Value-Sensitive Analysis for Looping Array Code

ANONYMOUS AUTHOR(S)

Imperative user-schedulable languages (USLs) enable performance optimization through explicit scheduling transformations, but their reliance on dependency analysis cannot identify optimizations requiring value-sensitive reasoning, such as redundant write elimination or sliding window transformations. We present a value-sensitive analysis framework for imperative USLs that tracks program values through abstract interpretation. Our key contribution is a novel cylindrical algebraic decomposition (CAD) tree domain paired with an efficient flood-fill widening operator. The CAD tree domain systematically partitions the iteration space to discover semantic values absent from source code, enabling precise reasoning about looping array code including complex patterns like array reverse that previous abstract interpretation-based array content analyses could not handle precisely. We implement three new scheduling operators, namely delete, insert, and bind, that leverage abstract value information for safe program optimizations. Evaluation on microbenchmarks demonstrates that our analysis precisely handles complex loop and array operations including multi-dimensional arrays, sliding windows, and reverse access patterns, and scales more effectively than prior work. In a case study optimizing RVV kernels, our generated code matches expert-written intrinsics performance with only modest analysis overhead during compilation, while enabling optimizations beyond the scope of existing USLs.

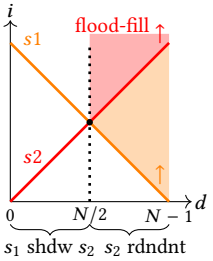
1 INTRODUCTION

User-schedulable languages (USLs) enable performance optimization by separating algorithm specification ("what to compute" as a program) from scheduling ("how to compute" the program), allowing rapid exploration of optimizations for locality, parallelism, and recomputation without modifying the core computation logic. While classic USLs like Halide [34, 35] and TVM [7] use functional programming paradigms, recent imperative USLs like Exo [23, 24] and Allo [6] better align with hardware execution models through explicit loads, stores, and control flow, offering more intuitive reasoning for performance engineers.

Unlike functional USLs, where computations are expressed as pure functions of their inputs, imperative USLs require reasoning about the safety of scheduling operations in the presence of state mutation and side effects, which poses unique challenges in program analysis. Traditional dependency analysis treats statements as opaque operations and justifies rewrites by proving that statements are independent–that they access disjoint sets of memory locations. However, some program optimizations can only be justified by examining the actual values being computed. We call this "value-sensitive analysis," which reasons about the specific values flowing through the program rather than just their dependencies, enabling us to identify when certain statements become redundant or when new computations can be inserted without changing program semantics.

To motivate value-sensitive analysis for performance optimization, consider the loop shown on the right. Each element of x is written twice with the same value, making it safe to remove either statement. However, traditional dead code analysis cannot identify this optimization opportunity. If we attempt to remove $s2$, we encounter a problem: on iteration 0, the effect $x[0] = y[0]$ is overwritten by $s1$ on the final iteration, which traditional analysis can detect. But on the final iteration $N-1$, the execution $x[N-1] = y[N-1]$ is not overwritten by $s1$ at all—instead, it is *redundant* because it writes the same value that already exists at that memory location. Traditional dead code analysis fails here because it does not consider whether the value being written is *the same as the right before* the statement executes. Since redundancy provides an equally valid justification for statement removal, $s2$ can be safely removed, but the reasoning differs across loop iterations.

```
   for i in 0..N:
s1:  x[N-1-i] = y[N-1-i]
s2:  x[i] = y[i]          ✓
```



Consider the space-time diagram on the right, depicting the contents of array x at location d (horizontal axis) after iteration i (vertical axis) completes. Statements $s1$ and $s2$ appear as diagonal lines marking the points in space and time where they modify array x. This diagram shows that the statement which writes the final (vertically highest) value depends on array position: in the first half of the array, we are justified in deleting $s2$ because it is overwritten, or *shadowed*, by $s1$ (shown as s_1 shadow (shdw) s_2). In the second half, this justification is no longer sound. Instead, we must verify that values written by $s1$ reach $s2$ uninterrupted and that $s2$ writes identical values (shown as s_2 redundant (rdndnt)). This requires value-sensitive analysis that tracks x across both space and time. Existing array content analyses using abstract interpretation [11, 12, 15, 30] cannot discover that partitioning should occur at index $N/2$, since this value is absent from the source code, leading to imprecision.

We introduce a *cylindrical algebraic decomposition (CAD) tree* domain and *flood-fill* widening to address this issue. The CAD tree domain decomposes the array's iterator-offset space (i, d) into sign-invariant cells and enables us to systematically find partitions for values that only appear semantically in the code (such as $\lfloor N/2 \rfloor$), as CAD's projection phase finds the resultant of input polynomials (namely $2d-N+1$ in this case) from which it trivially obtains the partition. Furthermore, the flood-fill widening allows us to precisely capture values flowing through the loop by propagating stored values upwards in time in the iterator-offset space (shown by the upward arrow in the inset figure). This allows us to track x in space to see if $s1$ reaches $s2$ uninterrupted and with what value.

Our system, Prexo, builds on top of Exo [23] and combines the obtained abstract value information with the scheduling transformations to enable value-sensitive analysis for imperative USLs. We propose a new analysis API that constructs safety check queries using abstract values, which is further used in the implementation of three scheduling operations that enable safe program transformations for inserting, deleting, and binding mutable variable stores.

We evaluate Prexo from two perspectives. First, we measure the expressiveness and runtime performance of the CAD tree domain design and flood-fill widening on targeted microbenchmarks. We demonstrate that our analysis is both more expressive and scales better than a previous abstract interpretation approach for array analysis [15]. Second, we evaluate Prexo by optimizing kernels on the RVV architecture to assess its practical utility and expressiveness in performance engineering applications. RVV's vector length agnostic instructions pose challenges to compilers due to the global "vector length" (*vl*) state, which modifies the vector length behaviour of executed instructions. Naive dependency analysis cannot safely reason about rewriting *vl*, which motivates our value-sensitive analysis approach. We achieve performance comparable to expert-written DAXPY code with modest analysis runtime overhead, with end-to-end compilation completing in under 3 seconds.

2 MOTIVATIONAL EXAMPLES

In many performance-critical scenarios, transformations cannot be handled by dependency analysis. We illustrate this with two concrete examples.

2.1 Example 1: Sliding Window Optimization

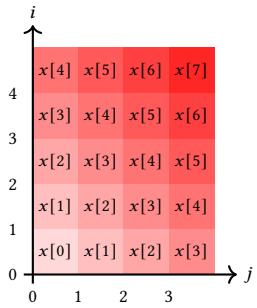
We present code examples using Exo IR rather than Prexo IR for readability. Although our analysis operates on Prexo IR (defined in Section 4), its SSA-form makes it verbose for illustrative purposes. As shown in Section 3, Prexo takes Exo IR as input and converts it to Prexo IR for analysis. In Exo IR, both procedure arguments and variable declarations follow the syntax $\langle name \rangle : \langle type \rangle [\langle size \rangle]$. In `foo` (Figure 1a), `f32` is a numeric type for 32-bit floating point. The $\langle size \rangle$s may be constants, or refer dependently to other arguments. For instance, in `foo`, arrays can use dependent sizes like `[n + m]` that reference the procedure's dimension parameters. Sequential loops are written as `for i in seq(0, m)`, iterating from 0 to $m - 1$ (inclusive).

```
def foo(n: size, m: size, a: f32,
        res: f32, inp: f32[n + m]):
    x: f32[n + m]
    for i in seq(0, n):
        for j in seq(0, m):
            x[i + j] = inp[i + j] ←
            res += x[i + j] * a
```

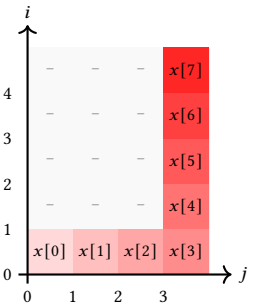
(a) Original code for `foo`

```
def foo(...):
    x: f32[n + m]
    for i in seq(0, n):
        for j in seq(0, m):
            if i == 0 or j == m - 1:
                x[i + j] = inp[i + j] ←
            res += x[i + j] * a
```

(b) Optimized code for `foo`



(c) Iteration space for $x[i+j] = inp[i+j]$ of (a)



(d) Iteration space for $x[i+j] = inp[i+j]$ of (b)

Fig. 1. Code snippets for `foo` illustrating sliding window optimization from (a) to (b). (c) and (d) shows the iteration space when $n = 5$ and $m = 4$.

Function `foo` in Figure 1a takes two size parameters `n` and `m`, scalar values `a` and `res`, and an input array `inp` of size `n + m`. It creates a temporary buffer `x` of the same size and copies the input array through nested loops that iterate over indices `i` from 0 to `n-1` and `j` from 0 to `m-1`, while accumulating `x[i+j]*a` to `res`. While the temporary buffer `x` serves no functional purpose in this example since it merely duplicates the input array, it demonstrates a common optimization pattern for utilizing vector registers or accelerator scratchpads.

Figure 1c shows the iteration space of the load statement $x[i+j] = inp[i+j]$, where array elements with identical offsets are highlighted in matching red tones. This redundancy occurs when multiple (i, j) pairs produce the same index $i + j$, such as $(0, 1)$ and $(1, 0)$ both yielding $i + j = 1$. Writing identical values to the same location wastes computational resources, making it essential to remove redundant memory operations for optimal performance. To achieve this, the original code can be transformed into the new code shown in Figure 1b, which includes a guard `if i == 0 or j == m - 1` that shields the code from redundant loads. The iteration space of the transformed code, shown in Figure 1d, demonstrates this removal, with values from 0 to $n + m - 2$ being loaded only once.

This pattern is common in many high-performance kernels, such as convolution, due to overlapping regions in the updated buffer. However, this poses a challenge for program analysis, as it requires safely reasoning about the removal of redundant writes when the newly written value matches previously computed value. Symbolically, this can be represented as:

$$x[\text{idx}] = v_1; \quad x[\text{idx}] = v_2 \;\rightsquigarrow\; \begin{cases} x[\text{idx}] = v_1, & \text{if } v_1 = v_2, \\ x[\text{idx}] = v_1;\ x[\text{idx}] = v_2, & \text{otherwise.} \end{cases}$$

This type of transformation depends not only on iteration space properties but also on data values. Therefore, dependency analysis, which relies on iteration-domain and access-pattern analysis, fails to directly capture these opportunities. Value-sensitive analysis is, therefore, essential in cases where eliminating or merging redundant writes can significantly optimize the program runtime.

2.2 Example 2: Loop Fusion and Redundant Writes

```
x : f32[n+1]
for i in seq(0, n):
  x[i] = y[i]
for i in seq(0, n):
  x[i+1] = y[i+1]
```

(a) Unfused

```
x : f32[n+1]
for i in seq(0, n):
  x[i] = y[i]
  x[i+1] = y[i+1]
```

(b) Fused

```
x : f32[n+1]
for i in seq(0, n+1):
  x[i] = y[i]
```

(c) Redundant write removed

Fig. 2. Code snippets illustrating loop optimization from (a) to (b) to (c).

Consider a second example in Figure 2. The initial code in Figure 2a has two separate loops: the first copies elements from `y` to `x` for indices 0 through `n-1`, while the second copies elements for indices 1 through `n`. This results in redundant writes to `x[i]` for $i \in [1, n-1]$. Figures 2b and 2c show a loop optimization sequence. First, loop fusion transforms (a) to (b) by combining both loops into a single loop body, eliminating the overhead of two separate loop structures. However, this intermediate form (b) still contains redundant memory operations. Memory location of `x[i+1]` has already been written by `x[i]` in the previous iteration with the same value for $i \in [0, n-1]$. The second optimization step transforms (b) to (c) by recognizing this pattern and extending the loop bounds to `seq(0, n+1)` with a single write operation, eliminating all redundant stores.

The safety of both transformations hinges on value-sensitive analysis. Traditional dependency-based approaches, such as those in Exo for loop fusion, would reject both transformations. Standard fusion analysis requires that loop effects commute, but the two loops in Figure 2a write to overlapping memory regions (`x[1]` through `x[n-1]`), making their effects non-commutative. The fundamental limitation is that dependency analysis only tracks *which* locations are accessed, not *what* values are written. Value-sensitive analysis addresses this limitation by recognizing that both loops copy identical values from `y[i]` to `x[i]` within the overlapping region.

3 SYSTEM OVERVIEW

As shown in Figure 3, value-sensitive analysis is embedded within Exo [23]'s compilation and optimization stack, which provides type and bound checking, scheduling operations, and a backend for memory and precision checks as well as code generation. While these components are not our contributions, they provide the foundation for Prexo. Our value-sensitive analysis operates exclusively on the Prexo IR, and Exo IR is converted to Prexo IR as shown in Figures 3 and 4. For the syntax of Exo IR, we refer readers to the original Exo paper [23], as its details are not central to this paper. The syntax and semantics of Prexo IR are defined in the next section; intuitively, the transformation rewrites imperative array code with loops into a system of space–time arrays. This conversion occurs at Exo's compilation time through a standard SSA conversion process:

(1) Inlining: All function calls and window statements in the Exo IR are inlined.
(2) Arguments: Exo IR's function arguments are not converted to Prexo IR, which has no notion of program parameters. Instead, they surface as free variables in the resulting Prexo IR.
(3) SSA-fy Assignments: Each assignment in the Exo IR introduces a fresh symbol for Prexo IR. Exo IR's allocation statements are ignored, as SSA allocates new variables upon assignment.
(4) Space-time Arrays: When loops enclose arrays, loop counters are added as new array dimensions. For example, a one-dimensional array in Exo IR becomes three-dimensional in Prexo IR when nested within two loops. The resulting dimensions track both space (via array offsets) and time (via loop iteration counters), as formalized in the following sections.
(5) Merging Control Flows: At control flow merge points (loop entries, loop exits, and if exits), additional Prexo IR assignment statements are inserted. These serve as phi nodes, similar to those in SSA-based IRs like LLVM IR, allowing Prexo IR to preserve functional behavior with Exo IR even without explicit branching and loop constructs.

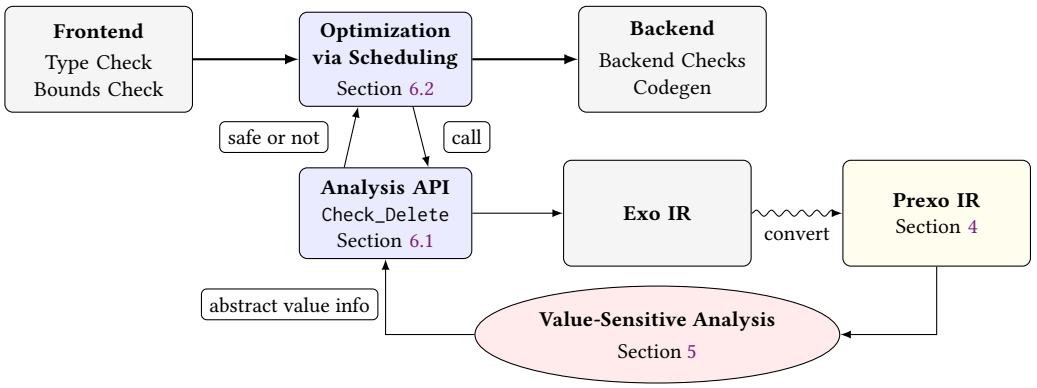


Fig. 3. System overview of Prexo, showing how value-sensitive analysis integrates into the compiler stack of Exo, a performance optimization compiler. The Prexo IR (Section 4), value-sensitive analysis (Section 5), and analysis API (Section 6) represent the novel contributions of this paper. Gray boxes (frontend, backend, Exo IR) are from the original Exo paper [23] and remain unchanged.

```
for i in seq(0, 10):
    x[i] = 0.0
```

convert

```
x_1 =\i \d0 (i == 0 ? x[d0] : x_2[i - 1, d0]) # LoopStart
x_2 =\i \d0 (i == d0 ? 0.0 : x_1[i, d0])
x_3 =\d0 (10 > 0 ? x_2[10 - 1, d0] : x[d0]) # LoopExit
```

(a) Exo IR (b) Prexo IR

Fig. 4. Conversion from Exo IR to Prexo IR.

p : Proc ::=	Fix s	program
s : Stmt ::=	$s_1; s_2$	sequencing
	$y = \lambda i^*.(b\,?\,c_t : c_f)$	control assign
	$x = \lambda i^*.\lambda \eta^*.(b\,?\,e_t : e_f)$	data assign
v : Var ::=	$x \in \mathbb{X}$	data variable
	$y \in \mathbb{Y}$	control variable
n	$\in \mathbb{Z}$	integer
t	$\in \mathbb{B}$	boolean
d	$\in \mathbb{D}$	data values

c : CExpr ::=	n	integer
	$y \mid y[c^*]$	scalar/array reads
	$c_1{+}c_2 \mid c_1{-}c_2 \mid c{*}n \mid c/n$	(quasi-) affine arith
e : DExpr ::=	d	data value
	$x \mid x[c^*]$	scalar/array reads
	$e_1{+}e_2 \mid e_1{-}e_2 \mid e_1{*}e_2 \mid e_1/e_2$	compound expr
b : BExpr ::=	t	boolean
	b_1 and $b_2 \mid b_1$ or b_2	logical ops
	$c_1 == c_2 \mid c_1 < c_2 \mid c_1 \leq c_2$	relational ops

Fig. 5. The syntax of Prexo IR. i^* and η^* are tuples of control variables $y \in \mathbb{Y}$, each denoting loop-iterator variables and array-offset variables.

4 PROGRAM SYNTAX AND CONCRETE SEMANTICS

4.1 Syntax of Prexo IR

Figure 5 defines the syntax of Prexo IR. Prexo explicitly distinguishes between control and data expressions and is a static control program, where control flow depends only on loop indices and control/constant parameters, not on data computed at runtime. We denote the sets of data variables and control variables by $x \in \mathbb{X}$ and $y \in \mathbb{Y}$, respectively. Each control variable y has an iterator offset arity $\mathrm{itr}(y)$ and an array offset space $\mathbb{Z}^{\mathrm{itr}(y)}$. Each data variable x has an iterator offset arity $\mathrm{itr}(x)$, an array offset arity $\dim(x)$, and an array offset space $\mathbb{Z}^{\mathrm{itr}(x)} \times \mathbb{Z}^{\dim(x)}$. The truth values $\mathbb{B} \triangleq \{\mathrm{true}, \mathrm{false}\}$ are ranged over by a metavariable t, and n is the metavariable for $\mathbb{Z} \triangleq \{\cdots, -1, 0, 1, 2, \cdots\}$. The metavariable d ranges over data values in $\mathbb{D}$, including 8-, 32-, and 64-bit integers as well as single- and double-precision floating-point numbers.

