Accelerating Loops with Arrays

Florian Frohn and Jürgen Giesl

RWTH Aachen University, Aachen, Germany

Abstract. We propose a novel *acceleration technique* for loops operating on arrays. The goal of acceleration is to characterize the transitive closure of loops in a logic which is suitable for automated reasoning. Using the new notion of *inductive lvalues*, our technique can handle loops where previous techniques fail, and it *unifies* acceleration for arrays and scalar variables by regarding scalars as arrays of dimension 0. Moreover, our approach uses λs instead of quantifiers. Then the resulting SMT problems can be solved via *lemmas on demand*. An empirical evaluation of our implementation in the tool LoAT shows the power of our approach.

1 Introduction

Model checking programs operating on arrays is notoriously difficult. One reason is that interesting properties of array programs often involve quantifiers, but reasoning about quantifiers is still challenging for SMT solvers. In particular, they can hardly find models for such formulas, but models are crucial for model checking algorithms that use them to prove unsafety and for abstraction refinement.

Alternatives to abstraction refinement are based on *acceleration techniques* [5, 9–11, 16, 19, 31, 32] that try to characterize the transitive closure of a single-path loop. Then algorithms like Acceleration Driven Clause Learning [21] or Accelerated Bounded Model Checking (ABMC) [23] can lift acceleration from single-path loops to full programs. As acceleration does not over-approximate, it is particularly useful for finding counterexamples, but it can also be used for proving safety [12].

We present a new acceleration technique for loops operating on arrays. In contrast to earlier approaches [1–3, 28, 33], it applies to a broader class of loops, due to the novel notion of *inductive lvalues*. The core idea of inductive lvalues is that the effect of assignments like $a[i+1] \leftarrow a[i]+1; i \leftarrow i+1$ in loop bodies can be summarized by recurrence equations. Here, the value $a[i]$ that is read is equal to the value that has been written to $a[i+1]$ in the previous iteration.

Another benefit of inductive lvalues is that they allow us to *unify* loop acceleration for arrays and scalar variables by regarding scalar variables as arrays of dimension 0. In contrast, earlier approaches treat arrays and scalars differently.

One more key difference to earlier approaches is that we use λs and a specialized quantifier elimination technique to avoid quantifiers. Even though most SMT solvers have very limited support for λs, the resulting SMT problems can usually be solved with a refinement loop in the spirit of *lemmas on demand* [14, 37, 38].

2 Overview and Contributions

We start with an informal overview, where we illustrate our pipeline for verifying array programs, and explain which parts are contributions of this paper. Consider the C program on the right, which writes $a[0]$ to $a[1], \ldots, a[k]$. Here, nondet(0, k) returns some integer between 0 and k. So the final assertion always fails. There exist various techniques to transform imperative programs into more formal representations like *transition systems*, resulting in:

```
int i = 0;
int k = 10000;
while(i < k) {
    a[i+1] = a[i];
    i = i + 1;
}
int j = nondet(0, k);
assert(a[j] != a[0]);
```

$$\begin{aligned}
\mathsf{init}(a) &\to \mathsf{while}(a, 0, 10000) && \\
\mathsf{while}(a, i, k) &\to \mathsf{while}(\lambda j.\ \textbf{if}\ (j = i+1)\ a[i]\ \textbf{else}\ a[j], i+1, k) && \textbf{if}\ i < k \qquad (\textsc{Loop}) \\
\mathsf{while}(a, i, k) &\to \mathsf{check}(a, j) && \textbf{if}\ i \geq k \wedge 0 \leq j \leq k \\
\mathsf{check}(a, j) &\to \mathsf{err} && \textbf{if}\ a[j] = a[0]
\end{aligned}$$

In our evaluation (see Sect. 7), this step is done by our tool HornKlaus that transforms C programs into linear Constrained Horn Clauses, which are equivalent to transition systems. In the transition (Loop), "$\lambda j.$ **if** $(j = i+1)$ $a[i]$ **else** $a[j]$" denotes the array that results from writing $a[i]$ to $a[i+1]$.

To prove unsafety, one has to show that the location err can be reached from init. The core idea of acceleration is to replace loops like (Loop) with transitions like the one below that can simulate arbitrarily many iterations in a single step:

$$\begin{aligned}
\mathsf{while}(a, i, k) \to \mathsf{while}(a', i+n, k) \quad & \textbf{if}\ n > 0 \wedge i + n \leq k \wedge \forall j \in \mathbb{Z} \qquad (\textsc{Accel}) \\
& (i < j \leq i + n \implies a'[j] = a[i]) \wedge \\
& (j \leq i \vee i + n < j \implies a'[j] = a[j])
\end{aligned}$$

It simulates n iterations, where n is chosen non-deterministically.

For this step, two ingredients are needed: First, one needs a *closed form* for each variable x that expresses the value of x after n iterations. For scalar variables like i, *closed form expressions* are computed by solving recurrence equations, resulting in the closed form $i + n$. So a closed form expression for a variable x involves program variables and the additional variable n, and its value is equal to the value of x after n iterations, for all instantiations of the occurring variables.

