# The Virtual Recency Abstraction: Strong Updates for Abstract Interpreters with Shared State

Sven Keidel, Raphaël Monat, Sebastian Erdweg

► **To cite this version:**

Sven Keidel, Raphaël Monat, Sebastian Erdweg. The Virtual Recency Abstraction: Strong Updates for Abstract Interpreters with Shared State. ECOOP 2026 - 40th European Conference on Object-Oriented Programming, Jun 2026, Brussels, Belgium. ⟨hal-05604182⟩

**HAL Id: hal-05604182**
**https://inria.hal.science/hal-05604182v1**

Submitted on 27 Apr 2026

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

# The Virtual Recency Abstraction

## Strong Updates for Abstract Interpreters with Shared State

**Sven Keidel**
Fraunhofer SIT | ATHENE, Darmstadt, Germany

**Raphaël Monat**
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

**Sebastian Erdweg**
KIT, Karlsruhe, Germany

**Abstract**

Abstract interpreters enable sound static analysis, but are hard to develop. In recent years, researchers have proposed a component-based approach to developing abstract interpreters, where different parts of the abstract domain (e.g., numeric, call frame, heap) are handled by isolated components. This works well as long as components do not share or expose their internal state: Any state update that is locally sound is also globally sound. However, some abstract domains require shared state, most prominently relational abstract domains, which use symbolic expressions such as $2x + 5$ to represent abstract values. As the relational component performs a strong update of $x$, the abstract value $2x+5$ can change non-monotonically, which breaks soundness. We propose a novel solution to this problem: A virtual recency abstractions that decouples the relational component, supports strong updates, and allows recursion. We prove that the virtual recency abstraction restores soundness: Any shared state wrapped in our virtual recency abstraction may be locally updated non-monotonically, while global soundness persists. We applied our approach to develop the first relational WebAssembly analysis, reusing many components from an existing inter-procedural abstract interpreter. Furthermore, we evaluate the recall, precision, and scalibility of this analysis to demonstrate the practicality of the virtual recency abstraction.

**2012 ACM Subject Classification** Software and its engineering → Automated static analysis

**Keywords and phrases** Relational Numerical Analysis, Recency Abstraction

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2026.14

**Supplementary Material** *Software (Artifact)*: `https://doi.org/10.5281/zenodo.19704087`

## 1 Introduction

Static analyzers are automated tools that extract information from computer programs without running them. They are used in compilers, integrated development environments, security scanners, bug trackers, and software verification. However, developing sound and precise static analyses for modern programming languages is a delicate balancing act: The analysis needs to faithfully encode the complexity of the language semantics and employ complex abstract domains.

One approach to reduce the development complexity of static analyses is to modularize their implementation, such that independent analysis components capture different aspects of the language semantics and the abstract domain. Such component-based design not only reduces the analysis' code complexity, it also enables reuse of language-independent components [18, 32, 33], allows swapping out components for more precise or higher performance components [29, 32, 27], and allows proving components sound independently from each other [35]. For example, a modular WebAssembly analysis includes analysis components for the byte-addressable memory, global variables, function tables, exceptions, the operand stack, and call frame of local variables [8]. And a Java call-graph analyses may be implemented

© S. Keidel, R. Monat, and S. Erdweg;
licensed under Creative Commons License CC-BY 4.0

modularly with analysis components for reflection, threads, and serialization [28], which are notoriously difficult to analyze [36, 1, 48].

Developing static analyses from modular analysis components works well when components do not share state. Specifically, when the concretization of a decomposed abstract domain is compositional $\gamma(A_1 \times \cdots \times A_n) = \gamma_1(A_1) \times \cdots \times \gamma_n(A_n)$, then each analysis component is responsible for its own soundness only. This is the case in all prior work on analysis components. However, we found that some analysis components $A_0$ require shared state $S$, such that $\gamma(A_0(S) \times A_1 \times \cdots \times A_n) = \gamma_0(A_0(S)) \times \gamma_1(A_1, S) \times \cdots \times \gamma_n(A_n, S)$. This means, we need to co-develop $A_0$ with all other analysis components, which breaks modularity.

Consider *relational abstractions* [15, 40], which infer relationships between program variables to improve precision. For example, the relationship $0 \leq$ `index` $<$ `array.length` proves safe accesses to `array` via `index`. Relational abstractions require shared state for the analysis component that manages local variables, such as the call frame. This is because relational analyses use symbolic expressions to represent abstract values, so that variable symbols have to be resolved by other analysis components, too. For example, consider a relational analysis for WebAssembly (WASM) using separate components for the operand stack and the local variables:

| Statement | Analysis 1 (Unsound) | | Analysis 2 (Imprecise) | | Analysis 3 (Non-Modular) | |
|---|---|---|---|---|---|---|
| | Stack | Local Vars. | Stack | Local Vars. | Stack | Local Vars. |
| `;; y ∈ [1,5]` | [] | $y \in [1,5]$ | [] | $y \in [1,5]$ | [] | $y \in [1,5]$ |
| `1: (local.set $x $y)` | [] | $x = y \ldots$ | [] | $x = y \ldots$ | [] | $x = y \ldots$ |
| `2: (local.get $x)` | [x] | $x = y \ldots$ | [x] | $x = y \ldots$ | [x] | $x = y \ldots$ |
| `3: (local.set $x (i32.add $y $y))` | [x] | $x = 2y \ldots$ | [x] | $y \leq x \leq 2y$ | [$x_1$] | $x = 2y, x_1 = y \ldots$ |
| `4: (local.get $x)` | [x, x] | $x = y \ldots$ | [x, x] | $x = y \ldots$ | [x, $x_1$] | $x = y \ldots$ |
| `5: (i32.eq)` | [true] | ... | [⊤] | ... | [false] | ... |

The analyzed program corresponds to `x := y; x == {x := y + y; x}` in an imperative language. The key point is that one operand of the comparison `==` is looked up before the second assignment happens, whereas the second operand is looked up after the second assignment. Hence, for $x \neq 0$, the comparison must yield false. Let's see how a component-based relational analysis can handle this program.

We consider three alternative analysis designs. Analysis 1 pushes the symbolic expression `x` on the stack to represent the abstract value corresponding to variable `$x` (Line 2). When re-assigning `x` (Line 3), the component responsible for local variables changes its internal state using a strong update to obtain $x = 2y$. However, the component responsible for the stack was relying on the previous state for the locals. Indeed, Analysis 1 subsequently infers that the comparison must yield `true`, which is unsound. The analysis could have prevented this unsoundness if it had avoided the strong update of `x`. Analysis 2 shows this behavior: It performs a weak update and assigns `x` to $y \sqcup 2y$. This yields a sound result, but is unnecessarily imprecise. To retain precision, the analysis must perform a strong update, but it may not break the stack component. To this end, Analysis 3 introduces a fresh variable $x_1$ to remember the old binding of `x` when doing the strong update. But this only helps when the analysis also updates the internal state of the other components. In Analysis 3, when the component for local variables handles an assignment, it also must adapt the internal state

of the stack component. This is sound and precise, but not modular and requires the two components to be co-developed and co-maintained.

None of the three analyses was able to deliver sound and precise analysis results, unless we give up on component-based analysis design. However, the problem we illustrated not only occurs in the operand stack, but in every component that accesses shared state. For relational analysis, this means any component that handles values must coordinate with the call frame to resolve symbolic expressions. In WebAssembly, this includes for example components for value-carrying exceptions, which we would rather not have to redesign. But, we also found that shared state can occur within the call-frame component itself. Specifically, when analyzing recursive functions, each function call spawns a fresh call frame, while the caller's call frame awaits. A strong variable update in the callee is necessary for precision, but it may not affect the variables of the caller, although they carry the same name and make use of the same relational domain. Moreover, we cannot allocate fresh variables to remember the old bindings like Analysis 3 did above, because this precludes a fixed-point for recursive functions.

We solve the above problems by introducing the *virtual recency abstraction* for local variables, inspired by the original recency abstraction for heaps [3]. The original recency abstraction distinguishes the most recent allocation at a memory location from all prior allocations at this location with two kinds of abstract addresses: $a_{\mathsf{recent}}$ and $a_{\mathsf{old}}$. We adapt this technique for local variables, so that we get precise strong updates for $\mathtt{x}_{\mathsf{recent}}$ yet sound weak updates for $\mathtt{x}_{\mathsf{old}}$. However, when $\mathtt{x}_{\mathsf{recent}}$ is re-assigned, the recency abstraction usually requires a complete rewrite of the analysis state to retire $\mathtt{x}_{\mathsf{recent}}$ references and replace them with $\mathtt{x}_{\mathsf{old}}$ instead. This is not only a performance bottleneck, it is also non-modular because analysis components need to be aware of recency addresses contained within them. We avoid this problem by adding a new layer of abstraction over recency addresses, similar to virtual and physical memory in an operating system [19]. Specifically, we define symbolic expressions over *virtual addresses*, which are then mapped to *physical addresses* by an *address translation*. The physical addresses must be bounded to ensure termination; we use recency addresses $\mathtt{x}_{\mathsf{recent}}$ and $\mathtt{x}_{\mathsf{old}}$ for that. However, the virtual addresses space can grow freely and virtual addresses do not need to be retired, which elevates the need for rewriting the analysis state.

For example, consider again our Wasm example from above and let us illustrate how our new virtual recency abstraction works. The crucial step is the re-assignment in statement 3.

| **Statement** | **Virtual Recency Abstraction** | | | |
|---|---|---|---|---|
| | Stack | Local Vars. | Address translation | Relations |
| `;; y ∈ [1,5]` | [] | $\mathtt{y} \mapsto \mathtt{y}_1$ | $\mathtt{y}_1 \mapsto \mathtt{y}_r$ | $\mathtt{y}_r \in [1,5]$ |
| `1: (local.set $x $y)` | [] | $\mathtt{x} \mapsto \mathtt{x}_1, \mathtt{y} \mapsto \mathtt{y}_1$ | $\mathtt{x}_1 \mapsto \mathtt{x}_r, \mathtt{y}_1 \mapsto \mathtt{y}_r$ | $\mathtt{x}_r = \mathtt{y}_r, \mathtt{y}_r \in [1,5]$ |
| `2: (local.get $x)` | $[\mathtt{x}_1]$ | ... | ... | ... |
| `3: (local.set $x (i32.add $y $y))` | $[\mathtt{x}_1]$ | $\mathtt{x} \mapsto \mathtt{x}_2, \mathtt{y} \mapsto \mathtt{y}_1$ | $\mathtt{x}_1 \mapsto \mathtt{x}_o, \mathtt{y}_1 \mapsto \mathtt{y}_r,$ $\mathtt{x}_2 \mapsto \mathtt{x}_r$ | $\mathtt{x}_o = \mathtt{y}_r, \mathtt{y}_r \in [1,5],$ $\mathtt{x}_r = 2\mathtt{y}_r$ |
| `4: (local.get $x)` | $[\mathtt{x}_2, \mathtt{x}_1]$ | ... | | |
| `5: (i32.eq)` | [false] | ... | | |

The virtual recency abstraction handles statement 3 by (i) retiring $\mathtt{x}_{\mathsf{recent}}$ and replacing it by $\mathtt{x}_{\mathsf{old}}$ in the address translation and relational state, (ii) creating a fresh virtual address $\mathtt{x}_2$ and assigning it $\mathtt{x}_{\mathsf{recent}}$, and (iii) initializing $\mathtt{x}_{\mathsf{recent}}$ with a strong update to $2\mathtt{y}_{\mathsf{recent}}$. All the while, the reference $\mathtt{x}_1$ in stack remains valid, and the test $\mathtt{x}_1 = \mathtt{x}_2$ eventually fails, because $y \neq 2y$ for $y > 0$. Specifically, note that the analysis result is sound and precise, and we never needed to make breaking changes to the stack component. Indeed, later in the paper, we develop

theories to show that analyses using the virtual recency abstraction for local variables are terminating and sound, and they can be defined modularly using analysis components.

To demonstrate the practicality of our solution and the reusability of analysis components, we apply the virtual recency abstraction to develop a flow-sensitive inter-procedural relational analysis for the research language *Tip* [42] and for *WebAssembly 1.0* [26]. Tip is a simple imperative language with functions, loops, and recursion. WebAssembly (Wasm) is a modern low-level language for the web, that is statically-typed and features structured control-flow. We implement a new component for the call frame that supports strong-updates by using the virtual recency abstraction, which we use for both languages. Our Wasm analysis soundly and precisely approximates 32 and 64-bit integer operation with overflow handling, IEEE754 floating-point operations with special value handling ($-\infty, \infty, -0.0, \mathrm{NaN}$), conversion operations between numeric types, assignment to local and global variables with strong-updates, the operand stack, and all other features of the Wasm language. We constructed the Wasm analysis by instantiating an existing analysis template for Wasm [8] called a *generic interpreter*. Generic interpreters capture analysis-independent functionality of a language and can be instantiated to obtain different analyses and also concrete interpreters [35, 6]. We instantiate the generic interpreter with new analysis components we developed specifically for the relational analysis, while soundly reusing existing components for Wasm's operand stack, function tables, exceptions, and failures.

To demonstrate the practicality of the virtual recency abstraction, we evaluate recall, scalability, and precision of the relational Wasm analysis on the Wasm 1.0 specification test suite, 7 compiled C programs from the benchmarks game suite, and 9 existing and 22 custom precision tests. We find that our Wasm analysis on the Wasm 1.0 specification test suite has a high recall of 100% for Polyhedra [15], 94% for Octagons [40], and 100% for Intervals [11]. Furthermore, on 7 compiled C programs of the benchmarks game suite with 255 to 1120 lines of code, the Wasm analysis has an average running time of 22s for Polyhedra, 13s for Octagons, and 1.3s for Intervals. Lastly, on the precision tests our Wasm analysis achieves 63% precision for Polyhedra, 57% for Octagons, and 29% for Intervals.

In summary, we make the following contributions:

- We introduce the virtual recency abstraction that enables strong updates for abstract interpreters with shared state (Section 2).
- We prove that analyses based on the virtual recency abstraction terminate (Section 3).
- We prove that the virtual recency abstraction allows to soundly reuse existing analysis components as-is (Section 4).
- We evaluate the modularity of the virtual recency abstraction by developing relational analyses for Tip and Wasm 1.0 that reuse existing analysis components (Section 5).
- We demonstrate the practicality of the relational Wasm analysis by evaluating recall, scalability, and precision (Section 6).

