SSA Translation Is an Abstract Interpretation

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

Static single assignment (SSA) form is a popular intermediate representation that helps implement useful static analyses, including global value numbering (GVN), sparse dataflow analyses, or SMT-based abstract interpretation or model checking. However, the precision of the SSA translation itself depends on static analyses, and a priori static analysis is even indispensable in the case of low-level input languages like machine code. To solve this chicken-and-egg problem, we propose to turn the SSA translation into a standard static analysis based on abstract interpretation. This allows the SSA translation to be combined with other static analyses in a single pass, taking advantage of the fact that it is more precise to combine analyses than applying passes in sequence. We illustrate the practicality of these results by writing a simple dataflow analysis that performs SSA translation, optimistic global value numbering, sparse conditional constant propagation, and loop-invariant code motion in a single small pass; and by presenting a multi-language static analyzer for both C and machine code that uses the SSA abstract domain as its main intermediate representation.

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: Static Single Assignment (SSA), Abstract interpretation, Cyclic term graph.

ACM Reference Format:
Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (January 2023), 30 pages. https://doi.org/10.1145/3571258

1 INTRODUCTION

"SSA translation is an abstract interpretation" may be a surprising statement. Usually, SSA translation [Cytron et al. 1991] is presented as a code transformation that modifies the input program so that each variable is assigned only once (the Static Single Assignment (SSA) property). Abstract interpretation, on the other hand, is a method for designing sound static analyses that retrieve information about the program. If a code transformation can be based on the results of an analysis, and the results of an analysis can be made more precise by a previous code transformation, both are usually clearly separated and done in separate passes. This is unfortunate as it is known that performing analyses in a sequence of passes yields less precise results than combining them in a single pass [Click and Cooper 1995; Cousot and Cousot 1979].

To solve this problem, we perform SSA translation using abstract interpretation, i.e. the program abstraction that the analysis computes is the SSA translation of the program. More precisely, our SSA abstraction builds upon a variant of a global value numbering (GVN) analysis that we call *symbolic expression* analysis (Section 4). This is unlike most existing work (e.g., [Alpern et al. 1988; Barthe et al. 2014; Rosen et al. 1988; Rüthing et al. 1999]), where GVN is performed *after* SSA translation. Indeed in existing work, GVN requires a previous SSA translation to name the subexpressions of the program so that GVN then only needs to partition the names in equality classes. By contrast, in our work, the underlying abstraction used by GVN is not SSA names, but a

Author's address: Matthieu Lemerre, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, matthieu.lemerre@cea.fr.

2475-1421/2023/1-ART65
https://doi.org/10.1145/3571258

cyclic term graph [Ariola and Klop 1996] called the global value graph [Reif and Lewis 1986]. This allows performing GVN directly on the original program.

Our SSA graph abstraction (Section 5) then builds upon the global value graph abstraction by adding control flow and conditionals. Like the symbolic expression analysis, the SSA translation is described as a standard dataflow analysis that directly applies to the source program. As SSA translation preserves the semantics of the original program, we prove that our SSA-abstraction analysis is both sound and complete. Interestingly, our proof method consists in finding a surjective strong homomorphism (a stronger variant of bisimulation) between the SSA translation and the original program, where the homomorphism directly derives from our symbolic expression analysis. Finally, we provide (Section 6) alternative transfer functions for our SSA translation that allows the analysis to be complete (i.e. excludes spurious behaviors) also on unfinished fixpoint computations, which makes our SSA translation algorithm incremental.

The syntax and semantics of our SSA abstraction are non-standard but can readily be translated to the usual SSA syntax by choosing an order of evaluation for expressions; moreover, our SSA translation enjoys the classic minimality and strictness [Rastello 2016] properties of SSA.

Motivation. Interesting theoretical contributions of this work are its alternative syntax and semantics for SSA form (both are new, concise and simple), the algorithm used for its translation (a standard dataflow analysis which builds on global value numbering and does not require the computation of any dominance properties), the theoretical connections it reveals between global value numbering, symbolic expression analysis and SSA, and the technique used to perform dataflow analysis on cyclic terms or systems of recursive equations in general (reusing cyclic names of the original program to detect cycles and guarantee termination).

A practical motivation for presenting SSA translation as an abstract domain is that abstract domains can be combined modularly [Cousot and Cousot 1979; Cousot et al. 2006; Journault et al. 2019]. An application of this is to explore alternative designs for the combination of compiler optimizations. In general, compiler optimizations are written as a succession of analysis and rewriting passes. Viewing SSA as an abstract domain allows combining the analysis and the rewriting pass in a single step, with the benefit that the SSA invariants (e.g., related to domination) are automatically maintained when the control-flow graph changes (e.g., when a branch is found dead, or a new target of a computed goto is discovered). This is beneficial as "maintaining SSA is often one of the more complicated and error-prone parts in such optimizations" [Hack 2016].

It is even possible to group all the different analyses and transformations in a single pass, avoiding the burden of performing any intermediate rewriting steps. As an example (Section 7), we implemented a simple dataflow analysis that performs global value numbering, loop-invariant code motion, sparse conditional constant propagation, and SSA translation in a single dataflow analysis pass. This pass is followed by a pass of linear complexity translating our internal SSA representation into LLVM bitcode, mainly consisting in making explicit the order of evaluation of the symbolic expressions appearing in our representation of SSA. Experimental results show that the implementation is small and the compiling time is compatible with inclusion in a compiler.

Another application is formal verification. In general, combining analyses yields an analysis which is more precise than performing different analyses in a sequence of passes [Click and Cooper 1995; Cousot and Cousot 1979]. Viewing SSA as an abstract domain is thus interesting when analyzing programs where the SSA translation can be improved by a static analysis, and the static analysis can itself be improved by more precise SSA translation. For instance, it is common to use SSA as an intermediate language before translation into logic formulas queried to an automated solver (e.g., [Brain et al. 2015; Gurfinkel et al. 2015; Henry et al. 2012; Leino 2005]). Combining the SSA translation with other abstract domains allows simplifying the generated formula, and

simplifications significantly impact the solver performance [Farinier et al. 2018; Gurfinkel et al. 2015]. In turn, the solver's response can improve the precision of the static analysis. One example application is the analysis of machine code, where the precision of the recovered control-flow graph is crucial, and SMT-based techniques can be used to ensure the reachability of the locations of the recovered CFG [Djoudi et al. 2016; Reinbacher and Brauer 2011]. Note, however, that using such a combination requires that the SSA form produced from an unfinished fixpoint iteration is complete (i.e., does not introduce any spurious behavior): this problem is solved in Section 6.

The benefits of this combination of domains grow in importance when analyzing lower-level languages, such as machine code or C. Indeed, more precise SSA translation in these languages demands that memory locations are promoted to SSA variables; this promotion requires precise memory and control flow analyses, which depend on a numeric analysis; the numeric analysis can be enhanced by more precise SSA translation. Having SSA as an abstract domain and using it with a combination of memory, numerical and control-flow abstractions solves this chicken-and-egg problem. For instance, we have implemented a static analyzer for both C and machine code programs that decompiles low-level code into a higher-level SSA abstraction using the SSA graph abstract domain; simultaneously, this SSA abstraction is used as the language on which the numerical analyses are performed. This paper is the first step in explaining how such an analysis works.

2 OVERVIEW

This section provides an overview of our main results on a running example (Figure 1b) that we will use in most of the paper. In our view, the essence of SSA is the addition of control flow information to a structure called the global value graph [Reif and Lewis 1986]; our presentation shows how to build these two abstractions in turn. We start from a program consisting of basic blocks, each associated with a unique program *location* $\ell_0, \ldots, \ell_3 \in \mathbb{L}$ labeling the basic block. Each block contains a sequence of instructions and performs nondeterministic jumps to other basic blocks; these jumps have guards that block some executions. The semantic is standard and consists of a transition between states, where a state contains both the current location and a *store* $\sigma \in \mathbb{\Sigma}$ (a mapping from *program variables* $x \in \mathbb{X}$ to values). The initial states are all the states starting at ℓ_0.

2.1 Symbolic Expressions, Global Value Numbering and the Global Value Graph

The Symbolic Expression Abstraction. Our main program abstraction $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ associates to each location $\ell \in \mathbb{L}$ an *abstract store* $\sigma^\sharp \in \mathbb{\Sigma}^\sharp_\bot$ describing the possible states before execution of the basic block. An abstract store is either $\bot$ or a mapping from program variables to *symbolic expressions*, i.e. terms containing *symbolic variables* $\in \mathbb{V}$. The result of the analysis for our example program is given on the first line of Figure 1a. For instance, the abstract store at $p^\sharp[\ell_2]$ refers to two symbolic variables, namely $\mathtt{i}_{\ell_1}$ and $\mathtt{time}_{\ell_1}$. An abstract store represents all the concrete stores that can be obtained using any valuation of the symbolic variables, e.g., the concrete store $[\mathtt{inc} \mapsto 1, \mathtt{time} \mapsto 0, \mathtt{i} \mapsto 0, \mathtt{hours} \mapsto 0]$ is represented by $p^\sharp[\ell_2]$ (using any valuation where both $\mathtt{i}_{\ell_1}$ and $\mathtt{time}_{\ell_1}$ are assigned 0). Note that the symbolic expressions allow describing relations between variables (e.g. `hours = time/(1 * 3600)` in the states at ℓ_2) or some numerical information (e.g. $2 * \mathtt{x}_{\ell_5}$ describes an even value). One particular relation between variables is equality, and indeed this abstraction can be used to detect equality between variables corresponding to *global value numbering* [Alpern et al. 1988; Rosen et al. 1988] (where the "number" uniquely identifies a subterm of a symbolic expression).

Note that symbolic variables $\in \mathbb{V} = \mathbb{X} \times \mathbb{L}$ are named using a program variable and a location: intuitively, $\mathtt{i}_{\ell_1}$ represents the value of program variable `i` at location ℓ_1 (in the SSA abstraction, as $\mathtt{i}_{\ell_1}$ is unconstrained in the symbolic expression abstraction). To express more equalities, we can reuse

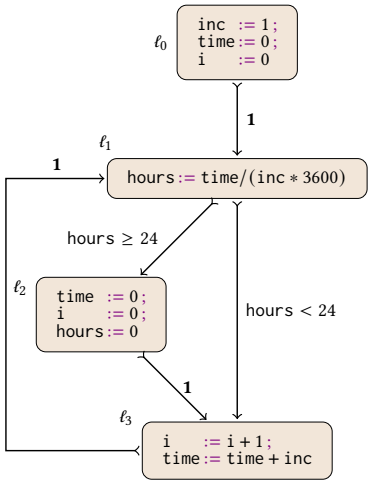
$p^\sharp$ when $\mathbb{V} = \mathbb{X} \times \mathbb{L}$

$$\begin{array}{c|c|c|c} \ell_0 & \ell_1 & \ell_2 & \ell_3 \\ \begin{bmatrix} \texttt{inc} & \mapsto \texttt{inc}_{\ell_0} \\ \texttt{time} & \mapsto \texttt{time}_{\ell_0} \\ \texttt{i} & \mapsto \texttt{i}_{\ell_0} \\ \texttt{hours} & \mapsto \texttt{hours}_{\ell_0} \end{bmatrix} & \begin{bmatrix} \texttt{inc} & \mapsto 1 \\ \texttt{time} & \mapsto \texttt{time}_{\ell_1} \\ \texttt{i} & \mapsto \texttt{i}_{\ell_1} \\ \texttt{hours} & \mapsto \texttt{hours}_{\ell_1} \end{bmatrix} & \begin{bmatrix} \texttt{inc} & \mapsto 1 \\ \texttt{time} & \mapsto \texttt{time}_{\ell_1} \\ \texttt{i} & \mapsto \texttt{i}_{\ell_1} \\ \texttt{hours} & \mapsto \texttt{time}_{\ell_1}/(1*3600) \end{bmatrix} & \begin{bmatrix} \texttt{inc} & \mapsto 1 \\ \texttt{time} & \mapsto \texttt{time}_{\ell_3} \\ \texttt{i} & \mapsto \texttt{i}_{\ell_3} \\ \texttt{hours} & \mapsto \texttt{hours}_{\ell_3} \end{bmatrix} \end{array}$$

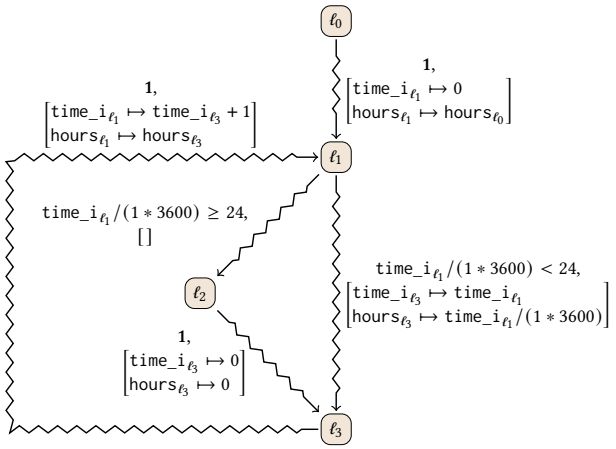
$p^\sharp$ when $\mathbb{V} = \mathcal{P}(\mathbb{X}) \times \mathbb{L}$

$$\begin{array}{c|c|c|c} \begin{bmatrix} \texttt{inc} & \mapsto \texttt{inc}_{\ell_0} \\ \texttt{time} & \mapsto \texttt{time}_{\ell_0} \\ \texttt{i} & \mapsto \texttt{i}_{\ell_0} \\ \texttt{hours} & \mapsto \texttt{hours}_{\ell_0} \end{bmatrix} & \begin{bmatrix} \texttt{inc} & \mapsto 1 \\ \texttt{time} & \mapsto \texttt{time\_i}_{\ell_1} \\ \texttt{i} & \mapsto \texttt{time\_i}_{\ell_1} \\ \texttt{hours} & \mapsto \texttt{hours}_{\ell_1} \end{bmatrix} & \begin{bmatrix} \texttt{inc} & \mapsto 1 \\ \texttt{time} & \mapsto \texttt{time\_i}_{\ell_1} \\ \texttt{i} & \mapsto \texttt{time\_i}_{\ell_1} \\ \texttt{hours} & \mapsto \texttt{time\_i}_{\ell_1}/(1*3600) \end{bmatrix} & \begin{bmatrix} \texttt{inc} & \mapsto 1 \\ \texttt{time} & \mapsto \texttt{time\_i}_{\ell_3} \\ \texttt{i} & \mapsto \texttt{time\_i}_{\ell_3} \\ \texttt{hours} & \mapsto \texttt{hours}_{\ell_3} \end{bmatrix} \end{array}$$

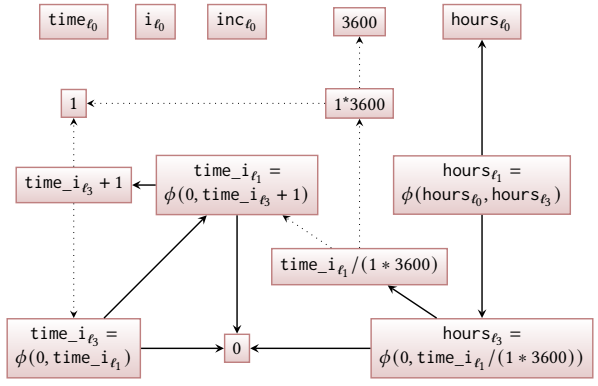
(a) Results of the basic (top) and improved (bottom) symbolic expression analyses (abstract domain: $\mathbb{L} \to \mathbb{S}^\sharp_\bot$).



