $C((\ell',\ \sigma'),\ (\Gamma'^{\#},\ \Gamma'))$, there exists $(\ell,\ \sigma)$ such that $(\ell,\ \sigma) \rightarrow_{\mathcal{G}} (\ell',\ \sigma')$ and $C((\ell,\ \sigma),\ (\Gamma^{\#},\ \Gamma))$.*
- *There exists $(\ell,\ \sigma)$ such that $C((\ell,\ \sigma),\ (\Gamma^{\#},\ \Gamma))$*

Proof sketch. We show the result on paths that don't go through $\mathtt{img}\, p^{\#}$, composing them if needed. $\Gamma'^{\#}$ is in the image of $p^{\#}$, so it has the same transitions as those to a $\mathcal{F}_g(p^{\#})(\ell')$ term. Thus, we have the same cases four cases as in Lemma A.6. We can use the no $p^{\#}$ on the path hypothesis to ensure the start $p^{\#}(\ell)$ of our path must match $\Gamma^{\#}$. This requires a bit of work for the length 0 case though. □

Proof. Take $(\Gamma^{\#},\ \Gamma) \rightsquigarrow^{*}_{\hat{\mathcal{G}}_{p^{\#}}} (\Gamma'^{\#},\ \Gamma')$ such that $\Gamma^{\#}$ and $\Gamma'^{\#}$ appear in $\mathtt{img}\, p^{\#}$. We show the result for when the path does not go through the image of $p^{\#}$, since the general case can be established by splitting the path around nodes that are in the image of $p^{\#}$ and applying the result to each segment.
$\Gamma'^{\#}$ is in the image of $p^{\#}$, so there exists ℓ' and $\sigma_1^{\#}$ such that $(\sigma_1^{\#}, \Gamma'^{\#}) = p^{\#}(\ell')$. We can then define $\sigma' \triangleq [x \in \mathbb{L} \mapsto \hat{\mathcal{E}}[\![\sigma_1^{\#}(x)]\!](\Gamma)]$ which shows existence.
For the rest of this proof, take ℓ' and σ' such that $C((\ell',\ \sigma'),\ (\Gamma'^{\#},\ \Gamma'))$.

The same reasoning as in the previous theorem shows we can take $\Gamma'^\#$ to be the same as $\mathcal{F}_g(p^\#)(\ell')$ without changing the path. The term built by $\mathcal{F}_g$ is a *join* of *apply* of the predecessors of ℓ'. These predecessors are in the image of $p^\#$.

Looking how these map to our SSA operations, we notice the same four cases as before in the previous lemma (Lemma A.6). They imply that a path from a predecessor not going through the image of $p^\#$ is either empty, an `Assûme`, a `Joîn`, or a `Joîn` of an `Assûme`. Furthermore, these are the only paths that reach $\Gamma'^\#$, so one of these predecessors must correspond to the path coming from $\Gamma^\#$. Let us denote it ℓ and its associated relation R.

If the path from $p^\#(\ell)$ is non-empty, it must correspond to $\Gamma^\#$ (Otherwise, the path from $\Gamma^\#$ would go through a point in $\mathtt{img}\, p^\#$). If the path is empty, then either $p^\#(\ell) \neq (\_, \Gamma^\#)$, then we can choose a $\ell \neq \ell'$ (else $\Gamma'^\#$ has only one parent, itself, and so $\Gamma^\# = \Gamma'^\#$). We can repeat this process until we find a ℓ that matches $\Gamma^\#$. This terminates because there is a finite number of program locations.

There must then exist $\sigma_0^\#$ such that $(\sigma_0^\#, \Gamma^\#) = p^\#(\ell)$.

- If there is no `Assûme`, then by inverting *apply*, we know that R is an assignment "$x := e$".
  If there is also no `Joîn`, the path is empty. $\Gamma^\# = \Gamma'^\#, \Gamma = \Gamma'$ and $\sigma_1^\# = \sigma_0^\# \left[x \mapsto \mathtt{subst}(e,\ \sigma_0^\#)\right]$. Choosing $\sigma \triangleq [x \in \mathbb{L} \mapsto \hat{\mathcal{E}}[\![\sigma_0^\#(x)]\!](\Gamma)]$ is then sufficient. It indeed verifies the compatibility with Γ, which, combined with the compatibility between Γ' and σ', implies $\sigma' = \sigma\,[x \mapsto \mathcal{E}[\![e]\!](\sigma)]$ by compatibility between `subst` and $\mathcal{E}[\![\cdot]\!]$.