Control expressions CExpr encompass integer, scalar and array control variable reads, and (quasi-) affine arithmetic operations. Data expressions DExpr are similarly defined, but operations are

not constrained to be affine. Boolean expressions BExpr include the control expressions with basic logical and relational operations. Indices c^* (we use $.^*$ to denote a tuple) used in array reads are specified as a tuple of control expressions, expressing an access to multi-dimensional arrays. In both data and control expressions, scalar read is a syntactic sugar for an array access of $c^* = \langle\,\rangle$ (empty tuple). A read $y[c^*]$ requires $|c^*| = \mathrm{itr}(y)$, and $x[c^*]$ requires $|c^*| = \dim(x) + \mathrm{itr}(x)$. Scalar reads x and y are well-typed only when $\mathrm{itr}(y) = 0$ and $\dim(x) = \mathrm{itr}(x) = 0$.

A statement $s \in \mathrm{Stmt}$ can be a sequencing, control-assignment, or data-assignment statement. Following prior work [26, 31], Prexo IR adopts the SSA form, where each assignment represents both the declaration and initialization of control or data variables. During the Exo IR to Prexo IR conversion, whenever an assignment overwrites an array, it is renamed to a fresh identifier, effectively treating it as a distinct array instance. The tuples ι^* and η^* consist of distinct λ-bound control variables from $\mathbb{Y}$, where ι^* represents the loop iterators (temporal dimension) and η^* represents the array offset variables (spatial dimension), together forming the space-time array of assign statements. Notably, the syntax excludes "if" and "loop" statements. Instead, conditional logic is expressed via ternary operators $(b\ ?\ e_t : e_f)$ and $(b\ ?\ c_t : c_f)$ within assignments, and loops are transformed into space-time arrays during the Exo IR to Prexo IR conversion. The conversion also inserts additional assignments to handle control flow merging. Free (unassigned) variables in the Prexo IR program Fix s represent the function input parameters in Exo IR.

4.2 Concrete Semantics of Prexo IR

Figure 6 shows denotational semantics of Prexo IR. Both control and data values range over flat lattices $\mathbb{Z}_\perp^\top \triangleq \mathbb{Z} \cup \{\perp, \top\}$ and $\mathbb{D}_\perp^\top \triangleq \mathbb{D} \cup \{\perp, \top\}$ with $\perp \sqsubseteq v \sqsubseteq \top$ for all concrete v. Let $\mathbb{B}_\perp^\top \triangleq \mathbb{B} \cup \{\perp, \top\}$ with the flat order $\perp \sqsubset \mathrm{false}, \mathrm{true} \sqsubset \top$ (and false, true incomparable). Control and data variables live in disjoint environments. A control environment $\sigma_c \in \Sigma_c$ maps each $y \in \mathbb{Y}$ to an array $\mathbb{Z}^{\mathrm{itr}(y)} \to \mathbb{Z}_\perp^\top$. A data environment $\sigma_d \in \Sigma_d$ maps each $x \in \mathbb{X}$ to an array $\mathbb{Z}^{\mathrm{itr}(x)} \times \mathbb{Z}^{\dim(x)} \to \mathbb{D}_\perp^\top$. Since data variables can depend on control variables but not vice versa, the evaluation functions are defined as $\mathcal{E}_c\,[\![s]\!] : \Sigma_c \to \Sigma_c$ and $\mathcal{E}_d\,[\![s]\!] : \Sigma_c \times \Sigma_d \to \Sigma_d$. We write $\sqcup_c$ (resp. $\sqcup_d$) for the join on Σ_c (resp. Σ_d), which are defined pointwise. Hence $(\Sigma_c, \sqsubseteq_c)$ and $(\Sigma_d, \sqsubseteq_d)$ are complete lattices.

Expressions are evaluated over the flat lattices $\mathbb{Z}_\perp^\top$, $\mathbb{D}_\perp^\top$, and $\mathbb{B}_\perp^\top$ with $\perp \sqsubseteq v \sqsubseteq \top$. All case splits below are applied in the written order (left-to-right). For each arithmetic operator $\mathrm{op} \in \{+, -, *, /\}$ we use the strict lift $u\ \widetilde{\mathrm{op}}\ v \;=\; \perp$ if $u{=}\perp$ or $v{=}\perp$; $= \top$ if $u{=}\top$ or $v{=}\top$; $= \mathrm{op}(u, v)$ when defined (e.g. $v \neq 0$ for '/'); and $= \perp$ otherwise. We interpret '/' as a floor division for control expressions. Relational and logical operators are lifted analogously: for $\mathrm{op} \in \{==, <, \leq, \mathsf{and}, \mathsf{or}\}$, $u\ \widetilde{\mathrm{op}}\ v = \perp$ if either operand is $\perp$, $= \top$ if either is $\top$, and $= \mathrm{op}(u, v)$ otherwise. Array reads are strict in each index: $y[c^*]$ and $x[c^*]$ first evaluate c^* via $\mathcal{E}_c^e$; if any index is $\perp$ (resp. $\top$) the read yields $\perp$ (resp. $\top$), otherwise the concrete lookup is used. The conditional is $\mathrm{ite}(b, u, v) = u$ if $b{=}\mathrm{true}$, v if $b{=}\mathrm{false}$, $\perp$ if $b{=}\perp$, and $u \sqcup v$ if $b{=}\top$, where $\sqcup$ is the flat join in the target lattice.

Sequential composition for control variables composes by standard function composition; y-assignments adds $a_y : \mathbb{Z}^{\mathrm{itr}(y)} \to \mathbb{Z}_\perp^\top$ defined pointwise with a parameter z^*. Data evaluation function composes by first evaluating the data store of s_1 under $(\mathcal{E}_c[\![s_1]\!](\sigma_c), \sigma_d)$ and then s_2 under the accumulated control store $\mathcal{E}_c[\![s_1; s_2]\!](\sigma_c)$ and the result of evaluating s_1. a_x also is defined pointwise. Control assignments leave σ_d unchanged, and data assignments leave σ_c unchanged.

A program is written Fix s and evaluated from an initial store (σ_c^0, σ_d^0). We compute a control fixed point $\sigma_c^* = \mathrm{lfp}(F_c)$ where $F_c(\sigma_c) = \sigma_c^0 \sqcup_c \mathcal{E}_c[\![s]\!](\sigma_c)$, then a data fixed point $\sigma_d^* = \mathrm{lfp}(F_d^{\sigma_c^*})$ where $F_d^{\sigma_c}(\sigma_d) = \sigma_d^0 \sqcup_d \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)$. Both fixed points exist by Tarski–Knaster because the stores form complete lattices and both F_c and $F_d^{\sigma_c}$ are monotone (proof in Appendix A). Although we write lfp to emphasize the order-theoretic construction, the semantics is deterministic: since statement execution is deterministic, the denotation of Fix s is uniquely determined by s and the initial store.

$$
\begin{array}{|l|}
\hline
\mathcal{E}_c^e\,[\![c]\!] : \Sigma_c \to \mathbb{Z}_\perp^\top \\
\qquad\qquad \mathcal{E}_c^e\,[\![n]\!]\,(\sigma_c) = n \\
\qquad\qquad \mathcal{E}_c^e\,[\![y]\!]\,(\sigma_c) = \sigma_c(y, \langle\,\rangle) \\
\qquad\qquad \mathcal{E}_c^e\,[\![y[c^*]]\!]\,(\sigma_c) = \sigma_c(y, \langle \mathcal{E}_c^e\,[\![c_i]\!]\,(\sigma_c)\,\rangle_i) \\
\qquad\qquad \mathcal{E}_c^e\,[\![c_1 \text{ op } c_2]\!]\,(\sigma_c) = \mathcal{E}_c^e\,[\![c_1]\!]\,(\sigma_c)\ \widetilde{\text{op}}\ \mathcal{E}_c^e\,[\![c_2]\!]\,(\sigma_c) \quad (\text{op} \in \{+, -, *, /\}) \\
\hline
\mathcal{E}_d^e\,[\![e]\!] : \Sigma_c \to \Sigma_d \to \mathbb{D}_\perp^\top \\
\qquad\qquad \mathcal{E}_d^e\,[\![d]\!]\,(\sigma_c, \sigma_d) = d \\
\qquad\qquad \mathcal{E}_d^e\,[\![x]\!]\,(\sigma_c, \sigma_d) = \sigma_d(x, \langle\,\rangle) \\
\qquad\qquad \mathcal{E}_d^e\,[\![x[c^*]]\!]\,(\sigma_c, \sigma_d) = \sigma_d(x, \langle \mathcal{E}_c^e\,[\![c_i]\!]\,(\sigma_c)\,\rangle_i) \\
\qquad\qquad \mathcal{E}_d^e\,[\![e_1 \text{ op } e_2]\!]\,(\sigma_c, \sigma_d) = \mathcal{E}_d^e\,[\![e_1]\!]\,(\sigma_c, \sigma_d)\ \widetilde{\text{op}}\ \mathcal{E}_d^e\,[\![e_2]\!]\,(\sigma_c, \sigma_d) \quad (\text{op} \in \{+, -, *, /\}) \\
\hline
\mathcal{E}_b\,[\![b]\!] : \Sigma_c \to \mathbb{B}_\perp^\top \\
\qquad\qquad \mathcal{E}_b\,[\![t]\!]\,(\sigma_c) = t \\
\qquad\qquad \mathcal{E}_b\,[\![b_1 \text{ op } b_2]\!]\,(\sigma_c) = \mathcal{E}_b\,[\![b_1]\!]\,(\sigma_c)\ \widetilde{\text{op}}\ \mathcal{E}_b\,[\![b_2]\!]\,(\sigma_c) \quad (\text{op} \in \{\text{and}, \text{or}\}) \\
\qquad\qquad \mathcal{E}_b\,[\![c_1 \text{ op } c_2]\!]\,(\sigma_c) = \mathcal{E}_c^e\,[\![c_1]\!]\,(\sigma_c)\ \widetilde{\text{op}}\ \mathcal{E}_c^e\,[\![c_2]\!]\,(\sigma_c) \quad (\text{op} \in \{==, <, \leq\}) \\
\hline
\mathcal{E}_c\,[\![s]\!] : \Sigma_c \to \Sigma_c \\
\qquad\qquad \mathcal{E}_c\,[\![s_1; s_2]\!]\,(\sigma_c) = \mathcal{E}_c\,[\![s_2]\!]\,\big(\mathcal{E}_c\,[\![s_1]\!]\,(\sigma_c)\big) \\
\qquad \mathcal{E}_c\,[\![\, y = \lambda\imath^*.(b\ ?\ c_t : c_f)\,]\!]\,(\sigma_c) = \sigma_c\big[y \mapsto a_y\big] \\
\text{where } a_y : \mathbb{Z}^{\text{itr}(y)} \to \mathbb{Z}_\perp^\top \text{ and } a_y(z^*) = \text{ite}\,\big(\mathcal{E}_b\,[\![b]\!]\,(\sigma_c[\imath^* \mapsto z^*]),\ \mathcal{E}_c^e\,[\![c_t]\!]\,(\sigma_c[\imath^* \mapsto z^*]),\ \mathcal{E}_c^e\,[\![c_f]\!]\,(\sigma_c[\imath^* \mapsto z^*])\big). \\
\hline
\mathcal{E}_d\,[\![s]\!] : \Sigma_c \to \Sigma_d \to \Sigma_d \\
\qquad\qquad \mathcal{E}_d\,[\![s_1; s_2]\!]\,(\sigma_c, \sigma_d) = \mathcal{E}_d\,[\![s_2]\!]\,\big(\mathcal{E}_c\,[\![s_1; s_2]\!]\,(\sigma_c),\ \mathcal{E}_d\,[\![s_1]\!]\,\big(\mathcal{E}_c\,[\![s_1]\!]\,(\sigma_c), \sigma_d\big)\,\big) \\
\mathcal{E}_d\,[\![\, x = \lambda\imath^*.\lambda\eta^*.(b\ ?\ e_t : e_f)\,]\!]\,(\sigma_c, \sigma_d) = \sigma_d\big[x \mapsto a_x\big] \\
\text{where } a_x : \mathbb{Z}^{\text{itr}(x)} \times \mathbb{Z}^{\dim(x)} \to \mathbb{D}_\perp^\top \text{ and } a_x(z_1^*, z_2^*) = \text{ite}\,\big(\mathcal{E}_b\,[\![b]\!]\,(\sigma_c'),\ \mathcal{E}_d^e\,[\![e_t]\!]\,(\sigma_c', \sigma_d),\ \mathcal{E}_d^e\,[\![e_f]\!]\,(\sigma_c', \sigma_d)\big) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sigma_c' = \sigma_c[\imath^* \mapsto z_1^*, \eta^* \mapsto z_2^*]. \\
\hline
\mathcal{E}\,[\![p]\!] : \Sigma_c \times \Sigma_d \to \Sigma_c \times \Sigma_d \\
\qquad\qquad \mathcal{E}\,[\![\text{Fix } s]\!]\,(\sigma_c^0, \sigma_d^0) = (\sigma_c^*,\ \sigma_d^*) \\
\qquad\qquad\qquad\qquad \sigma_c^* = \text{lfp}(F_c) \quad \text{where } F_c(\sigma_c) = \sigma_c^0 \sqcup_c\ \mathcal{E}_c[\![s]\!](\sigma_c) \\
\qquad\qquad\qquad\qquad \sigma_d^* = \text{lfp}(F_d^{\sigma_c^*}) \quad \text{where } F_d^{\sigma_c}(\sigma_d) = \sigma_d^0 \sqcup_d\ \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \\
\hline
\end{array}
$$

Fig. 6. Denotational semantics of Prexo IR.

5 VALUE-SENSITIVE ANALYSIS

To enable sound safety checking, we propose a value-sensitive analysis that tracks program variables and over-approximates their values. We introduce a novel abstract domain called the cylindrical algebraic decomposition (CAD) tree domain, and develop an efficient value-sensitive analysis on top of it by designing a new flood-fill widening operator. In this section, we define the CAD tree domain (Section 5.1) and its abstraction and concretization functions (Section 5.2). Subsequently, we detail the transfer functions (Section 5.3) and the flood-fill widening operator (Section 5.4) and establish the soundness guarantee of the analysis.

5.1 CAD Tree Domain

We separate the *base domain* into a control domain $\mathcal{B}_C$ and a data domain $\mathcal{B}_D$ (Fig. 7a). Elements of $\mathcal{B}_C$, written ζ, denote abstract control values and are closed under affine operators ($+$, $-$, $*n$), and a quasi-affine operator ($/n$), together with lattice elements $\top_\zeta$ and $\bot_\zeta$. Elements of $\mathcal{B}_D$, written δ, denote abstract data values: either a data value d, an array element $x[\zeta^*]$, or extreme elements $\top_\delta$ and $\bot_\delta$, where x is a free variable in the Prexo IR procedure and ζ^* is a tuple of control expressions from $\mathcal{B}_C$. Intuitively, $x[\zeta^*]$ carries relational information about arrays by referring to the value of the Exo IR procedure's data array parameter, which appears as a free variable in Prexo IR. For example, if ζ^* evaluates to index expressions $i_1, \ldots, i_k$, then $x[i_1, \ldots, i_k]$ is the abstract

$\zeta : \mathcal{B}_C$	$::= \top_\zeta \mid \bot_\zeta$	top/bot
	$\mid n \mid y \mid$	ctrl scalars
	$\mid \zeta_1{+}\zeta_2 \mid \zeta_1{-}\zeta_2 \mid \zeta{*}n$	affine arith
	$\mid \zeta/n$	quasi affine
$\delta : \mathcal{B}_D$	$::= d \mid x[\zeta^*]$	data/arrays
	$\mid \top_\delta \mid \bot_\delta$	top/bot
g : Guard	$::= t$	boolean
	$\mid \zeta_1 \text{ op } \zeta_2$	op $\in \{==, <, \leq\}$
	$\mid g_1 \text{ and } g_2$	logical and

(a) Base domains $\mathcal{B}_C$ and $\mathcal{B}_D$.

$\tau_\pi : \mathcal{T}\langle\pi\rangle$	$::= (\text{vars} : \iota^*\eta^*, \text{cuts} : \zeta^*, \text{tree} : \mathit{node}\langle\pi\rangle)$	
	$\mid \bot_{\mathcal{T}\langle\pi\rangle} \mid \top_{\mathcal{T}\langle\pi\rangle}$	
$\mathit{node}\langle\pi\rangle$	$::= \text{Leaf}(\pi, \mathbb{Z}^{\lvert\iota^*\rvert+\lvert\eta^*\rvert}) \mid \text{LinSplit}(\mathit{cell}\langle\pi\rangle^*)$	
$\mathit{cell}\langle\pi\rangle$	$::= \text{Cell}(g, \mathit{node}\langle\pi\rangle)$	

(b) CAD-tree family $\mathcal{T}\langle\pi\rangle$.

$\mathcal{T}_C$	$\triangleq \mathcal{T}\langle\zeta\rangle$	control tree domain
$\mathcal{T}_D$	$\triangleq \mathcal{T}\langle\delta\rangle$	data tree domain

(c) Control/data CAD-tree domains

Fig. 7. Syntax of the base domains (left) and the CAD-tree domains (right). In (b), $\mathcal{T}$ is parameterized by π to collapse control/data tree definitions in (c). ι^* and η^* are tuples of control variables $y \in \mathbb{Y}$, each denoting loop-iterator variables and array-offset variables.

value symbolically depicting the value stored *at the beginning of* the Exo IR procedure. Guards g compare ζ-expressions via op $\in \{==, <, \leq\}$ and a conjunction is expressed with and. Disjunction is represented structurally by sibling CAD cells; we therefore restrict guards to conjunctions.

Appendix B reviews classical CAD, a finite partition of $\mathbb{R}^n$ into semi-algebraic cells on which each input polynomial has constant sign [8, 9]. We restrict the input polynomials to affine, yielding a cylindrical, sign-invariant decomposition with (potentially) rational roots; a cell is declared infeasible when its induced polyhedron contains no integer point, verified via an ILP feasibility check. We encode this as a CAD-tree whose leaves (cells) carry a base-domain annotation. The CAD algorithm takes ordered variables (which we term "vars") $(y_1, \ldots, y_n)$ and affine expressions of those variables ("cuts") as inputs, and returns a CAD tree ("tree").