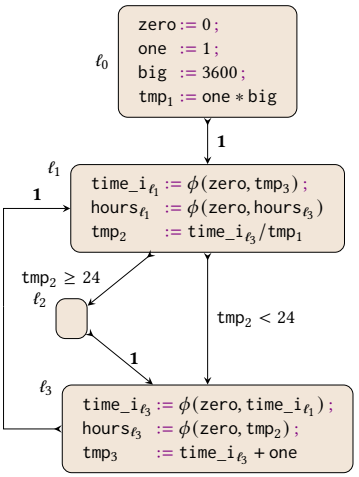
(b) Original program.



(c) Representation in the SSA abstract domain.



(d) The global value graph is a cyclic term graph.



(e) Usual SSA representation.

Fig. 1. Analyses and transformations of an SSA program.

the same name when symbolic variables are equal. This is why we shift to another representation $\mathbb{V} = \mathcal{P}(\mathbb{X}) \times \mathbb{L}$ where a symbolic variable is named using one location and a *set* of program variables; intuitively, $\texttt{time\_i}_{\ell_1}$ represents the value of both `i` and `time` at location ℓ_1 (thus, `i` = `time` at ℓ_1). The second line of Figure 1a presents the program abstraction $p^\sharp$ with this change.

Analysis Operations. We focus now our attention on how to compute $p^\sharp$. Many domain operations are simple and exact: notably, interpretation of expression substitutes program variables with their corresponding symbolic expression; interpretation of assignment updates the abstract store (observe, for instance, how `hours` is bound to $\texttt{time\_i}_{\ell_1}/(1 * 3600)$ in $p^\sharp[\ell_2]$ given $p^\sharp[\ell_1]$). The only operation in this domain that lose precision is the join operation $\sqcup^\ell$, which merges abstract stores at control-flow join points such as ℓ_3.

One way to perform these joins would be to add a *nondeterministic choice operator* ϕ, as a function symbol in the language of symbolic expressions. For instance, the value of the program variable `i` at ℓ_3, which is either 0 or $\texttt{time\_i}_{\ell_1}$ depending on the previous program location, could be represented by the term $\phi(0, \texttt{time\_i}_{\ell_1})$.

A first issue with this technique is the handling of nondeterminism: i.e. two terms $\phi(0, 1)$ and $\phi(0, 1)$ correspond to the same value if one is copied from the other but to different values if different nondeterministic choices between 0 and 1 must be made in the program. However, this problem could be mitigated by using different labels for each ϕ function (as in [Alpern et al. 1988; Rüthing et al. 1999]). A much more problematic issue is that this approach cannot terminate in the presence of loops, as each iteration would create a larger term without ever converging.

The Global Value Graph. Our solution to both of these issues is to use cyclic term graphs [Ariola and Klop 1996] instead of simple terms. In cyclic term graphs, recursion variables are introduced to name some nodes, which allows breaking cycles (in addition, sharing between terms is maximized). The whole cyclic graph is described using a set of recursive definitions, each definition mapping a recursion variable to a simple term that can refer to recursion variables. We apply this idea to our terms: the cyclic term graph corresponds to a structure called the *global value graph* [Reif and Lewis 1986] (sometimes also called the SSA graph [Stanier 2016]), and our symbolic variables play the role of the recursion variables of the cyclic term graph. In addition, symbolic variables are defined for all ϕ terms: i.e. every term $\phi(e_1, e_2)$ is named using a symbolic variable $\texttt{var}_\ell$. The global value graph corresponding to the program of Figure 1b is represented graphically in Figure 1d. In this graph, each node corresponds to a term; dotted lines correspond to the "direct subterm" relation (e.g., "$1 * 3600$" has "1" and "3600" as subterms); solid lines correspond to the direct subterm relation between a ϕ term $\phi(e_1, e_2)$ and its subterms e_1 and e_2 (they can also be seen as representing the definition of a symbolic variable as a ϕ term).

Switching from terms to cyclic terms and using symbolic variables to name ϕ terms solves *both* the nondeterminism and termination issues. The names differentiate two terms built from different nondeterministic choices between two values: e.g. if we have both $\texttt{x}_{\ell_5} = \phi(0, 1)$ and $\texttt{y}_{\ell_6} = \phi(0, 1)$, we can conclude that $\texttt{x}_{\ell_5} - \texttt{x}_{\ell_5} = 0$ but that $\texttt{x}_{\ell_5} - \texttt{y}_{\ell_6}$ may be different from 0. Furthermore, cyclic terms provide a finite representation of the values that exist in the program, which is necessary for the analysis to terminate.

Regarding termination, a standard approach consisting of widening [Cousot and Cousot 1977] two terms to create a cyclic term (e.g., "0" ∇ "$0 + 1$" = "let $x = \phi(0, x + 1)$ in x") would be difficult to implement. Instead, our idea is to view the program locations as the recursion variables in a cyclic structure, which is the control flow graph of the original program, and to make use of these names to help us terminate the analysis. This is the reason why our symbolic variables have a program location in their definition. Essentially, our analysis terminates if variables are named according to

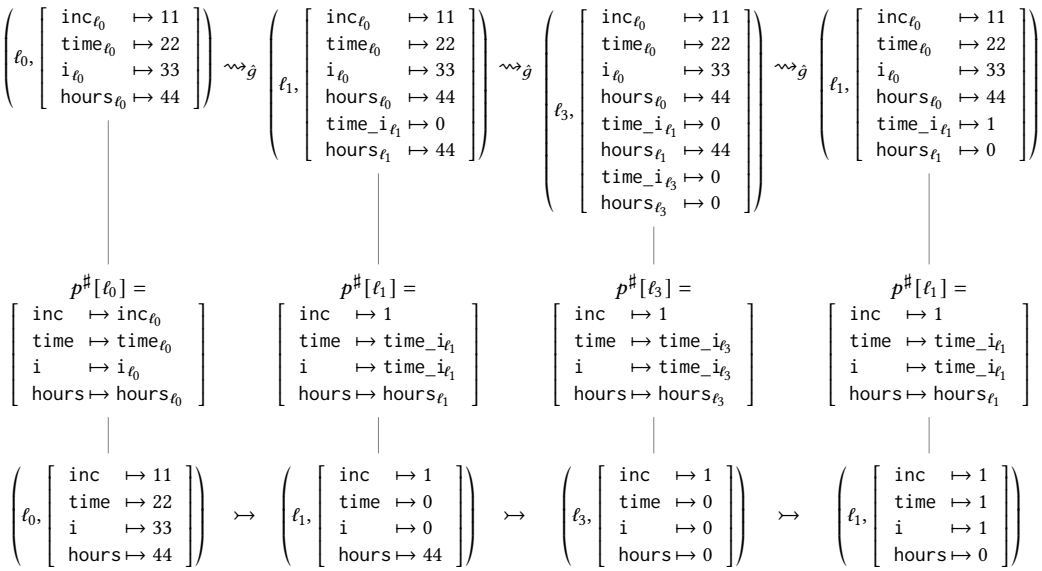


Fig. 2. Top: Execution trace of the SSA program of Figure 1c. Middle: The symbolic expression abstraction $p^\sharp$ seen as an homomorphism or bisimulation. Bottom: Execution trace of the original program of Figure 1b.

a "deterministic" naming scheme based on the program locations: in the actual implementation, we use integers instead of a set of program variables like `time_i` to name our symbolic variables. We prove that this technique is sound, but the soundness proofs are unusual.

If the global value graph of Figure 1d can be computed explicitly in the analysis, it is not necessary to do it: computing $p^\sharp$ (that never contains any ϕ term) is sufficient. Still, the value graph is useful to justify how $p^\sharp$ is built, in particular, how the transfer function chooses the name of fresh symbolic variables and how this choice allows the analysis to terminate.

2.2 The SSA Graph Abstract Domain

2.2.1 The SSA Graph Abstraction. Compared to $p^\sharp$, the value graph has more information as it links symbolic variables to their ϕ definition. But the value graph does not capture all the information about the initial program. First, it misses some relations between symbolic variables: for instance, we have $\texttt{hours}_{\ell_3} = \texttt{time\_i}_{\ell_3}/(1 * 3600)$, but this cannot be inferred from the global value graph as the definition of $\texttt{hours}_{\ell_3}$ and $\texttt{time\_i}_{\ell_3}$ are on different cycles. Second, the conditions in the original program appear nowhere in the global value graph.

What the global value graph lacks is information about control flow, which is necessary to encode the fact that different symbolic variables are updated simultaneously, and only when some conditions are met. Our SSA graph abstraction is built upon the idea of adding control flow information to the global value graph. More precisely, an SSA graph is a graph whose nodes are the locations of the original program, and whose edges are associated with a guard (translated from the guard of the original program) and a set of *bindings* from symbolic variables to symbolic expressions (corresponding to ϕ nodes in usual SSA). Figure 1c presents the SSA graph computed by our SSA abstract domain on the program of Figure 1b. Similarly to the global value graph, the SSA graph links symbolic variables to their ϕ definition; but unlike in the global value graph, it relates this information to the edges of the control flow. For instance, while the global value graph

only says that $\texttt{time\_i}_{\ell_3}$ is a nondeterministic choice between 0 and $\texttt{time\_i}_{\ell_1}$, the SSA graph also says that $\texttt{time\_i}_{\ell_3}$ is 0 at ℓ_3 when control comes from ℓ_2, and then $\texttt{hours}_{\ell_3}$ will also be 0; that when control comes from ℓ_1, $\texttt{time\_i}_{\ell_3}$ gets the same value that $\texttt{time\_i}_{\ell_1}$ had, and that this transition is possible only if $\texttt{time\_i}_{\ell_1}/(1 * 3600) < 24$.

The SSA graph can be given a simple semantics as a transition system: states consists in a location ℓ (the same as in the original program) and a *valuation* Γ (mapping from symbolic variables to values). The transition from a state (ℓ, Γ) to a state (ℓ', Γ') is possible only if there is an edge from ℓ to ℓ' in the SSA graph; the guard on this edge evaluates to true; and Γ' is updated from Γ using the bindings on the edge. Finally, a symbolic variable $\texttt{x}_\ell$ has a scope corresponding to all the locations dominated by ℓ, and variables of a valuation are unbound when they go out of scope. The top of Figure 2 provides an example of an execution trace for the SSA graph of Figure 1c.

2.2.2 A Sound and Complete Abstract Interpretation. One of the main result of this paper is that the behavior of the SSA graph computed using abstract interpretation is equivalent to the behavior of the original program. More precisely, the transition systems described by the original program and SSA graph are, once limited to the reachable states, related by a surjective strong homomorphism, a relation that is stronger than bisimulation [Sangiorgi 2009]. Furthermore, this surjective strong homomorphism directly derives from the symbolic expression abstraction $p^\sharp$: in essence, $p^\sharp$ is a map from SSA states to program states. Consider Figure 2 again : observe that the abstract stores in the middle can be used to retrieve values for program variables in the states at the bottom from the values of symbolic variables in the SSA states at the top. Thus, to each SSA execution trace corresponds a concrete execution trace (and conversely, every concrete execution traces can be simulated by one SSA execution trace).

This correspondence makes our abstract interpretation both sound (it does not miss any behavior) and complete (it does not include extra behavior). Furthermore, we can make the abstract interpretation complete even when the fixpoint is not finished (see Section 6).

2.2.3 Translation to the Usual SSA Representation. Our SSA graph representation is a faithful representation of the semantics of the original program, but it abstracts away some details of the usual SSA representation. Thus, translating from an SSA graph to standard SSA consists in additionally constraining the code to fill in these details. For instance, Figure 1e corresponds to a translation of the SSA graph of Figure 1c to a usual representation. The translation has to:

- Name every intermediate computation (i.e., flatten [Ariola and Klop 1996]) the term graph, as SSA representations are usually in three-address code. For instance, our representation names intermediate results like $\texttt{tmp}_1$.
- Determine the location where each intermediate computation is done. In Section 5.1 we define a notion of scope of symbolic expression that determines a suitable location for each intermediate computation (although other locations might be preferred). For instance, the assignment $\texttt{tmp}_1 := \texttt{one} * \texttt{big}$ has been moved outside the loop, because the scope $1 * 3600$ is ℓ_0 : i.e., SSA graphs naturally perform loop-invariant code motion.
- Explicit the order of evaluation of symbolic expressions by appropriately scheduling instructions. For instance in Figure 1c the order of assignment between one and big is arbitrary, but must be done before $\texttt{tmp}_1$ is assigned.

The SSA graph that we present is more appropriate for static analysis. It is however easy to modify SSA graphs so that it is more adapted to compilation, e.g. by using terminator instructions instead of guards on edges, by treating constant expressions specially, or by explicitly restricting the scope of symbolic expressions. This is what our experiment on compilation to LLVM (Section 7.1) does.

$$x \in \mathbb{X} \text{ (program variables)} \qquad z \in \mathbb{Z}(\text{ numeric constant)} \qquad \diamond \in \{+, -, *, /\}) \text{ (binary operation)}$$

$$e \in \mathbb{E}_{\mathbb{X}} \text{ (program expression)} \triangleq z \mid x \mid e \diamond e \qquad i \in \mathbb{I} \text{ (instruction)} \triangleq x := e \mid i\,;i$$

$$\ell \in \mathbb{L} \text{ (location)} \qquad \mathcal{G} \in \mathbb{G} \triangleq \mathbb{L} \times \mathbb{L} \rightharpoonup \mathbb{E}_{\mathbb{X}} \text{ (program graph)} \qquad p \in \mathbb{P} \triangleq (\mathbb{L} \to \mathbb{I}) \times \mathbb{G} \text{ (program)}$$

$$\mathbb{\Sigma} \triangleq \mathbb{X} \to \mathbb{Z} \text{ (stores)} \qquad \mathbb{S} \triangleq \mathbb{L} \times \mathbb{\Sigma} \text{ (states)} \qquad \mathbb{S}_0 \triangleq \{\ell_0\} \times \mathbb{\Sigma} \text{ (initial states)}$$

$$\mathcal{E}[\![\cdot]\!](\cdot) : \mathbb{E}_{\mathbb{X}} \times \mathbb{\Sigma} \rightharpoonup \mathbb{Z} \qquad \mathcal{E}[\![z]\!](\sigma) = z \qquad \mathcal{E}[\![x]\!](\sigma) = \sigma[x] \qquad \mathcal{E}[\![e_1 \diamond e_2]\!](\sigma) = \mathcal{E}[\![e_1]\!](\sigma) \diamond \mathcal{E}[\![e_2]\!](\sigma)$$

$$\mathcal{I}[\![\cdot]\!](\cdot) : \mathbb{I} \times \mathbb{\Sigma} \rightharpoonup \mathbb{\Sigma} \qquad \mathcal{I}[\![x := e]\!](\sigma) = \sigma[x \leftarrow \mathcal{E}[\![e]\!](\sigma)] \qquad \mathcal{I}[\![i_1\,;i_2]\!](\sigma) = \mathcal{I}[\![i_2]\!](\mathcal{I}[\![i_1]\!](\sigma))$$

$$\rightarrowtail \in \mathbb{S} \times \mathbb{S} \quad (\ell_1, \sigma_1) \rightarrowtail (\ell_2, \sigma_2) \triangleq \sigma_2 = \mathcal{I}[\![\text{instr}[\ell_1]]\!](\sigma_1) \wedge \exists e : ((\ell_1, \ell_2) \mapsto e) \in \mathcal{G} \wedge \mathcal{E}[\![e]\!](\sigma_2) \neq 0$$

Fig. 3. Syntax (top) and semantics (bottom) of the example language.