For arrays, existing acceleration techniques characterize closed forms via quantified formulas of the form "$\forall j \in \mathbb{Z}.\ \varphi$" instead of expressions, where j is used as array index, as in (Accel). The reason is that the standard theory of arrays [34] is not expressive enough to admit suitable closed form expressions.

In this paper, we propose to use λ-expressions as a suitable language to express closed forms for arrays. In our example, instead of (Accel), our approach yields:

$$\begin{aligned}
&\mathsf{while}(a, i, k) \to \\
&\mathsf{while}'(\lambda j.\ \textbf{if}\ (i < j \leq i+n)\ a[i]\ \textbf{else}\ a[j], i+n, k)
\end{aligned} \quad \textbf{if}\ n > 0 \wedge i + n \leq k. \qquad (\lambda)$$

Moreover, our approach can characterize closed forms exactly in cases where earlier approaches fail or approximate. To achieve this, we introduce *inductive lvalues*. An lvalue is an expression like i or $a[i]$ that refers to a memory location.

We call an lvalue like $a[i]$ in (Loop) *inductive*, as the value that is read from $a[i]$ is equal to the value that has been written to $a[i+1]$ in the previous iteration. Thus, the value that is referenced by $a[i]$ in the n^{th} iteration can be defined recursively, which gives rise to recurrence equations, as in the case of scalars. This allows us to accelerate loops like (Loop) where earlier approaches fail, and to handle arrays and scalars uniformly by regarding scalars as arrays of dimension 0.

While λs can replace the outermost universal quantifier (as in the step from (Accel) to (λ)), in general, nested quantifiers are needed to characterize closed forms for arrays. As most acceleration based verification techniques rely on SMT solvers, and SMT solvers still struggle with quantifiers, our goal is to also eliminate the remaining quantifiers. To this end, we identify an important class of loops where all quantifiers can indeed be eliminated, see Sect. 5.

The second ingredient for acceleration is a formula which guarantees that the loop can be executed n times. In our example, this is ensured by the constraint $i + n \leq k$. In our implementation, such formulas are synthesized using the technique from [19]. The details are beyond the scope of this paper, but one core idea is to exploit monotonicity: In our example, if $i < k$ holds after executing the loop body, then it also held before. So if $i < k$ holds before the n^{th} iteration, then the loop can be executed n times. At this point, the value of i is $i+n-1$, so we obtain $i+n \leq k$. While originally developed for integer loops, the approach from [19] immediately carries over to our setting.

Now our verification problem can be encoded into an SMT formula with λs:

$$
\begin{aligned}
&\overbrace{i = 0 \land k = 10000}^{\text{initialization}} \land \overbrace{n > 0 \land i + n \leq k}^{\text{guard of } (\lambda)} \land \\
&\left.i' = i + n \land a' = \lambda j.\ \textbf{if } (i < j \leq i + n)\ a[i]\ \textbf{else } a[j] \land \quad\right\}\text{update of } (\lambda) \\
&\underbrace{i' \geq k \land 0 \leq j \leq k}_{\text{transition to check}} \land \underbrace{a'[j] = a'[0]}_{\text{transition to err}}
\end{aligned}
$$

Unfortunately, the major SMT solvers usually fail if λs cannot be β-reduced (i.e., evaluated). However, SMT problems with λs can often be solved via *lemmas on demand* [14,37,38]. Here, the main idea is to abstract λs with uninterpreted functions, and refine this abstraction with suitable *lemmas* whenever a model for the abstraction cannot be lifted to the original problem. In this way, we can prove satisfiability of the formula above, which implies unsafety of our example.

To summarize, our main contributions are: **(1)** *inductive lvalues* that allow for accelerating loops where existing techniques fail, **(2)** the first *uniform* approach for handling scalars and arrays in acceleration, **(3)** the use of λs to accelerate arrays, where the resulting SMT problems can be solved via *lemmas on demand*, **(4)** a quantifier elimination technique which often yields quantifier-free closed forms (with λs) for arrays, and **(5)** a competitive implementation in our tool LoAT.

Outline: After introducing preliminaries in Sect. 3, our main contributions are presented in Sections 4 and 5. Related work is discussed in Sect. 6, and we evaluate our approach empirically and conclude in Sect. 7. App. A explains how we handle λs in SMT, and App. B contains all proofs.

3 Preliminaries

We write $\vec{v}$ and $\mathbf{v}$ for column and row vectors, and v_i is the i^{th} element of $\vec{v}$ or $\mathbf{v}$.

Arrays and Expressions: Let $\mathcal{V}$ be a countably infinite set of variables, where each $x \in \mathcal{V}$ has a designated *arity* $\mathsf{arity}(x) \in \mathbb{N}$. Then x ranges over $\mathbb{Z}^{\mathbb{Z}^{\mathsf{arity}(x)}}$, i.e., over all functions with domain $\mathbb{Z}^{\mathsf{arity}(x)}$ and codomain $\mathbb{Z}$ (i.e., x maps row vectors to integers). Intuitively, x represents an integer array of dimension $\mathsf{arity}(x)$, i.e., we only consider integer valued arrays. So x represents a scalar if $\mathsf{arity}(x) = 0$, and a 1-dimensional array which is infinite in both directions if $\mathsf{arity}(x) = 1$. Considering infinite arrays is in line with SMT-LIB [6]. To model finite arrays, array-accesses must be guarded with suitable constraints. Note that we interpret arrays as uncurried functions, i.e., $x[\mathbf{i}]$ is not a valid expression if $|\mathbf{i}| \neq \mathsf{arity}(x)$. So if a is a 2-dimensional array, then $a[0]$ is undefined. In contrast, $a[0]$ would be a 1-dimensional array in, e.g., C. Handling "curried arrays" is future work.