  If there is a `Joîn`, the path is a single transition labelled by bindings: $\Gamma^\# \overset{B}{\hookrightarrow}_\# \Gamma'^\#$. We therefore have $\Gamma' \triangleq \mathrm{unbind}_{\hat{\mathcal{G}}_{p^\#}}(\Gamma'^\#, \mathrm{bind}(B, \Gamma))$. Furthermore, if we denote $\sigma_2^\# \triangleq \sigma_0^\# \left[x \mapsto \mathtt{subst}(e,\ \sigma_0^\#)\right]$ $\sigma_1^\#$ is equal to $\sigma_2^\#$ on the variables not in $\mathtt{dom}\, B$ and renames those in $\mathtt{dom}\, B$ to $x_{\Gamma'^\#}$. Note that B maps these variables to their value in $\sigma_2^\#$. Therefore, for all x, $\hat{\mathcal{E}}[\![\sigma_1^\#(x)]\!](\Gamma') = \hat{\mathcal{E}}[\![\sigma_2^\#(x)]\!](\Gamma')$. So again, it is sufficient to choose $\sigma \triangleq [x \in \mathbb{L} \mapsto \hat{\mathcal{E}}[\![\sigma_0^\#(x)]\!](\Gamma)]$ and $\sigma' \triangleq [x \in \mathbb{L} \mapsto \hat{\mathcal{E}}[\![\sigma_1^\#(x)]\!](\Gamma)]$ to show the results.
- If there is an `Assûme`, then by inverting *apply*, we know that R is a guard "If e".
  If there is also no `Joîn`, the path is a single transition $\Gamma^\# \overset{e}{\hookrightarrow}_\# \Gamma'^\#$. This transition implies that $\Gamma = \Gamma'$, $\sigma_1^\# = \sigma_0^\#$ and $\hat{\mathcal{E}}[\![e]\!](\Gamma) \neq 0$. Choosing $\sigma \triangleq [x \in \mathbb{L} \mapsto \hat{\mathcal{E}}[\![\sigma_0^\#(x)]\!](\Gamma)]$ is then sufficient. It indeed verifies $\sigma = \sigma' \wedge \mathcal{E}[\![e]\!](\sigma) \neq 0$ and the compatibility with Γ by definition.
  If there is a `Joîn`, the path is made of two transitions:

  $$\Gamma^\# \overset{e}{\hookrightarrow}_\# \mathtt{Assûme}(e,\ \Gamma^\#) \overset{B}{\hookrightarrow}_\# \Gamma'^\#$$

  We therefore have $\Gamma' \triangleq \mathrm{unbind}_{\hat{\mathcal{G}}_{p^\#}}(\Gamma'^\#, \mathrm{bind}(B, \Gamma))$. Furthermore, $\sigma_1^\#$ is equal to $\sigma_0^\#$ on the variables not in $\mathtt{dom}\, B$ and renames those in $\mathtt{dom}\, B$ to $x_{\Gamma'^\#}$. Note that B maps these variables to their value in $\sigma_0^\#$. Therefore, for all x, $\hat{\mathcal{E}}[\![\sigma_1^\#(x)]\!](\Gamma') = \hat{\mathcal{E}}[\![\sigma_0^\#(x)]\!](\Gamma)$.
  So again, it is sufficient to choose $\sigma \triangleq [x \in \mathbb{L} \mapsto \hat{\mathcal{E}}[\![\sigma_0^\#(x)]\!](\Gamma)]$ to show the results. □

Theorem 7.2 (SSA compilation backward simulation). *For all SSA states $(\Gamma^\#, \Gamma)$ and $(\Gamma'^\#, \Gamma')$ where $\Gamma^\#$ and $\Gamma'^\#$ appear in $\mathtt{img}\, p^\#$ as images of widening or entry points, and for all $s' \in \mathbb{S}$ we have:*

$$(\Gamma^\#, \Gamma) \rightsquigarrow^+_{\hat{\mathcal{G}}_{p^\#}} (\Gamma'^\#, \Gamma') \ \wedge\ C(s',\ (\Gamma'^\#, \Gamma')) \quad \Rightarrow \quad \exists s \in \mathbb{S},\ \ C(s,\ (\Gamma^\#, \Gamma)) \ \wedge\ s \rightarrow^+_{\mathcal{G}} s'$$

Furthermore, there exists an $s' \in \mathbb{S}$ such that $C(s',\ (\Gamma'^\#, \Gamma'))$ holds.