3 BACKGROUND AND NOTATIONS

Notations. We view (partial) functions as sets of bindings $x \mapsto y$, replacing the curly braces $\{\}$ of set notation by square brackets $[]$, e.g., $[x \mapsto x + 1 \mid x \in \mathbb{Z}]$ defines the successor function while $[0 \mapsto 5, 1 \mapsto 7]$ represents an array of two elements. We write $[x \in \mathbb{S} \mapsto y]$ as a shorthand for $[x \mapsto y \mid x \in \mathbb{S}]$. We note the domain of a partial function f by $\text{dom}\, f$. We note by $A \rightharpoonup B$ the set of partial functions whose domain is contained in A and codomain is contained in B . The notation $A \to B$ denotes the set of total functions from A to B, i.e. partial functions whose domain is exactly A and whose codomain is a subset of B. If f is a function and $a \in \text{dom}\, f$, we note by $f[a]$ the image of a by f. We define $f[x \leftarrow y] \triangleq [a \in \text{dom}\, f \mapsto f[a] \mid a \neq x] \cup [x \mapsto y]$ the function f where the binding $x \mapsto y$ has been added (x may or may not be bound in f).

We represent graphs by partial functions from pairs of nodes (representing an edge from the first node to the second) to the label of the edge. Given a graph g and node n, we note $\text{preds}(n) \triangleq \{m : (m, n) \in \text{dom}\, g\}$ the set of *predecessors* of n in g.

Syntax. We express our analyses on the language defined in Figure 3, featuring assignments of expressions to variables, separation of conditionals into guards and nondeterminism [Dijkstra 1975], and unstructured control flow. It is close to the standard control-flow graph representation of imperative programs.

A program $p \in \mathbb{P} = (\text{instr}[\cdot], \mathcal{G})$ is expressed as a flow-chart doing transitions between *locations* $\ell \in \mathbb{L}$. To each location is mapped an *instruction* $\text{instr}[\ell] \in \mathbb{I}$, consisting in a sequence of assignments. The expressions $e \in \mathbb{E}_{\mathbb{X}}$ are standard (being either variables, constants, or binary operations); $\mathbb{E}_{\mathbb{X}}$ stands for "expressions over the program variables $\mathbb{X}$". The possibles transitions between locations are expressed using a graph $\mathcal{G} \in (\mathbb{L} \times \mathbb{L}) \rightharpoonup \mathbb{E}_{\mathbb{X}}$, mapping edges to a *guard* filtering the possible transitions. The choice between multiple outgoing edges is nondeterministic.

We assume the existence of an *initial location* ℓ_0 that has no predecessors. To simplify our definitions and proofs, we assume that there are at most 2 predecessors to any location ℓ.

Concrete Semantics. We denote by $\mathbb{\Sigma} \triangleq \mathbb{X} \to \mathbb{Z}$ the set of *stores*, i.e. mapping from variables to values, in the program, and by $\mathbb{S} \triangleq \mathbb{L} \times \mathbb{\Sigma}$ the set of states. The semantics is given as a transition system $(\mathbb{S}, \rightarrowtail)$, where $\rightarrowtail$ is a relation describing the transition between states. $\mathbb{S}_0 \triangleq \{\ell_0\} \times \mathbb{\Sigma}$ is the set of initial states. The full definition of $\rightarrowtail$ is given in Figure 3, and makes use of $\mathcal{E}[\![\cdot]\!](\cdot)$ and $\mathcal{I}[\![\cdot]\!](\cdot)$ (describing respectively the evaluation of expressions and instructions).

$$x_\ell \in \mathbb{V} \triangleq \mathbb{X} \times \mathbb{L} \text{ (symbolic variables)} \qquad e \in \mathbb{E}_\mathbb{V} \triangleq z \mid x_\ell \mid e \diamond e \text{ (symbolic expressions)}$$
$$\sigma^\sharp \in \mathbb{\Sigma}^\sharp \triangleq \mathbb{X} \to \mathbb{E}_\mathbb{V} \text{ (abstract stores)} \qquad \mathbb{\Sigma}^\sharp_\bot \triangleq \mathbb{\Sigma}^\sharp \cup \{\bot\} \qquad \Gamma \in \mathbb{\Gamma} \triangleq \mathbb{V} \rightharpoonup \mathbb{Z} \text{ (valuations)}$$

$$\gamma_{\mathbb{E}_\mathbb{V}} : \mathbb{E}_\mathbb{V} \to (\mathbb{\Gamma} \rightharpoonup \mathbb{Z}) \qquad \gamma_{\mathbb{E}_\mathbb{V}}(e) \triangleq [\Gamma \in \mathbb{\Gamma} \mapsto \mathcal{E}[\![e]\!](\Gamma)]$$
$$\gamma_{\mathbb{\Sigma}^\sharp} : \mathbb{\Sigma}^\sharp \to (\mathbb{\Gamma} \rightharpoonup \mathbb{\Sigma}) \qquad \gamma_{\mathbb{\Sigma}^\sharp}(\sigma^\sharp) \triangleq [\Gamma \in \mathbb{\Gamma} \mapsto [x \in \mathbb{X} \mapsto \gamma_{\mathbb{E}_\mathbb{V}}(\sigma^\sharp[x])(\Gamma)]]$$
$$\gamma_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot} : (\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot) \to \mathcal{P}(\mathbb{S}) \qquad \gamma_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp) \triangleq \{(\ell, f[\Gamma]) \mid \ell \in \mathbb{L}, p^\sharp[\ell] \neq \bot, f \in \gamma_{\mathbb{\Sigma}^\sharp}(p^\sharp[\ell]), \Gamma \in \mathbb{\Gamma}\}$$

$$\mathcal{E}^\sharp[\![\cdot]\!](\cdot) : \mathbb{E}_\mathbb{X} \times \mathbb{\Sigma}^\sharp \to \mathbb{E}_\mathbb{V} \qquad \mathcal{E}^\sharp[\![e]\!](\sigma^\sharp) \triangleq \mathrm{subst}(e, \sigma^\sharp)$$

$$\mathcal{I}^\sharp[\![\cdot]\!](\cdot) : \mathbb{I} \times \mathbb{\Sigma}^\sharp \to \mathbb{\Sigma}^\sharp \qquad \begin{array}{l} \mathcal{I}^\sharp[\![i_1\,;i_2]\!](\sigma^\sharp) \triangleq \mathcal{I}^\sharp[\![i_2]\!](\mathcal{I}^\sharp[\![i_1]\!](\sigma^\sharp)) \\ \mathcal{I}^\sharp[\![x := e]\!](\sigma^\sharp) \triangleq \sigma^\sharp[x \leftarrow \mathcal{E}^\sharp[\![e]\!](\sigma^\sharp)] \end{array}$$

$$\phi^{x,\ell} : \mathbb{E}_\mathbb{V} \times \mathbb{E}_\mathbb{V} \to \mathbb{E}_\mathbb{V} \qquad \phi^{x,\ell}(e_1, e_2) \triangleq \begin{cases} e_1 & \text{if } e_1 = e_2 \\ x_\ell & \text{otherwise} \end{cases}$$

$$\sqcup^\ell : \mathbb{\Sigma}^\sharp \times \mathbb{\Sigma}^\sharp \to \mathbb{\Sigma}^\sharp \qquad \sigma^\sharp_1 \sqcup^\ell \sigma^\sharp_2 \triangleq [x \in \mathbb{X} \mapsto \phi^{x,\ell}(\sigma^\sharp_1[x], \sigma^\sharp_2[x])]$$

$$\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp) \triangleq \left[\ell' \in \mathbb{L} \mapsto \begin{cases} [x \in \mathbb{X} \mapsto x_{\ell_0}] & \text{if } \ell' = \ell_0 \\ \bigsqcup^{\ell'} \{\mathcal{I}^\sharp[\![\mathrm{instr}[\ell]]\!](p^\sharp[\ell]) \mid \mathcal{G}[\ell, \ell'] \neq 0 \wedge p^\sharp[\ell] \neq \bot\} & \text{if } \ell' \neq \ell_0 \end{cases}\right]$$

Fig. 4. The symbolic expression domain: abstraction (top), concretizations (middle), and operations (bottom).

Least Fixpoint and Collecting Semantics. Given $\mathcal{F}_S \in S \to S$, we note by $\mathrm{lfp}(\mathcal{F}_S)$ the *least fixpoint* of $\mathcal{F}_S$, i.e. the smallest element $X \in S$ such that $\mathcal{F}_S(X) = X$. We will mainly use the set of reachable states as our main collecting semantics. It is expressed as the least fixpoint of the following function $\mathcal{F}_{\mathcal{P}(\mathbb{S})} \in \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$:

$$\mathcal{F}_{\mathcal{P}(\mathbb{S})}(X) \triangleq \mathbb{S}_0 \cup \{s' \mid \exists s \in X : s \rightarrowtail s'\}$$

The set of reachable states is noted by $\mathbb{S}^*$, i.e.

$$\mathbb{S}^* = \mathrm{lfp}(\mathcal{F}_{\mathcal{P}(\mathbb{S})}) = \{s \in \mathbb{S} : \exists s_0 \in \mathbb{S}_0, s_0 \rightarrowtail^* s\}$$

where $\rightarrowtail^*$ is the transitive and reflexive closure of the transition relation $\rightarrowtail$.

4 SYMBOLIC EXPRESSION ANALYSIS

This section formalizes the definition of the symbolic expression analysis presented in the overview (Section 2.1). The analysis is formalized as a forward abstract interpretation [Cousot and Cousot 1977].

4.1 The Symbolic Expression Abstract Domain

4.1.1 Symbolic Expression, Abstract Store, and Main Program Abstraction. The structure of our abstraction (fully defined in Figure 4) follows the concrete semantics.

Concrete values are abstracted by *symbolic expressions* $e \in \mathbb{E}_\mathbb{V}$, where $\mathbb{E}_\mathbb{V}$ is similar to expressions $\mathbb{E}_\mathbb{X}$ except that variables are not program variables $\mathbb{X}$, but elements from a set $\mathbb{V}$ of *symbolic variables*. In this section, $\mathbb{V} \triangleq \mathbb{X} \times \mathbb{L}$, and we note symbolic variables by x_ℓ where $x \in \mathbb{X}$ and $\ell \in \mathbb{L}$. Note that the fact that our symbolic variables are given a deterministic, unique name (instead of generating "fresh" symbolic variables as is common in existing formula-based abstract domains (e.g.,[Chang

and Leino 2005; Chang and Rival 2013; Gange et al. 2016; Illous et al. 2021]) is a distinctive feature of our analysis.

As is common in abstract interpretation [Cousot and Cousot 1977], symbolic expressions are given a meaning using a concretization function $\gamma_{\mathbb{E}_\mathbb{V}}$, that associates an abstract element to the set of concrete elements that it describes. Instead of concretizing a symbolic expression to a set of values, we concretize it as a function from valuations to values, where a valuation Γ is a function that defines how symbolic variables are bound to a concrete value. Given a symbolic expression e, $\gamma_{\mathbb{E}_\mathbb{V}}(e)$ is a function that takes a Γ and returns the value obtained by evaluation of the symbolic expression after replacement of the symbolic variables by their value in Γ. This function can be seen as representing a set of values (the image of the function); but the point of having a function is that different symbolic expressions can be related by sharing a Γ.

Concrete stores are abstracted by *abstract stores* $\sigma^\sharp \in \mathbb{\Sigma}^\sharp \triangleq \mathbb{X} \to \mathbb{E}_\mathbb{V}$, which simply maps program variables to symbolic expressions. Sometimes we need to represent a set of no abstract store, and use the symbol $\bot$ for this; we define $\mathbb{\Sigma}^\sharp_\bot \triangleq \mathbb{\Sigma}^\sharp \cup \{\bot\}$. The concretization function for abstract stores $\gamma_{\mathbb{\Sigma}^\sharp}$ is a function from valuations to concrete stores; here we observe that the same Γ is passed to the concretization of all the symbolic expression in the store to get the corresponding value, thereby allowing relations between symbolic expressions. For instance, the abstract store $[x \mapsto 2 * z_{\ell_1}; y \mapsto 6 * z_{\ell_1}; z \mapsto z_{\ell_3}]$ represents concrete stores where x is even, y is a multiple of x, and z is arbitrary. Again, the function returned by $\gamma_{\mathbb{\Sigma}^\sharp}$ can be seen as set of concrete stores (the image of the function).

Finally, as is standard in flow-sensitive dataflow analyses, our main program abstraction is an element $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ assigning to each program location an abstract store, or $\bot$ (which happens when the location is unreachable, or the computation of the program abstraction is incomplete). We concretize elements $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ as a set of states: a state (ℓ, σ) is described by $p^\sharp$ whenever σ is described by the abstract store $p^\sharp[\ell]$ (i.e., if Γ exists such that $\gamma_{\mathbb{\Sigma}^\sharp}([p^\sharp[\ell]])[\Gamma] = \sigma$).

4.1.2 Analysis Operations. The abstract evaluation of expressions ($\mathcal{E}^\sharp[\![\cdot]\!](\cdot)$) consists in substituting the free program variables in the expression with the corresponding symbolic expression in the abstract store. The abstract evaluation of instructions ($\mathcal{I}^\sharp[\![\cdot]\!](\cdot)$) mimics the concrete evaluation by updating the abstract store after abstract evaluation of the expression.

The most interesting operation is the join operation $\sqcup^\ell$, which rely on a kind of "join of expressions" performed by a function ϕ. The current definition of ϕ is simple: when the two arguments of ϕ differ, a new symbolic variable is created; when they are equal, that expression is returned (this latter case corresponds to the notion of minimal number of ϕ functions in an SSA translation [Cytron et al. 1991]). The additional x, ℓ argument allows deterministic creation of variable names, which is important to ensure termination of the analysis. The $\sqcup^\ell$ operation is extended to operate on a set (i.e. $\sqcup^\ell\{\} = \bot$, $\sqcup^\ell\{\sigma^\sharp\} = \sigma^\sharp$, $\sqcup^\ell\{\sigma^\sharp_1, \sigma^\sharp_2\} = \sigma^\sharp_1 \sqcup^\ell \sigma^\sharp_2$).

Example 4.1. When analyzing ℓ_1 after the first loop iteration, we have to join two abstract stores:

$$\sigma^\sharp_1 = [\texttt{inc} \mapsto 1, \texttt{time} \mapsto 0, \texttt{i} \mapsto 0, \texttt{hours} \mapsto \texttt{hours}_{\ell_0}] \text{ and}$$
$$\sigma^\sharp_2 = [\texttt{inc} \mapsto 1, \texttt{time} \mapsto 0 + 1, \texttt{i} \mapsto 0 + 1, \texttt{hours} \mapsto \texttt{hours}_{\ell_3}].$$

The result is in Figure 1a (top): `inc` is bound to 1 in both incoming abstract stores, and thus stays bound to 1; while a new symbolic variable is created for `time`, `i` and `hours` .

The main transfer function $\mathcal{F}_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot}$ does a classic fixpoint step in flow-sensitive dataflow analysis, by merging at each location ℓ' the states coming from its predecessors. Note that this transfer function makes use of the information that a predecessor location ℓ is unreachable (i.e., $p^\sharp[\ell] = \bot$)

or that the guard on an edge always evaluates to false (i.e. $\mathcal{G}[\ell, \ell'] = 0$) to improve its precision. When combined with other domains (e.g. using a reduced product [Cousot and Cousot 1979] with a numerical abstraction), it is easy for the symbolic expression domain to benefit from interaction with the other domains by determining more unreachable locations or dead branches; in the end this mechanism ensures the automatic preservation of SSA invariants when another domain discovers dead code.