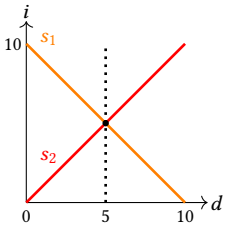
The CAD-tree domain is defined as a family $\mathcal{T}\langle\pi\rangle$ parameterized by the base domain type π (Fig. 7b). An abstract value $\tau_\pi \in \mathcal{T}\langle\pi\rangle$ is $\bot_{\mathcal{T}\langle\pi\rangle}$, $\top_{\mathcal{T}\langle\pi\rangle}$, or a triple $(\text{vars} : \iota^*\eta^*,\ \text{cuts} : \zeta^*,\ \text{tree} : \mathit{node}\langle\pi\rangle)$. Here, $\iota^*\eta^* = (\iota_1 \prec \cdots \prec \eta_1 \prec \cdots) \subset \mathbb{Y}$ are totally ordered CAD generator variables each denoting loop-iterator and array-offset variables, and ζ^* is a tuple of quasi-affine expressions over the generator variables. *vars* and *cuts* yield a CAD tree represented by a *tree*: $\text{LinSplit}(\mathit{cell}\langle\pi\rangle^*)$ captures the decomposition of *cells* with pairwise disjoint guards; each $\text{Cell}(g, \mathit{node}\langle\pi\rangle)$ pairs a guard g (over generator variables *vars*) with a further subtree; leaves are $\text{Leaf}(\pi, \mathbb{Z}^{\lvert\iota^*\rvert+\lvert\eta^*\rvert})$, which store a base domain value π with an integer tuple $(n_1, \cdots, n_{\lvert\iota^*\rvert+\lvert\eta^*\rvert})$ representing a sample point for vars $(\iota_1 = n_1, \cdots, \eta_1 = n_{\lvert\iota^*\rvert}, \cdots)$ that route to the current cell. We use two CAD tree instances (Fig. 7c): the control tree domain $\mathcal{T}_C \triangleq \mathcal{T}\langle\zeta\rangle$ and the data tree domain $\mathcal{T}_D \triangleq \mathcal{T}\langle\delta\rangle$, whose leaves carry, respectively, control base domain values and data base domain values. Note that $\eta^* = \langle\rangle$ for the control tree domain $\mathcal{T}_C$, as control variables do not have array offsets (Fig. 5).

Example 5.1. To build intuition for the CAD tree domain, consider the concrete example in Figure 8, which demonstrates τ_D for the Array Reverse Array-Write microbenchmark also shown in Figure 15 (m). In this example, each element of data array x is written twice with the same y value through two statements for each iteration i from 0 to 10. Figure 8b illustrates this behavior through a space-time diagram, where the horizontal axis represents the array location d and the vertical axis represents the iteration count i. The diagonal lines mark the points in space and time where statements $s1$ and $s2$ modify array x, creating a partition at $d = 5$. Figure 8c shows the corresponding τ_D resulting from invoking CAD with the parameters (vars=$\langle i, d\rangle$, cuts=$\langle i - d, i + d - 10\rangle$). The highlighted guards indicate which statement's expression generated each constraint.

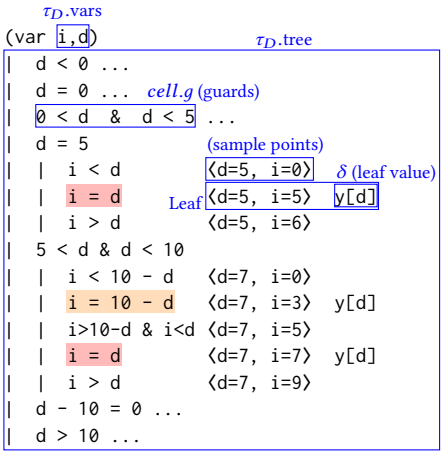
Employing the CAD tree domain as an abstract domain enables us to decompose the array's iterator-offset space into sign-invariant cells. This decomposition supports efficient flood-fill widening for propagating values forward in iteration time, as discussed in Section 5.4. Furthermore,

```
    for i in seq(0, 11):
s1:     x[10-i] = y[10-i]
s2:     x[i] = y[i]
```

(a) Array Reverse Array-Write in Exo IR



(b) Visualization of the CAD tree for (a)



(c) Pretty printing of the CAD tree for (a)

Fig. 8. CAD tree domain for array reversal. (a) Space-time diagram of x at location d after iteration i, with diagonal lines for s_1 and s_2. (b) CAD tree representation with implicit default leaf values $\perp_\delta$ (not shown).

CAD's projection phase computes the resultant of input cuts, with which we can systematically identify *partitions* for values that appear only semantically in the code, such as $d = 5$ in the example.

Definition 5.1 (Partial orders on base domains)**.** Both $\mathcal{B}_C$ and $\mathcal{B}_D$ are flat lattices modulo a canonical syntactic congruence $\equiv$. We lift $\equiv$ pointwise to tuples ($\equiv^*$), and define it on $\mathcal{B}_D$ structurally so that $x[\zeta^*] \equiv x'[\zeta'^*]$ iff $x = x'$, $|\zeta^*| = |\zeta'^*|$, and $\zeta^* \equiv^* \zeta'^*$. The partial orders are:

$$\zeta \sqsubseteq_{\mathcal{B}_C} \zeta' \iff (\zeta = \perp_\zeta) \vee (\zeta' = \top_\zeta) \vee (\zeta, \zeta' \notin \{\perp_\zeta, \top_\zeta\} \wedge \zeta \equiv \zeta'),$$
$$\delta \sqsubseteq_{\mathcal{B}_D} \delta' \iff (\delta = \perp_\delta) \vee (\delta' = \top_\delta) \vee (\delta, \delta' \notin \{\perp_\delta, \top_\delta\} \wedge \delta \equiv \delta').$$

For $\mathcal{B}_\pi \in \{\mathcal{B}_C, \mathcal{B}_D\}$, $\sqcup_\pi$ is the *flat* join modulo $\equiv$: $\perp_\pi$ is neutral, $\top_\pi$ is absorbing, and for non-extreme $\beta, \beta' \in \mathcal{B}_\pi$, $\beta \sqcup_\pi \beta' = \beta$ iff $\beta \equiv \beta'$, otherwise $\top_\pi$. $\mathcal{B}_C$ and $\mathcal{B}_D$ are complete lattices.

Definition 5.2 (Partial order $\sqsubseteq_\mathcal{T}$ on $\mathcal{T}\langle\pi\rangle$)**.** Let $\tau_i = (\text{vars} : \iota_i^* \eta_i^*, \text{cuts} : \zeta_i^*, \text{tree} : n_i) \in \mathcal{T}\langle\pi\rangle$ or $\tau_i \in \{\perp_{\mathcal{T}\langle\pi\rangle}, \top_{\mathcal{T}\langle\pi\rangle}\}$ for $i \in \{1, 2\}$. Let $\tau_i(\hat{n}) \in \mathcal{B}_C \cup \mathcal{B}_D$ denote the payload base-domain value at the leaf reached by traversing τ_i from the root, choosing at each LinSplit the child $\text{Cell}(g, \cdot)$ whose guard g is satisfied under $\hat{n}$, where $\hat{n} \in \mathbb{Z}^{|\iota^*|+|\eta^*|}$. Since CAD provides a complete decomposition, any $\hat{n} \in \mathbb{Z}^{|\iota^*|+|\eta^*|}$ maps to exactly one cell in the tree. Then

$$\tau_1 \sqsubseteq_{\mathcal{T}\langle\pi\rangle} \tau_2 \iff \big(\tau_1 = \perp_{\mathcal{T}\langle\pi\rangle}\big) \vee \big(\tau_2 = \top_{\mathcal{T}\langle\pi\rangle}\big) \vee \big(\iota_1^* \equiv \iota_2^* \wedge \eta_1^* \equiv \eta_2^* \wedge \forall \hat{n} \in \mathbb{Z}^{|\iota_1^*|+|\eta_1^*|}.\ \tau_1(\hat{n}) \sqsubseteq_\pi \tau_2(\hat{n})\big).$$

For $\tau_1, \tau_2 \in \mathcal{T}\langle\pi\rangle$, $\tau_1 \sqcup_{\mathcal{T}\langle\pi\rangle} \tau_2$ is $\top_{\mathcal{T}\langle\pi\rangle}$ if ($\iota_1^* \not\equiv \iota_2^*$ or $\eta_1^* \not\equiv \eta_2^*$); otherwise it is the pointwise lift of the base join: $\forall\, \hat{n}$, $(\tau_1 \sqcup_{\mathcal{T}\langle\pi\rangle} \tau_2)(\hat{n}) = \tau_1(\hat{n}) \sqcup_\pi \tau_2(\hat{n})$. Thus, $\mathcal{T}\langle\pi\rangle$ is a complete lattice.

5.2 Abstraction and Concretization

We define abstraction $\alpha : \mathcal{P}(\Sigma) \to \Sigma^\#$ and concretization $\gamma : \Sigma^\# \to \mathcal{P}(\Sigma)$ functions. The soundness (for every set of concrete states $S \subseteq \Sigma$, $S \subseteq \gamma(\alpha(S))$) is proved as Theorem C.1 in Appendix C.

Definition 5.3. (Abstract Environment) We denote the set of abstract environments by $\Sigma^\#$. Let $\Sigma^\# \triangleq \Sigma_c^\# \times \Sigma_d^\#$ with $\Sigma_c^\# \triangleq (\mathbb{Y} \to \mathcal{T}_C)$ and $\Sigma_d^\# \triangleq (\mathbb{X} \to \mathcal{T}_D)$.

Based on the definition of the partial order $\sqsubseteq_{\mathcal{T}\langle\pi\rangle}$ over the CAD tree domain $\mathcal{T}\langle\pi\rangle$, we can introduce the partial order over $\Sigma^\#$.

Definition 5.4 (Partial Order $\sqsubseteq_{\Sigma^\#}$). For $\sigma^\#_{c1}, \sigma^\#_{c2} \in \Sigma^\#_C$ and $\sigma^\#_{d1}, \sigma^\#_{d2} \in \Sigma^\#_D$, define the orders

$$\sigma^\#_{c1} \sqsubseteq_{\Sigma^\#_C} \sigma^\#_{c2} \iff \forall y \in \mathbb{Y}.\, \sigma^\#_{c1}(y) \sqsubseteq_{\mathcal{T}_C} \sigma^\#_{c2}(y), \qquad \sigma^\#_{d1} \sqsubseteq_{\Sigma^\#_D} \sigma^\#_{d2} \iff \forall x \in \mathbb{X}.\, \sigma^\#_{d1}(x) \sqsubseteq_{\mathcal{T}_D} \sigma^\#_{d2}(x),$$

where $\sqsubseteq_{\mathcal{T}_C}$ and $\sqsubseteq_{\mathcal{T}_D}$ are the CAD-tree orders of Def. 5.2. Bottom/top are pointwise: $(\bot_{\Sigma^\#_C})(y) \triangleq \bot_{\mathcal{T}_C}$, $(\top_{\Sigma^\#_C})(y) \triangleq \top_{\mathcal{T}_C}$ (and analogously for $\Sigma^\#_D$). Writing $\sigma^\#_1 = (\sigma^\#_{c1}, \sigma^\#_{d1})$ and $\sigma^\#_2 = (\sigma^\#_{c2}, \sigma^\#_{d2})$ for the control/data components of environments in $\Sigma^\#$, the partial order is

$$\sigma^\#_1 \sqsubseteq_{\Sigma^\#} \sigma^\#_2 \iff \sigma^\#_{c1} \sqsubseteq_{\Sigma^\#_C} \sigma^\#_{c2} \wedge \sigma^\#_{d1} \sqsubseteq_{\Sigma^\#_D} \sigma^\#_{d2},$$

with $\bot_{\Sigma^\#} \triangleq (\bot_{\Sigma^\#_C}, \bot_{\Sigma^\#_D})$ and $\top_{\Sigma^\#} \triangleq (\top_{\Sigma^\#_C}, \top_{\Sigma^\#_D})$.

Definition 5.5 (Join $\sqcup_{\Sigma^\#}$). The join is defined as the pointwise lift of the tree joins: for $\sigma^\#_i = (\sigma^\#_{ci}, \sigma^\#_{di})$,

$$\sigma^\#_1 \sqcup_{\Sigma^\#} \sigma^\#_2 \triangleq \big(y \mapsto \sigma^\#_{c1}(y) \sqcup_{\mathcal{T}_C} \sigma^\#_{c2}(y),\ \ x \mapsto \sigma^\#_{d1}(x) \sqcup_{\mathcal{T}_D} \sigma^\#_{d2}(x)\big).$$

It is the least upper bound in $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$, with $\bot_{\Sigma^\#}$ neutral and $\top_{\Sigma^\#}$ absorbing. Thus, $\Sigma^\#$ is a complete lattice.

We write $\lceil\cdot\rceil_C : \mathbb{Z} \to \mathcal{B}_C$ and $\lceil\cdot\rceil_D : \mathbb{D} \to \mathcal{B}_D$ for lifting from concrete values to base domain values: $\lceil n \rceil_C \triangleq n \quad (n \in \mathbb{Z})$ and $\lceil d \rceil_D \triangleq d \quad (d \in \mathbb{D})$. When indices are integers we implicitly coerce them into $\mathcal{B}_C$, e.g. $\lceil x[i_1, \ldots, i_k] \rceil_D = x[\lceil i_1 \rceil_C, \ldots, \lceil i_k \rceil_C]$. For guards $g_1, \ldots, g_k$, we define the syntactic conjunction $\bigwedge_{r=1}^{k} g_r$ as g_1 and $\cdots$ and g_k. For a variable tuple y^*, guard g, and an abstract value π, $\left[\, y^* \;\middle|\; g \Rightarrow \pi \,\right] \triangleq (\, y^*,\ \langle\rangle,\ \mathrm{LinSplit}(\langle \mathrm{Cell}(g, \mathrm{Leaf}(\pi, \langle\rangle))\rangle)\,)$. For any index set I and a tuple of cells $\langle\, [y^* \mid g_i \Rightarrow \pi_i] \,\rangle_{i \in I}$ we define the structural disjunction $\bigsqcup_{i \in I} [y^* \mid g_i \Rightarrow \pi_i] \triangleq \Big(\, y^*,\ \langle\rangle,\ \mathrm{LinSplit}\big(\langle \mathrm{Cell}(g_i, \mathrm{Leaf}(\pi_i, \langle\rangle))\rangle_{i \in I}\big)\,\Big)$. Increasing I simply adds sibling cells in the topmost LinSplit node (disjunction-as-structure).

Definition 5.6 (Abstraction α). We first define a per-state abstraction $\hat{\alpha} : \Sigma \to \Sigma^\#$ pointwise, and then lift it to sets by join: $\alpha(S) \triangleq \bigsqcup_{\Sigma^\#}^{\sigma \in S} \hat{\alpha}(\sigma)$ for $S \subseteq \Sigma$. For a concrete state $\sigma = (\sigma_c, \sigma_d)$, $\hat{\alpha}(\sigma_c, \sigma_d) \triangleq (\alpha_c(\sigma_c), \alpha_d(\sigma_c, \sigma_d))$, defined pointwise.

Control. For $y \in \mathbb{Y}$ with fresh control variables $\vec{\iota} = (\iota_1, \ldots, \iota_{\mathrm{itr}(y)})$,

$$\alpha_c(\sigma_c)(y) \triangleq \bigsqcup_{\vec{i} \in \mathbb{Z}^{\mathrm{itr}(y)}} \left[\, \vec{\iota} \;\middle|\; \bigwedge_{r=1}^{\mathrm{itr}(y)} (\iota_r = i_r) \Rightarrow \left\lceil \sigma_c(y)(\vec{i}) \right\rceil_C \,\right].$$

Data. Let $\mathrm{Lhs}_p \subseteq (\mathbb{X} \cup \mathbb{Y})$ be the variables that are being assigned in the concrete program p. For $x \in \mathbb{X}$ with fresh $\vec{\iota} = (\iota_1, \ldots, \iota_{\mathrm{itr}(x)})$ and $\vec{\eta} = (\eta_1, \ldots, \eta_{\dim(x)})$,

$$\alpha_d(\sigma_c, \sigma_d)(x) \triangleq \begin{cases} \left[\, \vec{\eta} \;\middle|\; \text{true} \Rightarrow \lceil x[\vec{\eta}] \rceil_D \,\right] & \text{if } x \notin \mathrm{Lhs}_p, \\ \displaystyle\bigsqcup_{\substack{\vec{i} \in \mathbb{Z}^{\mathrm{itr}(x)} \\ \vec{j} \in \mathbb{Z}^{\dim(x)}}} \left[\, \vec{\iota}\,\vec{\eta} \;\middle|\; \left(\bigwedge_{r=1}^{\mathrm{itr}(x)} \iota_r = i_r\right) \text{and} \left(\bigwedge_{s=1}^{\dim(x)} \eta_s = j_s\right) \Rightarrow \left\lceil \sigma_d(x)(\vec{i}, \vec{j}) \right\rceil_D \,\right] & \text{if } x \in \mathrm{Lhs}_p. \end{cases}$$

Definition 5.7 (Lifting $\mathcal{L}^\#$). The lifting function $\mathcal{L}^\# : \Sigma^\# \to \Phi$, defined in Figure 9, traverses an abstract environment and produces a first-order formula. Writing $\Sigma^\# = (\Sigma^\#_c, \Sigma^\#_d)$,

$$\mathcal{L}^\#(\Sigma^\#_c, \Sigma^\#_d) \triangleq \Big(\bigwedge_{y \in \mathrm{dom}(\Sigma^\#_c)} \mathcal{L}^\#_{\mathcal{T}\langle\zeta\rangle}(y, \Sigma^\#_c(y)) \Big) \wedge \Big(\bigwedge_{x \in \mathrm{dom}(\Sigma^\#_d)} \mathcal{L}^\#_{\mathcal{T}\langle\delta\rangle}(x, \Sigma^\#_d(x)) \Big).$$