We use c, i, j, k, m, n for scalar variables and integer constants, a, b for non-scalars (i.e., for arrays of dimension > 0), and x, y can represent both scalars and non-scalars. For any entity e, $\mathcal{V}(e)$ denotes the variables that occur freely in e.

An expression of the form $x[\mathbf{c}]$ where $\mathbf{c} \in \mathbb{Z}^{\mathsf{arity}(x)}$ refers to a *cell*, where each cell stores an integer value. The set *Rval* of *rvalues* is defined by the grammar

$$Rval ::= c \mid Rval \circ Rval \mid x[\mathbf{r}]$$

where $c \in \mathbb{Z}$, $\circ$ is an arithmetic operator,[1] $x \in \mathcal{V}$, and $\mathbf{r} \in Rval^{\mathsf{arity}(x)}$. The set *Lval* of *lvalues* is defined as

$$Lval := \{x[\mathbf{r}] \mid x \in \mathcal{V}, \mathbf{r} \in Rval^{\mathsf{arity}(x)}\}.$$

So an rvalue (lvalue) is an expression that may occur on the right-hand (left-hand) side of an assignment. We write i instead of $i[\,]$ if $i \in \mathcal{V}$ has arity 0. For rvalues $r \in Rval$ we define $Lval(r)$ to be the set of r's "top-level" lvalues, i.e.,

$$Lval(r) := \begin{cases} \{r\} & \text{if } r \in Lval \\ Lval(r_1) \cup Lval(r_2) & \text{if } r = r_1 \circ r_2 \\ \varnothing & \text{if } r \in \mathbb{Z} \end{cases}$$

So we have $Lval(a[i]+7) = \{a[i]\}$ and, importantly, $i \notin Lval(a[i]+7)$, i.e., $Lval(r)$ only contains the lvalues occurring in r that are not nested below other lvalues.

An *array expression* is a variable from $\mathcal{V}$, or of the form $\lambda\mathbf{i}.\ e$, where $\mathbf{i}$ is a vector of scalar variables, and $e \in Expr$ is an *expression*. We define $\mathsf{arity}(\lambda\mathbf{i} \ldots) := |\mathbf{i}|$. Expressions are generated by the grammar

$$Expr ::= c \mid Expr \circ Expr \mid p[\mathbf{e}] \mid \textbf{if } (\mu)\ Expr \textbf{ else } Expr$$

where $c \in \mathbb{Z}$, $\circ$ is again an arithmetic operator, p is an array expression, $\mathbf{e} \in Expr^{\mathsf{arity}(p)}$, and μ is a first-order formula over expressions involving the relations $>, \neq, \ldots$, and $|$, and the usual Boolean connectives. Here, $|$ denotes divisibility, i.e., $c \mid i$ is true iff c divides i. So "$\textbf{if } (j = i)\ a[i-1] \textbf{ else } a[j]$" is an expression, and "$\lambda j.\ \textbf{if } (j = i)\ a[i-1] \textbf{ else } a[j]$" is an array expression of arity 1. We use r for rvalues, ℓ for lvalues, p, q for array expressions, and e for expressions.

[1] We use $+, -, \cdot$, and div (for integer division), but other operators are possible as well.

Loops: We consider single-path loops (without branching in their body):

$$\textbf{while } \varphi \textbf{ do } \vec{\ell} \leftarrow \vec{r} \textbf{ done} \tag{$\mathcal{T}_{loop}$}$$

Here, φ is a conjunction of (in)equations over rvalues, $\vec{\ell} = \begin{pmatrix} x_1[\mathbf{r}_1] \\ \cdots \\ x_m[\mathbf{r}_m] \end{pmatrix}$ is a vector of lvalues such that[2]

$$\forall i, j \in [1\,..\,m].\ i \neq j \implies x_i \neq x_j \vee \mathbf{r}_i \neq \mathbf{r}_j \tag{Distinct}$$

holds, and $\vec{r} \in \mathit{Rval}^m$. Here, $[1\,..\,m]$ denotes the closed integer interval $\{1, \ldots, m\}$ (and "$[1\,..\,m)$" denotes a half-open integer interval). Intuitively, the body of $\mathcal{T}_{loop}$ updates all lvalues $\vec{\ell}$ simultaneously[3] by overwriting each $x_i[\mathbf{r}_i] \in \vec{\ell}$ with r_i, which is the i^{th} component of $\mathbf{r}$. We also use the notation $\mathsf{rhs}(x_i[\mathbf{r}_i]) := r_i$. Such loops correspond to recursive transitions like (Loop) in Sect. 2. (Distinct) ensures that two different lvalues cannot refer to the same cell, which prevents loops like

$$\textbf{while } a[j] = 0 \textbf{ do } \begin{pmatrix} a[i] \\ a[j] \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \textbf{ done}.$$

As $a[i]$ and $a[j]$ are written simultaneously, here it is unclear if $a[j]$ is 0 or 1 after the loop body if $i = j$. In the sequel, $\mathcal{T}_{loop}$ denotes an arbitrary but fixed loop.