There is an order on $\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ (see Section 6) and the lattice has a finite height (which is $|\mathbb{L}| \times |\mathbb{X}|$, where $|S|$ is the cardinal of S) : hence, a Kleene iteration [Cousot and Cousot 1977] of $\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}$ will find a fixpoint in a finite number of steps.

4.1.3 Soundness of the Analysis. Due to space constraints, the full proof of soundness of this analysis is put in Lemerre [2023a, Appendix A]. Here we summarize the main interesting points about the proof.

First, note that if the SSA abstraction is sound and complete, the symbolic expression analysis is sound, but incomplete. It is in fact an advantage to have an incomplete abstraction that goes along the exact SSA abstraction: while the SSA abstraction exactly translates the semantics of the original program, the concretization of the incomplete the symbolic expression domain allows assessing the precision of this translation.

That said, a second observation is that the $\mathcal{E}^\sharp[\![\cdot]\!](\cdot)$ and $\mathcal{I}^\sharp[\![\cdot]\!](\cdot)$ operations are exact, in addition to being sound. This property is important to prove the soundness and completeness of the SSA translation.

A last, unusual, point is that the $\sqcup^\ell$ operator *is not sound* when applied to any pair of elements in $\mathbb{\Sigma}^\sharp$. The problem comes from the fact that we reuse variable identifiers, which could thus create spurious relations between terms (this problem does not happen in analyses which only create fresh variables). Consider for instance:

$$[\mathsf{x} \mapsto 1 + \mathsf{y}_{\ell_6}, \mathsf{y} \mapsto 2] \sqcup^{\ell_6} [\mathsf{x} \mapsto 1 + \mathsf{y}_{\ell_6}, \mathsf{y} \mapsto 3] = [\mathsf{x} \mapsto 1 + \mathsf{y}_{\ell_6}, \mathsf{y} \mapsto \mathsf{y}_{\ell_6}]$$

The resulting abstract store implies the relation $\mathsf{x} = 1 + \mathsf{y}$ through the shared use of y_{ℓ_6}, but this relation is false in both arguments to $\sqcup^{\ell_6}$.

However, the complete analysis is sound, because 1. $\sqcup^\ell$ is sound in the case when ℓ does not occur in at least one of the arguments to $\sqcup^\ell$, and 2. during the fixpoint computation, $p^\sharp$ is always such that $\sqcup^\ell$ is applied in the sound case (because ℓ never occurs in the abstract stores on the simple paths from ℓ_0 to ℓ).

4.2 Building ϕ Dynamically to Find More Equalities

The abstract domain defined Figure 4 sometimes assigns different symbolic expressions to program variables that could be proved equal. Consider again Example 4.1 (page 10). The program variables `time` and `i` are bound to the same symbolic expression in each of $\sigma^\sharp_1$ and $\sigma^\sharp_2$, and are thus equal; it would thus be more precise to have these program variables be bound to the same symbolic variable. What we want is in Figure 1a ($p^\sharp[\ell_1]$ on the second line): there, both `time` and `i` are bound to the same symbolic variable $\texttt{time\_i}_{\ell_1}$, which represents the value of both `time` and `i` at ℓ_1.

Building ϕ Dynamically. For this to work, we need to perform small changes to our static analysis. The ϕ function can no longer have a static definition, but will be computed based on the arguments of $\sqcup^\ell$. For this, we define $\Phi \triangleq ((\mathbb{E}_\mathbb{V} \times \mathbb{E}_\mathbb{V}) \rightharpoonup \mathbb{E}_\mathbb{V})$ to be the set of ϕ functions, and we define $\Phi^\ell : \mathbb{\Sigma}^\sharp \times \mathbb{\Sigma}^\sharp \to \Phi$ as the function computing ϕ from two abstract states. A definition for Φ^ℓ that

returns a ϕ equivalent to the one statically defined in Figure 4 is as follows:

$$\Phi^\ell_= : \mathbb{\Sigma}^\sharp \times \mathbb{\Sigma}^\sharp \to \Phi \qquad \Phi^\ell_=(\sigma^\sharp_1, \sigma^\sharp_2) \triangleq [(e,e) \mapsto e \mid \exists x \in \mathbb{X} : \sigma^\sharp_1[x] = \sigma^\sharp_2[x] = e]$$

$$\Phi^\ell_\neq : \mathbb{\Sigma}^\sharp \times \mathbb{\Sigma}^\sharp \to \Phi \qquad \Phi^\ell_\neq(\sigma^\sharp_1, \sigma^\sharp_2) \triangleq [(\sigma^\sharp_1[x], \sigma^\sharp_2[x]) \mapsto x_\ell \mid x \in \mathbb{X} \wedge \sigma^\sharp_1[x] \neq \sigma^\sharp_2[x]]$$

$$\Phi^\ell : \mathbb{\Sigma}^\sharp \times \mathbb{\Sigma}^\sharp \to \Phi \qquad \Phi^\ell(\sigma^\sharp_1, \sigma^\sharp_2) \triangleq \Phi_\neq(\sigma^\sharp_1, \sigma^\sharp_2) \cup \Phi_=(\sigma^\sharp_1, \sigma^\sharp_2)$$

We actually defined three Φ functions: $\Phi^\ell_=$ returns the ϕ function for the cases where the arguments to ϕ are equal, $\Phi^\ell_\neq$ for the cases where they are different, and Φ^ℓ handles both cases. Later on, we will focus on $\Phi^\ell_\neq$ as it corresponds to the ϕ functions in standard SSA.

We now need to change the definition of $\sqcup^\ell$ to make use of Φ^ℓ:

$$\sigma^\sharp_1 \sqcup^\ell \sigma^\sharp_2 \triangleq \text{let } \phi \triangleq \Phi^\ell(\sigma^\sharp_1, \sigma^\sharp_2) \text{ in } [x \in \mathbb{X} \mapsto \phi(\sigma^\sharp_1[x], \sigma^\sharp_2[x])]$$

More Precise Global Value Numbering. With the above changes, we compute the same analysis as in the previous section, the only difference being that the ϕ function is computed dynamically. But we can now adapt the analysis to return the same symbolic variable when two pairs of variables point to the same symbolic expression. It suffices first to change the definition of $\mathbb{V}$ so that symbolic variables are named according to *a set of* program variables $X \in \mathcal{P}(\mathbb{X})$, instead of a *single* variable $x \in \mathbb{X}$: i.e., now $\mathbb{V} \triangleq \mathcal{P}(\mathbb{X}) \times \mathbb{L}$. Second, we change the definition of $\Phi^\ell_\neq$ as follows:

$$\Phi^\ell_\neq(\sigma^\sharp_1, \sigma^\sharp_2) \triangleq [\ (e_1, e_2) \mapsto X_\ell \mid e_1 \neq e_2 \ \wedge\ X \neq \{\} \ \wedge\ \forall x \in X : \sigma^\sharp_1[x] = e_1 \ \wedge\ \sigma^\sharp_2[x] = e_2\ ]$$

This new definition associates to each pair of different expressions e_1, e_2 all the variables that contain these expressions in the abstract stores. The new analysis is still sound, and all the theorems of Lemerre [2023a, Appendix A] still hold (only the proof of Lemma A.6 needs to be modified).

Note. The formalization requires $\sqcup^\ell$ to perform two passes (one to build ϕ, and one to use it), but it is feasible to do it in one pass. It suffices to return a fresh (integer) identifier every time a new pair of expressions e_1, e_2 is passed to ϕ, and modify, using a mutable state, the relation between the identifier and the set of variables that it represents. Actually the implementation may only use the fresh identifier and completely omit to represent the set of variables, which is useful only to understand what the integer identifier (i.e., value number [Rosen et al. 1988]) represents.

Building the Global Value Graph. We return to the ϕ function computed by $\Phi^{\ell_1}_\neq$ (Example 4.1), which is

$$\Phi^{\ell_1}_\neq(\sigma^\sharp_1, \sigma^\sharp_2) = [(0, \texttt{time\_i}_{\ell_3}) \mapsto \texttt{time\_i}_{\ell_1}, (\texttt{hours}_{\ell_0}, \texttt{hours}_{\ell_3}) \mapsto \texttt{hours}_{\ell_1}]$$

In other words, $\texttt{time\_i}_{\ell_1} = \phi(0, \texttt{time\_i}_{\ell_3})$ and $\texttt{hours}_{\ell_1} = \phi(\texttt{hours}_{\ell_0}, \texttt{hours}_{\ell_3})$: the computed ϕ function corresponds to the binding of a symbolic variable to a ϕ expression in usual SSA. Thus, if we first collect all the terms that appear in $p^\sharp$, the result of the symbolic analysis, with all the ϕ functions computed by $\mathcal{F}_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\perp}(p^\sharp)$ viewed as a definition of symbolic variables, we obtain a cyclic term graph which is the global value graph. This can be done efficiently and incrementally; we do not explain this further as the generation of SSA is a generalization of this construction (as SSA is the global value graph with additional information about the control-flow).

4.3 Note on Practical Implementation

Several points must be observed for an efficient practical implementation. Firstly, the symbolic expressions should be hash-consed (i.e. the terms should be built as a minimal DAG [Reif and Lewis 1986]). This ensures that the total memory taken by the symbolic expressions corresponds to the number of nodes in the global value graph, and that the complexity of a $\mathcal{E}^\sharp[\![e]\!](\sigma^\sharp)$ operation is in $O(|e|)$ (i.e. proportional to the number of nodes in e).

Secondly, the abstract stores $\sigma^\sharp \in \mathbb{\Sigma}^\sharp$ should be implemented using balanced binary search trees [Blanchet et al. 2002] or, better, Okasaki maps [Okasaki and Gill 1998] (see Section 7.1). This allows the complexity of insertion in the map for the interpretation of assignments to be in $O(\log |\mathbb{X}|)$, and maximum complexity of the $\sqcup^\ell$ operation to be in $O(|\mathbb{X}|)$ (but the average complexity of the $\sqcup^\ell$ operation is in $O(\Delta \log |\mathbb{X}|)$, where Δ is the number of variables that differs in the arguments of $\sqcup^\ell$).

Thirdly, a chaotic iteration strategy [Cousot 1977, 1978] should be used to propagate states efficiently. For instance, using the iterative iteration strategy of Bourdoncle [1993] ensures that each location of the program will be analyzed less than h time, where h is the height of the lattice of abstract stores. It can be shown that $h = |\mathbb{X}|$ (we can view $\mathbb{E}_\mathbb{V}$ as a lattice of height 2, and abstract stores contain $|\mathbb{X}|$ of them).

Thus, assuming that the size of all expressions in the program is bound by a small constant (in addition to the assumption that nodes have no more than 2 predecessors), the maximum complexity of our analysis is $O(|\mathbb{L}| * |\mathbb{X}|^2)$. However, in a typical analysis, only a few iterations of the outermost component will be needed for the fixpoint iteration to converge (in general almost all the program variables converge simultaneously), the number of join points is smaller than the number of instructions, and many program variables have the same expression during joins (especially if there are many variables), so the typical complexity is $O(|\mathbb{L}| * \log |\mathbb{X}|)$, which makes the analysis quite efficient in practice (see Section 7.1).

Finally, the evaluation of expressions may perform more than just substitutions – it can perform some rewriting operations such as arithmetic simplifications, constant folding, normalization of commutative operations, etc. so as to detect more equalities between variables. It should also be possible to replace symbolic expressions by equivalent classes of equal expressions – using equality saturation [Nelson 1980; Willsey et al. 2021] to find more equalities between expressions.

5 CONSTRUCTION OF AN SSA GRAPH USING ABSTRACT INTERPRETATION

The previous section explained how a symbolic expression abstraction $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ could be computed using an analysis based on abstract interpretation. This section explains how this abstraction can be turned into an SSA graph.

5.1 Scope and Dominance Properties

First, we study some properties of the result $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ of a symbolic expression analysis. More generally, in this section, we assume that $p^\sharp$ is a fixpoint of $\mathcal{F}_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot}$ (although the results also apply when $p^\sharp$ is a postfixpoint) in a program whose graph is $\mathcal{G}$. The study of these properties is important to define the notion of *scope* and the semantics of SSA graphs.

We recall the notion of *domination*: we say that ℓ_D *dominates* ℓ (in a graph $\mathcal{G}$) if every path from ℓ_0 to ℓ in the graph $\mathcal{G}$ must go through ℓ_D. We also define the *location of a variable* : if $v \in \mathbb{V}$ is a pair whose second element is a location ℓ, then $\mathrm{loc}(v) = \ell$. Informally, $\mathrm{loc}(v)$ corresponds, in traditional SSA, to the basic block where the variable v will be defined.

We saw that the symbolic expression analysis (alone or paired with other abstract domains) can determine if an edge or a location is unreachable. To simplify our definitions, we assume that dead code elimination has been performed on $\mathcal{G}$ – otherwise, excluding the paths that go through unreachable nodes would make some notions, like domination, more complex.

Strictness. First we prove a property corresponding to the *dominance* or *strictness* property of SSA [Rastello 2016]. In strict SSA, if a variable occurs at location ℓ, then it is defined at a location ℓ_D that dominates ℓ. Note however that our theorem apply to a symbolic expression abstraction $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$, which is a different mathematical object than an SSA program.

Theorem 5.1 (Strictness). $\forall \ell \in \mathbb{L}, x \in \mathbb{X}, v \in \mathbb{V} : \quad v$ *occurs in* $p^\sharp[\ell][x] \implies \mathrm{loc}(v)$ *dominates* ℓ.

The full proof is given in Lemerre [2023a, Appendix B].

Note that this result only applies when $p^\sharp$ is a fixpoint: during the fixpoint computation, there may be a variable x_{ℓ_D} that gets propagated to a location ℓ without ℓ being dominated by ℓ_D (e.g. when another path to ℓ is not yet propagated).

Scope. We now define a notion of *scope*, that intuitively corresponds to the set of locations on which an expression has a meaning: i.e. the locations that are dominated by the locations of all the variables of the expression.

Definition 5.2 (Scope). Given a graph G, the function $\mathrm{scope}_G : \mathbb{E}_\mathbb{V} \to \mathcal{P}(\mathbb{L})$ is defined as follows:

$$\mathrm{scope}_G(v) \triangleq \{\ell \in \mathbb{L} \mid \mathrm{loc}(v) \text{ dominates } \ell \text{ in the graph } G\} \qquad \mathrm{scope}_G(z) \triangleq \mathbb{L}$$

$$\mathrm{scope}_G(e_1 \diamond e_2) \triangleq \mathrm{scope}_G(e_1) \cap \mathrm{scope}_G(e_2)$$

Another interpretation of the scope of an expression e is the set of locations of $p^\sharp$ where e may appear when the analysis is complete.

Theorem 5.3 ($p^\sharp$ respects scope). $\forall \ell \in \mathbb{L} : \forall x \in \mathbb{X} : \ell \in \mathrm{scope}_\mathcal{G}(p^\sharp[\ell][x])$

Proof. Direct consequence of Theorem 5.1 and definition of scope. □

We say that the scope of e is *defined* when $\mathrm{scope}_G(e)$ is non-empty. Intuitively, if the scope of a symbolic expression e is undefined, then e is meaningless.