Proof. Apply Lemma A.7 as many times as the input path is long, and compose the results. □

A.5 Proofs: SSA based numerical analysis

Proof of Lemma 8.2:

Lemma 8.2 (Strong relative completeness). *Let $C \in \mathbb{G} \to \mathbb{G}$ be the transformation that flattens* Imp *program expression by writing all sub-expressions to new temporary variables, and π a projection that strips those newly introduced variables, then*

$$\mathtt{analyse}_{\mathcal{G}}(\mathit{Lift}(\widehat{\mathrm{N}})) = \pi(\mathtt{analyse}_{C(\mathcal{G})}(\mathit{Lift}(\widehat{\mathrm{N}})))$$

First, let us define the C transformation a bit more formally. We define C_{e} a transformation that transforms an expression into a list of assignments and a simple expression (either a constant or a variable) as follows:

$$\begin{aligned}
C_{\mathsf{e}} &\in \mathbb{E} \to (L(\mathbb{X} \times \mathbb{E}) \times \mathbb{E}) \qquad (L(X) \triangleq X^* \text{ is the set of finite lists of } X)\\
C_{\mathsf{e}}(z) &\triangleq ([], z)\\
C_{\mathsf{e}}(x) &\triangleq ([], x)\\
C_{\mathsf{e}}(e_\ell \diamond e_r) &\triangleq (a_\ell +\!\!+\, a_r +\!\!+ [(y, x_\ell \diamond x_r)], y) \qquad (+\!\!+ \text{ is list concatenation})\\
&\ y \text{ is fresh, } (a_i, x_i) \triangleq C_{\mathsf{e}}(e_i)\\
C_{\mathsf{e}}(e_c \,?\, e_t : e_f) &\triangleq (a_c +\!\!+\, a_t +\!\!+\, a_f +\!\!+ [(y, x_c \,?\, x_t : x_f)], y)\\
&\ y \text{ is fresh, } (a_i, x_i) \triangleq C_{\mathsf{e}}(e_i)
\end{aligned}$$

From this compilation of expression, C transforms full Imp programs by replacing every edge with a chain of edges, as generated C_{e}. All but the last edge of the chain are assignments, given by the list, and the last edge is the same as the original edge, but whose expression was replaced by the simple expression returned by C_{e}.

These definitions are meant to be close to those of Logozzo and Fähndrich [2008]. They are slightly adapted to better fit our model. We do not have two distinct language for source and target: our source is Imp and our target Imp but only using simple expressions (whose depth is at most 1).

Proof. The gist of the proof is to show that all domain operations verify the property, assuming their argument do. The result is then obtained by structural induction.

We first show that applying a single relation, or applying the chain of relation from the compiled version will lead to the same state (after removing the new temporary variables).

All but the last relation in the state are assignments. By definition of Lift.*apply*, these do not modify the SSA state, only the mapping of Imp variables to program expressions. For the last operation, the expression is first transformed to an SSA expression via `subst`. This function will unfold the definition of all variables until it reaches a constant or an SSA expression, thus undoing the compilation transformation.

We know that `subst` will remove all introduced variables since the only variables it cannot simplify are those mapped to SSA variables. As it stands, SSA variables are only introduced if a variable is used before it is defined ($x_{\widehat{entry}}$ variable), or if a variable has different values in different branches of a join (ϕ-variable, $x_{\widehat{join}\{\dots\}}$). Both cases cannot occur for the newly introduced variables since, by definition of C_{e}, it is clear that new variables are always defined before being used. Furthermore, new variables are only defined once, so they cannot appear in multiple branches of a join.

Thus evaluating (with `subst`) the full relation directly, or evaluating each assignment, then the compiled (simplified) expression will yield the same result. Furthermore, the SSA state in unchanged by evaluating the assignments, and only new variables in the variable store are changed. Hence, after removing the new temporary variables, both abstract states are the same.

To generalize this result, all we need is to show that the other domain operations (*entry*, *join*, *widen*) also verify this property. Lift($\widehat{\mathrm{N}}$).*entry* doesn't depend on the input program graph, so it is always true. For the Lift($\widehat{\mathrm{N}}$).*join*, notice that the SSA state is again unchanged in both versions, since it only depends on the parent SSA states, and the variables that differ in each branch. For the variable store, the join only keeps common variables, leaving them unchanged if they are equal in all branches, and introducing a new variable if not. As argued above, new variables are only introduced for original program variables. Thus removing the extra variables doesn't affect them and maintains the property.

Finally, the $\mathrm{Lift}(\widehat{\mathrm{N}}).\mathit{widen}$ is called at the same places on both graphs. It only affects the SSA state (which is the same in both graph), and only renames the newly introduced variables in the store (which are all variables from the original Imp program). Therefore, it also verifies the property.