Example 1 (Running Example). We use the following loop as running example:

$$\textbf{while } i < k \textbf{ do } \begin{pmatrix} i \\ a[i+1] \\ a[i] \end{pmatrix} \leftarrow \begin{pmatrix} i+1 \\ a[i] \\ a[i+1] \end{pmatrix} \textbf{ done} \tag{$\mathcal{T}_{swap}$}$$

It swaps the values of $a[i]$ and $a[i+1]$ in each iteration. So if the initial value of $a[i]$ is c, then after n iterations the value of $a[i+n]$ is c, and the initial values of $a[i+1], \ldots, a[i+n]$ have been shifted one position to the left.

For each $x \in \mathcal{V}$, $\mathsf{up}_x(\vec{\ell})$ is an array expression that describes the array x after one iteration of the loop body. More precisely, we define $\mathsf{up}_x(\vec{\ell})$ inductively as follows:

$$\mathsf{up}_x(\vec{\ell}) := \begin{cases} x & \text{if } \vec{\ell} \text{ is empty} \\ \lambda\mathbf{i}.\ \textbf{if } (\mathbf{i} = \mathbf{r})\ \mathsf{rhs}(x[\mathbf{r}]) \textbf{ else } \mathsf{up}_x(\vec{\ell}')[\mathbf{i}] & \text{if } \vec{\ell} = \begin{pmatrix} x[\mathbf{r}] \\ \vec{\ell}' \end{pmatrix} \\ \mathsf{up}_x(\vec{\ell}') & \text{if } \vec{\ell} = \begin{pmatrix} y[\mathbf{r}] \\ \vec{\ell}' \end{pmatrix} \text{ where } x \neq y \end{cases} \tag{read write}$$

Here, the scalars $\mathbf{i} \in \mathcal{V}^{\mathsf{arity}(x)}$ do not occur freely in $\mathbf{r}, \mathsf{rhs}(x[\mathbf{r}])$, or $\mathsf{up}_x(\vec{\ell}')$. Moreover, we define $\mathsf{up} := [x/\mathsf{up}_x(\vec{\ell}) \mid x \in \mathcal{V}]$, i.e., up is the substitution that replaces each variable x with the result of updating x according to the body of the loop.

Example 2 (up). For $\mathcal{T}_{swap}$, we have

$$\mathsf{up}_i \begin{pmatrix} i \\ a[i+1] \\ a[i] \end{pmatrix} \quad = \quad \lambda.\ \textbf{if } ([] = [])\ i+1 \textbf{ else } \mathsf{up}_i \begin{pmatrix} a[i+1] \\ a[i] \end{pmatrix} \quad = \quad i+1$$

where $\lambda.\ \ldots$ denotes a function of arity 0, and:

$$\mathsf{up}_a \begin{pmatrix} i \\ a[i+1] \\ a[i] \end{pmatrix} = \mathsf{up}_a \begin{pmatrix} a[i+1] \\ a[i] \end{pmatrix} = \lambda j.\ \textbf{if } (j = i+1)\ a[i] \textbf{ else } \mathsf{up}_a(a[i])[j]$$

[2] A less restrictive definition would require $\varphi \implies$ (Distinct). We use the definition above instead to simplify the presentation.

[3] Considering sequential updates instead requires additional technicalities, so we restrict our attention to simultaneous updates for the sake of simplicity.

$$= \lambda j.\ \textbf{if}\ (j = i+1)\ a[i]\ \textbf{else if}\ (j = i)\ a[i+1]\ \textbf{else}\ \mathsf{up}_a()[j]$$
$$= \lambda j.\ \textbf{if}\ (j = i+1)\ a[i]\ \textbf{else if}\ (j = i)\ a[i+1]\ \textbf{else}\ a[j]$$

The definition of $\boxed{\substack{read\\write}}$ corresponds to the *read-over-write axioms* [34]:

$$\mathbf{i} = \mathbf{r} \implies \mathsf{up}_x(x[\mathbf{r}])[\mathbf{i}] = \mathsf{rhs}(x[\mathbf{r}]) \qquad \mathbf{i} \neq \mathbf{r} \implies \mathsf{up}_x(x[\mathbf{r}])[\mathbf{i}] = x[\mathbf{i}]$$

So our notion of arrays is compatible with the standard first-order theory of arrays. However, our array expressions are more expressive than expressions in the standard theory. As an example, consider the array expression

$$\lambda i.\ \textbf{if}\ (0 \leq i \leq j)\ 0\ \textbf{else}\ a[i]. \tag{1}$$

Here, the number of elements where a and (1) differ is not fixed (it depends on j). Expressions for such arrays do not exist in the standard theory.