Definition 5.8 (Concretization γ). We write $a \models b$ to mean b is true in a. We interpret an abstract environment as the set of concrete states that satisfy the formula produced by $\mathcal{L}^\#$:

$$\gamma(\Sigma^\#) \triangleq \{\sigma \in \Sigma \mid \sigma \models \mathcal{L}^\#(\Sigma^\#)\}.$$

$$\mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle} : (\mathbb{X} \cup \mathbb{Y}) \to \mathcal{T}\langle\pi\rangle \to \Phi$$
$$\mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r,\ \bot_{\mathcal{T}\langle\pi\rangle}) = \mathit{false} \qquad \mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r,\ \top_{\mathcal{T}\langle\pi\rangle}) = \mathit{true}$$
$$\mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r,\ (\mathit{vars},\ \mathit{cuts},\ \mathit{tree})) \ =\ \mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\ \mathit{tree})$$

$$\mathcal{L}^{\#}_{n\langle\pi\rangle} : (\mathbb{X} \cup \mathbb{Y}) \to \mathit{node}\langle\pi\rangle \to \Phi$$
$$\mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\ \mathrm{Leaf}(\pi',\ \_)) \ =\ \mathcal{L}^{\#}_{\pi}(r,\ \pi')$$
$$\mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\ \mathrm{LinSplit}(\langle \mathrm{Cell}(g_i, t_i)\rangle_{i\in I})) = \bigvee_{i\in I}\big(\mathcal{L}^{\#}_{g}(g_i) \wedge \mathcal{L}^{\#}_{n\langle\pi\rangle}(r, t_i)\big)$$
$$\mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\ \mathrm{LinSplit}([])) \ =\ \mathit{false}$$

$$\mathcal{L}^{\#}_{g} : \mathrm{Guard} \to \Phi$$
$$\mathcal{L}^{\#}_{g}(\mathrm{true}) = \mathit{true} \qquad \mathcal{L}^{\#}_{g}(\mathrm{false}) = \mathit{false}$$
$$\mathcal{L}^{\#}_{g}(\zeta_1\ \mathrm{op}\ \zeta_2) = \nu^{C}_{\mathrm{new}}(y_1, y_2 : \mathbb{Y}).$$
$$\big(\textstyle\bigwedge_{i=1}^{2} \mathcal{L}^{\#}_{\zeta}(y_i,\ \zeta_i) \wedge [\![y_1\ \widetilde{\mathrm{op}}\ y_2]\!]\big) \quad \mathrm{op} \in \{==, <, \leq\}$$
$$\mathcal{L}^{\#}_{g}(g_1\ \mathrm{and}\ g_2) = \mathcal{L}^{\#}_{g}(g_1) \ \wedge\ \mathcal{L}^{\#}_{g}(g_2)$$

$$\mathcal{L}^{\#}_{\zeta} : \mathbb{Y} \to \mathcal{B}_C \to \Phi$$
$$\mathcal{L}^{\#}_{\zeta}(r,\ n) \ =\ [\![r = n]\!] \qquad \mathcal{L}^{\#}_{\zeta}(r,\ y) \ =\ [\![r = y]\!]$$
$$\mathcal{L}^{\#}_{\zeta}(r, \zeta_1\ \mathrm{op}\ \zeta_2) = \nu^{C}_{\mathrm{new}}(y_1, y_2 : \mathbb{Y}).$$
$$\big(\textstyle\bigwedge_{i=1}^{2} \mathcal{L}^{\#}_{\zeta}(y_i,\ \zeta_i) \wedge [\![r = y_1\ \widetilde{\mathrm{op}}\ y_2]\!]\big) \quad \mathrm{op} \in \{+, -, *, /\}$$
$$\mathcal{L}^{\#}_{\zeta}(r,\ \top_{\zeta}) \ =\ \mathit{true} \qquad \mathcal{L}^{\#}_{\zeta}(r,\ \bot_{\zeta}) \ =\ \mathit{false}$$

$$\mathcal{L}^{\#}_{\delta} : \mathbb{X} \to \mathcal{B}_D \to \Phi$$
$$\mathcal{L}^{\#}_{\delta}(r,\ d) \ =\ [\![\, r = d \,]\!]$$
$$\mathcal{L}^{\#}_{\delta}(r,\ x[\zeta_1, \ldots, \zeta_k]) = \nu^{C}_{\mathrm{new}}(y_1, \ldots, y_k\ : \mathbb{Y}).$$
$$\Big(\textstyle\bigwedge_{i=1}^{k} \mathcal{L}^{\#}_{\zeta}(y_i,\ \zeta_i)\Big) \ \wedge\ [\![\, r = \nu^{D}_{\mathrm{new}}(x,\ y_1, \ldots, y_k) \,]\!]$$
$$\mathcal{L}^{\#}_{\delta}(r,\ \top_{\delta}) \ =\ \mathit{true} \qquad \mathcal{L}^{\#}_{\delta}(r,\ \bot_{\delta}) \ =\ \mathit{false}$$

Fig. 9. Lifting functions $\mathcal{L}^{\#}$ from CAD trees to first-order logic. $[\![\ldots]\!]$ denotes the code quote of the first-order logic formula expressing that $r \in \mathbb{X} \cup \mathbb{Y}$ equals the given value or expression. $\nu^{D}_{\mathrm{new}} : \mathbb{X} \times \mathbb{Y}^* \to \mathbb{X}$ is an injective renaming from array–index tuples to variables in $\mathbb{X}$. $\nu^{C}_{\mathrm{new}}\ (\vec{y} : \mathbb{Y})$. introduces fresh control variables.

Example 5.2. Suppose we have a variable $x \in \mathbb{X}$ and an abstract value $\tau'_\delta = (\mathrm{vars} : \langle n\rangle, \mathrm{cuts} : \langle n\rangle, \mathrm{tree} : \mathrm{LinSplit}\langle \mathrm{Cell}(n < 0, \mathrm{Leaf}(\bot_\delta, \langle -1\rangle)),$ $\mathrm{Cell}(n = 0,\ \mathrm{Leaf}(1.0, \langle 0\rangle)), \mathrm{Cell}(n > 0,\ \mathrm{Leaf}(\top_\delta, \langle 1\rangle)))\rangle$. This tree is visualized as the inset figure on the right. The corresponding formulas obtained by lifting function $\mathcal{L}^{\#}_{\mathcal{T}\langle\delta\rangle}(x, \tau'_\delta)$ is:

```
(var n)
| n < 0 ⟨n=-1⟩ bot
| n = 0 ⟨n=0⟩  1.0
| n > 0 ⟨n=1⟩  top
```

$$(n < 0)\ ?\ \mathrm{false} : \big(n = 0\big)\ ?\ x == 1.0 : \mathrm{true}\big).$$

The lifting process traverses the CAD tree, converting each branch condition and leaf value into a boolean expression. Bottom leaves ($\bot_\pi$) lift to false (infeasibility), top leaves ($\top_\pi$) lift to true (all possible values), and concrete scalars like 1.0 lift to equality formulas $[\![x == 1.0]\!]$. The tree structure naturally translates to nested conditional expressions, preserving the logical flow of the original abstract value while converting it to formulas.

5.3 Transfer Function

We have constructed three lattices, namely $(\mathcal{T}_C, \sqsubseteq_{\mathcal{T}_C})$, $(\mathcal{T}_D, \sqsubseteq_{\mathcal{T}_D})$, and $(\Sigma^{\#}, \sqsubseteq_{\Sigma^{\#}})$. Based on them, we formulate the abstract semantics of Prexo IR through transfer functions that update abstract environment, faithfully aligning with the concrete semantics in Figure 6. We introduce *evaluation functions* $A^{\#}, B^{\#}, C^{\#}, D^{\#}$ that evaluate Prexo IR expressions over the abstract environment $\sigma^{\#}$ (Figure 10). $A^{\#}$ extracts affine forms from control expressions, $B^{\#}$ turns boolean guards into tuples for cuts, and $C^{\#}/D^{\#}$ lift control/data expressions to the control/data tree domains $\mathcal{T}_C$ and $\mathcal{T}_D$. Write Lhs_p for the set of program variables that are assigned in the procedure p. The environment $\sigma^{\#}$ stores abstract values only for variables in Lhs_p, since procedure parameters in Exo IR are free variables in Prexo IR. This explains the cases $y \in \mathrm{Lhs}_p$ and $x \in \mathrm{Lhs}_p$ in Fig. 10: if a variable is in Lhs_p we read its tracked abstract value from $\sigma^{\#}$ (so updates are reflected); if it is not, we produce a leaf that symbolically represents the corresponding free variable. For control variables y, $C^{\#}$ therefore either looks up $\sigma^{\#}(y)$ (when $y \in \mathrm{Lhs}_p$) or emits a leaf $[\langle\rangle \mid \mathrm{true} \Rightarrow y]$. For data variables x, $D^{\#}$ behaves analogously on scalars, and on arrays distinguishes two cases: (i) when $x \in \mathrm{Lhs}_p$, the read substitutes the concrete indices into the abstract tree stored at $\sigma^{\#}(x)$ (via the substitution θ); (ii) when $x \notin \mathrm{Lhs}_p$ (an input array parameter), a read $x[c_1, \ldots, c_n]$ becomes a symbolic value $x[A_1, \ldots, A_n]$

$$
\begin{array}{l}
A^\# : \mathrm{CExpr} \to \Sigma^\# \to \mathcal{B}_C \\
\qquad A^\#\,[\![n]\!]\;\sigma^\# \;=\; n \qquad A^\#\,[\![y]\!]\;\sigma^\# \;=\; y \qquad A^\#\,[\![y[c_1,\ldots,c_n]]\!]\;\sigma^\# \;=\; \top_\zeta \\
\qquad A^\#\,[\![c_1 \,\mathrm{op}\, c_2]\!]\;\sigma^\# \;=\; A_1, A_2 \notin \{\bot_\zeta, \top_\zeta\}\;?\; A_1 \,\widetilde{\mathrm{op}}\, A_2 \;:\; \top_\zeta, \quad \text{where } A_i = A^\#\,[\![c_i]\!]\;\sigma^\# \\
\hline
B^\# : \mathrm{BExpr} \to \Sigma^\# \to \zeta^* \\
\qquad B^\#\,[\![t]\!]\;\sigma^\# \;=\; \langle\,\rangle \qquad B^\#\,[\![b_1 \text{ and } b_2]\!]\;\sigma^\# \;=\; \Pi_1 \uplus \Pi_2 \qquad B^\#\,[\![b_1 \text{ or } b_2]\!]\;\sigma^\# \;=\; \Pi_1 \uplus \Pi_2, \quad \text{where } \Pi_i = B^\#\,[\![b_i]\!]\;\sigma^\# \\
\qquad B^\#\,[\![c_1 \bowtie c_2]\!]\;\sigma^\# \;=\; A_1, A_2 \notin \{\bot_\zeta, \top_\zeta\}\;?\; \langle\, A_1 - A_2 \,\rangle : \langle \top_\zeta \rangle, \quad \text{where } A_i = A^\#\,[\![c_i]\!]\;\sigma^\# \text{ and } \bowtie \in \{==, <, \leq\} \\
\hline
C^\# : \mathrm{CExpr} \to \Sigma^\# \to \mathcal{T}_C \\
\qquad C^\#\,[\![n]\!]\;\sigma^\# \;=\; [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow n\;] \qquad C^\#\,[\![y]\!]\;\sigma^\# \;=\; y \in \mathrm{Lhs}_p\;?\;\sigma^\#(y) : [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow y\;] \\
\qquad C^\#\,[\![c_1 \,\mathrm{op}\, c_2]\!]\;\sigma^\# \;=\; A \notin \{\bot_\zeta, \top_\zeta\}\;?\; [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow A\;] : \top_{\mathcal{T}_C}, \quad \text{where } A = A^\#\,[\![c_1 \,\mathrm{op}\, c_2]\!]\;\sigma^\# \\
C^\#\,[\![y[c_1,\ldots,c_n]]\!]\;\sigma^\# \;=\; \forall k.A_k \notin \{\bot_\zeta, \top_\zeta\}\;?\; (y^{\mathrm{itr}*}, \theta.\mathrm{cuts}, \theta.\mathrm{tree}) : \top_{\mathcal{T}_C}, \\
\qquad\qquad \text{where } (y^{\mathrm{itr}*}, \mathrm{cuts}, \mathrm{tree}) = \sigma^\#(y), \quad \theta = [\; y_k^{\mathrm{itr}} \mapsto A_k \;]_{k=1}^n \text{ and } \quad A_i = A^\#\,[\![c_i]\!]\;\sigma^\# \\
\hline
D^\# : \mathrm{DExpr} \to \Sigma^\# \to \mathcal{T}_D \\
\qquad D^\#\,[\![d]\!]\;\sigma^\# \;=\; [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow d\;] \qquad D^\#\,[\![x]\!]\;\sigma^\# \;=\; x \in \mathrm{Lhs}_p\;?\;\sigma^\#(x) : [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow x\;] \\
\qquad D^\#\,[\![e_1 \,\mathrm{op}\, e_2]\!]\;\sigma^\# \;=\; \top_{\mathcal{T}_D} \\
D^\#\,[\![x[c_1,\ldots,c_n]]\!]\;\sigma^\# \;=\; \begin{cases} (y^{\mathrm{itr}*}, \theta.\mathrm{cuts}, \theta.\mathrm{tree}), & x \in \mathrm{Lhs}_p \text{ and } \forall k.A_k \notin \{\bot_\zeta, \top_\zeta\} \\ [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow x[A_1,\ldots,A_n]\;], & \forall k.A_k \notin \{\bot_\zeta, \top_\zeta\} \\ [\;\langle\,\rangle \mid \mathrm{true} \Rightarrow \top_\delta\;], & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } (y^{\mathrm{itr}*}, \mathrm{cuts}, \mathrm{tree}) = \sigma^\#(x), \quad \theta = [\; y_k^{\mathrm{itr}} \mapsto A_k \;]_{k=1}^n \text{ and } \quad A_i = A^\#\,[\![c_i]\!]\;\sigma^\#
\end{array}
$$

Fig. 10. Abstract semantics of Prexo IR expressions. $\uplus$ denotes tuple concatenation, and Lhs_p denotes the set of program variables assigned in p. The notation $a?b:c$ is a shorthand for a case split: "if a then b else c".

$$
\begin{array}{l}
S^\# : \mathrm{Stmt} \to \Sigma^\# \to \Sigma^\# \\
\qquad S^\#\,[\![s_1; s_2]\!]\;\sigma^\# \;=\; S^\#\,[\![s_2]\!]\;(S^\#\,[\![s_1]\!]\;\sigma^\#) \\
S^\#\,[\![\; y = \lambda\imath^*.(\,b\,?\,c_t : c_f\,)\;]\!]\;\sigma^\# \;=\; \sigma^\#[\; y \mapsto \tau_{\mathrm{new}}^{y\,\prime}\;] \\
\qquad \tau_t = C^\#\,[\![c_t]\!]\;\sigma^\#, \quad \tau_f = C^\#\,[\![c_f]\!]\;\sigma^\#, \quad \Pi = B^\#\,[\![b]\!]\;\sigma^\# \\
\qquad \tau_{\mathrm{new}}^{y} = \mathrm{CAD}_\bot^\top(\mathrm{vars}, \mathrm{cuts}), \quad \mathrm{vars} = \tau_t.\mathrm{vars} \uplus \tau_f.\mathrm{vars} \uplus \imath^*, \quad \mathrm{cuts} = \tau_t.\mathrm{cuts} \uplus \tau_f.\mathrm{cuts} \uplus \Pi \\
\qquad \tau_{\mathrm{new}}^{y\,\prime} = \tau_{\mathrm{new}}^{y}\big[\mathrm{Leaf}(\_, \hat{n}) \mapsto \mathrm{Leaf}(v, \hat{n})\big], \text{where } v = \text{if } \mathcal{E}_b\,[\![b]\!]\;(\{\imath_k \mapsto \hat{n}_k\}) \text{ then } \tau_t(\hat{n}) \text{ else } \tau_f(\hat{n}) \\
S^\#\,[\![\; x = \lambda\imath^*.\lambda\eta^*.(\,b\,?\,e_t : e_f\,)\;]\!]\;\sigma^\# \;=\; \sigma^\#[\; x \mapsto \tau_{\mathrm{new}}^{x\,\prime}\;] \\
\qquad \tau_t = D^\#\,[\![e_t]\!]\;\sigma^\#, \quad \tau_f = D^\#\,[\![e_f]\!]\;\sigma^\#, \quad \Pi = B^\#\,[\![b]\!]\;\sigma^\# \\
\qquad \tau_{\mathrm{new}}^{x} = \mathrm{CAD}_\bot^\top(\mathrm{vars}, \mathrm{cuts}), \quad \mathrm{vars} = \tau_t.\mathrm{vars} \uplus \tau_f.\mathrm{vars} \uplus \imath^* \uplus \eta^*, \quad \mathrm{cuts} = \tau_t.\mathrm{cuts} \uplus \tau_f.\mathrm{cuts} \uplus \Pi \\
\qquad \tau_{\mathrm{new}}^{x\,\prime} = \tau_{\mathrm{new}}^{x}\big[\mathrm{Leaf}(\_, \hat{n}) \mapsto \mathrm{Leaf}(v, \hat{n})\big], \text{where } v = \text{if } \mathcal{E}_b\,[\![b]\!]\;(\{\imath_k \mapsto \hat{n}_k\}) \text{ then } \tau_t(\hat{n}) \text{ else } \tau_f(\hat{n})
\end{array}
$$

Fig. 11. Abstract transfer $S^\#$ for the Prexo IR. We define $\mathrm{CAD}_\bot^\top$ as an extension of the standard CAD: it returns $\top_{\mathcal{T}\langle\pi\rangle}$ if any cuts are $\bot_\zeta$ or $\top_\zeta$, and otherwise follows the normal CAD algorithm. Leaf values are initialized to $\top_\zeta$ or $\top_\delta$ by default. The notation $\tau(\hat{n})$ is introduced in Definition 5.2.

whenever the indices are affine ($A_i = A^\#[\![c_i]\!]\;\sigma^\#$), thereby preserving relational information among different accesses to the same input array; if the indices are not affine we return a $\top_\delta$.

The *transfer function* $S^\# : \mathtt{Stmt} \to \Sigma^\# \to \Sigma^\#$ is defined in Figure 11, which updates the environment by building a CAD over variables and cuts, then resolving leaves by the guard. As shown in Figure 11, for control $y = \lambda\imath^*.(b?c_t : c_f)$ and data $x = \lambda\imath^*.\lambda\eta^*.(b?e_t : e_f)$ assignments, we evaluate the branches to CAD-trees ($\tau_t = C^\#\,[\![c_t]\!]\;\sigma^\#$ or $D^\#\,[\![e_t]\!]\;\sigma^\#$; similarly τ_f) and transfer the guard to *cuts* $\Pi = B^\#\,[\![b]\!]\;\sigma^\#$. We then form $\tau_{\mathrm{new}} = \mathrm{CAD}_\bot^\top(\mathrm{vars}, \mathrm{cuts})$ with $\mathrm{vars} = \tau_t.\mathrm{vars} \uplus \tau_f.\mathrm{vars} \uplus \imath^*$ (and $\uplus\,\eta^*$ for data) and $\mathrm{cuts} = \tau_t.\mathrm{cuts} \uplus \tau_f.\mathrm{cuts} \uplus \Pi$; each leaf at a sample point $\hat{n}$ is set to $\tau_t(\hat{n})$ if $\mathcal{E}_b\,[\![b]\!]\;(\hat{n})$ else $\tau_f(\hat{n})$. According to the above definitions of the evaluation functions and the transfer function $S^\#$, we prove the soundness of the transfer function $S^\#$ in Theorem D.1 in Appendix D.