Since all domain operations yield the same state (after stripping extra variables) when run directly or run on the compiled version, it stands to reason that the full analysis (result of a finite number of applications of such domain operations) also has this property. ◻

B USING AN ORDER RELATION TO REACH A POST FIXED POINT

For our analysis in Section 3.3, the hypothesis WidenValid requires our domains widening to converge to a true fixed-point. Often in abstract interpretation, we relax that hypothesis to a post fixed-point. To do so, we need a new domain function, a pre-order relation on $\Sigma^\#$:

$$\sqsubseteq \ \in \ \mathcal{P}(\Sigma^\# \times \Sigma^\#)$$

It should be compatible with the domain order: $\forall\, s^\#\ t^\#,\ s^\# \sqsubseteq t^\# \implies \gamma(s^\#) \subseteq \gamma(t^\#)$. We extend $\sqsubseteq$ to a relation on $\Sigma^\#_\perp$ by having $\perp$ be a minimal element.

With this, we can weaken WidenValid to simply say the sequence $t^\#_{n+1} \triangleq \mathit{widen}(\ell,\ t^\#_n,\ s^\#_n)$ reaches a post-fixed point in finite time: $\exists\, n,\ t^\#_{n+1} \sqsubseteq t^\#_n$. However, to prove convergence we now need monotony hypotheses for two reasons:

- To ensure that once we will keep decreasing after we reach a post fixed point ($p^\#_{n+1} \sqsubseteq p^\#_n$), and so can stop at any time after that.
- To ensure that non widening points converge, since otherwise it is possible that although $p^\#_n$ decreases on the widening points, it does not do so on non-widening points.

Overall, this leads to a weaker hypothesis on *widen*, but requires stronger hypotheses on *apply* and *join*:

$$\forall\, s^\#\ t^\#\ R,\quad s^\# \sqsubseteq t^\# \implies \mathit{apply}(R,\ s^\#) \sqsubseteq \mathit{apply}(R,\ t^\#) \qquad \text{(ApplyMonotone)}$$

$$\forall\, S^\#\ T^\#,\quad (\forall\, s^\# \in S^\#,\ \exists\, t^\# \in T^\#,\ s^\# \sqsubseteq t^\#) \implies \mathit{join}(S^\#) \sqsubseteq \mathit{join}(T^\#) \qquad \text{(JoinMonotone)}$$

C PERFORMING FEWER WIDENINGS USING WIDENING EDGES

C.1 Definition of widening edges

Since widening (calling *widen*) can lead to loss of precision, we want to do so as little as possible. As defined the analysis (Section 3.3) calls *widen* on all loop heads. This notably includes heads of unreachable loops. To avoid this, a simple improvement is to only widen at $\ell \in W$ if one of its predecessors is not $\perp$.

While we are considering predecessors, we can do even better by limiting widening to predecessors coming from inside the loop. Indeed, it is possible that the analysis detects that one edge of the loop is not taken. This is especially possible with transformation functors (Section 5) and numerical analysis (Section 5.3). This is expresses by having the corresponding *apply* evaluate to $\perp$. In that case, the loop is already broken: we have no need to widen at the head.

To take advantage of this fact, we replace our set of widening points W with a set of *widening edges* $U \in \mathcal{P}_f(\mathbb{L} \times \mathbb{R} \times \mathbb{L})$. Formally, this is a subset of the program graph $\mathcal{G}$ that should contain at least one edge in every looping path. In practice, we use the weak topological order [Bourdoncle 1993] and define U as the set of edges returning to a component head from inside said component.

This choice is not only practical, it is also optimal. Indeed, if the analysis detects a false edge, all subsequent loop locations will be unreachable ($\perp$). So, if any edge of the loop is eliminated, then the last edge will also be eliminated.

With this new set, we can define a new widening operator:

$$\begin{array}{rcll} p^\# \nabla_U q^\# & \triangleq & \ell \mapsto \mathit{widen}(\ell,\ p^\#(\ell),\ q^\#(\ell)) & \text{if } \exists\, \ell',\ (\ell', R, \ell) \in U \wedge \mathit{apply}(R,\ p^\#(\ell')) \neq \perp \\ & | & \ell \mapsto q^\#(\ell) & \text{otherwise} \end{array}$$

Here, the condition for widening is more complex than that of ∇_W. We only widen if we are at the end point of a widening edge ($(\ell', R, \ell) \in U$) and that edge is taken ($\mathit{apply}(R,\ p^\#(\ell')) \neq \perp$).

C.2 Ensuring convergence

With this definition, we can still prove soundness (Lemma A.3) and completeness (Lemma A.3) fairly easily, but termination (Lemma A.1) is harder. Indeed, the set of widening points is no longer fixed. It might vary across iterations.