A *state* s maps each $x \in \mathcal{V}$ to a function of arity $\mathsf{arity}(x)$. The semantics $[\![e]\!]_s$ of an (array) expression e w.r.t. a state s is defined in the obvious way (see App. C), and it is lifted to formulas over (array) expressions as usual. In particular, the semantics of an array expression is a function. Given a first-order formula ψ, we often say that "ψ holds" or "ψ is valid", which means that $[\![\psi]\!]_s = \mathsf{true}$ for all states s. Now we can define the *transition relation* of $\mathcal{T}_{loop}$.

Definition 3 (Transition Relation). *We define the relation $\to_{\mathcal{T}_{loop}}$ on states as follows: We have $s \to_{\mathcal{T}_{loop}} s'$ if $[\![\varphi]\!]_s = \mathsf{true}$ and $s'(x) = [\![\mathsf{up}(x)]\!]_s$ for all $x \in \mathcal{V}$.*

Example 4 (Transition Relation). Let $s(i) = 0$, $s(k) = 5$, and $s(a) = \lambda j.\ j$. Then $s \to_{\mathcal{T}_{swap}} s'$ with $s'(i) = [\![\mathsf{up}(i)]\!]_s = [\![i+1]\!]_s = s(i) + 1 = 1$, $s'(k) = s(k) = 5$, and:

$$s'(a) = [\![\mathsf{up}(a)]\!]_s = [\![\lambda j.\ \textbf{if}\ (j = i+1)\ a[i]\ \textbf{else if}\ (j = i)\ a[i+1]\ \textbf{else}\ a[j]]\!]_s$$
$$= \lambda j.\ \textbf{if}\ (j = 1)\ 0\ \textbf{else if}\ (j = 0)\ 1\ \textbf{else}\ j$$

Acceleration: Our goal is to *accelerate* loops like $\mathcal{T}_{loop}$.

Definition 5 (Acceleration). *Let $\to^n_{\mathcal{T}}$ denote the n-fold closure of $\to_{\mathcal{T}}$. An* acceleration technique *is a partial function* accel*, mapping loops to first-order formulas over $\mathcal{V}$, $\mathcal{V}' := \{x' \mid x \in \mathcal{V}\}$, and a designated variable n such that*

$$\mathsf{accel}(\mathcal{T})\ [x/s(x), x'/s'(x) \mid x \in \mathcal{V}] \iff s \to^n_{\mathcal{T}} s'$$

holds for all $\mathcal{T} \in \mathrm{dom}(\mathsf{accel})$, $n \in \mathbb{N}_{>0}$, and all states s, s'. Here, $[x/s(x), x'/s'(x) \mid x \in \mathcal{V}]$ is the substitution that replaces x by $s(x)$ and x' by $s'(x)$, for all $x \in \mathcal{V}$.

So acceleration yields a formula over $\mathcal{V}$ and $\mathcal{V}'$, representing the values of the program variables before and after n loop iterations, and a dedicated loop counter n. This formula characterizes the n-fold closure of $\mathcal{T}_{loop}$ precisely.[4]

Example 6 (Acceleration). Consider the loop

$$\textbf{while}\ i > 0\ \textbf{do}\ i \leftarrow i - 1\ \textbf{done}$$

Here, an acceleration technique may yield the formula $i - n \geq 0 \land i' = i - n$.

As explained in Sect. 2, many acceleration techniques rely on *closed form expressions* that characterize the values of the variables after n iterations of the loop.

[4] Some acceleration techniques also construct under-approximations, but since we show how to accelerate certain loops precisely, Def. 5 does not allow approximations.

Definition 7 (Closed Forms). *We call an expression* $x^{(n)}$ *over* $\mathcal{V} \cup \{n\}$ *a* closed form *for* $x \in \mathcal{V}$ *if* $x^{(n)} = \mathsf{up}^n(x)$ *holds for all* $n \in \mathbb{N}$.

For integers, closed forms are usually computed via *recurrence solving*.

Example 8 (Closed Forms). For Ex. 6, a closed form for i is computed by solving the recurrence equation $i^{(n)} = i^{(n-1)} - 1$ with the initial condition $i^{(0)} = i$.

Given closed forms for each $x \in \mathcal{V}$, an acceleration technique can return

$$\left(\forall k \in [0 \mathinner{..} n).\ \varphi[x/x^{(k)} \mid x \in \mathcal{V}]\right) \wedge \bigwedge_{x \in \mathcal{V}(\mathcal{T}_{loop})} x' = x^{(n)}.$$

However, most acceleration techniques try to avoid quantifiers, as they are challenging for automated reasoning. Instead, the calculus from [19] can be used (see Sect. 2), which can often compute suitable quantifier-free formulas. Therefore, in the sequel, we focus on computing closed forms for loops involving arrays.

4 Closed Forms for Lvalues

We now show how to compute closed forms for array loops. To this end, we impose mild restrictions. First, we require that the indices in $\vec{\ell}$ behave monotonically w.r.t. the lexicographic order (where we have $\mathbf{r} < \mathbf{r}'$ if there is an $i \in [1 \mathinner{..} |\mathbf{r}|]$ such that $r_i < r'_i$ and $r_j = r'_j$ hold for all $j \in [1 \mathinner{..} i)$). $\mathcal{T}_{loop}$ is x*-increasing* if the indices $\mathbf{r}$ where x is updated by the loop increase their values weakly in each iteration.