## 2 Virtual Recency Abstraction for Relational Analyses

To showcase the relevance of our approach, we start by presenting some analysis definitions and their shortcomings: unsoundness (Section 2.1), non-termination (Section 2.2), or non-modularity (Section 2.3). Then, we introduce our solution (Section 2.4), and explain how the virtual recency abstraction enables sound strong updates in relational analyses, while reusing existing modular analysis components as-is. Our solution will build upon various components of the alternative analyses.

## 2.1 Naive Strong Updates to Local Variables (Unsound)

We start with a relational analysis that performs *naive* strong updates on local variables. Local variables are managed in the *call frame*. The concrete call frame is a mapping from local variables of a function to their assigned values ($\mathsf{CallFrame} = \mathsf{Map}[\mathsf{Var}, \mathsf{Val}]$). For the sake of a simpler presentation, we focus on languages with integer values for now ($\mathsf{Val} = \mathbb{Z}$), but our relational analyses for Tip and WebAssembly have to abstract over values of different types (Section 5). A relational analysis abstracts call frames with a numerical relational abstraction over local variables such as convex Polyhedra [15] or Octagons [40] ($\widehat{\mathsf{CallFrame}}_1 = \mathsf{Rel}[\mathsf{Var}, \mathbb{Z}]$).[1] In case of convex Polyhedra, $\mathsf{Rel}[\mathsf{Var}, \mathbb{Z}]$ contains linear equalities and inequalities over variables in the program ($a_1\mathtt{x}_1 + \dots a_n\mathtt{x}_n \leq c$). Octagons sacrifice some precision compared to Polyhedra by limiting equalities and inequalities to two variables with unit coefficients ($\pm\mathtt{x}_1 \pm \mathtt{x}_2 \leq c$), in order to gain an exponential performance speedup. The inner working of these abstract domains is not relevant here and we refer interested readers to the primary sources listed above.

The abstract call frame defines three key operations: variable assignment, variable lookup, and assertion checking.

$$\widehat{\mathsf{assign}} : \mathsf{Var} \times \widehat{\mathsf{Val}}_1 \times \widehat{\mathsf{CallFrame}}_1 \to \widehat{\mathsf{CallFrame}}_1$$
$$\widehat{\mathsf{assign}}(x, v, \mathit{callFrame}) = \mathit{callFrame}[x \mapsto v]$$

$$\widehat{\mathsf{lookup}} : \mathsf{Var} \times \widehat{\mathsf{CallFrame}}_1 \to \widehat{\mathsf{Val}}_1$$
$$\widehat{\mathsf{lookup}}(x, \mathit{callFrame}) = \mathbf{if}(x \in \mathit{callFrame})\ x \ \mathbf{else}\ \bot$$

$$\widehat{\mathsf{assert}} : \widehat{\mathsf{BoolVal}}_1 \times \widehat{\mathsf{CallFrame}}_1 \to \widehat{\mathsf{Bool}}$$
$$\widehat{\mathsf{assert}}(\mathit{cons}, \mathit{callFrame}) = \begin{cases} \top & \text{if } \mathit{callFrame} \text{ satisfies } \mathit{cons} \text{ and } \mathsf{not}(\mathit{cons}) \\ \mathsf{true} & \text{if } \mathit{callFrame} \text{ satisfies } \mathit{cons} \\ \mathsf{false} & \text{if } \mathit{callFrame} \text{ satisfies } \mathsf{not}(\mathit{cons}) \end{cases}$$

$$\widehat{\mathsf{Val}}_1 = \mathsf{RelExpr}[\mathsf{Var}]$$
$$\mathsf{RelExpr}[V] ::= \bot \mid V \mid \mathsf{Const}(\mathsf{Interval}) \mid \mathsf{Add}(\mathsf{RelExpr}[V], \mathsf{RelExpr}[V]) \mid \dots$$
$$\widehat{\mathsf{BoolVal}}_1 = \mathsf{RelCons}[\mathsf{Var}]$$
$$\mathsf{RelCons}[V] ::= \mathsf{Eq}(\mathsf{RelExpr}[V], \mathsf{RelExpr}[V]) \mid \mathsf{Leq}(\mathsf{RelExpr}[V], \mathsf{RelExpr}[V]) \mid \dots$$

Operation $\widehat{\mathsf{assign}}(x, v, \mathit{callFrame})$ strongly updates variable $x$ within the relational abstract domain by overwriting its prior value. Operation $\widehat{\mathsf{lookup}}(x, \mathit{callFrame})$ returns a symbolic reference to variable $x$ if it is bound in *callFrame* and $\bot$ otherwise. Operation $\widehat{\mathsf{assert}}(\mathit{cons}, \mathit{callFrame})$ asserts if *callFrame* does or does not satisfy a given constraint *cons*. These operations are defined on numeric values abstracted with symbolic numeric expressions over local variables ($\widehat{\mathsf{Val}}_1 = \mathsf{RelExpr}[\mathsf{Var}]$) and boolean values abstracted with symbolic boolean expressions ($\widehat{\mathsf{BoolVal}}_1 = \mathsf{RelCons}[\mathsf{Var}]$). Symbolic expressions are computed by the analysis as overapproximations of program expressions. For example, the abstract interpreter evaluates program expression $\mathtt{abs(x)}$ to symbolic expression $\mathtt{x}$ in case $x \geq 0$ and $\mathtt{x} \sqcup \mathsf{Negate}(\mathtt{x})$ otherwise.

With these call frame operations we reproduce the unsoundness problem of $\mathsf{Analysis}_1$ in the first example of Section 1:

[1] We use hats to distinguish abstract definitions from their same named concrete definitions without a hat. The number in the subscript after call frames distinguishes different versions of the call frame introduced throughout this section.

| Abstract Operations | $\widehat{\mathsf{CallFrame}}_1$ | $\widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_1]$ |
|---|---|---|
| 1: $c_1 \coloneqq \mathsf{assign}(\mathtt{y}, \mathsf{Const}([1,5]), [])$ | $[\mathtt{y} \in [1,5]]$ | $[]$ |
| 2: $c_2 \coloneqq \mathsf{assign}(\mathtt{x}, \mathtt{y}, c_1)$ | $[\mathtt{x} = \mathtt{y},\ \mathtt{y} \in [1,5]]$ | $[]$ |
| 3: $s_1 \coloneqq \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_2), [])$ | $[\mathtt{x} = \mathtt{y},\ \mathtt{y} \in [1,5]]$ | $[\mathtt{x}]$ |
| 4: $c_3 \coloneqq \mathsf{assign}(\mathtt{x}, \mathsf{Add}(\mathtt{y}, \mathtt{y}), c_2)$ | $[\mathtt{x} = 2\mathtt{y},\ \mathtt{y} \in [1,5]]$ | $[\mathtt{x}]$ |
| 5: $s_2 \coloneqq \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_3), s_1)$ | $[\mathtt{x} = 2\mathtt{y},\ \mathtt{y} \in [1,5]]$ | $[\mathtt{x}, \mathtt{x}]$ |
| 6: $\mathsf{assert}(\mathsf{Eq}(\mathsf{peek}(s_2, 1), \mathsf{peek}(s_2, 2)), c_3) = \mathsf{true}$ | | |

The abstract stack is a list of abstract values, where operation $\mathsf{push} : \widehat{\mathsf{Val}}_1 \times \widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_1] \to \widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_1]$ pushes an element on top of the stack and operation $\mathsf{peek} : \widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_1] \times \mathbb{N} \to \widehat{\mathsf{Val}}_1$ retrieves the element on the stack at a given index. The assertion in the last line confirms that the top two elements on the stack are equal in the latest call frame $c_3$. This is unsound because the value of variable x has been overwritten line 4, so we would expect that top two elements on the stack refer to different values and are not equal. However, because both elements on the stack refer to the same symbolic expression x, the assertion holds. The problem is that the naive strong update overwrites the value of x, however, this also changes the value of symbolic expression x on the stack, which is unsound.

A sound solution would be to weakly update local variables, joining newly assigned values into prior values: $\widehat{\mathsf{assign}}(x, v, \mathit{callFrame}) = \mathit{callFrame} \sqcup [x \mapsto v]$. With weak updates, $c_3$ in the example above would be $[\mathtt{y} \leq \mathtt{x} \leq 2\mathtt{y}, y \in [0,5]]$ and the assertion would return $\top$. This is sound, but imprecise because weak updates loose many benefits that relational analyses have over non-relational, making the performance trade-off not worth it.

## 2.2 Constant References to Local Variables (Non-Terminating)

The previous subsection has shown that naive strong updates to local variables also changes the value of references to these variables in other analysis components, which is unsound. This problem can be solved by implementing a call frame in which references to variables remain constant throughout the analysis. This is achieved by allocating a fresh reference – i.e, an *abstract address* – whenever a variable assignment is analyzed. This way references to variables via addresses remain constant.

To ensure that the analysis never runs out of addresses and can always assign new values to a freshly-generated addresses, the analysis defines addresses as the variable name paired with a natural number ($\widehat{\mathsf{Addr}}_2 = \mathsf{Var} \times \mathbb{N}$). These abstract addresses are added to the call frame ($\widehat{\mathsf{CallFrame}}_2 = \mathsf{Map}[\mathsf{Var}, \mathcal{P}(\widehat{\mathsf{Addr}}_2)] \times \mathsf{Rel}[\widehat{\mathsf{Addr}}_2, \mathbb{Z}]$) [51] and to abstract values ($\widehat{\mathsf{Val}}_2 = \mathsf{RelExpr}[\widehat{\mathsf{Addr}}_2]$, $\widehat{\mathsf{BoolVal}}_2 = \mathsf{RelCons}[\widehat{\mathsf{Addr}}_2]$). The powerset ($\mathcal{P}(\widehat{\mathsf{Addr}}_2)$) ensures that call frames can be joined if different addresses have been assigned to the same variable.

The operations on the new call frame are now defined as follows:

$$\mathsf{lookup} : \mathsf{Var} \times \widehat{\mathsf{CallFrame}}_2 \to \widehat{\mathsf{Val}}_2 \times \widehat{\mathsf{CallFrame}}_2$$
$$\mathsf{lookup}(x, \mathit{callFrame}@(\mathit{addrFrame}, _)) =$$
$$\quad \mathbf{if}(x \in \mathit{addrFrame})\ \mathsf{join}(\{a \mid a \in \mathit{addrFrame}(x)\}, \mathit{callFrame})\ \mathbf{else}\ (\bot, \bot)$$
$$\mathsf{assign} : \mathsf{Var} \times \widehat{\mathsf{Val}}_2 \times \widehat{\mathsf{CallFrame}}_2 \to \widehat{\mathsf{CallFrame}}_2$$
$$\mathsf{assign}(x, v, (\widehat{\mathit{addrFrame}}, \widehat{\mathit{relStore}})) =$$
$$\quad \mathbf{let}\ \mathit{addr} = \mathsf{fresh}(x)\mathbf{in}\ (\widehat{\mathit{addrFrame}}[x \mapsto \{\mathit{addr}\}], \widehat{\mathit{relStore}}[\mathit{addr} \mapsto v])$$

Operation lookup returns a reference to the addresses assigned to a local variable. If the call frame assigns multiple addresses to the variable, helper function $\mathsf{join} : \mathcal{P}(\widehat{\mathsf{Val}}) \times \widehat{\mathsf{CallFrame}}_2 \to \widehat{\mathsf{Val}} \times \widehat{\mathsf{CallFrame}}_2$ allocates a fresh address and joins the value of the other addresses into the

fresh address. Operation assign allocates a fresh address to hold the new value, which avoids overwriting values of symbolic expressions that reference the variable.

Going back to our running example with this new call frame, we now obtain a sound result: the strong update to variable x does not change the value of symbolic expression $\mathtt{x}_1$ on the stack.

| **Abstract Operations** | $\widehat{\mathsf{CallFrame}}_2$ | $\widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_2]$ |
|---|---|---|
| 1: $c_1 := \mathsf{assign}(\mathtt{y}, \mathsf{Const}([1,5]), [])$ | $[\mathtt{y} \mapsto \{\mathtt{y}_1\}], [\mathtt{y}_1 \in [1,5]]$ | $[]$ |
| 2: $c_2 := \mathsf{assign}(\mathtt{x}, \mathtt{y}_1, c_1)$ | $[\mathtt{x} \mapsto \{\mathtt{x}_1\}, \mathtt{y} \mapsto \{\mathtt{y}_1\}], [\mathtt{x}_1 = \mathtt{y}_1,\ \mathtt{y}_1 \in [1,5]]$ | $[]$ |
| 3: $s_1 := \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_2), [])$ | -"- | $[\mathtt{x}_1]$ |
| 4: $c_3 := \mathsf{assign}(\mathtt{x}, \mathsf{Add}(\mathtt{y}_1, \mathtt{y}_1), c_2)$ | $[\mathtt{x} \mapsto \{\mathtt{x}_2\}, \mathtt{y} \mapsto \{\mathtt{y}_1\}],$ $[\mathtt{x}_2 = 2\mathtt{y}_1,\ \mathtt{x}_1 = \mathtt{y}_1, \mathtt{y}_1 \in [1,5]]$ | $[\mathtt{x}_1]$ |
| 5: $s_2 := \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_3), s_1)$ | -"- | $[\mathtt{x}_2, \mathtt{x}_1]$ |
| 6: $\mathsf{assert}(\mathsf{Eq}(\mathsf{peek}(s_2, 1), \mathsf{peek}(s_2, 2)), c_3) = \mathsf{false}$ | | |

Unfortunately, the allocation of fresh addresses created a termination problem: the number of addresses that the analysis allocates is unbounded and hence the analysis does not terminate.

## 2.3 Recency Abstraction for Local Variables (Non-Modular)

The previous subsection has shown that assigning new values to freshly-generated addresses to keep references to addresses constant, created a termination problem. To solve this termination problem, the set of addresses must be finite and addresses must be reused on assignment. This can be done by adapting the *recency abstraction* [3], which traditionally has been used to handle dynamic memory allocation with a good precision-performance ratio. The recency abstraction distinguishes the most recent allocation of an object ($x_{\mathsf{recent}}$) from all prior allocations ($x_{\mathsf{old}}$). Whenever a new allocation is performed for the same object, $x_{\mathsf{recent}}$ is first retired with all other prior allocations, to make it usable through strong updates for the newly allocated object.