Increasing it is fine: if we can prove that once we start widening at a given point we always will, we can still show convergence. This is because in that case, the set of widening points is increasing across iterations, and bounded by the finite set of points that appear in U. Thus, it converges and the proof from Lemma A.1 still works.

Lemma C.1 (Monotone case). *If our transfer functions apply and join are monotone (verify ApplyMonotone and JoinMonotone), then the set of widening points is monotone.*

Proof. Start by proving that for all n, we have $\forall\, \ell,\ p_n^\#(\ell) \sqsubseteq p_{n+1}^\#(\ell)$. This can be done by strong induction on n.

- For $n = 0$, consider the case $\ell \neq \ell_0$. Then $p_0^\#(\ell) = \perp$, which is a minimal element. For the ℓ_0 case, we have $p_1^\#(\ell_0) = \mathit{entry} = p_0^\#(\ell_0)$.
- For the induction case, we can have $p_{n-1}^\#(\ell) \sqsubseteq p_n^\#(\ell)$ by induction hypothesis.
  - If we widen at ℓ at $n + 1$, the result is given by WSound.
  - If there is no widening the result is shown by monotonicity of $\mathcal{F}_g$, immediate given ApplyMonotone and JoinMonotone.
  - We cannot widen at step n but not $n+1$. Otherwise, there exists ℓ' such that $(\ell', R, \ell) \in U$ (does not depend on n); $p_{n-1}^\#(\ell') \neq \perp$ (also true at n since $p_{n-1}^\#(\ell') \sqsubseteq p_n^\#(\ell')$ by induction and $\perp$ is minimal); and $\mathit{apply}(R,\ p_n^\#(\ell')) \neq \perp$ (also true at n since *apply* is monotone and the same reasoning as the previous point). Thus, we will also widen at step $n + 1$.

Repeat the reasoning of the last point to show that $\forall\, n\, \ell,\ p_n^\#(\ell) \sqsubseteq p_{n+1}^\#(\ell)$ implies the lemma. □

Non-monotone non-convergence. For the non-monotone case, we can find examples of non convergence. For instance, consider the interval domain of Section 3.4. We could define non-monotone transfer functions. Consider this function for the + operator:

$$\begin{array}{rcl} [1:1] \vec{+} [1:1] & = & [1:4] \\ [1:+\infty] \vec{+} [1:1] & = & [2:+\infty] \not\sqsupseteq [1:4] \end{array}$$

Combine with edge elimination for false guards, this can lead to non-convergence. An example of such a program is given in Figure 14. The analysis will not converge since will alternate between widening at point 1, which leads to a more precise value at point 2, which leads to not considering the widening-edge $2 \longrightarrow 1$, so no longer widening at 1, which leads to imprecision at point 2 which fails to eliminate the back edges...

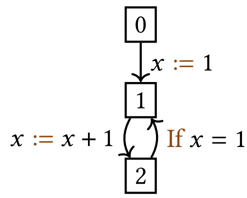


Fig. 14. Example of non-convergence for non-monotone programs

Forcing monotony. One workaround is to force the monotony of the set of widening points. We remember the points where we widened in previous iteration and keep widening there. Formally, we define the analysis no longer just on a function $p^\#$, but on a pair $(p^\#, L)$ where $L \in \mathcal{P}(\mathbb{L})$, starting at $(p_0^\#, \emptyset)$. The transfer function just applies $\mathcal{F}_g$ to the first component. For the widening change the condition in ∇_U to "$\ldots \vee \ell \in L$", and change L to the set of points where we widened.

C.3 Performance cost of widening edges

The new widening operator ∇_U can be computed with almost the same complexity as ∇_W. Indeed, the $\mathit{apply}(R,\ p^\#(\ell')) \neq \bot$ part of the condition is already computed in $\mathcal{F}_g$. By memorizing the points for which this is true, we can evaluate the widening condition by simply testing, for each of these predecessors, if they correspond to a widening edge. This means we have replaced one set lookup per point ℓ (check if $\ell \in W$) to multiple set lookups (for all predecessors ℓ', check if $(\ell', R, \ell) \in U$). Thus added complexity comes from points which have lots of predecessors, or loops with many paths returning to the head (e.g. loops with `continue` statements, which can lead to U being much larger than W). In most programs, both of these will be bounded by reasonable constants.

Another performance lost comes from slower convergence. Since we only widen when we find a loop, we need to propagate through each loop at least twice to ensure convergence. Once before widening, and once more after (since widening likely changes the loop head). This wasn't a problem with widening points since we always widened at the head, and thus avoided the first pre-widening pass.

Received 2023-11-16; accepted 2024-03-31