Definition 9 (Monotonic Loop). $\mathcal{T}_{loop}$ *is* x-increasing *or* x-decreasing *if*

$$\bigwedge_{x[\mathbf{r}] \in \vec{\ell}} \mathbf{r} \leq \mathsf{up}(\mathbf{r}) \qquad or \qquad \bigwedge_{x[\mathbf{r}] \in \vec{\ell}} \mathbf{r} \geq \mathsf{up}(\mathbf{r})$$

is valid, respectively. If $\mathcal{T}_{loop}$ *is* x*-increasing or* x*-decreasing, then it is* x-monotonic. *If* $\mathcal{T}_{loop}$ *is* x*-monotonic for each* $x \in \mathcal{V}$, *then* $\mathcal{T}_{loop}$ *is* monotonic.

Example 10. Reconsider $\mathcal{T}_{swap}$, where a is accessed via the indices i and $i+1$ in $\vec{\ell}$. Since we have $i \leq \mathsf{up}(i) = i + 1$, $\mathcal{T}_{swap}$ is a-increasing and thus a-monotonic. As i is a scalar variable, $\mathcal{T}_{swap}$ is also i-monotonic. Thus, $\mathcal{T}_{swap}$ is monotonic.

For simplicity, in the sequel we assume that x-monotonic loops are x-increasing. Lifting our approach to arbitrary monotonic loops is straightforward.

A monotonic loop $\mathcal{T}_{loop}$ is *array-solvable*, or *a-solvable* for short, if each set $Lval(r_i)$ only refers to array cells that have been written in the previous iteration, or only to array cells that have not been written yet.

Definition 11 (A-Solvable Loop). *Let* $\mathcal{L}$ *be the smallest set such that:*

$$Lval(\vec{r}) \subseteq \mathcal{L} \qquad and \qquad if\ x[\mathbf{r}] \in \mathcal{L},\ then\ Lval(\mathbf{r}) \subseteq \mathcal{L}$$

An lvalue $x[\mathbf{r}] \in \mathcal{L}$ *is*

- trivial *if for all* $y[\mathbf{r}'] \in \vec{\ell}$, *we have* $y \notin \mathcal{V}(x[\mathbf{r}])$
- inductive *if* $x[\mathsf{up}(\mathbf{r})] \in \vec{\ell}$
- displacing *if for all* $\mathbf{r}'$ *such that* $x[\mathbf{r}'] \in \vec{\ell}$, *we have* $\mathbf{r}' < \mathsf{up}(\mathbf{r})$

$\mathcal{T}_{loop}$ *is* a-solvable *if it is monotonic, each element of* $\mathcal{L}$ *is trivial, inductive, or displacing, and one of the following holds for each* $i \in [1 \mathinner{..} |\vec{r}|]$*:*
(a) all lvalues in $Lval(r_i)$ *are trivial or inductive*
(b) all lvalues in $Lval(r_i)$ *are displacing*

So $x[\mathbf{r}]$ is trivial if none of the variables occurring in $x[\mathbf{r}]$ are changed by $\mathcal{T}_{loop}$. Inductive lvalues $x[\mathbf{r}]$ give rise to inductive definitions, as reading $x[\mathbf{r}]$ in the n^{th} iteration yields the value that has been written to the same cell (i.e., to $x[\mathsf{up}(\mathbf{r})]$) in the $(n-1)^{th}$ iteration. In contrast, the cell that is referenced by a displacing lvalue $x[\mathbf{r}]$ has not been written before it is being read, as all (increasing) indices that were used for write accesses to x in earlier iterations are smaller than $\mathbf{r}$ in the n^{th} iteration. Note that trivial lvalues are a special case of displacing lvalues. For x-decreasing loops, the inequation $\mathbf{r}' < \mathsf{up}(\mathbf{r})$ becomes $\mathbf{r}' > \mathsf{up}(\mathbf{r})$.

Example 12 (A-Solvable). For $\mathcal{T}_{swap}$, we have $r_1 = i+1$, $r_2 = a[i]$, $r_3 = a[i+1]$, and $\mathcal{L} = \{i, a[i], a[i+1]\}$. For $Lval(r_1) = \{i\}$, we have $i \in \vec{\ell}$, i.e., i is inductive. For $Lval(r_2) = \{a[i]\}$, we have $a[\mathsf{up}(i)] = a[i+1] \in \vec{\ell}$, so $a[i]$ is inductive, too. For $Lval(r_3) = \{a[i+1]\}$, we have $a[\mathsf{up}(i+1)] = a[i+2]$. So $a[i+1]$ is displacing, as $i < i+2$ and $i+1 < i+2$ hold for the accesses $a[i]$ and $a[i+1]$ to a in $\vec{\ell}$. Hence, $\mathcal{T}_{swap}$ is a-solvable.