Here, we use the recency abstraction to abstract the addresses from the previous subsection ($\widehat{\mathsf{Addr}}_2 = \mathsf{Var} \times \mathbb{N}$) into a finite set of addresses ($\widehat{\mathsf{Addr}}_3 = \mathsf{Var} \times \{\mathsf{recent}, \mathsf{old}\}$) as shown with the following abstraction function:

$$\begin{array}{ccc} \mathsf{Var} \times \mathbb{N} & \{x_n\} & \{x_{n-1}, x_{n-2}, \dots x_1\} \\ \downarrow \alpha_{\mathsf{Recency}} & \downarrow & \downarrow \\ \mathsf{Var} \times \{\mathsf{recent}, \mathsf{old}\} & x_{\mathsf{recent}} & x_{\mathsf{old}} \end{array}$$

Thanks to the recency abstraction, we can perform strong updates on recent local variables, while soundly referring to old values.

While the lookup and assertion operations stay mostly the same for the new call frame, variable assignment changes as follows:

$$\begin{aligned} &\mathsf{assign} : \mathsf{Var} \times \widehat{\mathsf{Val}}_3 \times \widehat{\mathsf{CallFrame}}_3 \times \widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_3] \to \widehat{\mathsf{CallFrame}}_3 \times \widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_3] \\ &\mathsf{assign}(x, v, (\mathit{addrFrame}, \mathit{relStore}), \mathit{stack}) = \\ &\quad \mathbf{let}\ \mathit{addrFrame}' = \mathit{addrFrame}[x \mapsto \{x_{\mathsf{recent}}\}] \\ &\quad\quad \mathit{relStore}' = \mathsf{fold}(\mathit{relStore}, x_{\mathsf{recent}}, x_{\mathsf{old}})[x_{\mathsf{recent}} \mapsto v] \\ &\quad\quad \mathit{stack}' = [v'[x_{\mathsf{recent}}/x_{\mathsf{old}}] \mid v' \in \mathit{stack}] \\ &\quad \mathbf{in}\ ((\mathit{addrFrame}', \mathit{relStore}'), \mathit{stack}') \end{aligned}$$

Operation $\widehat{\mathsf{assign}}(x, v, \mathit{callFrame}, \mathit{stack})$ joins the prior recent value $x_{\mathsf{recent}}$ into address $x_{\mathsf{old}}$ and then strongly updates address $x_{\mathsf{recent}}$ to the newly assigned value $v$. The retirement of address $x_{\mathsf{recent}}$ into $x_{\mathsf{old}}$ is performed by the relational fold operator [23]. Afterwards, operation $\widehat{\mathit{assign}}$ broadcasts this retirement to all other components of the analysis: this is a crucial step, which would otherwise make the analysis unsound. In particular, all references to addresses $x_{\mathsf{recent}}$ on the stack are renamed into $x_{\mathsf{old}}$.

We obtain a sound result on the running example:

| **Abstract Operations** | $\widehat{\mathsf{CallFrame}}_3$ | $\widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_3]$ |
|---|---|---|
| 1: $c_1, s_1 := \mathsf{assign}(\mathtt{y}, \mathsf{Const}([1,5]), [], [])$ | $[\mathtt{y} \mapsto \{\mathtt{y}_{\mathsf{recent}}\}], [\mathtt{y}_{\mathsf{recent}} \in [1,5]]$ | $[]$ |
| 2: $c_2, s_2 := \mathsf{assign}(\mathtt{x}, \mathtt{y}_{\mathsf{recent}}, c_1, s_1)$ | $[\mathtt{x} \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{y} \mapsto \{\mathtt{y}_{\mathsf{recent}}\}]$, $[\mathtt{x}_{\mathsf{recent}} = \mathtt{y}_{\mathsf{recent}},\ \mathtt{y}_{\mathsf{recent}} \in [1,5]]$ | $[]$ |
| 3: $s_3 := \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_2), s_2)$ | -"- | $[\mathtt{x}_{\mathsf{recent}}]$ |
| 4: $c_3, s_4 := \mathsf{assign}(\mathtt{x}, \mathsf{Add}(\mathtt{y}_{\mathsf{recent}}, \mathtt{y}_{\mathsf{recent}}), c_2, s_3)$ | $[\mathtt{x}_{\mathsf{recent}} = 2\mathtt{y}_{\mathsf{recent}},\ \mathtt{x}_{\mathsf{old}} = \mathtt{y}_{\mathsf{recent}} \ldots]$ $[\mathtt{x} \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{y} \mapsto \{\mathtt{y}_{\mathsf{recent}}\}]$, | $[\mathtt{x}_{\mathsf{old}}]$ |
| 5: $s_5 := \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_3), s_4)$ | -"- | $[\mathtt{x}_{\mathsf{recent}}, \mathtt{x}_{\mathsf{old}}]$ |
| 6: $\mathsf{assert}(\mathsf{Eq}(\mathsf{peek}(s_5, 1), \mathsf{peek}(s_5, 2)), c_3) = \mathsf{false}$ | | |

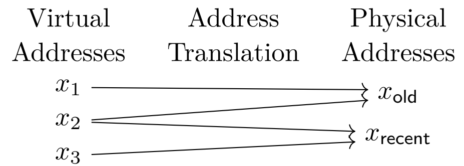
The key difference to before is that the assignment of variable `x` in line 4 does not overwrite the old value referenced by a symbolic expression on the stack. Instead, the call frame assigns the new value to address $\mathtt{x}_{\mathsf{recent}}$ and refers to the old value with $\mathtt{x}_{\mathsf{old}}$. This way the assertion in the last line confirms that the top two values on the stack are not equal, because one refers to $\mathtt{x}_{\mathsf{recent}}$ and the other to $\mathtt{x}_{\mathsf{old}}$.

Unfortunately, the recency abstraction causes a new modularity problem: the retirement of address $\mathtt{x}_{\mathsf{recent}}$ into $\mathtt{x}_{\mathsf{old}}$ needs to be broadcast and applied to all analysis components of the analysis state. This is not only slow, but also breaks the interface of analysis components, which now need to implement an address retirement operation. Furthermore, existing soundness proofs for these components become invalid and need to be reestablished for relational abstractions.

## 2.4 Virtual Recency Abstraction (Sound, Modular, Terminating)

To summarize, naive strong updates are unsound for relational abstractions, because they unsoundly change value to references in other analysis components. Assigning new values to freshly-generated addresses to keep references to other addresses constant is sound, but does not terminate because the number of addresses is unbounded. The recency abstraction is a finite abstraction over the infinite set of freshly-generated addresses and enables strong updates of local variables, however, the retirement of addresses requires to replace addresses in other analysis components, which is non-modular. We solve the final modularity problem by combining the recency abstraction with the generation of fresh addresses. Specifically, we add an extra layer of indirection between the two address types, similar to virtual memory in operating systems [19]. Therefore, we call this the *Virtual Recency Abstraction*.

The virtual recency abstraction defines a mapping between *virtual addresses* ($\mathsf{VirtAddr} = \mathsf{Var} \times \mathbb{N}$, addresses from Section 2.2) and symbolic expressions. Virtual addresses are first mapped to physical addresses ($\mathsf{PhysAddr} = \mathsf{Var} \times \{\mathsf{recent} \times \mathsf{old}\}$, addresses from Section 2.3), which are then used as variables in the relational domain. We call the mapping between virtual and physical addresses the *address translation* ($\widehat{\mathsf{AddrTrans}} = \mathsf{Map}[\mathsf{VirtAddr}, \mathcal{P}(\mathsf{PhysAddr})]$). The call frame



$$
\begin{aligned}
&\mathsf{lookup} : \mathsf{Var} \times \widehat{\mathsf{CallFrame}}_4 \to \widehat{\mathsf{Val}}_4 \times \widehat{\mathsf{CallFrame}}_4 \\
&\mathsf{lookup}(x, \mathit{callFrame}@(\mathit{frame}, \mathit{addrTrans}, \mathit{relStore})) = \\
&\quad \mathbf{if}(x \in \mathit{frame})\ \mathbf{let}\ \mathit{virtAddr} = \mathsf{fresh}(x) \\
&\qquad\qquad\qquad \mathit{addrTrans}' = \mathit{addrTrans} \sqcup [\mathit{virtAddr} \mapsto \bigcup_{v \in \mathit{frame}(x)} \mathit{addrTrans}(v)] \\
&\qquad\qquad \mathbf{in}\ (\mathit{virtAddr}, (\mathit{frame}, \mathit{addrTrans}', \mathit{relStore})) \\
&\quad \mathbf{else}\ (\bot, \bot) \\
&\mathsf{assign} : \mathsf{Var} \times \widehat{\mathsf{Val}}_4 \times \widehat{\mathsf{CallFrame}}_4 \to \widehat{\mathsf{CallFrame}}_4 \\
&\mathsf{assign}(x, v, \mathit{callFrame}) = \\
&\quad \mathbf{let}\ (v', (\mathit{frame}, \mathit{addrTrans}, \mathit{relStore})) = \mathsf{toPhysExpr}(v, \mathit{callFrame}) \\
&\qquad \mathit{virtAddr} = \mathsf{fresh}(x) \\
&\qquad \mathit{frame}' = \mathit{frame}[x \mapsto \{\mathit{virtAddr}\}] \\
&\qquad \mathit{addrTrans}' = \mathit{addrTrans}[x_n \mapsto \{x_{\mathsf{old}}\} \mid x_n \in \mathit{addrTrans}][\mathit{virtAddr} \mapsto \{x_{\mathsf{recent}}\}] \\
&\qquad \mathit{relStore}' = \mathsf{fold}(\mathit{relStore}, x_{\mathsf{recent}}, x_{\mathsf{old}})[x_{\mathsf{recent}} \mapsto v'] \\
&\quad \mathbf{in}\ (\mathit{frame}', \mathit{addrTrans}', \mathit{relStore}') \\
&\quad \mathbf{where}\ \mathsf{toPhysExpr} : \mathsf{RelExpr}[\mathsf{VirtAddr}] \times \widehat{\mathsf{CallFrame}}_4 \to \mathsf{RelExpr}[\mathsf{PhysAddr}] \times \widehat{\mathsf{CallFrame}}_4 \\
&\mathsf{assert} : \widehat{\mathsf{BoolVal}}_4 \times \widehat{\mathsf{CallFrame}}_4 \to \widehat{\mathsf{Bool}} \times \widehat{\mathsf{CallFrame}}_4 \\
&\mathsf{assert}(\mathit{cons}, \mathit{callFrame}) = \\
&\quad \mathbf{let}\ (\mathit{cons}', \mathit{callFrame}'@(_, _, \mathit{relStore})) = \mathsf{toPhysCons}(\mathit{cons}, \mathit{callFrame}) \\
&\quad \mathbf{in} \begin{cases} (\top, \mathit{callFrame}') & \text{if } \mathit{relStore} \text{ satisfies } \mathit{cons}' \text{ and } \mathsf{not}(\mathit{cons}') \\ (\mathsf{true}, \mathit{callFrame}') & \text{if } \mathit{relStore} \text{ satisfies } \mathit{cons}' \\ (\mathsf{false}, \mathit{callFrame}') & \text{if } \mathit{relStore} \text{ satisfies } \mathsf{not}(\mathit{cons}') \end{cases} \\
&\quad \mathbf{where}\ \mathsf{toPhysCons} : \mathsf{RelCons}[\mathsf{VirtAddr}] \times \widehat{\mathsf{CallFrame}}_4 \to \mathsf{RelCons}[\mathsf{PhysAddr}] \times \widehat{\mathsf{CallFrame}}_4
\end{aligned}
$$

**Figure 1** Operations of the Virtual Recency Abstraction

maps variables to virtual addresses, includes the address translation from virtual to physical addresses, and defines the relational abstract domain over physical addresses:

$$\widehat{\mathsf{CallFrame}}_4 = \mathsf{Map}[\mathsf{Var}, \mathcal{P}(\mathsf{VirtAddr})] \times \mathsf{Map}[\mathsf{VirtAddr}, \mathcal{P}(\mathsf{PhysAddr})] \times \mathsf{Rel}[\mathsf{PhysAddr}, \mathbb{Z}]$$

We now define the abstract operations for the call frame of the virtual recency abstraction (Figure 1). Operation $\mathsf{lookup}(x, \mathit{callFrame})$ returns a fresh virtual address that refers to all physical addresses that correspond to $x$ within the address translation. Operation $\mathsf{assign}(x, v, \mathit{callFrame})$ first converts the virtual addresses contained in value $v$ to physical addresses with helper function toPhysExpr. It then points variable $x$ to a freshly generated virtual address, assigns this virtual address to $x_{\mathsf{recent}}$ in the address translation, folds $x_{\mathsf{recent}}$ into $x_{\mathsf{old}}$ in the relational abstract domain, and finally assigns the converted value to $x_{\mathsf{recent}}$. Helper function toPhysExpr replaces virtual addresses in symbolic expression with physical addresses, such that the resulting expression can be assigned to the relational abstract domain. In case virtual address $x_n$ points to multiple physical addresses $\{x_{\mathsf{recent}}, x_{\mathsf{old}}\}$, toPhysExpr joins $x_{\mathsf{recent}}$ into $x_{\mathsf{old}}$ in the relational abstract domain and replaces virtual address $x_n$ with physical address $x_{\mathsf{old}}$ in the symbolic expression. Similarly, operation assert first converts

constraints over virtual addresses to constraints over physical addresses and afterwards decides if the resulting constraint is satisfied by the relational abstract domain.