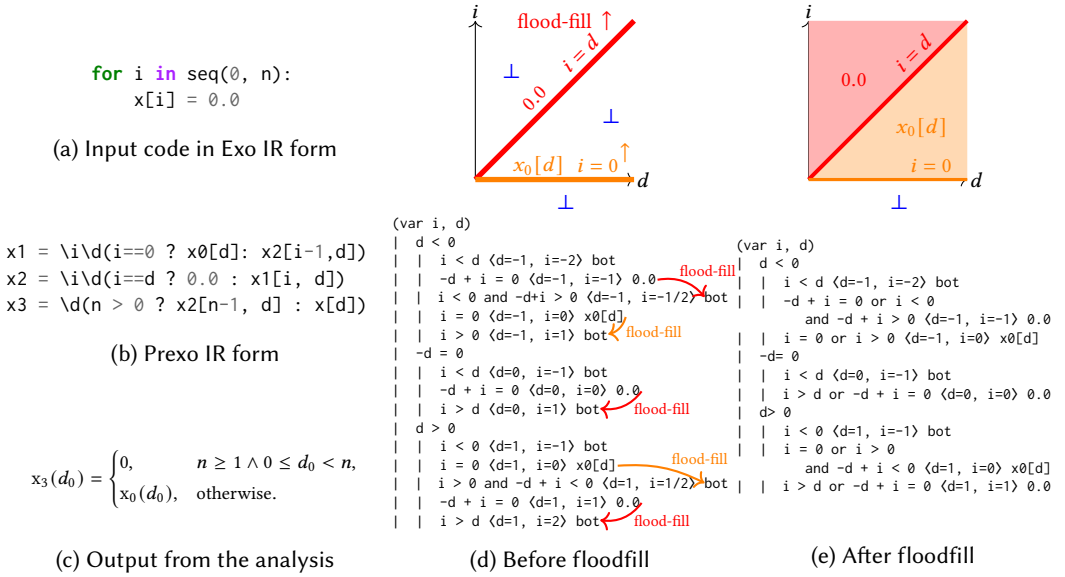


Fig. 12. Visualization of the floodfill widening on Array Init 1D example

Definition 5.9 (Abstract transfer on programs)**.** For a program $p = \mathrm{Fix}\, s$ and an abstract initial store $\sigma^{\#0} \in \Sigma^{\#}$, we define the *abstract program transfer* $P^{\#}\,[\![p]\!]\,(\sigma^{\#0}) = \sigma^{\#*}$ by the fixed point

$$\sigma^{\#*} \;=\; \mathrm{lfp}\big(F^{\#}\big), \quad \text{where} \quad F^{\#}(\sigma^{\#}) \;\triangleq\; \sigma^{\#0} \;\sqcup_{\Sigma^{\#}}\; S^{\#}\,[\![s]\!]\;(\sigma^{\#})$$

Since $\Sigma^{\#}$ is a complete lattice and $S^{\#}$ is monotone, $F^{\#}$ is monotone; hence the fixed points exist by Tarski–Knaster. We prove the soundness of the transformer $P^{\#}$ in Theorem E.2 in Appendix E.

5.4 Widening

In this subsection, we introduce an efficient "flood-fill" widening operator $\nabla : \Sigma^{\#} \times \Sigma^{\#} \to \Sigma^{\#}$ that *accelerates* the fixpoint computation of the abstract program transfer (Definition 5.9). The operator safely extrapolates joins, $(\sigma_a^{\#} \sqcup_{\Sigma^{\#}} \sigma_b^{\#}) \;\sqsubseteq_{\Sigma^{\#}}\; (\sigma_a^{\#} \nabla \sigma_b^{\#})$ for all $\sigma_a^{\#}, \sigma_b^{\#} \in \Sigma^{\#}$, and it enforces finite convergence: for any initial abstract state $\widetilde{\sigma}_0^{\#} \in \Sigma^{\#}$, a sequence $\widetilde{\sigma}_{i+1}^{\#} = \widetilde{\sigma}_i^{\#} \nabla F^{\#}(\widetilde{\sigma}_i^{\#})$ is an ascending chain and stabilizes after finitely many steps.

Definition 5.10 (Widening $\nabla : \Sigma^{\#} \times \Sigma^{\#} \to \Sigma^{\#}$)**.** $X \nabla Y \;\triangleq\; \mathrm{Floodfill}\big(X \;\sqcup_{\Sigma^{\#}}\; Y\big)$.

To build intuition for the floodfill algorithm used in our widening, we present a concrete example in Figure 12. The figure demonstrates the analysis of a simple one-dimensional array initialization program. Figure 12a shows the original Exo IR code, which initializes elements of an array x to zero for indices 0 to $n-1$. This code is transformed into Prexo IR shown in Figure 12b, where i denotes the loop iterator, d represents the array dimensions, and `x0` is the initial value of x at the beginning of the procedure. `x1` through `x3` represent different versions of the same array variable x at distinct program points: `x1` corresponds to the loop start, `x2` to the assignment, and `x3` to the loop exit node. The analysis produces the final abstract interpretation output shown in Figure 12c, which precisely captures that array elements in the range $[0, n)$ are set to zero when $n \geq 1$. Figure 12d illustrates the abstract domain value for x_2 after the first fixpoint iteration, with both a visualization (top diagram) and the corresponding CAD tree representation (bottom diagram). In the visualization, the red line represents cells where $i = d$ with value 0.0, while the orange line represents cells where $i = 0$ with value $x_0[d]$; all other cells contain $\bot_{\delta}$. The flood-fill operation (shown as arrows) then propagates these values "upward in time", from *sections* to *sectors* immediately following them, resulting in the filled regions shown in Figure 12e.

```
for i in seq(0, 10):
    x[i] = 0.0
#  ..................  {x3 : τ3}
for i in seq(0, 10):
    x[i] = 0.0
#  ..................  {x6 : τ6}
...
```

(a) Exo IR code example where s can be removed.

```
for i in seq(0, 10):
    x[i] = -x[i]
#  ..................
for i in seq(0, 10):
    x[i] = x[i] + 2.0
#  ..................
...
```

(b) Exo IR code example where s cannot be removed.

```
x_1 =\i \d0 (i == 0 ? x[d0] : x_2[i - 1, d0]) # LoopStart
x_2 =\i \d0 (i == d0 ? 0.0 : x_1[i, d0])
x_3 =\d0 (10 > 0 ? x_2[10 - 1, d0] : x[d0]) # LoopExit
x_4 =\i_1 \d0 (i_1 == 0 ? x_3[d0] : x_5[i_1 - 1, d0]) # LoopStart
x_5 =\i_1 \d0 (i_1 == d0 ? 0.0 : x_4[i_1, d0])
x_6 =\d0 (10 > 0 ? x_5[10 - 1, d0] : x_3[d0]) # LoopExit
...
```

(c) Prexo IR of (a)

Fig. 13. Deleting the second loop in (a) is correct, while (b) is incorrect. $\{x_3 : \tau_3\}$ and $\{x_6 : \tau_6\}$ represents Prexo IR program environment (variable-abstract value pair) at that program location.

Definition 5.11 (Flood-fill). Fix $\pi \in \{\zeta, \delta\}$ and the flat base domain $\mathcal{B}_\pi$. For $(\sigma_c^\#, \sigma_d^\#) \in \Sigma^\#$ define pointwise $\mathrm{Floodfill}\big(\sigma_c^\#, \sigma_d^\#\big) \triangleq \big(\, y \mapsto \mathrm{Floodfill}_t\big(\sigma_c^\#(y)\big)\,,\ x \mapsto \mathrm{Floodfill}_t\big(\sigma_d^\#(x)\big)\,\big)$, where $\mathrm{Floodfill}_t : \mathcal{T}\langle\pi\rangle \to \mathcal{T}\langle\pi\rangle$ is the node-level flood-fill defined as follows. Let $\mathrm{Floodfill}_t(\bot_{\mathcal{T}\langle\pi\rangle}) = \bot_{\mathcal{T}\langle\pi\rangle}$ and $\mathrm{Floodfill}_t(\top_{\mathcal{T}\langle\pi\rangle}) = \top_{\mathcal{T}\langle\pi\rangle}$. For $\tau = (\mathrm{vars} : \iota^*\eta^*,\ \mathrm{cuts} : \zeta^*,\ \mathrm{tree} : n)$, define

$$\mathrm{Floodfill}_t(\tau) \triangleq (\mathrm{vars} : \iota^*\eta^*,\ \mathrm{cuts} : \zeta^*,\ \mathrm{tree} : m) \quad \text{where} \quad (m, \_) = \mathrm{Floodfill}_n(n,\ \bot_\pi).$$

Define $\mathrm{Floodfill}_n : \ node\langle\pi\rangle \times \mathcal{B}_\pi \ \to \ node\langle\pi\rangle \times \mathcal{B}_\pi$ as

$$\mathrm{Floodfill}_n\big(\mathrm{Leaf}(\ell, \mathbf{s}),\ c\big) \triangleq \big(\mathrm{Leaf}(\ell', \mathbf{s}),\ \ell'\big) \quad \text{with} \quad \ell' = (\ell = \bot_\pi)\ ?\ c\ :\ell$$

$$\mathrm{Floodfill}_n(\mathrm{LinSplit}\big((g_i, n_i)_{i=1}^k\big), c) \triangleq \big(\ \mathrm{LinSplit}\big(\mathrm{Cell}(g_1, m_1), \ldots, \mathrm{Cell}(g_k, m_k)\big)\ ,\ c_k\ \big).$$

where $c_0 \triangleq c$ and $(m_i, c_i) \triangleq \mathrm{Floodfill}_n(n_i, c_{i-1})\ \ (i = 1, \ldots, k)$. $\mathrm{Floodfill}_t$ preserves `vars`/`cuts` and overwrites only $\bot_\pi$-payload leaves in the tree; sample points $\mathbf{s}$ are preserved.

Theorems F.3 and F.6 in Appendix F establish that ∇ soundly over-approximates the join and that the iteration $\widetilde{\sigma}_{i+1}^\# = \widetilde{\sigma}_i^\# \nabla F^\#(\widetilde{\sigma}_i^\#)$ stabilizes in finitely many steps.

6 INTEGRATING ABSTRACT VALUES TO AN IMPERATIVE USL

6.1 Analysis API

This subsection introduces the analysis API `Check_Delete(p, s)`, which determines whether a statement s can be safely removed from program p. Consider the examples in Figures 13a and 13b. In (a), loop statement s writes 0.0 to array x, duplicating the value already written by the previous loop. Since this is redundant, statement s can be safely removed. In contrast, (b) shows a case where s assigns a different value to x, making deletion unsafe as it would alter program behavior.

Figure 13c shows the Prexo IR of example (a). In this representation, x_3 denotes the program variable for the buffer x before executing statement s, while x_6 denotes the variable after s. The corresponding abstract values are τ_3 for x_3 and τ_6 for x_6. This can be expressed using Hoare logic notation as $\{x_3 : \tau_3\}\ s\ \{x_6 : \tau_6\}$, where the pre- and post-conditions represent the program variable-abstract value pairs before and after executing s (as illustrated in Figure 13a). To determine whether a statement s can be safely deleted, we must verify that all buffers modified by s retain identical values before and after its execution. Since statement s modifies only buffer x, the safety check reduces to verifying whether the program variables x_3 and x_6 are equivalent.

Scheduling operation	Code transformation	Safety conditions
`delete(p, s)`	s1 s ⇝ s2 → s1 s2	`Check_Delete(p, s)` holds.
`insert(p, s)`	s1 → s1 s2 ⇝ s s2	`Check_Delete(p',s)` holds, where `p' = p[s1;s2 ↦ s1;s;s2]`.
`bind(p, e, v)`	s ⇝ v=e s[e↦v]	`Check_Delete(p',v=e)` holds, where `p' = p[s ↦ v=e;s]`.

Fig. 14. The list of new scheduling operators introduced by Prexo.

To determine whether the program variables x_3 and x_6 are equivalent, we compare their abstract values τ_3 and τ_6 using the lifting function $\mathcal{L}^\#$ (Definition 5.7). We require that for every Exo IR buffer modified in s, where $r_{\text{pre}} \in \mathbb{X} \cup \mathbb{Y}$ and $r_{\text{post}} \in \mathbb{X} \cup \mathbb{Y}$ denote the Prexo IR program variable before and after s, and where corresponding abstract values are given by $\tau_{\text{pre}} = \sigma^\# [\![r_{\text{pre}}]\!]$ and $\tau_{\text{post}} = \sigma^\# [\![r_{\text{post}}]\!] \in \mathcal{T}_{\langle\pi\rangle}$, the following condition must hold: if both $\mathcal{L}^\#(r_{\text{pre}}, \tau_{\text{pre}})$ and $\mathcal{L}^\#(r_{\text{post}}, \tau_{\text{post}})$ hold, then the abstract value must be unchanged by s, that is, $r_{\text{pre}} = r_{\text{post}}$. Formally, the safety condition of `Check_Delete` can be writen as:

$$\mathcal{L}^\#(r_{\text{pre}}, \tau_{\text{pre}}) \wedge \mathcal{L}^\#(r_{\text{post}}, \tau_{\text{post}}) \implies (r_{\text{pre}} = r_{\text{post}}).$$

Applying this to our deletion problem: if x_3 and x_6 are equivalent under this definition, then statement s produces no observable change to variable x, making it a candidate for safe deletion. This condition ensures that if the pre- and post-abstract values of x concretize to the same concrete values, then deleting s will not alter the value of x in the concrete environment. We use `Check_Delete(p, s)` to denote that the formula above is verifiable for a procedure p and a statement s.

In our implementation, $\mathcal{L}^\#$ translates CAD trees into first-order formulas that we discharge to an SMT solver. By construction, the image of $\mathcal{L}^\#$ stays within linear arithmetic: control expressions are closed under affine operators (no variable–variable products; divisions are by constants), guards compare affine terms with $\{=, <, \leq\}$ and combine by conjunction while LinSplit yields only finite disjunctions, and array reads $x[\zeta^*]$ are renamed to fresh scalars via the injective ν^D_{new}, eliminating arrays/UFs. This yields a formula in QF-LIA, for which SMT solvers provide complete decision procedures; hence every query we emit is decidable.

6.2 Scheduling Operations

Scheduling in the USL context refers to a set of meta-programming operators that transform the object code, while preserving semantic equivalence. On top of the existing scheduling operations of Exo [23], Prexo introduces three scheduling operations that leverage value-sensitive analysis to enable transformations that were unsupported or unsafe in Exo. Figure 14 shows these operations: `delete`, `insert`, and `bind`. The `delete` operation removes a statement `s` from program point `p` when `Check_Delete(p, s)` holds. As demonstrated in Section 6.1, this safety condition verifies that no subsequent computations depend on values produced by `s`, ensuring the removal preserves program semantics. The `insert` operation adds a statement `s` at program point `p`, with safety condition `Check_Delete(p', s)` where `p'` represents the transformed program containing `s`. This condition reuses the deletion check to ensure the inserted statement is redundant (i.e., without altering the program semantics) and therefore safe to add. The `bind` operation introduces a variable binding `v = e` and replaces all occurrences of expression `e` with `v`. It requires that the binding itself be deletable in the resulting program. These operations enable scheduling transformations while maintaining semantic equivalence through value-sensitive analysis, pushing the boundary of safe program transformations with imperative USLs.

```
x[0] = 2.0
if n < 3:
    x[n] = 3.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 3, & n < 3 \wedge d_0 = n, \\ 2, & d_0 = 0 \wedge n \neq 0, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(a) Simple Conditional

```
for i in seq(0, n):
    x[i] = 0.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 0, & n \geq 1 \wedge 0 \leq d_0 < n, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(b) Array Init 1D [11, 15, 30]

```
for i in seq(0, 10):
    for j in seq(0, 20):
        x[i] = 2.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 2, & d_0 \leq 9, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(c) Array Init 1D Multi-loop

```
for i in seq(0, 10):
    for j in seq(0, 20):
        x[i, j] = 0.0
```

$$\mathrm{x_o}(d_0, d_1) = \begin{cases} 0, & 0 \leq d_0 < 10 \\ & \wedge\ 0 \leq d_1 < 20, \\ \mathrm{x_i}(d_0, d_1), & \text{otherwise.} \end{cases}$$

(d) Array Init 2D Const

```
for i in seq(0, n):
    for j in seq(0, m):
        x[i, j] = 0.0
```

$$\mathrm{x_o}(d_0, d_1) = \begin{cases} 0, & 0 \leq d_0 < n \\ & \wedge\ 0 \leq d_1 < m, \\ \mathrm{x_i}(d_0, d_1), & \text{otherwise.} \end{cases}$$

(e) Array Init 2D Non-Const [30]

```
for i in seq(0, n):
    x[i] = y[i]
```

$$\mathrm{x_o}(d_0) = \begin{cases} \mathrm{y}(d_0), & n > 0 \wedge 0 \leq d_0 < n, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(f) Array Copy [15, 19, 30]

```
for i in seq(0,10):
  for j in seq(0,20):
    x[i + j] = 2.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 2, & d_0 \leq 28, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(g) SW Const

```
for i in seq(0,10):
  for j in seq(0,20):
    if i==0 or j==19:
      x[i + j] = 2.0
```

(h) SW Guard

```
for i in seq(0, n):
    for j in seq(0, m):
        x[i + j] = 2.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 2, & d_0 \leq n + m - 2, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(i) SW Non-Const

```
for i in seq(0, 10):
    x[i] = 2.0
for i in seq(0, 5):
    x[i] = 0.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 0, & d_0 \leq 4, \\ 2, & 4 < d_0 \leq 9, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(j) Array Loop Fuse

```
for i in seq(lo, hi):
    x[i] = 0.0
```

$$\mathrm{x_o}(d_0, lo, hi) = \begin{cases} 0.0, & lo \leq d_0 < hi, \\ \mathrm{x_i}[d_0], & \text{otherwise.} \end{cases}$$

(k) Array Slice Init [30]

```
for i in seq(0, N):
    x[N - i - 1] = 3.0
    x[i] = 1.0
```

$$\mathrm{x_o}(d_0, N) = \begin{cases} 3.0, & N > 0 \wedge 0 \leq d_0 < \lfloor N/2 \rfloor, \\ 1.0, & N > 0 \wedge \lfloor N/2 \rfloor \leq d_0 < N, \\ \mathrm{x_i}[d_0], & \text{otherwise.} \end{cases}$$

(l) Array Reverse Const-Write

```
for i in seq(0, 11):
    x[10-i] = y[10-i]
    x[i] = y[i]
```

$$\mathrm{x_o}(d_0) = \begin{cases} \mathrm{y}[d_0], & 0 \leq d_0 \leq 10, \\ \mathrm{x_i}[d_0], & \text{otherwise.} \end{cases}$$

(m) Array Reverse Array-Write

Fig. 15. The micro-benchmark programs and the array contents at the last program locations ($\mathrm{x_o}$). $\mathrm{x_i}$ represents the initial array content before entering the loops. SW in (g), (h), and (i) indicates Sliding Window.