Corollary 5.4. $\forall \ell \in \mathbb{L} : \forall x \in \mathbb{X} : \mathrm{scope}_\mathcal{G}(p^\sharp[\ell][x])$ *is defined.*

Example 5.5. Here is the scope of some expressions for the program of Figure 1b, where $\mathcal{G}$ corresponds to the graph of the program. All these expressions are defined except $\texttt{time\_i}_{\ell_2} + \texttt{hours}_{\ell_3}$.

$$\mathrm{scope}_\mathcal{G}(\texttt{hours}_{\ell_1}) = \{\ell_1, \ell_2, \ell_3\} \qquad \mathrm{scope}_\mathcal{G}(1 * 3600) = \mathbb{L} = \{\ell_0, \ell_1, \ell_2, \ell_3\}$$
$$\mathrm{scope}_\mathcal{G}(\texttt{time\_i}_{\ell_2}) = \{\ell_2\} \qquad \mathrm{scope}_\mathcal{G}(\texttt{hours}_{\ell_1} + \texttt{hours}_{\ell_3}) = \{\ell_1, \ell_2, \ell_3\} \cap \{\ell_3\} = \{\ell_3\}$$
$$\mathrm{scope}_\mathcal{G}(\texttt{hours}_{\ell_3}) = \{\ell_3\} \qquad \mathrm{scope}_\mathcal{G}(\texttt{time\_i}_{\ell_2} + \texttt{hours}_{\ell_3}) = \{\ell_2\} \cap \{\ell_3\} = \{\}$$

Single-Location Scope. The scope of any expression, when defined, can be represented by a single location; this property comes from the fact that domination is a tree relation [Appel 1998a]. The following function $\overline{\mathrm{scope}}$ expresses how this location can be computed (where the min of two locations is the location dominated by the other; this min exists when the scope of e is defined).

$$\overline{\mathrm{scope}}_G(v) \triangleq \mathrm{loc}(v) \qquad \overline{\mathrm{scope}}_G(z) \triangleq \ell_0 \qquad \overline{\mathrm{scope}}_G(e_1 \diamond e_2) \triangleq \min(\overline{\mathrm{scope}}_G(e_1), \overline{\mathrm{scope}}_G(e_2))$$

Theorem 5.6. $\mathrm{scope}_\mathcal{G}(e) = \{\ell \mid \overline{\mathrm{scope}}_\mathcal{G}(e)$ *dominates* $\ell\}$.

The full proof can be found in Lemerre [2023a].

We use this $\overline{\mathrm{scope}}$ operation when translating an expression in our SSA representation into assignment of a temporary variable in usual SSA: $\overline{\mathrm{scope}}(e)$ represents the most hoisted location where to place the assignment.

5.2 The SSA graph

We can now present (Figure 5) the syntax and semantics of our variant of SSA, the SSA graph (note that some authors [Stanier 2016] call the global value graph the SSA graph, which should not be confused with the SSA graph that we present here). The main use of this form of SSA is to be used as an abstraction of an abstract domain; however, its simple syntax and semantics may be useful in

$$\hat{\mathbb{G}} \triangleq (\mathbb{L} \times \mathbb{L}) \rightharpoonup (\mathbb{E}_{\mathbb{V}} \times (\mathbb{V} \rightharpoonup \mathbb{E}_{\mathbb{V}})) \text{ (SSA graph)} \quad \hat{\mathbb{S}} \triangleq \mathbb{L} \times \mathbb{\Gamma} \text{ (SSA state)} \quad \mathbb{\Gamma} \triangleq \mathbb{V} \rightharpoonup \mathbb{Z} \text{ (valuation)}$$

$$\text{unbind}_{\hat{g}} : \mathbb{L} \times \mathbb{\Gamma} \to \mathbb{\Gamma} \qquad \text{unbind}_{\hat{g}}(\ell, \Gamma) \triangleq [\, v \in \text{dom}\,\Gamma \mapsto \Gamma[v] \mid \ell \in \text{scope}_{\hat{g}}(v) \,]$$

$$\text{bind} : (\mathbb{V} \rightharpoonup \mathbb{E}_{\mathbb{V}}) \times \mathbb{\Gamma} \to \mathbb{\Gamma} \qquad \text{bind}(B, \Gamma) \triangleq \left[v \in \mathbb{V} \mapsto \begin{cases} \mathcal{E}[\![B[v]]\!](\Gamma) & \text{if } v \in \text{dom}\, B \\ \Gamma[v] & \text{otherwise} \end{cases} \right]$$

$$\rightsquigarrow_{\hat{g}} \in \hat{\mathbb{S}} \times \hat{\mathbb{S}} \qquad (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma') \triangleq \exists e \in \mathbb{E}_{\mathbb{V}}, B \in \mathbb{V} \rightharpoonup \mathbb{E}_{\mathbb{V}} : \begin{array}{l} (\ell, \ell') \mapsto (e, B) \in \hat{g} \\ \wedge\, \mathcal{E}[\![e]\!](\Gamma) \neq 0 \\ \wedge\, \Gamma' = \text{unbind}_{\hat{g}}(\ell', \text{bind}(B, \Gamma)) \end{array}$$

Fig. 5. Syntax and semantics of SSA graphs.

other contexts. A distinctive feature of this form (compared to traditional SSA form) is that it is *implicitly scoped*: we define the location of definition only for the symbolic variables (corresponding to the phi variables in traditional SSA form), and not for every expression (corresponding to other variables in traditional three-address SSA form). However, the largest scope in which an expression e is defined is given by $\overline{\text{scope}}(e)$ in the previous section; hence the scope of e is defined implicitly (and we can use this scope when translating to traditional SSA).

Syntax of SSA Graphs. Our SSA representation is a graph $\hat{g} \in \hat{\mathbb{G}}$ with $\hat{\mathbb{G}} \triangleq (\mathbb{L} \times \mathbb{L}) \rightharpoonup \mathbb{E}_{\mathbb{V}} \times (\mathbb{V} \rightharpoonup \mathbb{E}_{\mathbb{V}})$. The graph consists in a set of bindings $(\ell, \ell') \mapsto (e, B)$ representing an edge from ℓ to ℓ' labeled by a guard e and binding B, where a binding is a mapping from some variables to terms. Figure 1c presents the SSA graph produced by our analysis of the program of Figure 1b.

Semantics of SSA Graphs. A graph $\hat{g} \in \hat{\mathbb{G}}$ represents a transition system $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ between symbolic states $\hat{s} \in \hat{\mathbb{S}} \triangleq \mathbb{L} \times \mathbb{\Gamma}$ where a symbolic state consists of a control location ℓ and a valuation. The valuations $\Gamma \in \mathbb{\Gamma} \triangleq \mathbb{V} \rightharpoonup \mathbb{Z}$ are those used in the concretization of our symbolic expression abstraction, and assign values to symbolic variables. The initial states $\hat{\mathbb{S}_0} \triangleq \{(\ell_0, \Gamma) \mid \text{dom}\,\Gamma = \{x_{\ell_0} \mid x \in \mathbb{X}\}\}$ intuitively describes the initial value of each program variable at the starting location ℓ_0.

The transition relation $\rightsquigarrow_{\hat{g}}$, formally defined in Figure 5, works as follows : transition from a state (ℓ, Γ) to a state at location ℓ' in the graph is possible only if 1. there is a $(\ell, \ell') \mapsto (e, B)$ edge in the graph $\hat{g}$, and 2. the guard e evaluates to true. The transition then consists in adding some new bindings to Γ using the definitions provided by B (operator bind), but also to unbind the variables that go out of scope in the new control location ℓ', using the notion of scope that we defined in the previous section. An example of execution trace for the SSA program given in Figure 1c is given Figure 2 (first line).

The use of unbind is unusual in SSA semantics: generally SSA variables are assigned rather than bound (see e.g. [Barthe et al. 2014; Cytron et al. 1991; Zhao et al. 2012]). Our proof of soundness still holds with the assignment semantics (and is actually simpler), but unbind is important when SSA graphs are used in a combination with other abstract domains (e.g., to limit the scope where information about a variable must be retained).

Note that the semantics of SSA uses $\text{scope}_{\hat{g}}$ which depends on the SSA graph $\hat{g}$ instead of the program graph $\mathcal{G}$, as we want the SSA semantics to be independent of the original program. Note however, that $\text{scope}_{\hat{g}}$ and $\text{scope}_{\mathcal{G}}$ are the same when we define $\mathcal{G}$ to be the subset of the original program that excludes the parts that the symbolic expression analysis can prove unreachable.

$$\begin{aligned}
\mathrm{graph}(p^\sharp) &\triangleq \bigcup_{(\ell,\ell')\in \mathrm{dom}\,\mathcal{G}} \mathrm{edge}(p^\sharp,\ell,\ell') \\
\mathrm{edge}(p^\sharp,\ell,\ell') &\triangleq [(\ell,\ell') \mapsto (\hat{e}, B) \mid e \neq 0 \wedge p^\sharp[\ell] \neq \bot] \text{ where} \\
&\quad e = \mathcal{G}[\ell,\ell'], \\
&\quad \sigma'^\sharp = \mathcal{I}^\sharp[\![\mathrm{instr}[\ell]]\!](p^\sharp[\ell]), \\
&\quad \hat{e} = \mathcal{E}^\sharp[\![e]\!](\sigma'^\sharp), \\
&\quad B = \{p^\sharp[\ell'][x] \mapsto \sigma'^\sharp[x] \mid x \in \mathbb{X} \wedge p^\sharp[\ell'][x] \neq \sigma'^\sharp[x]\}
\end{aligned}$$

Fig. 6. Translation of a symbolic expression abstraction to an SSA graph.

5.3 From Symbolic Expressions to SSA Graphs

We now explain how the symbolic expression abstraction $p^\sharp$ can be translated into an SSA graph (Figure 6). The idea is that each edge of the original program corresponds to at most one edge in the SSA translation : if we detect that the source location ℓ is unreachable (i.e., $p^\sharp[\ell] = \bot$) or the guard is not satisfiable (i.e. $e = 0$), we leave the corresponding edge out of the SSA graph. The translation is done as follows: we first compute the abstract store $\sigma'^\sharp$ after execution of the block, and use it to translate the original guard e into an SSA guard $\hat{e}$; finally we examine the difference between $\sigma'^\sharp$ and $p^\sharp[\ell']$, the abstract store at ℓ', so that there is a binding in B whenever a new symbolic variable was introduced.

Example 5.7. Let $p^\sharp$ be the result of the analysis of our running example (Figure 1a). Then, $p^\sharp[\ell_1] \neq \bot$ and $\mathcal{G}[\ell_1,\ell_3] =$ "$\mathtt{hours} < 24'' \neq 0$, thus we have an edge from ℓ_1 to ℓ_3 in the SSA graph. Moreover:

$$\mathrm{edge}(p^\sharp,\ell_1,\ell_3) = \{(\ell_1,\ell_3) \mapsto (\mathcal{E}^\sharp[\![e]\!](\sigma'^\sharp), \{p^\sharp[\ell_3][x] \mapsto \sigma'^\sharp[x] \mid x \in \mathbb{X} \wedge p^\sharp[\ell_3][x] \neq \sigma'^\sharp[x]\})\}$$

where

$$\sigma'^\sharp = \mathcal{I}^\sharp[\![\mathrm{instr}[\ell_1]]\!](p^\sharp[\ell_1]) = \begin{bmatrix} \mathtt{inc} & \mapsto 1 \\ \mathtt{time} & \mapsto \mathtt{time\_i}_{\ell_1} \\ \mathtt{i} & \mapsto \mathtt{time\_i}_{\ell_1} \\ \mathtt{hours} & \mapsto \mathtt{time\_i}_{\ell_1}/(1*3600) \end{bmatrix} \text{ and } p^\sharp[\ell_3] = \begin{bmatrix} \mathtt{inc} & \mapsto 1 \\ \mathtt{time} & \mapsto \mathtt{time\_i}_{\ell_3} \\ \mathtt{i} & \mapsto \mathtt{time\_i}_{\ell_3} \\ \mathtt{hours} & \mapsto \mathtt{hours}_{\ell_3} \end{bmatrix}$$

The result is

$$\mathrm{edge}(p^\sharp,\ell_1,\ell_3) = \left\{(\ell_1,\ell_3) \mapsto \left(\mathtt{time\_i}_{\ell_1}/(1*3600) < 24, \begin{bmatrix} \mathtt{time\_i}_{\ell_3} \mapsto \mathtt{time\_i}_{\ell_1} \\ \mathtt{hours}_{\ell_3} \mapsto \mathtt{time\_i}_{\ell_1}/(1*3600) \end{bmatrix}\right)\right\}$$

as in Figure 1c.

5.4 The SSA Abstract Domain

5.4.1 Abstract Elements. We can now describe our SSA abstract domain (Figure 7). Elements of this domain are a pair $(p^\sharp, \hat{g})$ where $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ is an element of the symbolic expression abstract domain and $\hat{g} \in \hat{\mathbb{G}}$ is an SSA graph. Note that $p^\sharp$ is a flow-sensitive abstraction while $\hat{g}$ is a flow-insensitive abstraction.

5.4.2 Concretization. The central idea of the concretization is that an element $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ can be viewed as a function $\mathcal{H}_{p^\sharp} \in \hat{\mathbb{S}} \rightharpoonup \mathbb{S}$ mapping SSA states to concrete states. More precisely, given an SSA state $\hat{s} = (\ell, \Gamma)$ composed of a location ℓ and a valuation Γ, the abstract store $p^\sharp[\ell]$ can be viewed as a function from valuations to concrete stores using the concretization $\gamma_{\mathbb{\Sigma}^\sharp}$: indeed, $\gamma_{\mathbb{\Sigma}^\sharp}(p^\sharp[\ell]) \in \mathbb{\Gamma} \rightharpoonup \mathbb{\Sigma}$. $\mathcal{H}_{p^\sharp}$ retrieves a concrete store by applying Γ to this function, keeps the location ℓ, and use both to create a concrete state.

$$\mathbb{SSA}^\sharp = (\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot) \times \hat{\mathbb{G}}$$

$$\mathcal{H}_{p^\sharp} : \hat{\mathbb{S}} \rightharpoonup \mathbb{S} \qquad \gamma_{\hat{\mathbb{G}}}(\hat{g}) : \hat{\mathbb{G}} \to \mathcal{P}(\mathcal{P}(\hat{\mathbb{S}} \times \hat{\mathbb{S}}))$$

$$\mathcal{H}_{p^\sharp}(\ell, \Gamma) \triangleq (\ell, \gamma_{\mathbb{\Sigma}^\sharp}(p^\sharp[\ell])(\Gamma)) \qquad \gamma_{\hat{\mathbb{G}}}(\hat{g}) \triangleq \rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*}$$

$$\gamma_{\mathbb{SSA}^\sharp} : \mathbb{SSA}^\sharp \to \mathcal{P}(\mathcal{P}(\mathbb{S} \times \mathbb{S}))$$

$$\gamma_{\mathbb{SSA}^\sharp}(p^\sharp, \hat{g}) \triangleq \{(\mathcal{H}_{p^\sharp}(\hat{s}), \mathcal{H}_{p^\sharp}(\hat{s}')) \mid (\hat{s}, \hat{s}') \in \gamma_{\hat{\mathbb{G}}}(\hat{g})\}$$

$$\mathcal{F}_{\mathbb{SSA}^\sharp}(p^\sharp, \hat{g}) \triangleq (p'^\sharp, \mathrm{graph}(p'^\sharp)) \quad \text{where} \quad p'^\sharp = \mathcal{F}_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot}(p^\sharp)$$

Fig. 7. The SSA abstract domain : definition (top), concretizations (middle), transfer function (bottom)

Once we have the $\mathcal{H}_{p^\sharp}$ function, the meaning of the SSA abstraction $(p^\sharp, \hat{g})$ is given as the image of the transition system $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ by $\mathcal{H}_{p^\sharp}$. This image is a transition system $(\mathbb{S}, \hookrightarrow)$ where $\hookrightarrow$ is defined as follows:

$$\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}} : \hat{s} \rightsquigarrow_{\hat{g}} \hat{s}' \Leftrightarrow \mathcal{H}_{p^\sharp}(\hat{s}) \hookrightarrow \mathcal{H}_{p^\sharp}(\hat{s}'). \tag{1}$$

This definition implies that $\mathcal{H}_{p^\sharp}$ is a strong homomorphism from $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ to $(\mathbb{S}, \hookrightarrow)$ (indeed, the $\mathcal{H}$ in $\mathcal{H}_{p^\sharp}$ stands for homomorphism). The goal of our soundness theorem will be to establish that $\hookrightarrow$ overapproximates $\rightarrowtail$ (and, since the SSA translation is sound but also complete, we will also prove that $\hookrightarrow$ and $\rightarrowtail$ are equal). Figure 2 illustrates the $\mathcal{H}_{p^\sharp}$ homomorphism.