To demonstrate that the virtual recency abstraction solves the unsoundness and modularity problem, we again reproduce the running example:

| Abstract Operations | $\widehat{\mathsf{CallFrame}}_4$ | $\widehat{\mathsf{Stack}}[\widehat{\mathsf{Val}}_4]$ |
|---|---|---|
| 1: $c_1 \coloneqq \mathsf{assign}(\mathtt{y}, \mathsf{Const}([1,5]), [])$ | $[\mathtt{y} \mapsto \{\mathtt{y}_1\}], [\mathtt{y}_1 \mapsto \{\mathtt{y}_{\mathsf{recent}}\}], [\mathtt{y}_{\mathsf{recent}} \in [1,5]]$ | $[]$ |
| 2: $c_2 \coloneqq \mathsf{assign}(\mathtt{x}, \mathtt{y}_1, c_1)$ | $[\mathtt{x} \mapsto \{\mathtt{x}_1\}, \mathtt{y} \mapsto \{\mathtt{y}_1\}]$, $[\mathtt{x}_1 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{y}_1 \mapsto \{\mathtt{y}_{\mathsf{recent}}\}]$, $[\mathtt{x}_{\mathsf{recent}} = \mathtt{y}_{\mathsf{recent}},\ \mathtt{y}_{\mathsf{recent}} \in [1,5]]$ | $[]$ |
| 3: $s_3 \coloneqq \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_2), [])$ | -"- | $[\mathtt{x}_1]$ |
| 4: $c_3, s_4 \coloneqq \mathsf{assign}(\mathtt{x}, \mathsf{Add}(\mathtt{y}_1, \mathtt{y}_1), c_2)$ | $[\mathtt{x} \mapsto \{\mathtt{x}_2\}, \mathtt{y} \mapsto \{\mathtt{y}_1\}]$, $[\mathtt{x}_2 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{x}_1 \mapsto \{\mathtt{x}_{\mathsf{old}}\} \ldots]$, $[\mathtt{x}_{\mathsf{recent}} = 2\mathtt{y}_{\mathsf{recent}},\ \mathtt{x}_{\mathsf{old}} = \mathtt{y}_{\mathsf{recent}} \ldots]$ | $[\mathtt{x}_1]$ |
| 5: $s_5 \coloneqq \mathsf{push}(\mathsf{lookup}(\mathtt{x}, c_3), s_4)$ | -"- | $[\mathtt{x}_2, \mathtt{x}_1]$ |
| 6: $\mathsf{assert}(\mathsf{Eq}(\mathsf{peek}(s_5, 1), \mathsf{peek}(s_5, 2)), c_3) = \mathsf{false}$ | | |

The analysis in this example is sound because the assignment in line 4 does not overwrite the prior value of virtual address $\mathtt{x}_1$, but moves it from physical address $\mathtt{x}_{\mathsf{recent}}$ to $\mathtt{x}_{\mathsf{old}}$. The new value for virtual address $\mathtt{x}_2$ is then written to a physical address $\mathtt{x}_{\mathsf{recent}}$. Furthermore, the analysis is modular, because virtual address $\mathtt{x}_1$ is retired to $\mathtt{x}_{\mathsf{old}}$ by adjusting the address translation, without the need to change addresses on the stack.

To summarize, the virtual recency abstraction solves all the issues of soundness, termination and modularity illustrated in the previous subsections. The next sections will provide further details on the features of our virtual recency abstraction:

**Termination** The virtual recency abstraction terminates, thanks to an ordering of the call frame based on physical addresses, which is a finite set. We detail this claim in Section 3.

**Soundness** The virtual recency abstraction enables sound strong updates by assigning to recent addresses and retiring references to old. We prove soundness formally in Section 4.

**Modularity** The virtual recency abstraction solves the modularity problem, allowing to reuse existing analysis components as-is. For example, our relational analysis for WebAssembly reuses existing analysis components for the stack, memory, global variables, function tables, exceptions, and failures, which were originally designed for a non-relational analysis (Section 5).

In the following parts of the paper, we keep the callframe abstraction described in this subsection: $\widehat{\mathsf{CallFrame}} = \widehat{\mathsf{CallFrame}}_4$.

## 3 Termination of the Virtual Recency Abstraction

In this section, we address the termination of analyses that use the virtual recency abstraction. Specifically, we describe a procedure that terminates analyses despite the infinite supply of virtual addresses.

### 3.1 Terminate Infinite Ascending Chains on Call Frames

One problem that threatens termination of the analysis is the unbounded number of virtual addresses in the call frame. To solve this problem, we define an ordering on call frames that does not compare virtual addresses directly. Instead, the ordering maps virtual addresses to

physical addresses, which is a finite set.

$$
\begin{aligned}
&(\mathit{frame}_1, \mathit{addrTrans}_1, \mathit{relStore}_1) \sqsubseteq_{\widehat{\mathsf{CallFrame}}} (\mathit{frame}_2, \mathit{addrTrans}_2, \mathit{relStore}_2) \\
&\quad \text{iff} \;\; (\mathit{addrTrans}_1 \circ \mathit{frame}_1) \sqsubseteq_{\mathsf{Map}[\mathsf{Var},\mathcal{P}(\mathsf{PhysAddr})]} (\mathit{addrTrans}_2 \circ \mathit{frame}_2) \\
&\quad \wedge \;\; \mathit{relStore}_1 \sqsubseteq_{\mathsf{Rel}[\mathsf{PhysAddr},\mathbb{Z}]} \mathit{relStore}_2
\end{aligned}
$$

This pre-order on call frames is not anti-symmetric ($c_1 \overset{\sqsubseteq}{\sqsupseteq} c_2$ does not imply $c_1 = c_2$), e.g.,

$$
\begin{aligned}
c_1 &= ([\mathtt{x} \mapsto \{\mathtt{x}_3\}], [\mathtt{x}_3 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{x}_2 \mapsto \{\mathtt{x}_{\mathsf{old}}\}, \ldots], [\mathtt{x}_{\mathsf{recent}} = \mathtt{x}_{\mathsf{old}} - 1 \wedge 0 \le \mathtt{x}_{\mathsf{old}} \le 100]) \\
c_2 &= ([\mathtt{x} \mapsto \{\mathtt{x}_4\}], [\mathtt{x}_4 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{x}_3 \mapsto \{\mathtt{x}_{\mathsf{old}}\}, \ldots], [\mathtt{x}_{\mathsf{recent}} = \mathtt{x}_{\mathsf{old}} - 1 \wedge 0 \le \mathtt{x}_{\mathsf{old}} \le 100])
\end{aligned}
$$

The lack of anti-symmetry is crucial for the virtual recency abstraction, because normalizing call frames to a unique representation would require updating virtual addresses in analysis components, which is not modular (Section 2.3).

Another problem that threatens termination are infinite ascending chains on relational abstract domains ($\mathsf{Rel}[\mathsf{PhysAddr}, \mathbb{Z}]$) contained within the call frame. To this end, we define a widening operator on call frames that applies a widening operator on the contained relational abstract domain. A *widening operator* [12] $\nabla : L \times L \to L$ computes an upper bound of its operands $x \sqsubseteq x \nabla y \sqsupseteq y$ and terminates an infinite ascending chain $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \ldots$ when folded over it: $x_1 \sqsubseteq x_1 \nabla x_2 \sqsubseteq (x_1 \nabla x_2) \nabla x_3 \sqsubseteq \ldots$ The widening operator on call frames is defined as follows:

$$
\begin{aligned}
&(\mathit{frame}_1, \mathit{addrTrans}_1, \mathit{relStore}_1) \nabla_{\widehat{\mathsf{CallFrame}}} (\mathit{frame}_2, \mathit{addrTrans}_2, \mathit{relStore}_2) = \\
&\quad \mathbf{let}\ \mathit{frame} = [x \mapsto \mathit{frame}_1(x) \cup \mathit{frame}_2(x) \mid x \in \mathit{frame}_1 \vee x \in \mathit{frame}_2] \\
&\qquad \mathit{addrTrans} = [v \mapsto \mathit{addrTrans}_1(v) \cup \mathit{addrTrans}_2(v) \mid v \in \mathit{addrTrans}_1 \vee v \in \mathit{addrTrans}_2] \\
&\qquad \mathit{relStore} = \mathit{relStore}_1 \nabla_{\mathsf{Rel}[\mathsf{PhysAddr},\mathbb{Z}]} \mathit{relStore}_2 \\
&\quad \mathbf{in}\ (\mathit{frame}, \mathit{addrTrans}, \mathit{relStore})
\end{aligned}
$$

This widening operator terminates an infinite ascending chain of call frames $\mathit{callFrame}_1 \sqsubseteq \mathit{callFrame}_2 \ldots$, because the address frame after mapping virtual addresses to physical addresses is a lattice of finite height ($\mathsf{Map}[\mathsf{Var}, \mathcal{P}(\mathsf{PhysAddr})]$). Furthermore, the relational abstract domain terminates by its own widening operator.

Lastly, we describe a fixpoint algorithm that iterates a monotone analysis function $\widehat{f} : P \to P$ over a pre-order $(P, \sqsubseteq, \bot)$ with least element $\bot$ and ensures termination with a widening operator $\nabla : P \times P \to P$. The fixpoint algorithm iterates $(x_i)_{i \in \mathbb{N}}$ defined by $x_1 = \bot$ and $x_{n+1} = x_n \nabla \widehat{f}(x_n)$ until the ascending chain terminates after $p \in \mathbb{N}$ steps with $x_p \sqsubseteq x_{p-1}$. This final iteratee $x_p$ soundly overapproximates the least fixpoint of a concrete semantics function $f : L \to L$ over lattice $(L, \preceq)$:

▶ **Theorem 1** (Fixpoint Iteration with Widening over Pre-Orders [10, Theorem 34]). *Let $\widehat{f} : P \to P$ be a monotone analysis function over pre-order $(P, \sqsubseteq, \bot)$ that soundly overapproximates the monotone $f : L \to L$ over lattice $(L, \preceq)$ such that for a monotone concretization $\gamma : P \to L$ it holds $\forall x \in P. \gamma(\widehat{f}(x)) \succeq f(\gamma(x))$.*

1. *The fixpoint iteration $(x_i)_{i \in \mathbb{N}}$ defined above reaches a post fixpoint $x_p \sqsupseteq \widehat{f}(x_p)$.*
2. *Every such post fixpoint $x_p$ overapproximates the least fixpoint of $f$: $\gamma(x_p) \succeq \mathsf{lfp}(f)$.*

**Proof.** Proofs can be found in our supplementary material and [10, Theorem 34]. ◀

## 3.2 Terminate Infinite Ascending Chains on Values and Analysis States

Another problem that threatens the termination of the analysis is the unbounded number of abstract values ($\widehat{\mathsf{Val}} = \mathsf{RelExpr}[\mathsf{VirtAddr}]$). To solve this problem, the ordering on abstract values does not compare symbolic expressions $v_1, v_2$ syntactically, but compares them semantically by evaluating them in the given call frames $c_1, c_2$:

$$(v_1, c_1) \sqsubseteq_{\widehat{\mathsf{Val}}} (v_2, c_2) \text{ iff } c_1' = \widehat{\mathsf{assign}}(\mathit{fresh}, v_1, c_1) \wedge c_2' = \widehat{\mathsf{assign}}(\mathit{fresh}, v_2, c_2) \wedge c_1' \sqsubseteq_{\widehat{\mathsf{CallFrame}}} c_2'$$

For example, expressions $\mathsf{Add}(\mathtt{x}_1, \mathtt{y}_1)$ and $\mathsf{Mul}(\mathtt{x}_1, \mathsf{Const}(2))$ are indistinguishable by the value ordering in call frame $(\ldots, [\mathtt{x}_1 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{y}_1 \mapsto \{\mathtt{y}_{\mathsf{recent}}\}], [1 \leq \mathtt{x}_{\mathsf{recent}} = \mathtt{y}_{\mathsf{recent}} \leq 10])$.

The widening operator on abstract values takes two values and two call frames as input.

$$\nabla_{\widehat{\mathsf{Val}}} : (\widehat{\mathsf{Val}} \times \widehat{\mathsf{CallFrame}}) \times (\widehat{\mathsf{Val}} \times \widehat{\mathsf{CallFrame}}) \to \widehat{\mathsf{Val}} \times \widehat{\mathsf{CallFrame}} \times \widehat{\mathsf{CallFrame}}$$

The full definition of the operator is given in Definition 2 in the supplementary material. In this section, we discuss the operator at a high-level. Depending on the expressions contained in the abstract values, the operator implements different widening strategies:

- In case the expressions are both constant, i.e. contain no variables, each expression is evaluated in the respective call frame and the resulting intervals are widened.
- In case the expressions are structurally equal, i.e. differ only in the recency of their addresses, the widening operator attempts a structural join. This happens often when the fixpoint algorithm widens two analysis states and handling this case avoids adding addresses to the relational abstract domain to hold the join of expressions. For example, the analysis of loop `for(int i = 0; ...; i++)` requires to widen two expressions for the incrementation of the index variable `i`: $\mathsf{Add}(\mathtt{i}_{n-1}, \mathsf{Const}(1))$ where $\mathtt{i}_{n-1}$ is old and $\mathsf{Add}(\mathtt{i}_n, \mathsf{Const}(1))$ where $\mathtt{i}_n$ is recent. The widening operator will join these two expressions into $\mathsf{Add}(\mathtt{i}_{n+1}, \mathsf{Const}(1))$, where $\mathtt{i}_{n+1}$ mapped to $\mathsf{recent}$ and $\mathsf{old}$ by the address translation.
- In all other cases the operator assigns expressions $v_1, v_2$ to a *join address* $\mathit{ctx}_k$ in the respective call frame.