7 EVALUATION

7.1 Micro-Benchmark Evaluation

Evaluation Setting. We evaluated the expressiveness and runtime performance of Prexo on microbenchmarks (Figure 15, Table 1) drawn from prior work [11, 15, 19, 30] as well as manually crafted programs. Figure 15 shows the input Exo IR and ground truth array contents at program termination. These programs exhibit diverse features that challenge static analyzers: branches (br) require state spliting and joining; single/multi-loops (sl, ml) require widening operators for termination; multi-loops (ml) and multi-dimensional arrays (md) induce case explosion in the number of symbolic cases that the analysis must track; array relations (ar) require reasoning about dependencies among multiple unbounded arrays with unknown initial values; array loop fusion (af) necessitates distinguishing overlapping writes across sequential loops; sliding window access (sw), common in image processing and stencil computations, demands precise reasoning to verify the functional equivalence between guarded and unguarded loop bodies. Finally, reverse access (ra) requires introducing new partitions like $\lfloor N/2 \rfloor$, motivating our CAD abstract domain design.

For each benchmark, we report the maximum number of variables (#vars), cuts (#cuts), and generated cells (#cells) to quantify computational complexity of CAD. We compare Prexo against two

Micro-benchmarks	Traits	Prexo	#vars	#cuts	#cells	TVLA-Arr	Prexo-NW
a. Simple Conditional	br	0.16s	2	4	28	–	0.15s
b. Array Init 1D [11, 15, 30]	sl	0.61s	2	6	136	1.7s	0.51s ●
c. Array Init 1D Multi-loop	ml	13.78s	3	26	1,511	–	4.52s ●
d. Array Init 2D Const	ml,md	81.05s	4	24	17,190	–	56.82s ●
e. Array Init 2D Non-Const [30]	ml,md	1,243.3s	5	24	264,331	–	816.44s ●
f. Array Copy [15, 19, 30]	ar	2.3s	2	11	238	338.1s	0.61s ●
g. Sliding Window Const	ml,sw	38.85s	3	21	8,161	–	28.45s ●
h. Sliding Window Guard	ml,sw,br	172.38s	3	41	24,779	–	146.34s ●
i. Sliding Window Non-Const	ml,sw	714.79s	4	21	145,314	–	467.46s ●
j. Array Loop Fuse	af	1.16s	2	15	192	–	0.98s ●
k. Array Slice Init [30]	sl	8.03s	3	6	2895	–	5.39s ●
l. Array Reverse Const-Write	ra	50.53s	3	9	13123	–	42.48s ●
m. Array Reverse Array-Write	ra,ar	17.97s	2	18	1354	–	2.73s ●

Table 1. The statistics of Prexo across micro-benchmarks

baselines: TVLA-Arr [15], an extension of TVLA supporting numeric array abstract interpretation, and Prexo-NW, a simplified variant of Prexo that replaces our widening with trivial flood-fill returning $\top_{\mathcal{T}_{\langle\pi\rangle}}$. We exclude Exo v1, which lacks array semantic reasoning entirely. While other studies on abstract interpretation [11, 19, 30] exist, they provide neither artifacts nor analysis result for comparison. All evaluations were conducted on a MacBook Pro with an Apple M1 Pro chip and 16 GB of unified memory, running macOS Sonoma (version 14.4).

Result. Prexo achieves the desired precision with acceptable runtime overhead. For each micro-benchmark, the computed abstract state exactly matches the ground truth shown in Figure 15, with ten out of thirteen programs analyzed within 100 seconds. The analysis runtime scales roughly linearly with the number of CAD cells generated (Table 1). Processing rates remain consistent across problem scales: 223 cells/s for 136 cells (0.61s total), 212 cells/s for 17,190 cells (81.05s total), and 212 cells/s for 264,331 cells (1243.3s total). Since TVLA-Arr [15] provides concrete results only for Array Init 1D and Array Copy, we compare these against ground-truth in Figure 15. For Array Init 1D, TVLA-Arr requires 1.7 seconds while Prexo completes in 0.61 seconds with the same expressivity. For Array Copy, TVLA-Arr requires 338.1 seconds while Prexo completes in only 2.3 seconds. While different benchmark architectures preclude direct comparison, relative performace is instructive: Array Copy shows a 200× slowdown compared to Array Init 1D under TVLA-Arr, but only 3.7× under Prexo, indicating better scalability.

To the best of our knowledge, no existing analysis can precisely handle micro-benchmarks (l) and (m), as syntactic segmentation cannot introduce new partitions (such as $\lfloor N/2 \rfloor$). Prexo consructs such partitions trivially during CAD's lifting phase. The Prexo-NW column shows the ablation variant's runtime. Except for Simple Conditional, Prexo-NW fails to derive precise array contents because its trivial flood-fill aggressively sets the state to $\top_{\mathcal{T}_{\langle\pi\rangle}}$, losing precision. Compared to Prexo-NW, Prexo incurs only a modest average runtime increase of ~2×, while preserving full precision, indicating that CAD tree construction dominates the computational cost, not widening.

7.2 Case Study: RVV Kernel Optimization

Preliminaries. We evaluate Prexo on real-world performance optimization scenarios with RISC-V's Vector Extension (RVV). RVV, like Arm's SVE, adopts a vector-length-agnostic SIMD model where instructions scale to any hardware-provided vector width, eliminating architecture-specific tail code without sacrificing performance. Performance engineers targeting RVV must account for VLEN, an implementation-defined constant that specifies the bit width of each vector register. At runtime, `vsetvli`/`vsetvl` instructions return *vl*—the number of elements fitting into one register group for the chosen element size and LMUL factor. The loop body repeats until *vl* naturally

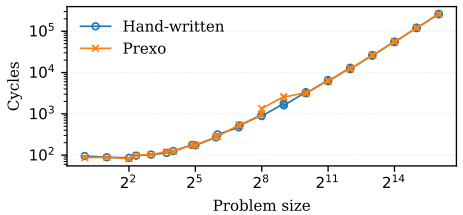
```
for io in seq(0, (n+31)/32):
    for ii in seq(0, min(32, n-32*io)):
        y[32*io+ii] += x[32*io+ii] * a
```

(a) Exo IR code before binding `rvv.vl`

```
for io in seq(0, (n+31)/32):
    rvv.vl = min(32, n-32*io)
    for ii in seq(0, rvv.vl):
        y[32*io+ii] += x[32*io+ii] * a
```

(b) Exo IR code after binding `rvv.vl`

Fig. 16. DAXPY kernel (a) before and (b) after applying `bind`. `rvv.vl` is a globally persistent configuration state. To ensure sound transformation, it is essential to verify that writing a new value to this state preserves functional equivalence. VLMAX = 32 elements for VLEN = 256 bits, SEW = 64 bits, and LMUL = 8.



(a) Prexo vs. expert-written runtime.

Kernel	NCD	#vars	#cuts	#cells	Prexo
DAXPY	1.61s	3	6	93	2.28s
SAXPY	1.62s	3	6	93	2.36s

(b) End-to-end compilation time of Prexo-NCD vs. Prexo.

Fig. 17. DAXPY kernel optimization results: performance and compilation time comparison.

shrinks to finish the workload [40]. Vector-length-agnostic architectures pose unique challenges for compilers. RVV programmers typically hand-write assembly using a "strip-mining" style that tracks remaining elements, decrements the count by *vl* each iteration, and exits when it reaches zero. However, compilers lack static visibility into elements per iteration or loop termination, and `vsetvl` appears as an opaque function call unless explicitly modeled in software. This creates significant challenges for USLs, which must ensure optimized RVV code processes the same `n` elements as the original code—requiring compiler analysis to track *vl* and determine elements per iteration. Exo and other USLs relying on dependency analysis are inadequate for this. To our knowledge, no USLs currently support RVV safely. To illustrate this, consider transforming code from (a) to (b) in Figure 16. While the transformation itself is straightforward, the analysis is non-trivial. Since `rvv.vl` is a global parameter, we must handle potential writes elsewhere (e.g., at the end of the `ii` loop) and ensure all uses remain consistent while processing every array element.

Evaluation Setting. We optimize DAXPY (standard kernel in BLAS [20], double-precision a-x-plus-y) using the `bind` scheduling operation from Section 6.2, which internally calls `Check_Delete`, where we integrated our analysis. We use Exo's scheduling operations for other optimizations, since dependency analysis suffices for those transformations. We run the abstract interpretation only for variables requested by the scheduling operation—in this case, just `rvv.vl`, not `y` or `x`. We benchmark performance on Saturn, a parameterized RVV implementation [40], using Verilator cycle counts with `GENV256D128ShuttleConfig` (a dual-issue core with 256-bit VLEN, 128-bit SIMD datapath, and separate floating-point and integer vector units). We compare the optimized DAXPY kernel against expert-written assembly from Saturn benchmarks [41] and an ablation variant, Prexo-NCD, which bypasses `Check_Delete` analysis to measure the API's overhead. Since standard Exo lacks safety checking for `bind`, Prexo-NCD effectively represents vanilla Exo applied to RVV.

Result. Figure 17a shows that Prexo-optimized DAXPY code matches the performance of hand-written assembly because Prexo supports the same key optimizations: *vl* binding, staging intermediate operations in RVV registers, and generating RVV intrinsic calls. Table 17b compares end-to-end compilation times between Prexo-NCD and full Prexo. Although Prexo compiles approximately 40% slower, its compilation time remains practical since dataflow analysis runs only on `rvv.vl`, keeping the number of variables and cells far below the values reported in Table 1.

8 RELATED WORK

8.1 Array Reasoning

Reasoning about programs that manipulate arrays poses a fundamental challenge in static analysis [12, 14, 15, 37]. Arrays may have unbounded lengths, and loops can execute an unbounded number of iterations, complicating the inference of properties such as the absence of out-of-bounds accesses [17, 38], invariants over array contents [12, 13], and loop termination [37]. Over the past two decades, a wide range of techniques have been developed to analyze arrays and loops in a sound manner. Abstract interpretation typically offers a general static analysis framework by interpreting program constructs over abstract domains [10]. Several frameworks summarize sections of arrays within numeric domains [15], while others partition arrays into segments to track segment-specific properties [11]. Also, symbolic abstract domains have been proposed for array reasoning to better capture memory updates [12, 13], and non-contiguous array partitioning techniques have been designed to infer precise invariants even when similar elements are dispersed across an array [28]. Instead of targeting general pointers or containers, we target a domain-specific language that excludes pointer manipulations and restricts control flows to affine expressions. Languages with these restrictions are still expressive enough for optimizing many HPC kernels like BLAS [20], which includes widely-used linear algebra kernels like GEMM, GEMV, and SAXPY/DAXPY. The CAD tree domain and the flood-fill widening are designed to maximally exploit this domain-specificity.

The parameterized *decision-tree* abstract domain of Urban & Miné [37] is the closest in spirit to our use of trees: internal nodes test linear constraints and leaves range over a chosen numerical domain. They target conditional termination, where leaves denote (pieces of) ranking functions and widening is based on left-unification plus leaf extrapolation. In contrast, Prexo is designed around *array relations*. Prexo's leaves carry value information including symbolic reads of input arrays (e.g., $x[\zeta^*]$), which lets us relate results back to inputs without knowing concrete contents—precisely the kind of value sensitivity needed for array transforms. A second difference is how disjunctions arise: instead of growing trees only from syntactic tests, we build trees by CAD over iterator/offset variables. CAD introduces *semantic* splitters that may not appear in the code (e.g., the $\lfloor N/2 \rfloor$ boundary in our reverse kernels, Section 7.1), yielding strictly finer partitions and, in practice, more precise array-content invariants. Finally, whereas Urban & Miné widen by unifying tree shapes, we use the flood-fill widening on space–time arrays: after a join, we propagate leaf values forward along the iteration direction, efficiently carrying array values through the fixpoint. These choices specialize the tree structure and payload base domains to value-sensitive array analysis and explain the precision gains we observe on sliding-window and reverse-access patterns.

8.2 Polyhedral Compilers

Polyhedral compilers analyze nested loops using polyhedra to represent iteration spaces and dependencies, enabling affine transformations like loop fusion, tiling, and parallelization. These compilers ensure soundness by preserving all dependencies through dependency analysis that guarantees $T(s) \leq T(e)$ for each dependency edge $s \rightarrow e$, where T is the transformation. Some systems can automatically generate affine transformations [3, 4, 16], while others accept user-provided transformations [1].

In contrast to polyhedral compilers, our approach incorporates value-sensitive analysis that tracks not only *which* memory locations are accessed but also *what* values are written to those locations. While polyhedral frameworks focus on preserving dependencies between statement instances, they fundamentally lack the ability to reason about the actual data values being computed and stored. This limitation prevents classical polyhedral methods from performing optimizations such as redundant write elimination.

8.3 User Schedulable Languages

Functional v.s. Imperative. Halide [34, 35] has popularized USLs over the past decade, inspiring numerous domain-specific variants for machine learning, sparse computation, and tensor optimization [1, 7, 18, 22, 25, 36, 39]. These Halide-like USLs employ functional object code where algorithms are expressed as pure functions, providing elegant abstractions and concise scheduling operations. However, recent systems like Exo [23, 24] and Allo [6] have adopted imperative object code to make performance-critical details such as hardware states and memory instructions explicit, which are obscured in functional code. For instance, sliding window optimization can only be expressed *implicitly* in Halide, meaning that programmers must rely on Halide's backend to perform sliding window optimization implicitly during code generation, instead of being able to express it explicitly as a loop optimization. Yet the shift to imperative USLs introduces new challenges—state mutation in imperative programs complicates analysis and potentially limits applicable transformations.

Safety guarantees. Prior work in USLs has established various methods to guarantee functional equivalence between original and optimized code. In functional USLs, Halide lacks sound compiler analysis and relies on scheduling operation-specific compiler passes with interval analysis, while ATL provides foundational verification of scheduling transformations using Rocq [27]. In imperative USLs, Exo offers value-sensitive support but is limited to scalar variables and excludes many performance optimizations on arrays. Allo [6] uses an external equivalence checker based on symbolic execution [33], though this approach is restricted to handling bounded loops and lacks both termination guarantees for infinite loops and soundness guarantees.

Despite their differences, USLs share a fundamental constraint: each USL must prove the soundness of its supported transformations, with the scope of possible transformations inherently bounded by the capabilities of the language's analysis engine. Imperative USLs face additional challenges in reasoning about state mutations and side effects. The central challenge we address in this paper is value-sensitive analysis—determining when program statements can be safely modified without affecting program output. Prexo aims to extend imperative USLs' transformation expressiveness by incorporating a precise abstract value information into the safety checks of scheduling operations.

9 LIMITATIONS AND FUTURE WORK

We introduced a novel CAD-tree domain with an efficient flood-fill widening algorithm for value-sensitive analysis. Although we believe our analysis scales better than existing abstract interpretation-based techniques, runtime remains a significant slowdown compared to lightweight dependency analyses, which usually finish in milliseconds for thousands of lines of code. In our evaluation, the array initialization with 2D non-constant bounded loops required the longest execution time at 20 minutes. While we can restrict the analysis to specific variables of interest (such as `rvv.vl` in the evaluation), the overhead becomes prohibitive when applied to heavily nested multi-dimensional arrays, since the worst-case complexity of CAD is doubly exponential.

Emerging hardware features—such as Hopper GPUs' asynchronous TMA copy instruction and RVV bit masks—allow developers to define custom behaviors for out-of-bounds memory accesses (e.g., zero, NaN, or undisturbed), whose results propagate through subsequent computations. Accurately reasoning about these effects requires analysis that goes beyond dependency analysis; only value-sensitive approaches can ensure that special values (e.g., NaNs) do not silently contaminate valid data. Beyond our current RVV kernels, we envision applying this approach to sparse matrix multiplication and wavefront optimizations, where value-sensitive reasoning becomes even more critical, and generalizing our implementation to support arbitrary architecture vector lengths. We believe that value-sensitive analysis is not just a theoretical contribution, but a practical foundation for building reliable, high-performance systems across diverse architectures.

REFERENCES

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019* (Washington, DC, USA). IEEE, Piscataway, NJ, USA, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[2] Hans Bekic. 1984. Definable Operation in General Algebras, and the Theory of Automata and Flowcharts. In *Programming Languages and Their Definition*. Springer, 30–55.

[3] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–303.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[5] Christopher W. Brown. 2001. Improved Projection for Cylindrical Algebraic Decomposition. *Journal of Symbolic Computation* 32, 5 (2001), 447–465. https://doi.org/10.1006/jsco.2001.0463

[6] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. https://doi.org/10.1145/3656401

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[8] George E. Collins. 1976. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition: a synopsis. *SIGSAM Bull.* 10, 1 (Feb. 1976), 10–12. https://doi.org/10.1145/1093390.1093393

[9] George E. Collins. 1998. Quantifier Elimination by Cylindrical Algebraic Decomposition — Twenty Years of Progress. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, B. F. Caviness and J. R. Johnson (Eds.). Springer, Vienna, 8–23. https://doi.org/10.1007/978-3-7091-9459-1_2

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/1926385.1926399

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid updates: beyond strong vs. weak updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems* (Paphos, Cyprus) *(ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 246–266. https://doi.org/10.1007/978-3-642-11957-6_14

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 187–200. https://doi.org/10.1145/1926385.1926407

[14] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. 2019. Precise reasoning with structured time, structured heaps, and collective operations. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 157 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360583

[15] Denis Gopan, Thomas Reps, and Mooly Sagiv. 2005. A framework for numeric analysis of array operations. *SIGPLAN Not.* 40, 1 (Jan. 2005), 338–350. https://doi.org/10.1145/1047659.1040333

[16] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.* 22 (2012). https://api.semanticscholar.org/CorpusID:18533155

[17] Yiyuan Guo, Peisen Yao, and Charles Zhang. 2024. Precise Compositional Buffer Overflow Detection via Heap Disjointness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 63–75.