Now as we will see, the original transition system $(\mathbb{S}, \rightarrowtail)$ and the corresponding SSA transition system $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ only match when both transition systems are restricted to the reachable states. This justifies why in Figure 7 we concretize SSA graphs as a transition system between the reachable SSA states $\hat{\mathbb{S}}^*$. Formally, given $\hat{g}$, we define $\hat{\mathbb{S}}^*$ as $\hat{\mathbb{S}}^* \triangleq \{\hat{s} \in \hat{\mathbb{S}} : \exists \hat{s}_0 \in \hat{\mathbb{S}}_0, \hat{s}_0 \rightsquigarrow^*_{\hat{g}} \hat{s}\}$ where $\rightsquigarrow^*_{\hat{g}}$ represent the reflexive and transitive closure of $\rightsquigarrow_{\hat{g}}$. We define $\rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*} \triangleq \{(\hat{s}, \hat{s}') \in \rightsquigarrow_{\hat{g}} \mid \hat{s} \in \hat{\mathbb{S}}^*\}$.

5.4.3 Transfer Functions. The transfer function for the SSA abstraction consists in updating $p^\sharp$ at every step (using the previously defined $\mathcal{F}_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot}$), and using the graph operator to accordingly update the SSA graph $\hat{g}$.

This formal analysis can be made more efficient in practice. Note that the computation of the SSA graph at each iteration does not require the SSA graph at the previous iteration. Thus, in analyses where the SSA abstraction is only used to get an SSA translation, and is not used to improve the precision of other abstract domains, it suffices to call graph once at the end. This is what our evaluation of Section 7.1 does.

If the SSA graph is used in a combination with other domains, we can avoid recomputing the whole SSA graph at each iteration and instead compute the SSA graph incrementally along with a chaotic iteration. In Section 6 we present an algorithm that does this (and also ensures completeness of the SSA abstraction during the fixpoint computation).

5.4.4 Soundness of the Analysis. Our main soundness theorem states that once the analysis reaches a fixpoint, the SSA abstraction is sound and complete representation of the transition system of the original program.

Theorem 5.8 (Soundness and completeness of the SSA abstraction computation). *Let $(p^\sharp, \hat{g})$ be a fixpoint of $\mathcal{F}_{\mathbb{SSA}^\sharp}$. Then:* $\gamma_{\mathbb{SSA}^\sharp}(p^\sharp, \hat{g}) = \rightarrowtail_{|\mathbb{S}^*}$

Proof sketch. First, we show that $\mathcal{H}_{p^\sharp}$ is a strong homomorphism from $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ to $(\mathbb{S}, \rightarrowtail)$. The main idea of this proof is that $\mathcal{E}^\sharp[\![\cdot]\!](\cdot)$ and $\mathcal{I}^\sharp[\![\cdot]\!](\cdot)$ are exact operations, and the bindings to symbolic variables created by edge operator compensates the loss of precision introduced by $\sqcup^\ell$. Note that $\mathcal{H}_{p^\sharp}$ is still a strong homomorphism if we use the assignment semantics of SSA (i.e. without the unbind operation) for $\rightsquigarrow_{\hat{g}}$, as, following Theorem 5.1, in a transition from ℓ to ℓ' the unbound variables are those that no longer occur $p^\sharp[\ell']$.

This strong homomorphism allows proving completeness, but not soundness. Indeed, $p^\sharp$ does not represent all the states, only the reachable states (i.e. in general $\gamma_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp) \subset \mathbb{S}$). Because of this, $\rightsquigarrow_{\hat{g}}$ cannot represent all the transitions that $\rightarrowtail$ can. However, it can represent the transitions between the reachable states. Thus, we restrict $\rightarrowtail$ to $\rightarrowtail_{|\mathbb{S}^*} \triangleq \{(s, s') \in \rightarrowtail \mid s \in \mathbb{S}^*\}$. If we want $\mathcal{H}_{p^\sharp}$ to remain a strong homomorphism, we also need to restrict $\rightsquigarrow_{\hat{g}}$ to $\rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*}$. In that case, $\mathcal{H}_{p^\sharp}$ becomes a *surjective* strong homomorphism (and hence, a bisimulation [Sangiorgi 2009]) which implies that $\gamma_{\mathbb{SSA}^\sharp} = \rightarrowtail_{|\mathbb{S}^*}$. □

The full proof is given in Lemerre [2023a, Appendix C].

We can make two observations about this theorem. Firstly, a transition system is one of the most precise semantics in the hierarchy of program semantics [Cousot 2002], being equivalent to the maximal trace semantics[1]. Thus, the fact that the SSA abstraction is sound and complete when concretized into a transition system implies that it is also sound and complete for standard semantics such as set of traces, set of input/output relations, or set of reachable states.

Secondly, throughout the proof we are only interested in the soundness and completeness when the fixpoint is reached, which is unlike the usual proof structure in static analyses based on abstract interpretation. Indeed, the concretization of successive iterations of $\mathcal{F}_{\mathbb{SSA}^\sharp}$ is not monotonic, and some intermediate iterations include spurious traces (i.e. are not complete) that disappear when the fixpoint is reached (by contrast, the computation of successive iterations of $\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}$ was monotonic). The next section fixes this completeness issue in intermediate iterations, but note that the computation of the SSA abstraction will still be non-monotonic.

6 INCREMENTALLY COMPLETE CONSTRUCTION OF SSA

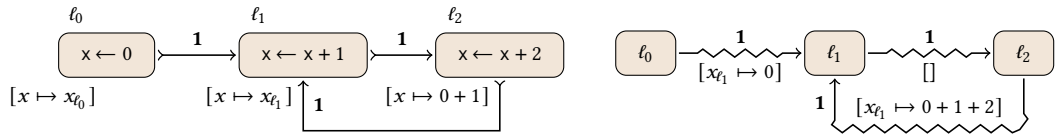
This section first explains why the previous construction of SSA abstraction is not incrementally complete, and provides a practical solution to fix the issue.

6.1 Normal Construction of SSA Abstraction is Not Incrementally Complete

Our computation of SSA using abstract interpretation is both sound (it describes all the behaviors of the original program) and complete (all the behaviors that it describes are present in the original program). However, it is not *continuously* sound and complete, i.e. intermediate computations may produce abstractions that are neither sound nor complete. While the fact that intermediate computations are not sound is expected (as abstract interpretation grows the abstraction until the fixpoint is reached), the incompleteness of intermediate computations is more problematic. This has practical consequences: if other domains rely on the intermediate SSA construction (e.g. for translation into a logic formula given to an SMT solver [Gurfinkel et al. 2015], to compute an exact abstract transformer [Reps et al. 2004], to retrieve the target of indirect jumps when analyzing machine code [Djoudi et al. 2016], etc.), this may lead to imprecisions of the static analysis.

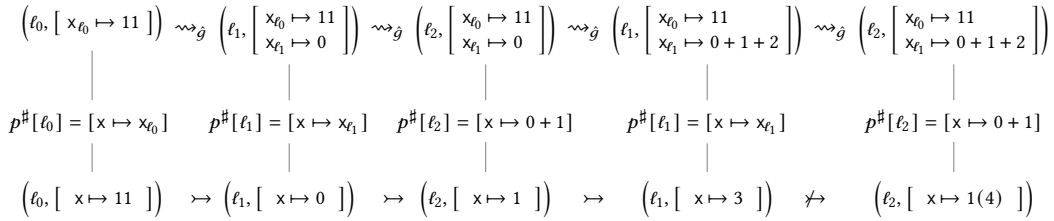
Consider the example of Figure 8. Figure 8a presents a program and the computation of the term abstraction $p^\sharp$ after 4 iterations of $\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}$ (the abstract store $p^\sharp[\ell]$ is given next to each program

[1] It is usual to define the set of traces from a transition system, but we can also define a transition relation from the successive states that appear in all the traces.



(a) Unfinished global value numbering (after 4 steps)

(b) Corresponding SSA program



(c) Faulty bisimulation. Top: Execution trace of the intermediate SSA program (b). Middle: Intermediate term abstraction. Bottom: Execution trace obtained by homomorphism (and execution trace of the original program).

Fig. 8. Example of an incomplete intermediate SSA abstraction, with the wrong execution trace.

location ℓ). We see that the fixpoint computation has joined, at the loop entry ℓ_1, the abstract stores coming from the beginning of the program ℓ_0 and from the loop back edge ℓ_2, creating the symbolic variable x_{ℓ_1}, but x_{ℓ_1} has not yet been propagated to ℓ_2.

Figure 8b presents the corresponding graph $\hat{g} \triangleq \mathrm{graph}(p^\sharp)$. We can see that the back-edge is labeled with the binding $[\mathsf{x}_{\ell_1} \mapsto 0 + 1 + 2]$, which is not the correct binding that we will get at the end of the analysis (the correct binding is $[\mathsf{x}_{\ell_1} \mapsto \mathsf{x}_{\ell_1} + 1 + 2]$).

However, the fact that the bindings on the SSA graph are "wrong" is only a consequence of the real issue: because the analysis is unfinished, the intermediate versions of $p^\sharp$ do not represent all the possible program states (e.g., $p^\sharp[\ell_2]$ only represents the program states at ℓ_2 for the first iteration of the loop); and at the same time, the edges present in $\hat{g}$ allows the transition system that $\hat{g}$ represents to follow paths that lead to program states that are not yet represented by $p^\sharp$. The erroneous execution trace of Figure 8c illustrates this: the program path $\ell_0{-}\ell_1{-}\ell_2{-}\ell_1{-}\ell_2$ leads to a concrete store $[x \mapsto 4]$ which is not represented by $p^\sharp[\ell_2]$, leading to the fact that this path in the original program cannot be matched in the SSA graph (instead, the path continues with a wrong transition). Note that this problem can also appear in loop-free programs.

6.2 Making SSA Construction Incrementally Complete

To make the SSA construction complete on intermediate steps, we need to limit the SSA graph so that transition to states that are not represented by $p^\sharp$ is impossible. For instance in Figure 8, if we remove the transition from ℓ_2 to ℓ_1 (for instance, by keeping only the transition between ℓ_0 and ℓ_1), then the SSA program represents a subset of the reachable program transitions of the original program.

Our idea is that the SSA graph should contain an edge from ℓ to ℓ' only if

$$\forall \sigma, \sigma' \in \mathbb{\Sigma} : (\ell, \sigma) \in \gamma_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot}(p^\sharp) \wedge (\ell, \sigma) \rightarrowtail (\ell', \sigma') \Rightarrow (\ell', \sigma') \in \gamma_{\mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot}(p^\sharp)$$

i.e. if all states represented by $p^\sharp[\ell]$, will have the next state s' be represented by $p^\sharp[\ell']$. We call such an edge *fully-propagated*. For instance, the edge from ℓ_1 to ℓ_2 in the SSA graph in Figure 8 is

not fully propagated, because $(\ell_1, [x \mapsto 8]) \in \gamma_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp)$ and $(\ell_1, [x \mapsto 8]) \rightarrowtail (\ell_2, [x \mapsto 9])$, but $(\ell_2, [x \mapsto 9]) \notin \gamma_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp)$. On the contrary, the edge from ℓ_0 to ℓ_1 is fully-propagated.

This idea of a fully-propagated edge (ℓ, ℓ') is that the stores that can be seen at ℓ' include the stores that come from ℓ. In the abstract semantics, it corresponds to an (abstract) inclusion between abstract stores, that we define below.

Definition 6.1 (Order relation on $\mathbb{\Sigma}^\sharp_\bot$, the abstract stores extended with $\bot$).

$$\forall \ell \in \mathbb{L}, \forall \sigma^\sharp_1, \sigma^\sharp_2, \sigma^\sharp_3 \in \mathbb{\Sigma}^\sharp : \qquad \bot \sqsubseteq^\ell \bot \qquad \bot \sqsubseteq^\ell \sigma^\sharp_2 \qquad \sigma^\sharp_1 \sqsubseteq^\ell \sigma^\sharp_2 \triangleq \exists \sigma^\sharp_3 : \sigma^\sharp_2 = \sigma^\sharp_1 \sqcup^\ell \sigma^\sharp_3$$

This definition is equivalent to the usual algebraic definition $\sigma^\sharp_1 \sqsubseteq^\ell \sigma^\sharp_2 \triangleq \sigma^\sharp_2 = (\sigma^\sharp_1 \sqcup^\ell \sigma^\sharp_2)$ ([Lemerre 2023a, Lemma D.1]), but it is closer to the intended meaning of why the $\sqsubseteq^\ell$ relation is needed in our case (i.e. $\sigma^\sharp_1 \sqsubseteq^\ell \sigma^\sharp_2$ if we can obtain $\sigma^\sharp_2$ by joining $\sigma^\sharp_1$ with something else).

Equipped with this order relation, we can define the maximal set of fully-propagated edges (or, maximal fully-propagated subgraph) given $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$ as follows:

$$\mathrm{maxG}(p^\sharp) \triangleq \{(\ell, \ell') \in \mathrm{dom}\, \mathcal{G} \mid \mathcal{G}[\ell, \ell'] = 0 \vee p^\sharp[\ell] = \bot \vee \mathcal{I}^\sharp[\![\mathrm{instr}[\ell]]\!](p^\sharp[\ell]) \sqsubseteq^{\ell'} p^\sharp[\ell']\}$$

This definition is derived from the definition of $\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}$ describing the transfer function over edges in Figure 4. On the example of Figure 8, we have $\mathrm{maxG}(p^\sharp) = \{(\ell_0, \ell_1)\}$.

Then, we define a modified transfer function for the SSA abstract domain, consisting in limiting the SSA graph to the maximal fully-propagated subgraph of the current value of $p^\sharp$. Formally:

$$\mathcal{F}'_{\mathbb{SSA}^\sharp}(p^\sharp, \hat{g}) \triangleq \mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp), \mathrm{graph}(\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp))_{|\mathrm{maxG}(p^\sharp)}$$

where we note by $\hat{g}_{|G} \triangleq \{(\ell, \ell') \mapsto (\hat{e}, B) \mid (\ell, \ell') \mapsto (\hat{e}, B) \in \hat{g} \wedge (\ell, \ell') \in G\}$ the restriction of an SSA graph $\hat{g}$ to the edges in G. By restricting the SSA graph to fully-propagated edges of $p^\sharp$, we can prove that intermediate computation performed by this new transfer function is complete:

Theorem 6.2 (Incremental SSA graphs are complete).
Let $p^\sharp \in \mathbb{L} \to \mathbb{\Sigma}^\sharp_\bot$, $G = \mathrm{maxG}(p^\sharp)$, and $\hat{g} = \mathrm{graph}(\mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp))$. Then: $\gamma_{\mathbb{SSA}^\sharp}(p^\sharp, \hat{g}_{|G}) \subseteq \rightarrowtail_{|\mathbb{S}^*}$.