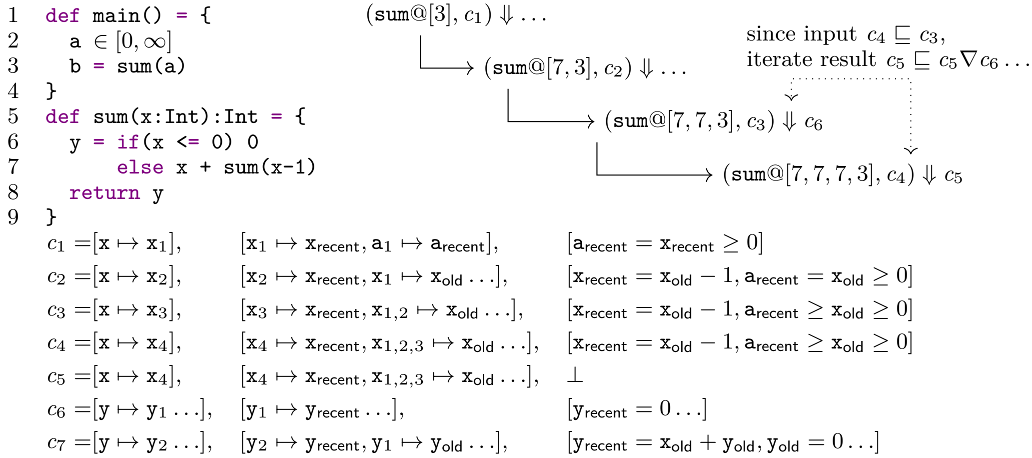
$$\begin{aligned}
&(v_1, (\mathit{frame}_1, \mathit{addrTrans}_1, \mathit{relStore}_1)) \nabla_{\widehat{\mathsf{Val}}} (v_2, (\mathit{frame}_2, \mathit{addrTrans}_2, \mathit{relStore}_2)) = \\
&\quad \textbf{let } \mathit{ctx}_k = \mathsf{alloc}(\mathit{expr}_1, \mathit{expr}_2) \\
&\qquad \mathit{callFrame}_1' = (\mathit{frame}_1, \mathit{addrTrans}_1[\mathit{ctx}_k \mapsto \{\mathit{ctx}_{\mathsf{old}}\}], \mathit{relStore}_1[\mathit{ctx}_{\mathsf{old}} \mapsto v_1]) \\
&\qquad \mathit{callFrame}_2' = (\mathit{frame}_2, \mathit{addrTrans}_2[\mathit{ctx}_k \mapsto \{\mathit{ctx}_{\mathsf{old}}\}], \mathit{relStore}_2[\mathit{ctx}_{\mathsf{old}} \mapsto v_2]) \\
&\quad \textbf{in } (\mathit{ctx}_k, \mathit{callFrame}_1', \mathit{callFrame}_2')
\end{aligned}$$

  The context of join address $\mathit{ctx}_n$ is decided by function $\mathsf{alloc}$, which is an additional parameter of the widening operator. Depending on where operator $\nabla_{\widehat{\mathsf{Val}}}$ is called, different contexts may be allocated. For example, when two stack entries are widened, the allocator might return a context $\mathsf{Stack}(\mathit{position}, \mathit{instruction}, \mathit{function})$ to indicate the position on the stack, the current instruction, and function where the widening occurred.

Finally, we describe a widening operator $\nabla_{\widehat{\mathsf{State}}}$ on compound analysis states, whose type $\widehat{\mathsf{State}}$ we do not specify for generality. Since two compound analysis states $S, S'$ with respective call frames $c_1, c_1'$ contain many values $v_1 \ldots v_n$ and $v_1' \ldots v_n'$ that need to be widened pair-wise, we fold operator $\nabla_{\widehat{\mathsf{Val}}}$ over all values $(v_i, c_i) \nabla_{\widehat{\mathsf{Val}}} (v_1', c_i') = (v_i'', c_{i+1}, c_{i+1}')$ where $1 \leq i \leq n$ and finally apply the widening operator on the two resulting call frames $c_n \nabla_{\widehat{\mathsf{CallFrame}}} c_n'$. This terminates infinite ascending chains of compound analysis states because abstract values are compared by evaluating them in respective call frames, which themselves are widened during every step.

## 3.3 Terminating Analysis of Recursive Functions

In this subsection, we describe how the virtual recency abstraction terminates the analysis of recursive functions. To ensure termination of the analysis, we use an existing fixpoint algorithm [34], which we illustrate at the example of the recrusive `sum` function:




The graph to the top-right shows a trace of the analysis, where $(\texttt{sum}@ctx, c) \Downarrow c'$ can be read as function `sum` in calling context *ctx* under input call frame $c$ evaluates to output call frame $c'$. We use short-hand notation for the frame and address translation, e.g., $[\mathsf{x}_{1,2} \mapsto \mathsf{x}_{\mathsf{old}}]$ stands for $[\mathsf{x}_1 \mapsto \{\mathsf{x}_{\mathsf{old}}\}, \mathsf{x}_2 \mapsto \{\mathsf{x}_{\mathsf{old}}\}]$. The fixpoint algorithm first recursively analyzes function `sum`, widening consecutive input call frames after the function parameters have been assigned, until they do not grow any further at $c_4 \sqsubseteq c_3$. The algorithm then iterates between the recurrent recursive calls $\texttt{sum}@[7,7,7,3]$ and $\texttt{sum}@[7,7,3]$ until the widened output call frames $c_5, c_5 \nabla c_6, (c_5 \nabla c_6) \nabla c_7 \ldots$ do not grow anymore. Once the iteration is finished, the algorithm returns to the top-most call with final result:

$$[\mathsf{a}_{\mathsf{recent}} \geq \mathsf{x}_{\mathsf{old}} \geq \mathsf{x}_{\mathsf{recent}} \geq 0,\ \mathsf{x}_{\mathsf{recent}} \geq \mathsf{x}_{\mathsf{old}} - 1,\ \mathsf{b}_{\mathsf{recent}} = \mathsf{y}_{\mathsf{recent}} = \mathsf{x}_{\mathsf{old}} + \mathsf{y}_{\mathsf{old}} \geq \mathsf{x}_{\mathsf{old}}]$$

The inequalities $\mathsf{a}_{\mathsf{recent}} \geq \mathsf{x}_{\mathsf{old}} \geq \mathsf{x}_{\mathsf{recent}} \geq 0$ show that parameter `x` decreases from `a` down to 0. Furthermore, $\mathsf{y}_{\mathsf{recent}} = \mathsf{x}_{\mathsf{old}} + \mathsf{y}_{\mathsf{old}} \geq \mathsf{x}_{\mathsf{old}}$ show that the last result is the sum of any `x` with any result `y`, which is greater than any input $\mathsf{x}_{\mathsf{old}}$.

# 4 Soundness of the Virtual Recency Abstraction

In this section, we prove that existing analysis components remain sound when instantiated relational values in form of symbolic expressions. We prove this by showing that abstract components remain sound overapproximations of concrete components on all operations of the abstract call frame. To prove soundness, we first define concretization functions on abstract call frames, values, and compound analysis states.

## 4.1 Concretizations on Call Frames, Values, and Analysis States

Typically, soundness would be proven with respect to Galois connections [12]. However, some relational abstract domains, such as polyhedra, do not enjoy Galois connections [41, Paragraph "Absence of Galois Connection"]. The problem is that the abstraction function $\alpha$, that computes the best possible approximation of a concrete property in the abstract domain, may not exist. Instead, we prove soundness with respect to monotone concretization

functions $\gamma$ from abstract domain to concrete domain, which is also used in mechanized soundness proofs [31]. The full definitions of the concretization functions can be found in Definitions 3-5 in the supplementary material. Here we discuss the concretizations at a high-level by example.

The concretization function for abstract call frames is the composition of two functions:

$$\begin{array}{ll}\widehat{\mathsf{CallFrame}} & = \mathsf{Map}[\mathsf{Var}, \mathcal{P}(\mathsf{VirtAddr})] \times \mathsf{Map}[\mathsf{VirtAddr}, \mathcal{P}(\mathsf{PhysAddr})] \times \mathsf{Rel}[\mathsf{PhysAddr}, \mathbb{Z}] \\ \quad\big\downarrow \gamma_{\widehat{\mathsf{CallFrame}}} & \\ \mathcal{P}(\mathsf{CallFrame}') & = \mathcal{P}(\mathsf{Map}[\mathsf{Var}, \mathcal{P}(\mathsf{VirtAddr})] \times \mathsf{Map}[\mathsf{VirtAddr}, \mathbb{Z}]) \\ \quad\big\downarrow \gamma_{\mathsf{CallFrame}'} & \\ \mathcal{P}(\mathsf{CallFrame}) & = \mathcal{P}(\mathsf{Map}[\mathsf{Var}, \mathbb{Z}])\end{array}$$

This two-step-concretization is needed because the concretization for values evaluates them in $\mathsf{CallFrame}'$. Concretization $\gamma_{\widehat{\mathsf{CallFrame}}}$ expands the codomain of the relational abstraction to virtual addresses and then concretizes it. This is necessary to assign virtual addresses on the same physical address their own individual values. Concretization $\gamma_{\mathsf{CallFrame}'}$ maps to the result of the first to regular concrete call frames. To illustrate the concretization on call frames, consider the following example:

$$\begin{aligned} c &= ([\mathtt{x} \mapsto \{\mathtt{x}_3\}], [\mathtt{x}_1 \mapsto \{\mathtt{x}_{\mathsf{old}}\}, \mathtt{x}_2 \mapsto \{\mathtt{x}_{\mathsf{old}}\}, \mathtt{x}_3 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}], [\mathtt{x}_{\mathsf{old}} < \mathtt{x}_{\mathsf{recent}}]) \\ \gamma_{\widehat{\mathsf{CallFrame}}}(c) &= \bigcup_{i<k, j<k} \{\ ([\mathtt{x} \mapsto \{\mathtt{x}_3\}], [\mathtt{x}_1 \mapsto i, \mathtt{x}_2 \mapsto j, \mathtt{x}_3 \mapsto k])\ \} \\ & \qquad \{([\mathtt{x} \mapsto \{\mathtt{x}_3\}], [\mathtt{x}_1 \mapsto 1, \mathtt{x}_2 \mapsto 1, \mathtt{x}_3 \mapsto 2]), \\ & \qquad\ ([\mathtt{x} \mapsto \{\mathtt{x}_3\}], [\mathtt{x}_1 \mapsto 1, \mathtt{x}_2 \mapsto 2, \mathtt{x}_3 \mapsto 3]), \ldots\} \\ \gamma_{\mathsf{CallFrame}'}(\gamma_{\widehat{\mathsf{CallFrame}}}(c)) &= \{[\mathtt{x} \mapsto i] \mid i \in \mathbb{Z}\} = \{\ldots [\mathtt{x} \mapsto -1], [\mathtt{x} \mapsto 0], [\mathtt{x} \mapsto 1] \ldots\}\end{aligned}$$

The concretization function for abstract values ($\widehat{\mathsf{Val}} = \mathsf{RelExpr}[\mathsf{VirtAddr}]$) evaluates symbolic expressions in a given $\mathsf{CallFrame}'$:

$$\gamma_{\widehat{\mathsf{Val}}} : \mathsf{CallFrame}' \to \widehat{\mathsf{Val}} \to \mathcal{P}(\mathsf{Val})$$

Symbolic expressions are not evaluated in $\widehat{\mathsf{CallFrame}}$ because this would loose the relations between variables, which makes the soundness proof meaningless [43]. To illustrate the concretization of abstract values, consider the interpretation of $(x_3 - x_1) * [1, 3]$ in a call frame $c' \in \gamma_{\widehat{\mathsf{CallFrame}}}(c)$:

$$\begin{aligned} c' &= (\ldots, [\mathtt{x}_1 \mapsto 1, \mathtt{x}_2 \mapsto 1, \mathtt{x}_3 \mapsto 2]) \\ \gamma_{\widehat{\mathsf{Val}}}(c')(\mathsf{Mul}(\mathsf{Sub}(\mathtt{x}_3, \mathtt{x}_1), \mathsf{Const}([1, 3]))) &= \{(2-1)*1, (2-1)*2, (2-1)*3\} = \{1, 2, 3\}\end{aligned}$$

Next, we describe the concretization of analysis components. To stay as generic as possible, we do not require any structure of analysis components. The only requirement is that analysis must be parametric in the type of values ($\widehat{\mathsf{Component}}[\widehat{\mathsf{Val}}]$), such that they can be reused in a relational analysis. Because components are parametric in the type of values, their concretization function is also parameterized by a concretization function for values:

$$\gamma_{\widehat{\mathsf{Component}}} : \forall \widehat{V}.\ (\widehat{V} \to \mathcal{P}(\mathsf{Val})) \to (\widehat{\mathsf{Component}}[\widehat{V}] \to \mathcal{P}(\mathsf{Component}))$$

As an example, consider the concretization function for a stack analysis component:

$$\widehat{\mathsf{Stack}}[\mathit{Val}] = \mathsf{List}[\mathit{Val}] \qquad \mathsf{Stack}[\mathit{Val}] = \mathsf{List}[\mathit{Val}]$$

$$\gamma_{\widehat{\mathsf{Stack}}} : \forall \widehat{\mathit{Val}}, \mathit{Val}.\, (\widehat{\mathit{Val}} \to \mathcal{P}(\mathit{Val})) \to \widehat{\mathsf{Stack}}[\widehat{\mathit{Val}}] \to \mathcal{P}(\mathsf{Stack}[\mathit{Val}])$$

$$\gamma_{\widehat{\mathsf{Stack}}}(\gamma_{\widehat{\mathit{Val}}})(\widehat{\mathit{stack}}) = \{\mathit{stack} \mid \mathsf{size}(\mathit{stack}) = \mathsf{size}(\widehat{\mathit{stack}}) \wedge \forall i.\ \mathit{stack}[i] \in \gamma_{\widehat{\mathit{Val}}}(\widehat{\mathit{stack}}[i])\}$$

To use the concretization function for stacks, we apply it to the concretization function for abstract values:

$$\gamma_{\widehat{\mathsf{Stack}}}(\gamma_{\widehat{\mathsf{Val}}}(\dots, [\mathtt{x}_1 \mapsto 1, \mathtt{x}_2 \mapsto 1, \mathtt{x}_3 \mapsto 2]))([\mathtt{x}_1, \mathtt{x}_3]) = \{[1, 2]\}$$

Finally, we describe the concretization function on compound analysis states. The analysis state consists of multiple components together with the call frame of the virtual recency abstraction:

$$\mathsf{State} = \mathsf{Component}_1 \times \dots \mathsf{Component}_n \times \mathsf{CallFrame}$$
$$\widehat{\mathsf{State}} = \widehat{\mathsf{Component}}_1[\widehat{\mathsf{Val}}] \times \dots \widehat{\mathsf{Component}}_n[\widehat{\mathsf{Val}}] \times \widehat{\mathsf{CallFrame}}$$

The concretization function $\gamma_{\widehat{\mathsf{State}}} : \widehat{\mathsf{State}} \to \mathcal{P}(\mathsf{State})$ partially applies a concretized call frame $c' \in \gamma_{\widehat{\mathsf{CallFrame}}}(\widehat{c})$ to the concretization function for values to obtain a concretization functions on components: $\gamma_{\widehat{\mathsf{Component}}_i}[\widehat{\mathsf{Val}}](\gamma_{\widehat{\mathsf{Val}}}(\mathit{callFrame}')) : \widehat{\mathsf{Component}}_i[\widehat{\mathsf{Val}}] \to \mathcal{P}(\mathsf{Component}_i)$.