Earlier acceleration techniques for arrays could essentially only handle displacing lvalues without approximations, i.e., loops where each cell is written at most once, see Sect. 6. Moreover, they treated arrays and scalars differently. Our new notion of inductive lvalues allows us to also handle examples like $\mathcal{T}_{swap}$, and to unify the treatment of arrays and scalars (by regarding them as arrays of dimension 0). Ex. 13 shows why we disallow mixing inductive and displacing lvalues in $Lval(r_i)$.

Example 13 (Mixing Inductive and Displacing Lvalues). Consider the loop

$$\textbf{while } i < k \textbf{ do } \left(\begin{smallmatrix} i \\ a[i+1] \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} i+1 \\ a[i]+a[i+1] \end{smallmatrix}\right) \textbf{ done}$$

where $a[i]$ is inductive and $a[i+1]$ is displacing. We have $\mathsf{up}^n(a[i]) = \sum_{j=i}^{i+n} a[j]$, which is not an expression as defined in Sect. 3. For such loops, extensions of the underlying theory are required, see, e.g., [29]. We leave that to future work.

We first show how to compute closed forms for all lvalues in $\mathcal{L}$.

Definition 14 (Closed Forms for Lvalues). *We call an expression* $\ell^{(n)}$ *over* $\mathcal{V} \cup \{n\}$ *a* closed form *for* $\ell \in Lval$ *if* $\ell^{(n)} = \mathsf{up}^n(\ell)$ *holds for all* $n \in \mathbb{N}$.

For trivial lvalues, computing closed forms is straightforward.

Lemma 15 (Closed Forms for Trivial Lvalues). *If* $\ell \in \mathcal{L}$ *is trivial, then* $\ell^{(n)} := \ell$ *is a closed form for* ℓ.

The next theorem yields closed forms for displacing lvalues $x[\mathbf{r}]$ if closed forms for all $\ell \in Lval(\mathbf{r})$ are known. Here, we use the notation $\mathbf{r}^{(n)} := \mathbf{r}[\ell/\ell^{(n)} \mid \ell \in Lval(\mathbf{r})]$.

Theorem 16 (Closed Forms for Displacing Lvalues). *If* $\mathcal{T}_{loop}$ *is monotonic and* $x[\mathbf{r}] \in \mathcal{L}$ *is displacing, then* $x[\mathbf{r}]^{(n)} := x[\mathbf{r}^{(n)}]$ *is a closed form for* $x[\mathbf{r}]$.

Example 17. Reconsider $\mathcal{T}_{swap}$, where the only displacing lvalue is $a[i+1]$. Indeed,

$$\mathsf{up}^n(a[i+1]) = a[(i+1)^{(n)}] = a[i+n+1]$$

holds for all $n \in \mathbb{N}$, i.e., the value that is referenced by the lvalue $a[i+1]$ after n iterations is equal to the value that is referenced by $a[i+n+1]$ initially.

For inductive lvalues, a-solvable loops give rise to systems of recurrence equations.[5]

Definition 18 (Recurrence Equations). *Let $\mathcal{R}$ be a set of symbols with $n \notin \mathcal{R}$. A* recurrence equation *(over $\mathcal{R}$) is an equation $\mathsf{rec}' = e$ where $\mathsf{rec} \in \mathcal{R}$ and e is an arithmetic expression over $\mathcal{R}$. A substitution θ which maps symbols from $\mathcal{R}$ to arithmetic expressions over $\mathcal{R} \cup \{n\}$ is a* solution *for $\mathsf{rec}' = e$ if*

$$\theta(\mathsf{rec})[n/n+1] = \theta(e) \qquad \textit{and} \qquad \theta(\mathsf{rec})[n/0] = \mathsf{rec}$$

are valid. A set of recurrence equations is also called a system*, and θ is a solution for such a system if it is a solution for each of its elements.*

Example 19. Consider the recurrences $i' = i + 1$ and $j' = j + i$. Then θ with $\theta(i) = i + n$ and $\theta(j) = j + \frac{1}{2} \cdot n^2 + n \cdot i - \frac{1}{2} \cdot n$ is a solution, as we have:

$$\theta(i)[n/n+1] = i + n + 1 = \theta(i) + 1 = \theta(i+1) \qquad \theta(i)[n/0] = i + 0 = i$$

$$\theta(j)[n/n+1] = j + \tfrac{1}{2} \cdot n^2 + \tfrac{1}{2} \cdot n + n \cdot i + i = \theta(j) + n + i = \theta(j) + \theta(i) = \theta(j+i)$$

$$\theta(j)[n/0] = j + \tfrac{1}{2} \cdot 0^2 + 0 \cdot i - \tfrac{1}{2} \cdot 0 = j$$

To construct such systems, we apply the *lvalue-substitution* $\sigma_{\mathsf{rec}} := [\ell/\mathsf{rec}_\ell \mid \ell \in \mathit{Lval}]$, which replaces every top-level occurrence of an lvalue ℓ by a fresh symbol rec_ℓ. Now we define the recurrences that yield closed forms for inductive lvalues.