[18] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

[19] Nicolas Halbwachs and Mathias Péron. 2008. Discovering properties about arrays in simple programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 339–348. https://doi.org/10.1145/1375581.1375623

[20] Richard J. Hanson, Fred T. Krogh, and Charles L. Lawson. 1973. A proposal for standard linear algebra subprograms. https://api.semanticscholar.org/CorpusID:60683717

[21] H. Hong. 1990. An improvement of the projection operator in cylindrical algebraic decomposition. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (Tokyo, Japan) *(ISSAC '90)*. Association for Computing Machinery, New York, NY, USA, 261–264. https://doi.org/10.1145/96877.96943

[22] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* 38, 6 (2019), 201:1–201:16. https://doi.org/10.1145/3355089.3356506

[23] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[24] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 426–444. https://doi.org/10.1145/3669940.3707218

[25] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–29. https://doi.org/10.1145/3133901

[26] Avery Laird, Bangtian Liu, Nikolaj S. Bjørner, and Maryam Mehri Dehnavi. 2024. SpEQ: Translation of Sparse Codes using Equivalences. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1680–1703. https://doi.org/10.1145/3656445

[27] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization Via High-level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[28] Jiangchao Liu and Xavier Rival. 2015. Abstraction of Arrays Based on Non Contiguous Partitions. In *Proc. Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 282–299.

[29] Scott McCallum. 1998. An Improved Projection Operation for Cylindrical Algebraic Decomposition. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, B. F. Caviness and J. R. Johnson (Eds.). Springer, Vienna, 242–268. https://doi.org/10.1007/978-3-7091-9459-1_12

[30] David Monniaux and Francesco Alberti. 2015. A Simple Abstraction of Arrays and Maps by Program Translation. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–234.

[31] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 119–134. https://doi.org/10.1145/3572848.3577475

[32] David Michael Ritchie Park. 1969. Fixpoint Induction and Proofs of Program Properties. In *Machine Intelligence Volume 5*. Edinburgh University Press, 59–78.

[33] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 97–107. https://doi.org/10.1145/3626202.3637563

[34] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[36] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Piscataway, NJ, USA, 74–85. https://doi.org/10.1109/CGO.2017.7863730

[37] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 302–318.

[38] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities.. In *NDSS*, Vol. 20. 0.

[39] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491

[40] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanović. 2024. *The Saturn Microarchitecture Manual.* Technical Report UCB/EECS-2024-215. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-215.html

[41] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanović. 2024. The Saturn Vector Unit. https://github.com/ucb-bar/saturn-vectors/tree/master. Accessed: 2025-11-07.

A MONOTONICITY PROOF

Lemma A.1 (Monotonicity of expression semantics). *For every control expression c, $\mathcal{E}_c^e[\![c]\!] : \Sigma_c \to \mathbb{Z}_\perp^\top$ is monotone. For every data expression e, $\mathcal{E}_d^e[\![e]\!] : \Sigma_c \times \Sigma_d \to \mathbb{D}_\perp^\top$ is monotone. For every boolean expression b, $\mathcal{E}_b[\![b]\!] : \Sigma_c \to \mathbb{B}_\perp^\top$ is monotone.*

Proof. By structural induction. Constants are constant; variables are store lookups, hence monotone. Composite forms apply lifted operators/relations and the lifted array read, each monotone in its arguments by case analysis on $\perp \sqsubseteq v \sqsubseteq \top$ (strict in $\perp$, absorbing at $\top$). □

Lemma A.2 (Monotonicity of statement semantics). *For every statement s, $\mathcal{E}_c[\![s]\!] : \Sigma_c \to \Sigma_c$ is monotone, and $\mathcal{E}_d[\![s]\!] : \Sigma_c \times \Sigma_d \to \Sigma_d$ is monotone in both arguments.*

Proof. By induction on s. Sequential composition uses composition of monotone maps. For a control assignment $y = \lambda\imath^*.(b?c_t : c_f)$, the updated array a_y is defined pointwise via ite applied to monotone subterms (Lemma A.1), hence the store update is monotone. For a data assignment $x = \lambda\imath^*.\lambda\eta^*.(b?e_t : e_f)$, by pointwise order on arrays a_x the store update $(\sigma_c, \sigma_d) \mapsto \sigma_d[x \mapsto a_x]$ is monotone in both arguments. Control assignments leave σ_d unchanged and data assignments leave σ_c unchanged, which is trivially monotone. □

Lemma A.3 (Existence of the least fixed points). *Let $F_c(\sigma_c) \triangleq \sigma_c^0 \sqcup_c \mathcal{E}_c[\![s]\!](\sigma_c)$ and $F_d^{\sigma_c}(\sigma_d) \triangleq \sigma_d^0 \sqcup_d \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)$. Then $\mathrm{lfp}(F_c)$ exists, and for every σ_c, $\mathrm{lfp}(F_d^{\sigma_c})$ exists.*

Proof. By Lemma A.2, $\mathcal{E}_c[\![s]\!]$ and $\mathcal{E}_d[\![s]\!]$ are monotone; joins with fixed stores σ_c^0, σ_d^0 preserve monotonicity, so F_c and $F_d^{\sigma_c}$ are monotone. The environments Σ_c and Σ_d are complete lattices (flat lifts with $\perp, \top$ are complete; finite products over complete lattices are complete under pointwise order). By the Tarski-Knaster theorem, $\mathrm{lfp}(F_c)$ and, for each σ_c, $\mathrm{lfp}(F_d^{\sigma_c})$ exist. □

B PRELIMINARIES: CYLINDRICAL ALGEBRAIC DECOMPOSITION

Cylindrical algebraic decomposition (CAD) decomposes $\mathbb{R}^n$ into finitely many semi-algebraic cells where each input polynomial has constant sign [8, 9]. The algorithm takes as input a tuple of ordered variables (which we term "vars") $(y_1, \ldots, y_n)$ and a set of polynomials in those variables ("cuts") $F = \{f_1, \ldots, f_m\} \subset \mathbb{Q}[y_1, \ldots, y_n]$, and returns a CAD tree structure ("tree") that encodes the decomposition.

Definition B.1 (Cylindrical Algebraic Decomposition)**.** Let $F \subset \mathbb{Q}[y_1, \ldots, y_n]$ be a finite set of polynomials with variable ordering $y_1 \prec \cdots \prec y_n$.

Projection. Set $P_n := F$. For $k = n, \ldots, 2$ define

$$P_{k-1} \;=\; \Big\{ \mathrm{coeff}_{y_k}(p),\ \mathrm{disc}_{y_k}(p),\ \mathrm{res}_{y_k}(p, q) \;\Big|\; p, q \in P_k,\ p \neq q \Big\} \subset \mathbb{Q}[y_1, \ldots, y_{k-1}],$$

where $\mathrm{coeff}_{y_k}(p)$ denotes every coefficient of p viewed as a univariate in y_k, $\mathrm{disc}_{y_k}(p)$ is the discriminant of p with respect to y_k, and $\mathrm{res}_{y_k}(p, q)$ is the resultant of p and q with respect to y_k.

Lifting.

(1) (*Base*) Isolate the ordered real roots $r_1 \prec \cdots \prec r_s$ of the polynomials in P_1 and form $\mathcal{D}_1 = \big\{(-\infty, r_1), \{r_1\}, (r_1, r_2), \ldots, \{r_s\}, (r_s, \infty)\big\}$.
(2) (*Induction*) For $k = 2, \ldots, n$: for each $C \in \mathcal{D}_{k-1}$, choose a sample point $s_C \in C$ and substitute it into all $p \in P_k$. Let $\alpha_1 \prec \cdots \prec \alpha_m$ be the real roots of the resulting polynomials in y_k. Partition $C \times \mathbb{R}$ into:
   - *sections*: $C \times \{\alpha_i\}$ for $i = 1, \ldots, m$
   - *sectors*: $C \times (\alpha_{i-1}, \alpha_i)$ for $i = 0, \ldots, m + 1$

where $\alpha_0 = -\infty$ and $\alpha_{m+1} = \infty$. $\mathcal{D}_k$ is the collection of all these sections and sectors.

Output. $\mathrm{CAD}(F) := \mathcal{D}_n$, a finite partition of $\mathbb{R}^n$ into connected semi-algebraic cells that is *cylindrical* (the projection of any cell onto the first k coordinates is a union of cells of $\mathcal{D}_k$ for every $k < n$) and *sign-invariant* for F (each $p \in F$ has constant sign on each cell).

The algorithm first computes the projection sets $P_{n-1}, \ldots, P_1$ by successively eliminating the variables $y_n, \ldots, y_2$; each P_{k-1} collects all coefficients, discriminants, and pairwise resultants required to mark where the original polynomials can change sign in the lower-dimensional space. In the lifting phase, space is rebuilt coordinate by coordinate. Beginning with the line decomposition $\mathcal{D}_1$ obtained from the real roots of P_1, the algorithm iteratively refines: for every cell $C \in \mathcal{D}_{k-1}$ a sample point s_C is chosen, substituted into the polynomials of P_k, and the resulting real roots $\alpha_1 \prec \cdots \prec \alpha_m$ in y_k are isolated. These roots define *sections* $C \times \{\alpha_i\}$ and the open intervals between them called *sectors* $C \times (\alpha_{i-1}, \alpha_i)$, yielding $\mathcal{D}_k$. Repeating for $k = 2, \ldots, n$ produces the cylindrical family $\mathcal{D}_n$; its cells and their sample points form a rooted tree whose edges record how each higher-dimensional cell projects onto its parent, fully encoding the cylindrical structure.

Numerous projection operators have been introduced to reduce the size of the projection sets, such as McCallum's reduced projection [29] and Brown–McCallum's minimal projection [5]. We use Hong's projector [21] in Prexo, which guarantees correctness without additional well-orientedness conditions while often yielding smaller sets than Collins's original projector.

C SOUNDNESS OF α AND γ

Theorem C.1 (Soundness of α and γ). *For every set of concrete states $S \subseteq \Sigma$, $S \subseteq \gamma\big(\alpha(S)\big)$. Equivalently, for each $\sigma \in S$ we have $\sigma \models \mathcal{L}^{\#}\big(\alpha(S)\big)$.*

Proof. Let $S \subseteq \Sigma$ and fix $\sigma = (\sigma_c, \sigma_d) \in S$. Because $\mathcal{L}^{\#}$ conjoins the per-variable formulas (Def. 5.7), it suffices to prove, for each variable $r \in \mathbb{X} \cup \mathbb{Y}$, that

$$\sigma \models \mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}\big(r, \hat{\alpha}(\sigma)(r)\big),$$

where $\pi = \zeta$ for control variables and $\pi = \delta$ for data variables. The theorem will then follow by conjoining these per-variable facts and lifting from $\hat{\alpha}(\sigma)$ to $\alpha(S)$ via join-weakening.

Control $y \in \mathbb{Y}$. By the definition of α_c (Def. 5.6),

$$\alpha_c(\sigma_c)(y) \;=\; \bigsqcup_{\vec{i} \in \mathbb{Z}^{\mathrm{itr}(y)}} \Big[\, \vec{\imath} \;\Big|\; \bigwedge_r (\imath_r = i_r) \;\Rightarrow\; \Big\lceil \sigma_c(y)(\vec{i}) \Big\rceil_C \,\Big],$$

where $\vec{\imath}$ is the tuple of control variables and $\vec{i}$ ranges over integer tuples of the same arity. Applying the lifting $\mathcal{L}^{\#}$ (Fig. 9), a top-level LinSplit yields

$$\mathcal{L}^{\#}_{\mathcal{T}\langle\zeta\rangle}\big(y, \alpha_c(\sigma_c)(y)\big) \;=\; \bigvee_{\vec{i}} \Big(\mathcal{L}^{\#}_g\big(\textstyle\bigwedge_r (\imath_r = i_r)\big) \;\wedge\; \mathcal{L}^{\#}_{\zeta}\big(y, \Big\lceil \sigma_c(y)(\vec{i}) \Big\rceil_C\big) \Big).$$

Let $\vec{i}^{\sigma}$ be the concrete integer tuple chosen by σ (Def. 5.2). Then the guard $\bigwedge_r (\imath_r = i_r^{\sigma})$ is true under that valuation, whereas all guards for $\vec{i} \neq \vec{i}^{\sigma}$ are false. Hence exactly one disjunct remains relevant; its leaf payload is $\Big\lceil \sigma_c(y)(\vec{i}^{\sigma}) \Big\rceil_C$, and the leaf clause of $\mathcal{L}^{\#}$ yields a base constraint (e.g. $y = \sigma_c(y)(\vec{i}^{\sigma})$) that holds in σ by construction. Therefore $\sigma \models \mathcal{L}^{\#}_{\mathcal{T}\langle\zeta\rangle}\big(y, \alpha_c(\sigma_c)(y)\big)$.

Data $x \in \mathbb{X}$. If $x \notin \mathrm{Lhs}_p$ (a free variable), α_d produces a single *unguarded* leaf with payload $\lceil x[\vec{\eta}] \rceil_D$; the corresponding base clause in the lifting is tautological (e.g. $x[\vec{j}] = x[\vec{j}]$), so it holds in

every σ. Otherwise,

$$\alpha_d(\sigma_c,\sigma_d)(x) \;=\; \bigsqcup_{\vec{i},\vec{j}} \Big[\, \vec{\imath}\,\vec{\eta} \;\Big|\; \big(\bigwedge_r \imath_r = i_r\big) \wedge \big(\bigwedge_s \eta_s = j_s\big) \;\Rightarrow\; \Big\lceil \sigma_d(x)(\vec{i},\vec{j}) \Big\rceil_D \,\Big].$$

Let $(\vec{i}^{\sigma},\vec{j}^{\sigma})$ be the concrete indices selected by σ. As above, only the disjunct with that guarded pair can be true, and at its leaf the payload is exactly $\Big\lceil \sigma_d(x)(\vec{i}^{\sigma},\vec{j}^{\sigma}) \Big\rceil_D$, so the resulting atomic constraint holds in σ.

Combining the two cases, we obtain $\sigma \models \mathcal{L}^{\#}\big(\hat{\alpha}(\sigma)\big)$.

□

D SOUNDNESS OF $S^{\#}$

Theorem D.1 (Soundness of $S^{\#}$). *For every statement s, concrete stores $(\sigma_c,\sigma_d) \in \Sigma$ and abstract store $\sigma^{\#} \in \Sigma^{\#}$,*

$$\hat{\alpha}(\sigma_c,\sigma_d) \;\sqsubseteq_{\Sigma^{\#}}\; \sigma^{\#} \;\Longrightarrow\; \hat{\alpha}\big(\,\mathcal{E}_c[\![s]\!](\sigma_c),\; \mathcal{E}_d[\![s]\!](\sigma_c,\sigma_d)\,\big) \;\sqsubseteq_{\Sigma^{\#}}\; S^{\#}[\![s]\!]\,\sigma^{\#}.$$

Proof. We proceed by structural induction on s. We rely on the following standard properties (all hold for the given definitions; proofs are routine inductions and omitted for brevity).

(F1) *Pointwise soundness of $C^{\#}$:* If $\hat{\alpha}(\sigma_c,\sigma_d) \sqsubseteq \sigma^{\#}$ and c is a control expression, then for $\iota^* = \mathbb{FV}(c)$ and for every integer tuple $z^* \in \mathbb{Z}$,

$$\Big\lceil \mathcal{E}^{e}_{c}[\![c]\!]\big(\sigma_c[\iota^* \mapsto z^*]\big)\Big\rceil_C \;\sqsubseteq_{\mathcal{B}_C}\; (C^{\#}[\![c]\!]\,\sigma^{\#}[\iota^* \mapsto z^*])\langle\rangle.$$

When all control variables in the control expression maps to a concrete integer ($\iota^* \mapsto z^*$), all the branch in $C^{\#}[\![c]\!]\,\sigma^{\#}$ should be either trivially false or true, thus leading to only one leaf value. Therefore, applying an empty tuple $\langle\rangle$ yields one and only reachable base domain value. The proof is by induction on c, using the given clauses for $C^{\#}$ and $A^{\#}$.

(F2) *Componentwise update for $\hat{\alpha}$:* If $\hat{\alpha}(\sigma_c,\sigma_d) \sqsubseteq \sigma^{\#}$ and for some y we have a function $f : \mathbb{Z}^{\mathrm{itr}(y)} \to \mathbb{Z}^{\top}_{\bot}$ and a tree $\tau \in \mathcal{T}_C$ such that

$$\forall z^*.\; \lceil f(z^*)\rceil_C \;\sqsubseteq_{\mathcal{B}_C}\; \tau(z^*),$$

then $\hat{\alpha}\big(\sigma_c[y \mapsto f],\sigma_d\big) \;\sqsubseteq_{\Sigma^{\#}}\; \sigma^{\#}[y \mapsto \tau]$.

(Sequence) $s \equiv s_1; s_2$. From the IH for s_1 and the premise $\hat{\alpha}(\sigma_c,\sigma_d) \sqsubseteq \sigma^{\#}$ we get

$$\hat{\alpha}\big(\mathcal{E}_c[\![s_1]\!](\sigma_c),\; \mathcal{E}_d[\![s_1]\!](\sigma_c,\sigma_d)\big) \;\sqsubseteq\; S^{\#}[\![s_1]\!]\,\sigma^{\#}.$$

Applying the IH to s_2 with concrete input $\big(\mathcal{E}_c[\![s_1]\!](\sigma_c),\; \mathcal{E}_d[\![s_1]\!](\sigma_c,\sigma_d)\big)$ and abstract input $S^{\#}[\![s_1]\!]\,\sigma^{\#}$ yields

$$\hat{\alpha}\,\big(\mathcal{E}_c[\![s_2]\!](\mathcal{E}_c[\![s_1]\!](\sigma_c)),\; \mathcal{E}_d[\![s_2]\!](\mathcal{E}_c[\![s_1]\!](\sigma_c),\; \mathcal{E}_d[\![s_1]\!](\sigma_c,\sigma_d))\big) \;\sqsubseteq\; S^{\#}[\![s_2]\!]\big(S^{\#}[\![s_1]\!]\,\sigma^{\#}\big).$$

This is the desired inequality since both concrete and abstract semantics compose.