## 4.2 Soundness Theorem

Now we are ready to state the main soundness theorem. The theorem guarantees that existing analysis components can be soundly reused in a relational analysis that uses the virtual recency abstraction.

▶ **Theorem 2** (Modular Soundness of the Virtual Recency Abstraction)**.** *Given a modular analysis consisting of the components* $\widehat{\mathsf{Component}}_1, \dots \widehat{\mathsf{Component}}_n, \widehat{\mathsf{CallFrame}}$. *All analysis components remain sound overapproximations of their concrete counterparts on all operations of the call frame of the virtual recency abstraction:*

$$\begin{aligned}
&\forall (\mathit{comps}, \mathit{callFrame}) \in \gamma_{\widehat{\mathsf{State}}}(\widehat{\mathit{comps}}, \widehat{\mathit{callFrame}}),\ v \in \gamma_{\widehat{\mathsf{Val}}}(\widehat{v}, \mathit{callFrame}),\\
&\quad b \in \gamma_{\widehat{\mathsf{BoolVal}}}(\widehat{b}, \mathit{callFrame}),\ x \in \mathsf{Var}.\\
&\qquad (\mathit{comps}, \pi_2(\mathsf{lookup}(x, \mathit{callFrame}))) \in \gamma_{\widehat{\mathsf{State}}}(\widehat{\mathit{comps}}, \pi_2(\widehat{\mathsf{lookup}}(x, \widehat{\mathit{callFrame}}))) \wedge\\
&\qquad (\mathit{comps}, \mathsf{assign}(x, v, \mathit{callFrame})) \in \gamma_{\widehat{\mathsf{State}}}(\widehat{\mathit{comps}}, \widehat{\mathsf{assign}}(x, \widehat{v}, \widehat{\mathit{callFrame}})) \wedge\\
&\qquad (\mathit{comps}, \pi_2(\mathsf{assert}(b, \mathit{callFrame}))) \in \gamma_{\widehat{\mathsf{State}}}(\widehat{\mathit{comps}}, \pi_2(\widehat{\mathsf{assert}}(\widehat{b}, \widehat{\mathit{callFrame}}))).
\end{aligned}$$

**Proof.** The proof is in the supplementary material accompanying this paper. ◀

# 5 Modular Reuse of Existing Analysis Components

In this section, we evaluate the modularity of the virtual recency abstraction by using it to develop relational analyses for two languages and report on the reuse of existing analysis functionality. Specifically, we developed analyses for the research language *Tip* and *WebAssembly 1.0*.

## 5.1 Relational Analysis for Tip

*Tip* [42] is a small imperative research language with functions, conditionals, loops, integers, references, and records.

We developed a relational analysis for Tip in Scala in the Sturdy framework by reusing code of an existing non-relational analysis for Tip.[2] Figure 3 shows a Venn diagram that

[2] We intend to submit an artifact including the code of the analyses.

| **Relational Tip Analysis**<br>Total 6360 loc (100.0%) | **Reused Code**<br>Total 2642 loc (41.5%) | **New Code**<br>Total 3718 loc (58.5%) |
|---|---|---|
| **Language-Independent**<br>Total 5200 loc (81.8%) | Functions<br>References & Records<br>Console Inp. & Out.<br>Failure<br>Loops & Recursion<br>2369 loc (37.2%) | Apron Integration<br>Call Frame<br>Heap<br>Integer Operations<br>Eq./ Ord. Operations<br>2831 loc (44.5%) |
| **Tip-Specific**<br>Total 1160 loc (18.2%) | Tip Generic Interpreter<br>Tip Values<br>Tip Records<br>Tip Functions<br>273 loc (4.3%) | Tip Analysis Config.<br>Tip Fixpoint Algo.<br>Tip Integer Values<br>Tip Reference Values<br>887 loc (13.9%) |

**Figure 2** Reuse and language-independence of analysis functionality in our relational Tip analysis. Lines of code (loc) were measured with `cloc` (`https://github.com/AlDanial/cloc`). Percentages are computed with respect to the total 6360 loc of the relational Tip analysis.

reports what analysis functionality was reused and what new code needed to be implemented. Furthermore, the diagram shows which analysis functionality is specific to Tip and which is language-independent. *Our evaluation shows that 86.1% of the code of the analysis is either reused as-is or new code that is language-independent. We demonstrate that the code is language-independent by reusing it in our WebAssembly analysis.*

**Generic Interpreter** We now describe in more detail the different parts of the Tip analysis, which parts could be reused, and are language-independent. We implemented the relational analysis for Tip by instantiating an existing generic interpreter for Tip. A *generic interpreter* captures analysis-independent language semantics and can be used to derive different analyses and even a concrete semantics [35]. The generic interpreter for Tip has been used to derive a non-relational interval analysis. Our relational analysis reuses the code of the generic interpreter as-is, because the generic interpreter is parametric in the values, addresses, and effects.

To derive an analysis, the generic interpreter must be instantiated with *analysis components* [33]. Analysis components describe analysis functionality, such as the virtual recency abstraction for call frames or abstract operations on integers. To this end, we distinguish analysis components for values (called *value components*) and analysis components for effects (called *effect components*).

**Value Components** The Tip language has boolean, integer, reference, function, and record values. The abstraction for abstract values joins values of different types to $\top$. Numeric operators support detection of integer overflows. *To summarize, for the implementation of the relational Tip analysis, we reused code for analyzing functions, and record values, and implemented new code for analyzing integer, boolean, and reference values.*

**Effect Components** The Tip language features a call frame of local variables, a heap of dynamically-allocated values, a console for printing and reading user input, and abort execution due to a program failure.

To analyze the call frame and the heap, we refactor the virtual recency abstraction from Section 2:

$$\widehat{\mathsf{CallFrame}}[Ctx, Val] = \mathsf{Map}[\mathsf{Var}, \mathcal{P}(\mathsf{VirtAddr}[Ctx])] \times \mathsf{RecencyStore}[Ctx, Val]$$
$$\widehat{\mathsf{Heap}}[Ctx, Val] = \mathsf{RecencyStore}[Ctx, Val]$$
$$\begin{aligned}\mathsf{RecencyStore}[Ctx, Val] = {} & \mathsf{Map}[\mathsf{VirtAddr}[Ctx], \mathcal{P}(\mathsf{PhysAddr}[Ctx])] \\ & \times \mathsf{Map}[\mathsf{PhysAddr}[Ctx], Val] \times \mathsf{Rel}[\mathsf{PhysAddr}[Ctx], \mathbb{Z} \cup \mathbb{R}]\end{aligned}$$

Specifically, we split the abstract call frame into the address frame and a new abstraction called *recency store*. The recency store contains the address translation and relational abstract domain. The same recency store is also used to analyze the heap, which allows relations to span over local and heap-allocated variables. Furthermore, we added a map to the recency store that holds non-relational values, which is also used to store meta-data of numeric variables such as the their types. When looking up from the call frame, the value stored in the non-relational map is joined with the value in the relational abstract domain, if there is any.

For example, analyzing the Tip program $l_1$ : `x = console.input();` $l_2$ : `y = alloc x` results in the following abstract call frame:

$$\begin{aligned}
addrFrame : {} & [\mathtt{x} \mapsto \{\mathtt{x}_1\}, \mathtt{y} \mapsto \{\mathtt{y}_1\}] \\
addrTrans : {} & [\mathtt{x}_1 \mapsto \{\mathtt{x}_{\mathsf{recent}}\}, \mathtt{y}_1 \mapsto \{\mathtt{y}_{\mathsf{recent}}\}, \mathsf{Input}(l_1)_1 \mapsto \{\mathsf{Input}(l_1)_{\mathsf{recent}}\}, \\
& \;\mathsf{Alloc}(l_2)_1 \mapsto \{\mathsf{Alloc}(l_2)_{\mathsf{recent}}\}] \\
nonRelStore : {} & [\mathtt{x}_{\mathsf{recent}} \mapsto \mathsf{IntVal}(\mathsf{Const}(\bot, \varnothing, \mathsf{Int})), \mathtt{y}_{\mathsf{recent}} \mapsto \mathsf{RefVal}(\{\mathsf{Alloc}(l_2)_1\}), \\
& \;\mathsf{Input}(l_1)_{\mathsf{recent}} \mapsto \mathsf{IntVal}(\mathsf{Const}(\bot, \varnothing, \mathsf{Int})), \mathsf{Alloc}(l_2)_{\mathsf{recent}} \mapsto \mathsf{IntVal}(\mathsf{Const}(\bot, \varnothing, \mathsf{Int}))] \\
relStore : {} & [\mathtt{x}_{\mathsf{recent}} = \mathsf{Input}(l_1), \mathsf{Alloc}(l_2)_{\mathsf{recent}} = \mathtt{x}_{\mathsf{recent}}]
\end{aligned}$$

The relational abstract domain used in the abstract call frame is implemented in the *Apron library* [29]. Apron provides a unified interface for relational abstract domains: changing its underlying domains does not require changes to the high-level analysis. Apron supports various domains such as Polyhedra [15] and Octagons [40].

*To summarize, for the implementation of the relational Tip analysis, we reused code for console input and output, and failure, and implemented new code for the call frame of the virtual recency abstraction and the heap.*

**Fixpoint Algorithm** Our relational Tip analysis reuses a fixpoint algorithm for big-step abstract interpreters that handles both loops and recursive functions [34], demonstrating that the reused fixpoint algorithm also works for relational abstractions. The vast majority of the fixpoint algorithm is language-independent and reused (717 loc). A smaller part is Tip-specific and reused (73 loc) and only 9 loc are not reused.

## 5.2 Relational Analysis for WebAssembly

WebAssembly (Wasm for short) is a modern low-level language for the web that is supported in all major browsers. The language is statically-typed, features an operand stack, a byte-addressable heap, global and local variables, function tables, modules, structured control-flow and exceptions. Furthermore, Wasm 1.0 has four types of values: 32 and 64-bit integers and 32 and 64-bit IEEE754 floating-point numbers.

| **Rel. Wasm Analysis**<br>Total 11389 loc (100%) | **Reused Code**<br>Total 8228 loc (72.2%) | **New Code**<br>Total 3161 loc (27.8%) |
|---|---|---|
| **Language-Independent**<br>Total 7049 loc (61.9%) | Local Variables<br>Operand Stack<br>Function Tables<br>Exceptions & Failures<br>Loops & Recursion<br>Apron Integration<br>Int./Eq./Ord. Ops<br>5914 loc (51.9%) | Global Variables<br>Byte Memory<br>Floating Point Ops.<br>Num. Conversion Ops.<br>1135 loc (10.0%) |
| **Wasm-Specific**<br>Total 4340 loc (38.1%) | Wasm Generic Interpreter<br>Wasm Modules & Funcs.<br>Wasm Values<br>Wasm Exceptions<br>2314 loc (20.3%) | Wasm Analysis Config.<br>Wasm Integer Values<br>Wasm Float Values<br>Wasm Addrs. & Types<br>2026 loc (17.8%) |

**Figure 3** Reuse and language-independence of existing functionality in our relational Wasm analysis. Percentages are computed with respect to the total 11389 loc of the relational Wasm analysis.

We developed a relational analysis for Wasm 1.0 in the Sturdy framework by reusing code of an existing non-relational analysis for Wasm [8] and code developed for the relational Tip analysis. Figure 3 reports what analysis functionality was reused and what new code needed to be implemented. Our relational analysis demonstrates that the Wasm generic interpreter and many existing non-relational analysis modules could be soundly reused as-is. *Our evaluation shows that 82.2% of the code of the analysis is either reused as-is or new code that is language-independent.*

**Value Components** Wasm has values of four types: 32 and 64-bit integers and 32 and 64-bit IEEE 754 floating point numbers. Our Wasm analysis uses the following abstraction for values:

$$\widehat{\mathsf{Val}} = \mathsf{I32Val}(\mathsf{VI32}) \mid \mathsf{I64Val}(\mathsf{VI64}) \mid \mathsf{F32Val}(\mathsf{VF32}) \mid \mathsf{F64Val}(\mathsf{VF64}) \mid \top$$
$$\mathsf{VI32} = \mathsf{RelExpr}[\widehat{\mathsf{Addr}}, \mathsf{Type}] \mid \mathsf{RelBool}[\widehat{\mathsf{Addr}}, \mathsf{Type}]$$
$$\mathsf{VI64} = \mathsf{VF32} = \mathsf{VF64} = \mathsf{RelExpr}[\widehat{\mathsf{Addr}}, \mathsf{Type}]$$
$$\mathsf{Type} = \mathsf{I32Type} \mid \mathsf{I64Type} \mid \mathsf{F32Type} \mid \mathsf{F64Type}$$

The type includes information about the bit-width (32 and 64-bit) as well as how the number is represented in the relational abstract domain (integer, real). The type information is used when detecting integer overflow and floating-point special values.

The Wasm analysis reuses integer operations developed for the Tip analysis by generically lifting operations on $\mathsf{RelExpr}$ to operations on Wasm's $\widehat{\mathsf{Val}}$. $\mathsf{I32Val}$ values can either be a numeric expression ($\mathsf{RelExpr}$) or a boolean expression ($\mathsf{RelBool}$). This is because Wasm encodes boolean `true` as 32-bit integer 1 and `false` as 0. However, to represent the result of a boolean operator more precisely, our analysis returns a relational boolean expression. The

conversion of a boolean expression to a numeric expression is only performed if absolutely necessary, for example, if a boolean expression is applied to a non-boolean numeric operator.

To analyze WASM's floating-point numbers, we developed language-independent operations that approximate the IEEE754 floating-point standard. The floating-point operations are similar to integer operations in that they first create relational expression and then check for floating-point special values that need to be added $(-\infty, -0.0, \mathsf{NaN}, \infty)$.

Additionally, WASM allows converting each of its four numeric types into another other. Conversions to integer types can be signed or unsigned. To analyze these numeric conversions, we implemented 12 conversion operations in 256 loc.