Proof sketch. The proof goes as follows. We show that the notion of *fully-propagated graph* (a set of fully-propagated edges) G corresponds to being a post-fixpoint of the original program restricted to G. For this, we define the notion of post-fixpoint and show that the theorems that we proved when $p^\sharp$ is a fixpoint also apply when $p^\sharp$ is a post-fixpoint, and in particular the soundness and completeness of $p^\sharp, \hat{g}$ when it is a (post)fixpoint of the SSA analysis (Theorem 5.8). Using this theorem and the fact that we have a post-fixpoint of the original program restricted to G, we prove that $\gamma_{\mathbb{SSA}^\sharp}(p^\sharp, \hat{g}_{|G})$ is transition system of the original program limited to G, which is a subset of the original transition system: thus our SSA abstraction does not have spurious behaviours. □

The full proof is in Lemerre [2023a, Appendix D].

6.3 A Practical Algorithm for Incremental Computation of SSA Graphs

The fixpoint computation using $\mathcal{F}'_{\mathbb{SSA}^\sharp}$ above has two efficiency problems. The first is that it recomputes the whole $p^\sharp$ and $\hat{g}$ at every step even though these do not change much between two iterations. The solution for computing $p^\sharp$ incrementally is known, and consists in performing chaotic iteration [Bourdoncle 1993; Cousot 1977]. It is quite easy to also compute the incomplete version of the graph incrementally (i.e., make the transfer functions of Figure 7 incremental) by updating all the edges going to ℓ when $p^\sharp[\ell]$ is updated. However, it is more difficult to ensure that $\hat{g}$ is complete, and in particular to ensure that $\hat{g}$ only contains the edges of $\mathrm{maxG}(p^\sharp)$.

Input: A weak topological ordering (WTO) of locations wto
Output: A program abstraction ($p^\sharp$) and SSA graph ($\hat{g}$)
Main algorithm is
  $p^\sharp := []$;
  $\hat{g} := \{\}$;
  `sequence(`wto`)`;

Procedure `sequence(`$list$`)` **is**
  forall $elt \in list$ **do**
    if elt *is a single location* ℓ **then**
      `location(`ℓ`)`;
    else
      `component(`elt`)`;

Procedure `location(`ℓ`)` **is**
  $p^\sharp[\ell] := \mathcal{F}_{\mathbb{L}\to\mathbb{\Sigma}^\sharp_\bot}(p^\sharp)[\ell]$;
  $\hat{g} := \hat{g} \cup \{\text{edge}(p^\sharp, \ell_1, \ell) \mid (\ell_1, \ell) \in \text{dom}\,\mathcal{G}\}$;

Procedure `component(`$head :: tail$`)` **is**
  local variables $save\hat{g}$, $curhead$;
  $save\hat{g} := \hat{g}$;
  `location(`$head$`)`;
  repeat
    $curhead := p^\sharp[head]$;
    $\hat{g} := save\hat{g} \cup$
      $\{\,\text{edge}(p^\sharp, \ell, head) \mid (\ell, head) \in \text{dom}\,\mathcal{G} \wedge\ \ell \preceq head\,\}$;
    `sequence(`$tail$`)`;
    `location(`$head$`)`;
  until $curhead = p^\sharp[head]$;

Fig. 9. A practical algorithm for incremental computation of an SSA abstraction.

Our solution to this problem (Figure 9) is to rely on a property of the recursive fixpoint iteration strategy of Bourdoncle [1993], which is that when ℓ_c is the currently modified location and $p^\sharp$ the current program abstraction, the subgraph $\{(\ell, \ell') \in \mathbb{L} \times \mathbb{L} \mid \ell \preceq \ell_c \wedge \ell' \preceq \ell_c\}$ is a fully-propagated subgraph of $p^\sharp$ (here, $\preceq$ represents the weak topological ordering of Bourdoncle [1993]). Our algorithm builds on this observation and incrementally compute $\hat{g}$ so that the edges of $\hat{g}$ are limited to this subgraph. In more details, the algorithm mostly implements Bourdoncle's recursive iteration strategy, taking as input a WTO (which is a sequence of locations or (connected) components, where a component is a head location followed by a sequence of locations or components). The main change compared to the normal recursive iteration strategy is that the SSA graph $\hat{g}$ is saved in a snapshot ($save\hat{g}$) before processing a component, and this snapshot is retrieved when another iteration on the component is needed. The feedback edges (edges from later locations going to the head of the component) do not appear in the $\hat{g}$ graph until the fixpoint iteration over the component is over. The end result is that this algorithm continuously produces (e.g. before the call to every `location` function) a complete SSA abstraction of the program (and moreover, terminates with a sound and complete SSA abstraction of the program).

7 USE CASES

7.1 Single-Pass Optimized Translations to SSA

Implementation. In the first use case, we demonstrate the feasibility of doing SSA translation using abstract interpretation. We have implemented the algorithms described in the paper as a small plugin for the Frama-C platform and used the LLVM intermediate representation (which is in SSA form) as the target of our translation. In particular, we have implemented the symbolic expression analysis of Section 4, followed by a call to the graph operator, modified to directly produce LLVM code instead of SSA graphs. An important (but small) change compared to our formalization is that we have replaced guards and non-determinism with terminator instructions (i.e., branches and conditional branches). Following the formal definition of our analysis, the implementation of this plugin does SSA translation, Global value numbering, Global code motion, but also Sparse Conditional Constant

Propagation [Wegman and Zadeck 1991] (by rewriting constant terms when it can), all in only 970 lines of OCaml code (measured by Sloccount), excluding a 316 lines implementation of Okasaki's Patricia tries [Okasaki and Gill 1998], and code implementing the iterative and recursive iteration strategy on Bourdoncle's [Bourdoncle 1993] WTO (183 lines). The parser and WTO calculation is provided by Frama-C, and the LLVM translation by the standard LLVM OCaml bindings. The goal of this experiment is to demonstrate that doing SSA translation via abstract interpretation is not only of theoretical interest but that it can be done in practice as the translation achieves reasonable performances. The full description of this implementation and the results of the evaluation is provided in Lemerre [2023a, Appendix E], while the code for the experiment and its results can be retrieved at Lemerre [2023b]. Here we reproduce only the main elements and our conclusions.

Experimental Settings. We have used this implementation to evaluate both the soundness and performance of our algorithm. To avoid the problem of addressable variables in C that cannot be directly translated to SSA, we have used the Csmith [Yang et al. 2011] C generator, using options to generate only non-addressable integer variables and complex unstructured control flow, to generate 100 large C files. We used our plugin to convert the C code to LLVM code, then used LLVM (version 13) to compile it to native x86 code, that we executed.

Results and Conclusions. First, all the C code without undefined behavior that we generated, returned the same value as the one compiled using a standard compiler. This confirms that *SSA translation using abstract interpretation can be used to soundly translate programs to SSA*, which is not a surprise since we provided the proof for all of our algorithms.

Furthermore, *the time spent doing so is in the order of magnitude compatible for incorporation in a compiler*: if we add incorporated our SSA translation pass inside `clang`, the maximum observed slowdown compared to a native code compilation by `clang` would have been ×6.57, which is slower but acceptable (the median slowdown would have been ×4.30). This is a maximum, as such an incorporation would lead to removing existing passes in `clang`; furthermore, additional engineering (such as switching from OCaml to C) could further improve the performance. *One of the reason for this efficiency is that fixpoint iteration with the symbolic expression domain is usually obtained using few iterations*, even in large programs: in the iterative fixpoint iteration strategy of Bourdoncle [Bourdoncle 1993], we never needed more than 6 iterations to reach the fixpoint on any strongly connected component. The other part is explained by the use of functional maps for low-complexity operations on abstract stores. *One crucial engineering decision in our analysis performance was the use of Okasaki maps [Okasaki and Gill 1998] to implement the abstract stores*, instead of the more common binary search trees (as used in Astrée [Blanchet et al. 2002]), as otherwise the time spent in the $\sqcup^\ell$ operation dominates the time spent in the analysis. Okasaki maps allow more efficient handling of the parts of the abstract store that have not changed between when the different states that are joined with $\sqcup^\ell$. Finally, *combining SSA translation with other analyses such as constant propagation and dead code elimination can significantly improve the time spent in the analysis*: in this case, incorporating our pass in `clang` would make it only 4.73 time slower (instead of 6.57 times), at worst (and the median slowdown would have been ×3.19)

Whether, with careful engineering, implementing compilers as a single-pass combination of modular abstract domains, can outperform the existing designs as a sequence of passes, is an open question, but our initial results indicate that the single-pass design could be competitive in terms of compilation speed.

7.2 A static Analyzer for C and Machine Code

The main motivation for this work is to describe the working of Codex, a multi-language static analyzer that can process both C (as a Frama-C plugin) and machine code (as a Binsec plugin)

that was used as the basis of the experiments in Nicole et al. [2021b,a, 2022]. Codex analyses an input program by combining the symbolic expression and SSA abstract domains with control-flow, numeric and memory abstractions. The symbolic expressions and SSA abstract domains have three main uses:

(1) The symbolic expressions allow proving equality between values. Combined with term rewriting rules, it can be used to prove the preservation of complex predicates; one application is proving preservation of some OS kernel invariants in Nicole et al. [2021b].
(2) It is used to perform a sparse [Tavares et al. 2014; Wegman and Zadeck 1991] numerical analysis of the program simultaneously with its SSA translation. This numerical analysis, in turn, is used to improve the precision of the symbolic expressions, control-flow, and memory abstractions. The benefit of such a sparse analysis is that it is more efficient in terms of memory and CPU time, and that it allows propagating constraints across different statements, which is important when analyzing machine code (that consists of a lot of small instructions).
(3) The SSA translation allows generating SMT or Horn queries. Thanks to the combination of SSA and memory abstractions, this logic query is automatically simplified, which is important as SMT solvers do not handle memory very well [Farinier et al. 2018]. Thanks to the incrementally complete construction of SSA (Section 6), the SMT queries can be performed anytime during the analysis, and is used in particular to refine Boolean expressions such as assertions or opaque predicates, or to enumerate control-flow targets when needed.

The full description of this analysis and combination of domains in out of the scope of this paper, but the above description motivates the practical benefits of using the symbolic expressions and SSA abstractions.

8 RELATED WORK

8.1 Abstract Interpretation on Cyclic Terms and Computing the Global Value Graph

Having terms representing expressions or predicates inside an abstract domain is a natural idea already present in the seminal paper of Kildall [1973]. The main difficulty of using terms in a static analysis is dealing with cycles of the term (coming from loops in the original program), and ensuring termination. Indeed, many analyses (e.g., symbolic execution [King 1976], bounded model checking [Clarke et al. 2004], weakest precondition and strongest-postcondition [de Bakker and Meertens 1975; Dijkstra 1975; Leino 2005] calculus) transform the program into a single global term or formula, but sidestep the problem of termination by analyzing only loop-free subsets of the program (although they can be viewed as abstract interpretations [Cousot 1978, ch.3.3]).

Some static analyses [Gulwani and Necula 2004; Kildall 1973] include terms that refer to program variables instead of symbolic variables. This generally removes the need for the terms to be cyclic. However, cyclicity reappears when program variables are substituted with their definition; Miné [2006] solves this problem by using a special $\top$ variable where we would use symbolic variables, thereby losing some precision. Some static analyses [Chang and Leino 2005; Chang and Rival 2013; Gange et al. 2016; Illous et al. 2021] create fresh variables when needed, which is precise but requires special care to ensure termination when the program has loops. Their solution is to use renaming: in particular, inclusion to test if a post-fixpoint is reached is done modulo renaming of the variables in the abstract elements. One issue with renaming is that it makes the definition and implementation of an abstract domain more complex. Furthermore, these solutions can be used to get acyclic terms at different program points, but not to create one global cyclic term like the global value graph.

The most common cyclic term graph used by static analyses is the global value graph. The classic solution to get this graph is to do an SSA translation, which solves the problem of termination of

the analysis because SSA translation is done as a program translation [Cytron et al. 1991] rather than as an abstract interpretation. Analyses using this graph then only need to manipulate SSA variables instead of nodes in a cyclic term graph, and, for instance, the problem of global value numbering is reduced to computing equivalence classes of equality between SSA variables [Alpern et al. 1988; Barthe et al. 2014; Rosen et al. 1988; Rüthing et al. 1999] (the optimistic algorithm of Cooper and Simpson [1995] that detects equalities between SSA expressions in a fixpoint being the closest to ours). The drawback of this design is that SSA translation must be done sequentially with GVN and other analyses, instead of having mutually beneficial combination of domains.

Some papers predating SSA [Reif and Lewis 1986; Reif and Tarjan 1982] did build the global value graph and performed global value numbering directly on the original program. In particular, [Reif and Lewis 1986] provided a fixpoint characterization of the global value graph, and introduced mappings from program expressions to symbolic expressions (playing the same role as our symbolic expression abstraction $p^\sharp$). However, they dismissed the computation of the global value graph using dataflow analysis (proposing a complex sparse algorithm instead) for fear of it being too slow, and because their characterization did not lead to a sufficiently precise value graph; we solve these issues using efficient abstract interpretation techniques [Blanchet et al. 2002; Bourdoncle 1993; Cousot 1977], and by combining creation of the global value graph with global value numbering.

We propose to solve the problem of having cyclic terms in an abstract domain by introducing recursion variables [Ariola and Klop 1996] that have a deterministic name, and have this name follow the name of the recursion variables that already exists in the original program, i.e. the names of the control-flow locations. We apply this method to build the global value graph using a standard flow-sensitive analysis on the original program, where the graph is simplified by a simultaneous global value numbering. The method does not require a prior SSA translation (instead, it is used to perform SSA translation).

8.2 SSA Translation and Reconstruction

Since the invention of SSA form [Rosen et al. 1988], several algorithms have been created to compute SSA form. The classic algorithm of Cytron et al. [1991] works in two phases, first computing where to insert ϕ functions after computation of the dominance frontier, then applying a linear pass that assigns unique names to the different definitions of each variable. Sreedhar and Gao [1995] makes the ϕ-function insertion pass linear by proposing a data structure called DJ-graph, that builds upon the dominator tree. The algorithm of Aycock and Horspool [2000] performs a pessimistic insertion of ϕ-functions and then improves the translation using rewriting rules. The ϕ-function insertion phase of these algorithms is related to our symbolic expression analysis, as the introduction of symbolic variables introduced during a $\sqcup^\ell$ operation corresponds to the insertion of a ϕ node at ℓ.

Some algorithms can work directly from programs in AST form [Brandis and Mössenböck 1994; Braun et al. 2013], without requiring a prior analysis, like computation of the domination tree. In particular, the algorithm of Braun et al. [2013] produces minimal SSA on unstructured programs, and can perform "on the fly optimizations" (which are similar to rewriting terms in our symbolic expression analysis) to improve the precision and efficiency of SSA generation.