(Control-assign) $s \equiv y = \lambda\iota^*.(b?c_t : c_f)$. Let

$$\tau_t = C^{\#}[\![c_t]\!]\sigma^{\#}, \quad \tau_f = C^{\#}[\![c_f]\!]\sigma^{\#}, \quad \Pi = B^{\#}[\![b]\!]\sigma^{\#}.$$

Build the CAD skeleton from the branches (cf. Fig. 11):

$$\tau_{\mathrm{skel}} = \mathrm{CAD}^{\top}_{\bot}(\mathrm{vars},\mathrm{cuts}), \quad \mathrm{vars} = \tau_t.\mathrm{vars} \;\uplus\; \tau_f.\mathrm{vars} \;\uplus\; \iota^*, \quad \mathrm{cuts} = \tau_t.\mathrm{cuts} \;\uplus\; \tau_f.\mathrm{cuts} \;\uplus\; \Pi.$$

If any component is imprecise (unknown iterator or a $\top_\zeta/\bot_\zeta$ cut), then by definition $\mathrm{CAD}^{\top}_{\bot}$ returns $\top_{\mathcal{T}_C}$; since $\top_{\mathcal{T}_C}$ is greatest, the target inequality holds trivially.

Henceforth assume the precise case. By the concrete semantics (Fig. 6),

$$\mathcal{E}_c[\![s]\!](\sigma_c) = \sigma_c\big[y \mapsto a_y\big], \quad a_y(z^*) = \mathrm{ite}\big(\mathcal{E}_b[\![b]\!](\sigma_c[\imath^* \mapsto z^*]),\ \mathcal{E}^e_c[\![c_t]\!](\sigma_c[\imath^* \mapsto z^*]),\ \mathcal{E}^e_c[\![c_f]\!](\sigma_c[\imath^* \mapsto z^*])\big).$$

The abstract transformer constructs $\tau^{\,y}_{\mathrm{new}} = \tau_{\mathrm{skel}}$ and then performs the leaf substitution

$$\tau^{\,y}_{\mathrm{new}}{}' = \tau^{\,y}_{\mathrm{new}}\big[\mathrm{Leaf}(\_, \hat{n}) \mapsto \mathrm{Leaf}(v(\hat{n}), \hat{n})\big], \quad v(\hat{n}) = \begin{cases} \tau_t(\hat{n}) & \text{if } \mathcal{E}_b[\![b]\!](\{\imath^* \mapsto \hat{n}\}) = \mathtt{true}, \\ \tau_f(\hat{n}) & \text{otherwise.} \end{cases}$$

By (F2), it suffices to show the *pointwise* inequality

$$\forall z^*.\ \big\lceil a_y(z^*)\big\rceil_C \ \sqsubseteq_{\mathcal{B}_C}\ v(z^*). \tag{D.1}$$

Case $\mathcal{E}_b[\![b]\!](\{\imath^* \mapsto \hat{n}\}) = \mathtt{true}$. Then $a_y(z^*) = \mathcal{E}^e_c[\![c_t]\!](\{\imath^* \mapsto \hat{n}\})$. By (F1) with $c = c_t$,

$$\big\lceil a_y(z^*)\big\rceil_C = \big\lceil \mathcal{E}^e_c[\![c_t]\!](\{\imath^* \mapsto \hat{n}\})\big\rceil_C \ \sqsubseteq_{\mathcal{B}_C}\ \tau_t(z^*) \ = \ v(z^*).$$

Case $\mathcal{E}_b[\![b]\!](\{\imath^* \mapsto \hat{n}\}) = \mathtt{false}$. Symmetrically, $a_y(z^*) = \mathcal{E}^e_c[\![c_f]\!](\{\imath^* \mapsto \hat{n}\})$ and (F1) with $c = c_f$ gives $\big\lceil a_y(z^*)\big\rceil_C \ \sqsubseteq_{\mathcal{B}_C}\ \tau_f(z^*) = v(z^*)$.

Thus (D.1) holds for all z^*. Applying (F2) yields

$$\hat{\alpha}\big(\sigma_c[y \mapsto a_y], \sigma_d\big) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#[y \mapsto \tau^{\,y}_{\mathrm{new}}{}'] \ = \ S^\#[\![s]\!]\,\sigma^\#.$$

Finally, by the (implicit) definition of $\mathcal{E}_d$ for control-assign, the data store is unchanged: $\mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) = \sigma_d$, and the claim follows.

Proof for the data assignment statement is similar. □

E SOUNDNESS OF $P^\#$

Lemma E.1 (Fixpoint transfer). *$(\Sigma, \sqsubseteq_\Sigma)$ and $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$ are complete lattices. $\overline{F} : \Sigma \to \Sigma$ and $F^\# : \Sigma^\# \to \Sigma^\#$ are monotone, and $\hat{\alpha} : \Sigma \to \Sigma^\#$ preserves arbitrary joins. If*

$$\hat{\alpha} \circ \overline{F} \ \sqsubseteq_{\Sigma^\#}\ F^\# \circ \hat{\alpha},$$

then

$$\hat{\alpha}\big(\mathrm{lfp}(\overline{F})\big) \ \sqsubseteq_{\Sigma^\#}\ \mathrm{lfp}(F^\#).$$

Proof. For each $\sigma^\# \in \Sigma^\#$, define the candidate set

$$\Sigma_{\sigma^\#} \ \triangleq\ \{\, \sigma \in \Sigma \mid \hat{\alpha}(\sigma) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\# \,\},$$

and its join

$$\sigma_{\mathrm{join}} \ \triangleq\ \bigsqcup_{\Sigma} \{\, \sigma \in \Sigma \mid \hat{\alpha}(\sigma) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\# \,\}.$$

Note that $\Sigma_{\sigma^\#} \neq \varnothing$ because $\hat{\alpha}$ preserves the empty join, hence $\hat{\alpha}(\bot_\Sigma) = \bot_{\Sigma^\#} \sqsubseteq_{\Sigma^\#} \sigma^\#$ and thus $\bot_\Sigma \in \Sigma_{\sigma^\#}$. Since $\hat{\alpha}$ preserves arbitrary joins,

$$\hat{\alpha}\big(\sigma_{\mathrm{join}}\big) \ = \ \hat{\alpha}\left(\bigsqcup_{\Sigma}\{\sigma \in \Sigma \mid \hat{\alpha}(\sigma) \sqsubseteq_{\Sigma^\#} \sigma^\#\}\right) \ = \ \bigsqcup_{\Sigma^\#}\{\hat{\alpha}(\sigma) \mid \hat{\alpha}(\sigma) \sqsubseteq_{\Sigma^\#} \sigma^\#\} \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#,$$

so σ_{join} is the greatest element of Σ whose $\hat{\alpha}$-image lies below $\sigma^\#$.

Now fix any *pre-fixed point* $\sigma^\#$ of $F^\#$, i.e. $F^\#(\sigma^\) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#$. Using the assumption $\hat{\alpha} \circ \overline{F} \sqsubseteq_{\Sigma^\#} F^\# \circ \hat{\alpha}$ and monotonicity of $F^\#$, we have

$$\hat{\alpha}\left(\overline{F}\left(\sigma_{\mathrm{join}}\right)\right) \ \sqsubseteq_{\Sigma^\#}\ F^\#\left(\hat{\alpha}\left(\sigma_{\mathrm{join}}\right)\right) \ \sqsubseteq_{\Sigma^\#}\ F^\#(\sigma^\#) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#.$$

Hence $\overline{F}\left(\sigma_{\mathrm{join}}\right) \in \Sigma_{\sigma^\#}$, which implies

$$\overline{F}\left(\sigma_{\mathrm{join}}\right) \ \sqsubseteq_{\Sigma}\ \sigma_{\mathrm{join}}.$$

Thus σ_{join} is a pre-fixed point of $\overline{F}$, and by Park's induction (least pre-fixed–point rule) [32]

$$\text{lfp}(\overline{F}) \;\sqsubseteq_{\Sigma}\; \sigma_{\text{join}}.$$

Applying monotonicity of $\hat{\alpha}$ gives

$$\hat{\alpha}\big(\text{lfp}(\overline{F})\big) \;\sqsubseteq_{\Sigma^\#}\; \hat{\alpha}\left(\sigma_{\text{join}}\right) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#}.$$

Since this holds for every pre-fixed point $\sigma^{\#}$ of $F^{\#}$, we conclude

$$\hat{\alpha}\big(\text{lfp}(\overline{F})\big) \;\sqsubseteq_{\Sigma^\#}\; \bigsqcap_{\Sigma^\#}\{\, \sigma^{\#} \in \Sigma^{\#} \mid F^{\#}(\sigma^{\#}) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#} \,\} \;=\; \text{lfp}(F^{\#}).$$

□

Theorem E.2 (Soundness of $P^{\#}$). *Let $p \equiv \text{Fix}\, s$. Let initial concrete stores $\sigma_c^0 \in \Sigma_c$, $\sigma_d^0 \in \Sigma_d$ and an initial abstract store $\sigma^{\#0} \in \Sigma^{\#}$ satisfy $\hat{\alpha}(\sigma_c^0, \sigma_d^0) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#0}$. Write the concrete program semantics from Figure 6 as $\langle \sigma_c^*, \sigma_d^* \rangle = \left\langle \text{lfp}(F_c)\,,\ \text{lfp}\big(F_d^{\text{lfp}(F_c)}\big) \right\rangle$, where $F_c(\sigma_c) = \sigma_c^0 \sqcup_c \mathcal{E}_c[\![s]\!](\sigma_c)$ and $F_d^{\sigma_c}(\sigma_d) = \sigma_d^0 \sqcup_d \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)$. Let the abstract program semantics be $\sigma^{\#*} = \text{lfp}(F^{\#})$ with $F^{\#}(\sigma^{\#}) \triangleq \sigma^{\#0} \sqcup_{\Sigma^\#} S^{\#}[\![s]\!](\sigma^{\#})$ (Def. 5.9). Then*

$$\hat{\alpha}(\sigma_c^*, \sigma_d^*) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#*}.$$

Proof. Define the *simultaneous concrete operator* $\overline{F} : \Sigma \to \Sigma$ by

$$\overline{F}\big(\langle \sigma_c, \sigma_d \rangle\big) \;\triangleq\; \left\langle\, \sigma_c^0 \sqcup_c \mathcal{E}_c[\![s]\!](\sigma_c)\,,\ \sigma_d^0 \sqcup_d \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \,\right\rangle.$$

For any $\langle \sigma_c, \sigma_d \rangle \in \Sigma$,

$$\begin{aligned}
\hat{\alpha}\big(\overline{F}\langle \sigma_c, \sigma_d \rangle\big) &= \hat{\alpha}\Big(\, \langle \sigma_c^0, \sigma_d^0 \rangle \;\sqcup_{\Sigma}\; \langle \mathcal{E}_c[\![s]\!](\sigma_c),\ \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \rangle \,\Big) \\
&= \hat{\alpha}\langle \sigma_c^0, \sigma_d^0 \rangle \;\sqcup_{\Sigma^\#}\; \hat{\alpha}\Big\langle \mathcal{E}_c[\![s]\!](\sigma_c),\ \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \Big\rangle \quad (\hat{\alpha} \text{ is pointwise and join-preserving}) \\
&\sqsubseteq_{\Sigma^\#}\; \sigma^{\#0} \;\sqcup_{\Sigma^\#}\; S^{\#}[\![s]\!]\big(\hat{\alpha}\langle \sigma_c, \sigma_d \rangle\big) \quad (\text{init. assumption and Thm. D.1}) \\
&= F^{\#}\big(\hat{\alpha}\langle \sigma_c, \sigma_d \rangle\big).
\end{aligned}$$

Hence $\hat{\alpha} \circ \overline{F} \;\sqsubseteq\; F^{\#} \circ \hat{\alpha}$. By Lemma E.1, $\hat{\alpha}\big(\text{lfp}(\overline{F})\big) \;\sqsubseteq_{\Sigma^\#}\; \text{lfp}(F^{\#}) \;=\; \sigma^{\#*}$. From Bekic's theorem [2], the least fixpoint of $\overline{F}$ is

$$\text{lfp}(\overline{F}) \;=\; \left\langle\, \text{lfp}(F_c)\,,\ \text{lfp}\big(F_d^{\text{lfp}(F_c)}\big) \,\right\rangle,$$

which is exactly $\langle \sigma_c^*, \sigma_d^* \rangle$ from Figure 6. This gives $\hat{\alpha}(\sigma_c^*, \sigma_d^*) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#*}$, as required. □

F SOUNDNESS AND STABILIZATION OF ∇

Lemma F.1 (Floodfill$_n$ is inflationary and idempotent w.r.t. its first argument). *Fix $\pi \in \{\zeta, \delta\}$ and $c \in \mathcal{B}_\pi$. For every node n in $\mathcal{T}\langle\pi\rangle$, let* Floodfill$_n(n, c) = (m, c')$. *Then:*

(1) Inflationary: $n \;\sqsubseteq_{\mathcal{T}\langle\pi\rangle}\; m$.
(2) Idempotent (at fixed carry): Floodfill$_n(m, c) = (m, c')$.

Proof. By structural induction on n.

Leaf. If $n = \text{Leaf}(\ell, \mathbf{s})$ then $m = \text{Leaf}(\ell', \mathbf{s})$ with $\ell' = \ell$ when $\ell \neq \bot_\pi$ and $\ell' = c$ when $\ell = \bot_\pi$. In the flat base lattice, $\ell \sqsubseteq_\pi \ell'$; thus inflationarity holds. Reapplying Floodfill$_n$ with the same c does nothing (if $\ell' = \bot_\pi$ it stays $\bot_\pi$; otherwise it stays ℓ'), so idempotence holds.

Split. If $n = \text{LinSplit}((g_i, n_i)_{i=1}^k)$, write $c_0 = c$ and $(m_i, c_i) = \text{Floodfill}_n(n_i, c_{i-1})$ for $i = 1, \ldots, k$; then $m = \text{LinSplit}(\text{Cell}(g_1, m_1), \ldots, \text{Cell}(g_k, m_k))$. By the IH applied to each child, $n_i \sqsubseteq_{\mathcal{T}\langle\pi\rangle} m_i$, hence (by the partial order of Def. 5.2) $n \sqsubseteq_{\mathcal{T}\langle\pi\rangle} m$. For idempotence, apply the IH again: Floodfill$_n(m_i, c_{i-1}) =$

(m_i, c_i) for each i, so the second pass reconstructs exactly the same children and carries, yielding (m, c_k). □

Corollary F.2 (Floodfill is inflationary and idempotent). $\mathsf{Floodfill}_t$ *is inflationary and idempotent, and* $\mathsf{Floodfill} : \Sigma^\# \to \Sigma^\#$ *(Def. 5.11) is inflationary and idempotent.*

Theorem F.3 (Soundness of Widening). *For all* $X, Y \in \Sigma^\#$,

$$(X \sqcup_{\Sigma^\#} Y) \sqsubseteq_{\Sigma^\#} X \nabla Y = \mathsf{Floodfill}\big(X \sqcup_{\Sigma^\#} Y\big).$$

Proof. By Cor. F.2, Floodfill is inflationary; thus $(X \sqcup_{\Sigma^\#} Y) \sqsubseteq_{\Sigma^\#} \mathsf{Floodfill}(X \sqcup_{\Sigma^\#} Y)$. □

Lemma F.4 (Monotonicity of the widening iteration). *For any initial* $\widetilde{\sigma}_0^\# \in \Sigma^\#$, *define* $\widetilde{\sigma}_{i+1}^\# \triangleq \mathsf{Floodfill}\big(\widetilde{\sigma}_i^\# \sqcup_{\Sigma^\#} F^\#(\widetilde{\sigma}_i^\#)\big)$. *Then* $\widetilde{\sigma}_i^\# \sqsubseteq_{\Sigma^\#} \widetilde{\sigma}_{i+1}^\#$ *for all* i.

Proof. By Cor. F.2, Floodfill is inflationary. Hence $\widetilde{\sigma}_i^\# \sqsubseteq_{\Sigma^\#} \mathsf{Floodfill}\big(\widetilde{\sigma}_i^\# \sqcup_{\Sigma^\#} F^\#(\widetilde{\sigma} i^\#)\big) = \widetilde{\sigma}_{i+1}^\#$. □

Lemma F.5 ($\Sigma^\#$ is finite height). $\Sigma^\#$ *is finite height.*

Proof. By construction, the base domains $\mathcal{B}_\pi$ ($\pi \in \{\zeta, \delta\}$) are flat lattices and hence finite height. The total number of cells in any CAD is bounded by a finite constant $M \leq t^{2^{O(n)}}$, where n is the size of vars and t is the size of cuts. Since only a finite number of program variables and cuts can exist for a given program p, both $\mathcal{T}_C$ and $\mathcal{T}_D$ are finite-height lattices. By Definition 5.3, $\Sigma^\# \triangleq \Sigma_c^\# \times \Sigma_d^\#$ with $\Sigma_c^\# \triangleq (\mathbb{Y} \to \mathcal{T}_C)$ and $\Sigma_d^\# \triangleq (\mathbb{X} \to \mathcal{T}_D)$. Since there are only finite number of variables in a program and $\mathcal{T}_C$ and $\mathcal{T}_D$ are finite-height lattices, $\Sigma^\#$ is a finite height lattice. □

Theorem F.6 (Stabilization of ∇). *For any initial* $\widetilde{\sigma}_0^\# \in \Sigma^\#$, *define* $\widetilde{\sigma}_{i+1}^\# \triangleq \mathsf{Floodfill}\big(\widetilde{\sigma}_i^\# \sqcup_{\Sigma^\#} F^\#(\widetilde{\sigma}_i^\#)\big)$. *Then* $(\widetilde{\sigma}_i^\#)_{i \geq 0}$ *stabilizes in finitely many steps.*

Proof. $F^\#$ is monotone by Definition 5.9, and the widening iteration is monotone by Lemma F.4. Therefore, $(\widetilde{\sigma}_i^\#)_{i \geq 0}$ is an ascending chain in $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$. By Lemma F.5, $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$ has finite height. Every ascending chain in a finite-height poset stabilizes after finitely many steps; hence so does $(\widetilde{\sigma}_i^\#)_{i \geq 0}$. □