*To summarize, our relational WASM analysis reuses code for relational integer operations, but required new code for floating-point operations and numeric conversion operations.*

**Effect Components** WASM features local and global variables, an operand stack, a byte-addressable heap, function tables, jumps, exceptions, and failures. The WASM analysis implements local variables by reusing the call frame of the virtual recency abstraction. The operand stack, function tables, jumps, exceptions, and failures are implemented by reusing existing analysis components of Sturdy.

To analyze global variables, we developed a new relational abstraction similar to the call frame that uses the RecencyStore, extended so that local variables on the call frame can be referenced by their index within the frame, whereas global variables must be reference by their name.

To analyze the byte-addressable heap, we developed a new relational abstraction inspired by Balakrishnan's abstraction for the memory of x86 executables [2]. Our byte-memory abstraction reads in information of the static memory layout generated by LLVM (if available), including information about global variables, stack, and heap. On stores and loads, the abstract byte-memory determines for the given numeric address to which abstract memory location the address belongs. In case the numeric address cannot be resolved to a unqiue address, the address is assigned to dynamic heap contexts, which may overlap other statically-known memory regions.

To enable relations that span WASM local variables, global variables, operand stack positions, and byte memory locations, we implemented the following context type for virtual and physical:

$$\widehat{\mathsf{Addr}} = \mathsf{VirtAddr}[\mathsf{Ctx}]$$

$$\begin{aligned}\mathsf{Ctx} = {} & \mathsf{Local}(\mathit{callFrameIdx} : \mathsf{Int}, \mathit{fun} : \mathsf{Function}) \mid \mathsf{Global}(\mathit{addr} : \mathsf{Int}) \\ & \mid \mathsf{Stack}(\mathit{stackPos} : \mathsf{Int}, \mathit{programPos} : \mathsf{Label}, \mathit{fun} : \mathsf{Function}) \\ & \mid \mathsf{ByteMemory}(\mathit{memoryAddr} : \mathsf{MemoryAddr}) \mid \mathsf{Other}(\mathit{programPos} : \mathsf{Label}, \mathit{type} : \mathsf{Type})\end{aligned}$$

While relational variables based on Local and Global contexts are proactively allocated and added to the relational abstract domain, the other contexts are only allocated if necessary in joins of relational expressions. For example, context Stack is only allocated at a join point where the joined stacks contain different relational expressions at the same position that cannot be joined any other way. In all other places relational variables based on context Other are allocated, for example, for when joining expressions in the analysis of a non-linear integer expression like max.

*To summarize, our relational WASM analysis reuses effect components for local variables, the operand stack, function tables, jumps, exceptions, and failures. For better precision, we implemented new effect components for global variables and the byte-addressable heap.*

# 6 Evaluation

In this section, we assess the praticality of the virtual recency abstraction by empirically evaluating our relational analysis for WebAssembly that uses it. In this evaluation, we answer the following research questions about our relational Wasm analysis.

- **RQ1**: Does the relational Wasm analysis yield sound analysis results?
- **RQ2**: Does the relational Wasm analysis scale to realistic programs?
- **RQ3**: What is the precision of the relational Wasm analysis?

## 6.1 Soundness of the Relational Wasm Analysis

While we already proved the soundness of our approach in Section 4, we here empirically evaluate the soundness of our relational Wasm analysis to rule out implementation bugs. To this end, we measure the recall on the Wasm 1.0 specification test-suite.[3] The test-suite is typically used to evaluate the conformance of concrete Wasm interpreters and tests all primitive Wasm instructions with all of their edge-cases. The test-suite consists of 71 individual test files, from which we execute 16517 `assert_return` and `assert_trap` assertions (we skipped parser tests, type checker tests, etc). For each assertion, we check if the analysis result soundly overapproximates the concrete interpretation result.

**Table 1** Recall of our Wasm analysis measured on the Wasm 1.0 specification test-suite

| Category | Assertions | Polyhedra | Octagons | Intervals |
|---|---|---|---|---|
| Control Flow | 832 | 100.00% | 40.26% | 100.00% |
| Conversions | 593 | 100.00% | 100.00% | 100.00% |
| Floats | 12197 | 100.00% | 100.00% | 100.00% |
| Functions | 339 | 100.00% | 50.74% | 100.00% |
| Global Vars. | 48 | 100.00% | 0.00% | 100.00% |
| Integers | 909 | 100.00% | 100.00% | 100.00% |
| Local Vars. | 93 | 100.00% | 56.99% | 100.00% |
| Memory | 757 | 100.00% | 93.13% | 100.00% |
| Modules | 606 | 100.00% | 83.99% | 100.00% |
| Stack | 143 | 100.00% | 34.27% | 100.00% |
| Total | 16517 | 100.00% | 94.01% | 100% |

We repeated the experiment for Apron's implementation of Polyhedra [15] (`Polka`), Octagons [40] (`Octagon`) and Intervals [11] (`Box`). The results are shown in Table 1. For Polyhedra and Octagons, there are 4 assertions in `float_exprs.wast` that run out of memory, which we count as a top analysis result, which is sound. For Octagons, 989 (6.0%) of assertions were approximated unsoundly. After investigating the issue, we found that all unsound approximations stem from the same Wasm code that is shared between the tests. The code assigns values to multiple local floating-point variables at once, which triggers a bug in the underlying Apron library. We have reported this bug to the Apron developers. This bug did not occur in any of our other experiments, which use different Wasm programs.

*RQ1: The relational Wasm analyses yield sound results for Polyhedra and Intervals, but revealed a soundness bug in Apron for Octagons. We conclude that our approach enables sound relational analyses for Wasm.*

[3] `https://github.com/WebAssembly/spec/tree/main/test/core`

## 6.2 Scalability of the Relational Wasm Analysis

The scalability of relational analyses is a general concern due to the higher computational complexity of relational abstract domains. We inherit this complexity in our relational WASM analysis, yet want to demonstrate that it still scales to realistic WASM programs. To this end, we evaluate the running time of the relational WASM analysis on 7 compiled C programs from the *benchmarks game suite*[20]. From the benchmark suite of 9 programs, we excluded `k-nucleotide` and `pidigits` because these contain arrays of structs, which our WASM analysis does not currently support. The 7 remaining programs have 546 lines `Wat` code on average (Figure 4).

We compiled the C programs with `clang-19 -target=wasm32 -O3 -g` to WASM and linked our own C standard library stub for which we implemented summaries within our analysis. We ran the experiment on Ubuntu 24.04 with OpenJDK 21.0.9 (100GB heap space, 1GB stack space) on a machine with an AMD Ryzen Threadripper PRO 5975WX CPU. For each analyzed program, we warmed up the JVM with 3 runs of the analysis and then collected the running times over 5 runs with 3 seconds breaks for garbage collection.

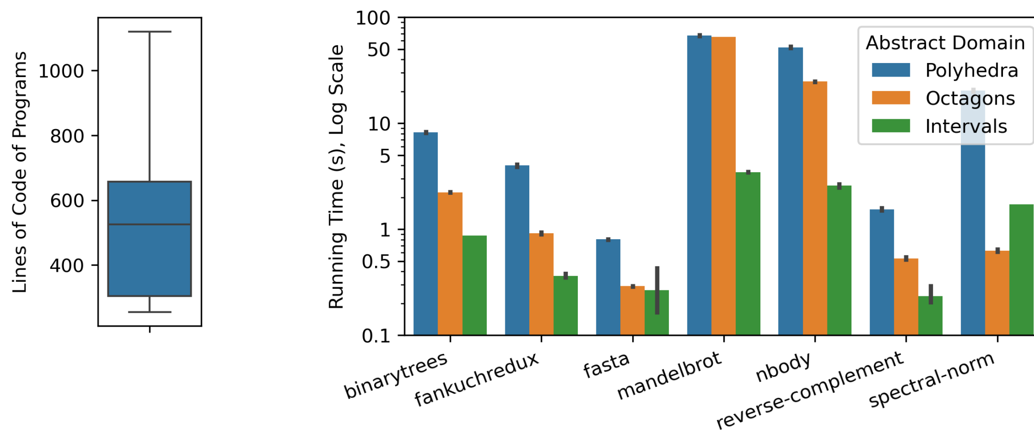



**Figure 4** Lines of `Wat` code of analyzed programs measured with `cloc` (left) and average running times of the relational WASM analysis per program (right).

The average running time per program are shown in Figure 4. As to be expected, Polyhedra are slowest as their worst-case execution time and space complexity is exponential [15]. Octagons are on average 7x faster and Intervals 11.6x faster than Polyhedra. The execution time of the relational analyses is heavily influenced by the number of local variables in the code, which in turn depends on the compiler optimization level: at higher optimization levels, the compiler moves more program data from the bytememory to local variables. For example, `mandelbrot` has 49 local variables and `nbody` has 67 local variables. Since the number of variables affects the number of dimensions of the relational abstract domain, these programs have the highest analysis times.

*RQ2: The relational WASM analyses scale to realistic WASM programs, compiled from C source code.*

## 6.3 Precision of the Relational Wasm Analysis

To enable the relational analysis of recursive functions, we introduced the virtual recency abstraction, which uses a single relational variable $\mathtt{x}_{\mathsf{old}}$ for all previous values of a variable `x`. This can lead to precision loss compared to a non-relational analysis. For example, consider the following code that increments variable `x` three times:

```
(func $plus_three (local $x i32)
  (local.set $x (call $itv (i32.const 0) (i32.const 10))) ;; x_recent ∈ [0, 10]
  (local.set $x (i32.add (local.get $x) (i32.const 1)))   ;; x_recent = x_old + 1, x_old ∈ [0, 10]
  (local.set $x (i32.add (local.get $x) (i32.const 1)))   ;; x_recent = x_old + 1, x_old ∈ [0, 11]
  (local.set $x (i32.add (local.get $x) (i32.const 1)))   ;; x_recent = x_old + 1, x_old ∈ [0, 12]
  (call $assert (i32.le_s (i32.const 3) (local.get $x)))) ;; Fails because x_recent ∈ [0, 13]
```

A non-relational interval analysis assigns $[0, 10] + [1, 1] + [1, 1] + [1, 1] = [3, 13]$ to the final value of `x` and is able to prove both assertions. In contrast, the virtual recency abstraction assigns $x_{recent} = x_{old} + 1$ with $x_{old} \in [0, 12]$, which fails the first assertion. This begs the question: *Is the precision loss of the virtual recency abstraction so severe that it negates the precision advantage of relational abstract domains?*

To answer this question, we measure the precision of our relational WASM analysis in Sturdy and compare it to state-of-the-art C analyzers *Goblint 2.7.1* [52] and *Mopsa 1.1* [32]. For this experiment, we created a benchmark of 35 precision tests, which we ported to WASM and C to be able to compare the analysis results. Of the 35 tests, 9 tests were taken from existing literature [41, 42, 39, 17] and 26 tests we implemented ourselves. The benchmark consists of 13 tests of common mathematical operations (linear arithmetic, maximum, absolut value, sine, etc.), 11 tests with variable reassignment of which 8 tests include loops, and 11 tests of common recursive functions (recursive identity, factorial, fibonacci, gaussian sum, even odd). Each test, like `plus_three` shown above, consists of a sequence of instructions with one or more assertions of numerical properties. The complete code of the benchmarks can be found in the supplementary material.

**Table 2** Precision advantage of relational abstract domains within the virtual recency abstraction

| | | Sturdy (Wasm) | | | Goblint (C) | | | Mopsa (C) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Tests | Poly. | Oct. | Itv. | Poly. | Oct. | Itv. | Poly. | Oct. | Itv. |
| Literature | 9 | 44% | 33% | 0% | 67% | 67% | 11% | 56% | 33% | 0% |
| Own Non-Rec. | 16 | 88% | 81% | 31% | 81% | 75% | 38% | 63% | 56% | 19% |
| Own Recursive | 10 | 40% | 60% | 60% | 20% | 30% | 20% | 0% | 0% | 0% |
| Total | 35 | 63% | 63% | 31% | 60% | 60% | 26% | 43% | 34% | 9% |

We executed Goblint[4] and MOPSA[5] with precise analysis configuration options and checked the console output if assertions could not be proven. Furthermore, we disabled sound overflow wrap-around in our WASM analysis for better comparability to C's signed integers, for which overflow is undefined behavior. The results of the experiment are shown in Table 2. While the virtual recency abstraction looses some precision, especially on the tests taken from literature, the relational abstractions polyhedra and octagons retain their precision advantage over non-relational intervals. Furthermore, on the benchmark the virtual recency abstraction analyzed recursive functions more precisely in contrast to Goblint which analyzes recursive function partially context-sensitively [21] and MOPSA which finitely inlines recursive functions.

*RQ3: While we may loose some precision due to the virtual recency abstraction, our relational WASM analyses were able to analyze recursive programs and have a precision advantage over a non-relational interval analyses.*

[4] `goblint --conf=examples/very_precise.json --set ana.context.widen true` (Polyhedra, Octagons, Intervals)

[5] `mopsa-c -config=c/cell-pack-rel-itv.json` (Polyhedra, Octagons) and `mopsa-c -config=c/cell-itv.json` (Intervals)

## 7 Related Work

**Recency Abstraction.** Balakrishnan and Reps [3] introduced the recency abstraction, which allows precise analyses of object initialized in loops by discriminating the last allocated object (called "recent") from others (called "old"). This is a highly popular allocation abstraction which has been extensively used and studied [30, 37, 46, 44]. However, whenever a new object is allocated, this abstraction needs to retire the previously recent object with the other old ones. We have shown in Section 2.3 that this retirement operation is non-modular: every abstract domain needs to support this retirement operation, breaking standard interfaces. One of the core contributions of this paper is to introduce a virtual recency abstraction, which is modular thanks to a new layer of virtual addresses, drawing from operating system design [19]. Additionally, we rely on the virtual recency abstraction to allocate variables: our analysis can refer to the values of a program variable at a previous program point, or in a different callstack. An in-depth study of variations in allocation strategies and their effect on performance and precision of the analysis is left as future work.