SSA translation based on abstract interpretation improves on the latter algorithm by performing "on the fly" global value numbering, sparse conditional constant propagation, or any optimization based on semantic properties obtained by combination with an abstract domain, in addition to not requiring a prior analysis. It is also the first algorithm to do SSA translation incrementally while maintaining completeness.

"Maintaining SSA is often one of the more complicated and error-prone part in [program] optimizations" [Hack 2016]. Combing our SSA abstract domain with another domain, and viewing a

program transformation as a reduced product [Cousot and Cousot 1979] between the two domains, is an interesting solution to this problem that can be used to create transformation passes that automatically maintain SSA.

8.3 Syntax and Semantics of SSA Programs

We represent a program in SSA form as a graph whose nodes are control locations and edges contain guards and bindings to symbolic variables, where the bindings correspond to ϕ functions in traditional SSA. We have proposed a semantics where SSA states consists of valuations of the symbolic variables that are in scope, so that SSA states and original program states are made bisimilar using the computed symbolic expressions as a bisimulation (and homomorphism) relation.

Usually, SSA is viewed as reusing the original program syntax, and SSA transformation consists in renaming the program variables and inserting the necessary ϕ functions. The SSA semantics is based on the assignment of SSA variables, and ϕ functions are also seen as doing assignments [Barthe et al. 2014; Cytron et al. 1991; Mansky and Gunter 2010; Zhao et al. 2012].

The idea of binding (instead of assigning) SSA variables to unbind those that go out of scope also exists when SSA is viewed as a functional program [Appel 1998b; Kelsey 1995], where the scope is represented explicitly in the syntax (rather than implicitly by domination). Schneider [2013] showed that both interpretations (one based on assignments and the other on bindings) can be used on functional representations of SSA. *We showed that we can also have both interpretations on a graph-based language without an explicit syntactic scope.* We preferred using a graph as the syntax for SSA rather than lambda terms, as graphs can be built incrementally using abstract domain operations, which seems more difficult to do when the abstraction is a lambda term.

The idea of not attaching sub-terms to a specific control location was proposed by Click and Paleczny [1995] in an SSA representation called Sea of Nodes, which is the closest to our SSA graphs (in particular, to a variant of our graphs that uses terminator instructions instead of guards + nondeterminism). The Sea of Nodes can be described as a low-level description of our SSA graphs, where their nodes would correspond to the memory blocks used to represent the graph and the edges to the pointers. Demange et al. [2018] proposed a formal semantics for Sea of Nodes using a mixture of big-step semantics for expressions and small-step semantics for instructions which is similar to our semantics: the main changes are that we propose to unbind variables that go out of scope, and we take advantage of the simpler, higher-level syntax of SSA graphs.

An important characteristic of SSA syntax and semantics, and of the SSA translation algorithm, is simplicity, particularly when the goal is to formally verify SSA transformations [Barthe et al. 2014; Buchwald et al. 2016; Zhao et al. 2012]. The entire syntax and semantics of our SSA abstraction fit in Figure 5, and our SSA translation algorithm is completely and formally described in a dozen of lines (coming from parts of Figure 4 and Figure 6), and are thus arguably very simple. Furthermore, this description includes GVN and dead code elimination, and other optimizations can be described in a very compact way as a reduced product with other abstract domains. It would thus be interesting to investigate if SSA translation based on abstract interpretation can be an effective basis for formal verification of SSA transformations.

8.4 SSA and Static Analysis

One of the motivations for our work is to combine the SSA abstraction with other abstractions. Such a combination benefit from interplay between SSA and static analyses that already exist.

One of the benefits of SSA form is that it allows more efficient *sparse* static analyses [Choi et al. 1991] (see, e.g., Tavares et al. [2014] for partitioned per variable analyses, and Mirliaz and Pichardie [2022] for analyses using relational domains). *Combining our SSA domain with a sparse analysis allows performing sparse analyses simultaneously with the SSA translation, and integrating*

sparse analyses in a standard [Blanchet et al. 2002; Blazy et al. 2017; Journault et al. 2019] abstract interpretation framework. Several work [Lerner et al. 2002; Ramsey et al. 2010; Rompf 2012] have shown that changing the program syntax is useful to communicate between abstract domains; our work allows seeing this as reduced products [Cousot and Cousot 1979] with the SSA abstract domain.

The need for static analyses and, in particular, memory, pointer or alias analyses to improve the precision of SSA is well-known. This requirement is very important, for instance, when the SSA form is used to generate logic formulas to an SMT solver [Clarke et al. 2004; Gurfinkel et al. 2015] to increase precision and decrease solving time. Static analyses are even indispensable for programs where the control-flow is not known in advance because usual SSA translations require a CFG (e.g., to compute dominance frontiers). In particular, this is the case when analyzing machine code, i.e. when doing decompilation [Brumley et al. 2013] to SSA, which is a popular target for decompilation [Van Emmerik 2007; Yadavalli and Smith 2019]. Precisely recovering control-flow in machine code requires doing control-flow and value analysis simultaneously [Bardin et al. 2011; Kinder et al. 2009]; and SSA is a good intermediate representation to do this value analysis. To deal with this phase ordering problem, Van Emmerik [2007] used a simple but inefficient method consisting in redoing the whole SSA decompilation every time an indirect jump target is discovered in a function. *Doing SSA translation as an abstract interpretation solves these phase ordering problems: we can use the incrementally built (Section 6) SSA representation to help the memory, numeric, or control-flow analyses that improve or are necessary to build the SSA representation.*

Outside SSA, is it well-known that symbolic domains can be used to improve the precision of abstract domains [Gange et al. 2016; Miné 2006], and our symbolic expression domain can be used for the same purpose (e.g., to detect equalities between variables that would be missed by numerical domains). Finally, Cousot and Cousot [2002] explains how abstract interpretation can help design and prove correct program transformations, and discusses the possibility of doing some transformations, like constant propagation and dead code elimination, using dataflow analysis.

9 CONCLUSION

SSA translation is not only a program transformation; it is the result of a static analysis based on abstract interpretation. This analysis has two parts: symbolic expression analysis, which maps program variables to symbolic expressions; and SSA graph creation, which expresses how the symbolic variables in the symbolic expressions change according to the control flow. The symbolic expression analysis (that also does a global value numbering) relates the SSA program graph to the original program graph by providing a strong homomorphism, a relation that implies bisimulation. Due to this bisimulation, the SSA-translation domain computes an abstraction of the original program which is both sound and complete. Furthermore, we can modify the analysis so that all intermediate abstractions are also complete. This static analysis can be done using relatively standard dataflow frameworks, which allows combination [Cousot and Cousot 1979] with other abstract domains. We explained, in this paper, how numerical abstract domains can strengthen the symbolic expression analysis and SSA abstractions, notably by detecting dead transitions or more equalities between symbolic expressions. Using a reduced product, we can build optimizing transformations from a source program to SSA in a single pass.

We have only sketched how the SSA abstraction can strengthen numerical abstract domains, and did not mention how the SSA abstraction can be combined with memory abstractions to do register promotion; these studies should be the target of future publications.

REFERENCES

Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting Equality of Variables in Programs. In *15th ACM Symposium on Principles of Programming Languages (POPL 1988)*. 1–11. https://doi.org/10.1145/73560.73561

Andrew W Appel. 1998a. *Modern compiler implementation in ML*. Cambridge University Press. https://doi.org/10.1017/CBO9780511811449

Andrew W. Appel. 1998b. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (apr 1998), 17–20. https://doi.org/10.1145/278283.278285

Zena M Ariola and Jan Willem Klop. 1996. Equational term graph rewriting. *Fundamenta Informaticae* 26, 3, 4 (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

Sébastien Bardin, Philippe Herrmann, and Franck Védrine. 2011. Refinement-Based CFG Reconstruction from Unstructured Programs. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011, Vol. 6538)*. Springer, 54–69. https://doi.org/10.1007/978-3-642-18275-4_6

Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014), 4:1–4:35. https://doi.org/10.1145/2579080

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones* 2566 (Oct. 2002), 85–108. https://doi.org/10.1007/3-540-36377-7_5

Sandrine Blazy, David Bühler, and Boris Yakobowski. 2017. Structuring Abstract Interpreters Through State and Value Abstractions. In *18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 112–130. https://doi.org/10.1007/978-3-319-52234-0_7

François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*. Springer, 128–141. https://doi.org/10.1007/BFb0039704

Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by k-Invariants and k-Induction. In *Static Analysis (SAS 2015)*, Sandrine Blazy and Thomas Jensen (Eds.). Springer, 145–161. https://doi.org/10.1007/978-3-662-48288-9_9

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

David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22th USENIX Security Symposium*. USENIX Association, 353–368. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. 2016. Verified construction of static single assignment form. In *25th International Conference on Compiler Construction (CC 2016)*. ACM, 67–76. https://doi.org/10.1145/2892208.2892211

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

Bor-Yuh Evan Chang and Xavier Rival. 2013. Modular Construction of Shape-Numeric Analyzers. In *Festschrift for Dave Schmidt (EPTCS, Vol. 129)*, Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff (Eds.). https://doi.org/10.48550/arXiv.1309.5138

Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *18th ACM Symposium on Principles of Programming Languages (POPL 1991)*, David S. Wise (Ed.). 55–66. https://doi.org/10.1145/99583.99594

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

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 (1995), 35–49. https://doi.org/10.1145/202530.202534

Keith D Cooper and Taylor Simpson. 1995. *SCC-Based Value Numbering*. Technical Report. Rice University.

Patrick Cousot. 1977. *Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice*. Technical Report. Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France. https://www.di.ens.fr/~cousot/publications.www/Cousot-IMAG-RR88-Sep-1977.pdf

Patrick Cousot. 1978. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Ph.D. Dissertation. Université Joseph Fourier, Grenoble, France. https://www.di.ens.fr/~cousot/publications.www/CousotTheseEsSciences1978.pdf

Patrick Cousot. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.* 277, 1-2 (2002), 47–103. https://doi.org/10.1016/S0304-3975(00)00313-3

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL 1977)*. 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 (POPL 1979)*. 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

Jacobus Willem de Bakker and Lambert Meertens. 1975. On the Completeness of the Inductive Assertion Method. *J. Comput. Syst. Sci.* 11, 3 (1975), 323–357. https://doi.org/10.1016/S0022-0000(75)80056-0

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

Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (aug 1975), 453–457. https://doi.org/10.1145/360933.360975

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

Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. 2018. Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing. In *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 57)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.). EasyChair, 363–380. https://doi.org/10.29007/dc9b

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, 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

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, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification (CAV 2015)*. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

Sebastian Hack. 2016. *SSA reconstruction* (1st ed.), Chapter 5. In [Rastello 2016].

Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: a path sensitive static analyzer. In *Tools for Automatic Program Analysis (TAPAS)*, Bertrand Jeannet (Ed.). https://doi.org/10.1016/j.entcs.2012.11.003

Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2021. A relational shape abstract domain. *Formal Methods Syst. Des.* 57, 3 (2021), 343–400. https://doi.org/10.1007/s10703-021-00366-4

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

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

Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, Neil D. Jones and Markus Müller-Olm (Eds.). Springer, 214–228. https://doi.org/10.1007/978-3-540-93900-9_19

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

K.R.M. Leino. 2005. Efficient weakest preconditions. *Inform. Process. Lett.* 93, 6 (2005), 281–288. https://doi.org/10.1016/j.ipl.2004.10.015

Matthieu Lemerre. 2023a. *SSA Translation is an Abstract Interpretation (full version with appendices).* Technical Report. CEA, LIST. https://binsec.github.io/assets/publications/papers/2023-popl-full-with-appendices.pdf

Matthieu Lemerre. 2023b. *SSA Translation is an Abstract Interpretation (artifact).* Artifact. CEA, LIST. https://doi.org/10.1145/3554341

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

William Mansky and Elsa L. Gunter. 2010. A Framework for Formal Verification of Compiler Optimizations. In *1st International Conference on Interactive Theorem Proving – ITP 2010 (Lecture Notes in Computer Science, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 371–386. https://doi.org/10.1007/978-3-642-14052-5_26

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

Solène Mirliaz and David Pichardie. 2022. A Flow-Insensitive-Complete Program Representation. In *23th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2022)*. https://doi.org/10.1007/978-3-030-94583-1_10

Greg Nelson. 1980. *Techniques for program verification.* Ph.D. Dissertation. Stanford University, CA, USA.

Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. 2021b. 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. 2021a. *Binsec/Codex, an abstract interpreter to verify safety and security properties of systems code.* Technical Report. https://binsec.github.io/assets/publications/papers/2021-rtas-technical-report-analysis.pdf

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

Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *Workshop on ML*. 77–86.

Norman Ramsey, João Dias, and Simon L. Peyton Jones. 2010. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell 2010)*, Jeremy Gibbons (Ed.). ACM, 121–134. https://doi.org/10.1145/1863523.1863539

Fabrice Rastello. 2016. *SSA-based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated.

John H. Reif and Harry R. Lewis. 1986. Efficient Symbolic Analysis of Programs. *J. Comput. Syst. Sci.* 32, 3 (1986), 280–314. https://doi.org/10.1016/0022-0000(86)90031-0

John H. Reif and Robert Endre Tarjan. 1982. Symbolic Program Analysis in Almost-Linear Time. *SIAM J. Comput.* 11, 1 (1982), 81–93. https://doi.org/10.1137/0211007

Thomas Reinbacher and Jörg Brauer. 2011. Precise control flow reconstruction using boolean logic. In *11th International Conference on Embedded Software (EMSOFT 2011)*, Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister (Eds.). ACM, 117–126. https://doi.org/10.1145/2038642.2038662

Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2004 (Lecture Notes in Computer Science, Vol. 2937)*, Bernhard Steffen and Giorgio Levi (Eds.). Springer, 252–266. https://doi.org/10.1007/978-3-540-24622-0_21

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 *15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1988)*. 12–27. https://doi.org/10.1145/73560.73562

Oliver Rüthing, Jens Knoop, and Bernhard Steffen. 1999. Detecting Equalities of Variables: Combining Efficiency with Precision. In *6th Static Analysis Symposium – SAS '99 (Lecture Notes in Computer Science, Vol. 1694)*, Agostino Cortesi and Gilberto Filé (Eds.). Springer, 232–247. https://doi.org/10.1007/3-540-48294-6_15

Davide Sangiorgi. 2009. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.* 31, 4 (2009), 15:1–15:41. https://doi.org/10.1145/1516507.1516510

Sigurd Schneider. 2013. *Semantics of an Intermediate Language for Program Transformation.* Master's thesis. Saarland University. https://www.ps.uni-saarland.de/~sdschn/msc.pdf

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

James Stanier. 2016. *Graphs and Gating Functions* (1st ed.), Chapter 14. In [Rastello 2016].

André Luiz Camargos Tavares, Benoit Boissinot, Fernando Magno Quintão Pereira, and Fabrice Rastello. 2014. Parameterized Construction of Program Representations for Sparse Dataflow Analyses. In *23rd International Conference on Compiler Construction – CC 2014 (Lecture Notes in Computer Science, Vol. 8409)*, Albert Cohen (Ed.). Springer, 18–39. https://doi.org/10.1007/978-3-642-54807-9_2

Michael James Van Emmerik. 2007. *Static single assignment for decompilation.* Ph.D. Dissertation. University of Queensland.

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. https://doi.org/10.1145/103135.103136

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304

S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*, Jian-Jia Chen and Aviral Shrivastava (Eds.). 213–218. https://doi.org/10.1145/3316482.3326354

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

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, John Field and Michael Hicks (Eds.). 427–440. https://doi.org/10.1145/2103656.2103709

Received 2022-07-07; accepted 2022-11-07