**Relational analysis of languages without an operand stack (C, Python, ...).** The analysis frameworks Astrée [14], Verasco [31], Mopsa [32], and Goblint [52, 47] implement relational analyses for C and other languages without an operand stack. Relational analyses for such languages can, under certain restrictions, strongly update local variables without running into a soundness problem. Strong updates to local variables is only sound if nothing in these analyses refers to an old state of local variables and thus updating local variables cannot be observed elsewhere. The exception to this are recursive functions where a strong update to a local variable can change the value in a recursive parent call frame. To solve this problem, fully context-sensitive analyses inline recursive calls up to a certain depth and tag variables in different call frames with the call string. This way each recursive call frame has its own fresh set of variables and strong updates never affect local variables in parent call frames. However, adding new variables to a relational abstract domain can be costly, as for example the time and memory complexity for Polyhedra grows exponentially with the number of variables. Furthermore, full context-sensitivity only works for recursive functions that can be fully explored by recursive unrolling up to a certain depth, but fails for example to analyze the recursive factorial function $\mathsf{fact}(n)$ for $0 \leq n \leq \infty$, which would require infinite unrolling. More recent work extends Goblint to handle recursive functions partially context-sensitively by annotating local variables with a context [21]. However, this can create scalability problems if too many contexts are distinguished.

In comparison, the virtual recency abstraction analyzes recursive functions while reusing the same set of variables. Specifically, variable updates in a recursive call retire virtual addresses in a parent call frame, ensuring that their value does not change non-monotonically. Furthermore, our relational analyses explore recursive functions fully by widening the analysis state at recursive calls with the state at parent calls. This ensures that every recursive call chain either terminates naturally or has a recurrent recursive call. In case of a recurrent recursive call, the fixpoint algorithm avoids recursing deeper by iterating between both occurences of the recurrent recursive call [34].

**Relational analyses for languages with an operand stack (assembly, bytecode, ...).** Many relational analyses for languages with an operand stack circumvent the problem of unsound strong updates by first translating to an intermediate language without a stack. Fähndrich and Logozzo [22] describe a relational analysis for .NET's bytecode representation CIL. They eliminate the operand stack before the analysis is executed, but they do not describe in

```
x = rand(0,10);
x = x + 1;
x = x + 1;
x = x + 1;
```

**(a)** Program

```
x1 = rand(0,10);
x2 = x1 + 1;
x3 = x2 + 1;
x4 = x3 + 1;
```

**(b)** SSA translation

SSA : $[\mathtt{x}_1 \in [0,10], \mathtt{x}_2 = \mathtt{x}_1 + 1, \mathtt{x}_3 = \mathtt{x}_2 + 1 \ldots]$
Virtual Recency Abstraction :
$[\mathtt{x} \mapsto \{x_4\}], [\mathtt{x}_4 \mapsto \{\mathsf{x}_{\mathsf{recent}}\}, \mathtt{x}_{3,2,1} \mapsto \{\mathsf{x}_{\mathsf{old}}\}],$
$[0 \leq \mathsf{x}_{\mathsf{old}} \leq 12, x_{\mathsf{recent}} = x_{\mathsf{old}} + 1]$

**(c)** Analysis states at the end of the program

**Figure 5** Comparison between relational analyses for SSA languages and the virtual recency abstraction.

detail what language they translate CIL programs into. Ballabriga et al. [4] developed a relational analysis for ARM assembly. As the first step ARM's stack is eliminated by translating it to a three-address code called `MEMP`, which is then analyzed. A problem is that the translation to three-address code introduces new variables for values that previously where stored on the stack. Their analysis adds all these variables to the relational abstract domain, which increases its size and could lead to scalibility problems. Bau et al. [5] analyze a stack-based smart contract language called Michelson. Instead of translating Michelson to an intermediate language without a stack, they analyze Michelson directly. Specifically, they introduce new variables into the relational abstract domain that represent a value at a positions on the stack at a position in the program. This solves the problem of unsound strong updates, but at the cost of introducing many new variables into the relational abstract domain.

In comparison, our relational analysis for Wasm analyzes the language directly without a preprocessing step. Our analysis approximates the operand stack with a list of abstract values, which are symbolic expressions over local variables. This avoids introducing new variables into the relational abstract domain for each stack position and program location. New variables are only introduced if necessary at control-flow joins in the program when the two stacks contains different symbolic expressions that cannot be joined any other way. Furthermore, our Wasm analysis solves the problem of unsound strong updates by using the virtual recency abstraction. The virtual recency abstraction ensures that symbolic expressions on the operand stack do not change its value non-monotonically in case of strong updates.

**Relational analysis of languages with static single assignment (LLVM, ...)** Languages with static single assignment (SSA) assign variables only once [16]. SSA partly solves the unsoundness problem of strong updates for relational analyses, because variables are never reassigned and thus their value does not change non-monotonically during analysis. For example, the SeaHorn verification framework [24, 9] implements an abstract interpreter Crab that analyzes CrabIR, an intermediate representation for LLVM [25]. However, SSA still does not solve the problem for recursive functions because recursive calls reuse the same set of local variables. Furthermore, the translation from a high-level language to SSA creates many new variables, that when added to the relational abstract domain, blow up its size and could cause scalibility issues.

Figure 5 compares relational analyses for an SSA language to the virtual recency abstraction at the example of a small C program. The program translated to SSA uses a fresh variable for each reassignment of `x`. The relational analysis of the SSA program adds each variable to the relational abstract domain and thus remembers each intermediate value of `x`. The interval of $\mathtt{x}_4$ after the last assignment is $[3, 13]$. In comparison, the virtual recency abstraction retires recent virtual addresses of `x` on each reassignment. This is less precise than the SSA analysis: the interval of $\mathsf{x}_{\mathsf{recent}}$ after the last assignment is $[1, 13]$. However, the virtual recency abstraction only requires 2 relational variables instead of 4 variables. In the

future, we want to investigate the precision and performance tradeoff of the virtual recency abstraction more thoroughly.

**Modular implementation of static analyses.** Prior works have modularized different aspects of static analyses:

**Generic Interpreters** A generic interpreter captures the semantics of the analyzed language without referring to analysis-specific details [50, 35, 6, 38, 31]. The generic interpreter can be instantiated to derive both concrete and abstract interpreters.

**Reusable Analysis Modules** Analyses are implemented modularly by reusing existing language-independent analysis components [29, 18, 33, 32, 27, 28, 45].

**Abstract Domain Combinators** Abstract domains can be modularly composed with domain combinators to improve their precision, e.g., *reduced products* [13] or *communication channels* [14, 31].

**Fixpoint modularity** Fixpoint algorithms can be modularized by different iteration strategies [7], by composing them from language-independent fixpoint combinators [34], or by relying on an external solver [49].

Our relational analyses in Section 5 combine all these types of modularization to achieve a high level of reuse and improve the maintainability of the analyses. To this end, we build the analyses within the Sturdy framework [35, 33, 34]. So far, Sturdy only supported non-relational numerical abstract domains and we needed to implement infrastructure to support relational abstract domains. Furthermore, our analyses reuse existing generic interpreters for Tip and Wasm [8] that up until were only instantiated with non-relational abstract domains. Finally, our analyses reuse existing language-specific fixpoint algorithms that have been modularly composed of language-independent fixpoint combinators [34].

## 8 Conclusion

This work shows how to integrate relational abstract domain within interpreter modular analyses platforms. We tackled the issue of shared state between abstract domains through the introduction of the virtual recency abstraction. We have shown our approach to be sound and terminating. We have implemented our approach within an existing modular framework without needing to change interfaces. This approach has permitted us to develop the first relational analysis for WebAssembly. In our evaluation, we find that the Wasm has a high recall of 100% for Polyhedra, 94% for Octagons, and 100% for Intervals. Furthermore, in our performance experiments, the Wasm analysis has average running times of 22s for Polyhedra, 13s for Octagons, and 1.3s for Intervals. Lastly, on the precision tests our Wasm analysis achieves 63% precision for Polyhedra, 57% for Octagons, and 29% for Intervals. These expriments demonstrate the practical applicability of the virtual recency abstraction.

## References

1 Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *13th Australian Software Engineering Conference (ASWEC 2001), 26-28 August 2001, Canberra, Australia*, pages 68–75. IEEE Computer Society, 2001. doi:10.1109/ASWEC.2001.948499.

2 Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004. doi:10.1007/978-3-540-24723-4_2.

3 Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006. doi:10.1007/11823230_15.

4 Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. Static analysis of binary code with memory indirections using polyhedra. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 114–135. Springer, 2019. doi:10.1007/978-3-030-11245-5_6.

5 Guillaume Bau, Antoine Miné, Vincent Botbol, and Mehdi Bouaziz. Abstract interpretation of michelson smart-contracts. In Laure Gonnord and Laura Titolo, editors, *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, pages 36–43. ACM, 2022. doi:10.1145/3520313.3534660.

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

7 François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993. URL: https://doi.org/10.1007/BFb0039704, doi:10.1007/BFB0039704.

8 Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. Modular abstract definitional interpreters for webassembly. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 5:1–5:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.5, doi:10.4230/LIPICS.ECOOP.2023.5.

9 Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, volume 8702 of *Lecture Notes in Computer Science*, pages 271–277. Springer, 2014. doi:10.1007/978-3-319-10431-7_20.

10 Patrick Cousot. Abstracting induction by extrapolation and interpolation. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 19–42. Springer, 2015. doi:10.1007/978-3-662-46081-8_2.

11 Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*, pages 106–130. Dunod, 1976.

12 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.

14 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer.

In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006. doi:10.1007/978-3-540-77505-8_23.

15 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978. doi:10.1145/512760.512770.

16 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi:10.1145/115372.115320.

17 Matthias Dangl, Stefan Löwe, and Philipp Wendler. Cpachecker with support for recursive programs and floating-point arithmetic - (competition contribution). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 423–425. Springer, 2015. doi:10.1007/978-3-662-46681-0_34.

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

19 Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970. doi:10.1145/356571.356573.

20 Isaac Gouy Dough Bagley, Brent Fulgham. The computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/, 2004–2026. Accessed: 2026-04-23.

21 Julian Erhard, Johanna Franziska Schinabeck, Michael Schwarz, and Helmut Seidl. When to stop going down the rabbit hole: Taming context-sensitivity on the fly. In Raphaël Monat and Cindy Rubio-González, editors, *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2024, Copenhagen, Denmark, 25 June 2024*, pages 35–44. ACM, 2024. doi:10.1145/3652588.3663321.

22 Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010. doi:10.1007/978-3-642-18070-5_2.

23 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2004. doi:10.1007/978-3-540-24730-2_38.

24 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015. doi:10.1007/978-3-319-21690-4_20.

**25** Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based memory model. In Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina, editors, *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, volume 13124 of *Lecture Notes in Computer Science*, pages 122–144. Springer, 2021. doi:10.1007/978-3-030-95561-8_8.

**26** Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.

**27** Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in OPAL. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 184–196. ACM, 2020. doi:10.1145/3368089.3409765.

**28** Dominik Helm, Tobias Roth, Sven Keidel, Michael Reif, and Mira Mezini. Unimocg: Modular call-graph algorithms for consistent handling of language features. In Maria Christakis and Michael Pradel, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 51–62. ACM, 2024. doi:10.1145/3650212.3652109.

**29** Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009. doi:10.1007/978-3-642-02658-4_52.

**30** Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. doi:10.1007/978-3-642-03237-0_17.

**31** Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

**32** Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2019. doi:10.1007/978-3-030-41600-3_1.

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

**34** Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. Combinator-based fixpoint algorithms for big-step abstract interpreters. *Proc. ACM Program. Lang.*, 7(ICFP):955–981, 2023. doi:10.1145/3607863.

**35** Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.*, 2(ICFP):72:1–72:26, 2018. doi:10.1145/3236767.

**36** Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection: literature review and empirical study. In Sebastián Uchitel, Alessandro Orso,

and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 507–518. IEEE / ACM, 2017. doi:10.1109/ICSE.2017.53.

**37** Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 411–427. ACM, 2010. doi:10.1145/1869459.1869494.

**38** Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. Abstract interpreters: A monadic approach to modular verification. *Proc. ACM Program. Lang.*, 8(ICFP):602–629, 2024. doi:10.1145/3674646.

**39** Antoine Miné. *Domaines numériques abstraits faiblement relationnels.* PhD thesis, Palaiseau, Ecole polytechnique, 2004.

**40** Antoine Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19(1):31–100, 2006. URL: https://doi.org/10.1007/s10990-006-8609-1, doi:10.1007/S10990-006-8609-1.

**41** Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Found. Trends Program. Lang.*, 4(3-4):120–372, 2017. doi:10.1561/2500000034.

**42** Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

**43** Raphaël Monat. *Static type and value analysis by abstract interpretation of Python programs with native C libraries. (Analyse statique, de type et de valeur, par interprétation abstraite, de programmes Python utilisant des librairies C).* PhD thesis, Sorbonne University, Paris, France, 2021. URL: https://tel.archives-ouvertes.fr/tel-03533030.

**44** Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Value and allocation sensitivity in static python analyses. In Paddy Krishnan and Christoph Reichenbach, editors, *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP@PLDI 2020, London, UK, June 15, 2020*, pages 8–13. ACM, 2020. doi:10.1145/3394451.3397205.

**45** Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of python programs with native C extensions. In Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi, editors, *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*, volume 12913 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2021. doi:10.1007/978-3-030-88806-0_16.

**46** Jihyeok Park, Xavier Rival, and Sukyoung Ryu. Revisiting recency abstraction for javascript: towards an intuitive, compositional, and efficient heap abstraction. In Karim Ali and Cristina Cifuentes, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 1–6. ACM, 2017. doi:10.1145/3088515.3088516.

**47** Simmo Saan, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. Goblint: Thread-modular abstract interpretation using side-effecting constraints - (competition contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 438–442. Springer, 2021. doi:10.1007/978-3-030-72013-1_28.

**48** Joanna C. S. Santos, Reese A. Jones, and Mehdi Mirakhorli. Salsa: static analysis of serialization features. In Wytse Oortwijn, editor, *FTfJP 2020: Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs, Virtual Event, USA, July 23, 2020*, pages 18–25. ACM, 2020. doi:10.1145/3427761.3428343.

**49** Helmut Seidl and Ralf Vogler. Three improvements to the top-down solver. *Math. Struct. Comput. Sci.*, 31(9):1090–1134, 2021. doi:10.1017/S0960129521000499.

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

51 Olin Grigsby Shivers. *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University, 1991.

52 Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the goblint approach. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 391–402. ACM, 2016. doi:10.1145/2970276.2970337.