Definition 20 (Rec). Rec *is the system of recurrence equations that contains $\mathsf{rec}'_{x[\mathbf{r}]} = r\sigma_{\mathsf{rec}}$ where $r = \mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])$ for each inductive $x[\mathbf{r}] \in \mathcal{L}$.*

So Rec is a system of recurrences over $\{\mathsf{rec}_\ell \mid \ell \in \mathcal{L}\}$. As the lvalues in $\mathbf{r}$ are updated simultaneously, the value of $x[\mathbf{r}]$ in the *next* iteration corresponds to the value that is written to $x[\mathsf{up}(\mathbf{r})]$ in the current iteration, so we have $r = \mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])$. This is in line with Def. 11, which ensures $x[\mathsf{up}(\mathbf{r})] \in \vec{\ell} = \mathrm{dom}(\mathsf{rhs})$.

Example 21 (Rec). For $\mathcal{T}_{swap}$, we get $\mathsf{Rec} = \{\mathsf{rec}'_i = \mathsf{rec}_i + 1, \mathsf{rec}'_{a[i]} = \mathsf{rec}_{a[i]}\}$. Note that we indeed have $\mathsf{rhs}(i) = i + 1$ and $\mathsf{rhs}(a[\mathsf{up}(i)]) = \mathsf{rhs}(a[i+1]) = a[i]$. Then θ with $\theta(\mathsf{rec}_i) = \mathsf{rec}_i + n$ and $\theta(\mathsf{rec}_{a[i]}) = \mathsf{rec}_{a[i]}$ is a solution of Rec, as:

$$\theta(\mathsf{rec}_i)[n/n+1] = (\mathsf{rec}_i + n)[n/n+1] = \mathsf{rec}_i + n + 1 = \theta(\mathsf{rec}_i + 1)$$

$$\theta(\mathsf{rec}_{a[i]})[n/n+1] = \mathsf{rec}_{a[i]}[n/n+1] = \mathsf{rec}_{a[i]} = \theta(\mathsf{rec}_{a[i]})$$

$$\theta(\mathsf{rec}_i)[n/0] = (\mathsf{rec}_i + n)[n/0] = \mathsf{rec}_i \qquad \theta(\mathsf{rec}_{a[i]})[n/0] = \mathsf{rec}_{a[i]}[n/0] = \mathsf{rec}_{a[i]}$$

To solve Rec, standard techniques for recurrence solving can be used (see, e.g. [4]). While these techniques are very powerful, there are of course cases where Rec cannot be solved, so that our approach cannot accelerate $\mathcal{T}_{loop}$. In the sequel, we assume that Rec can be solved. For inductive lvalues $\ell \in \mathcal{L}$, the following theorem shows that a solution of Rec allows us to construct a closed form.

[5] Our definition of recurrence equations is less general than usual, as we only need recurrences of order 1, and the initial conditions are always the same in our setting.

Theorem 22 (Closed Forms for Inductive Lvalues). *Let $\mathcal{T}_{loop}$ be a-solvable and let θ be a solution for* Rec*. Then for each inductive $\ell \in \mathcal{L}$, $\ell^{(n)} := \theta(\mathsf{rec}_\ell)\sigma_{\mathsf{rec}}^{-1}$ is a closed form for ℓ. Here, $\sigma_{\mathsf{rec}}^{-1}$ is the inverse of σ_{rec}.*

Example 23. Continuing Ex. 21, we get:

$$i^{(n)} = \theta(\mathsf{rec}_i)\sigma_{\mathsf{rec}}^{-1} = (\mathsf{rec}_i + n)\sigma_{\mathsf{rec}}^{-1} = i + n \qquad a[i]^{(n)} = \theta(\mathsf{rec}_{a[i]})\sigma_{\mathsf{rec}}^{-1} = \mathsf{rec}_{a[i]}\sigma_{\mathsf{rec}}^{-1} = a[i]$$

Indeed, we have $i + n = \mathsf{up}^n(i)$ and $a[i] = \mathsf{up}^n(a[i])$ for all $n \in \mathbb{N}$, i.e., the value that is referenced by $a[i]$ after n iterations is equal to the initial value of $a[i]$.

Lemma 15, Thm. 16, and Thm. 22 give rise to an algorithm for computing closed forms for *all* lvalues in $\mathcal{L}$ (as defined in Def. 11) whenever $\mathcal{T}_{loop}$ is a-solvable:

1. compute closed forms for all trivial lvalues via Lemma 15
2. compute closed forms for all inductive lvalues via Thm. 22
3. while there is an lvalue whose closed form has not yet been computed
   (a) pick an $x[\mathbf{r}] \in \mathcal{L}$ where the closed forms for all $\ell \in \mathit{Lval}(\mathbf{r})$ are known
   (b) compute the closed form for $x[\mathbf{r}]$ as in Thm. 16

For Step (3a), if $x[\mathbf{r}] \in \mathcal{L}$ is displacing and the closed form for $\ell \in \mathit{Lval}(\mathbf{r})$ has not yet been computed, then ℓ is displacing, too, and thus one can consider ℓ instead. As ℓ is a subterm of $x[\mathbf{r}]$, this yields a terminating approach for picking $x[\mathbf{r}]$.

5 Closed Forms for Arrays

We now show how to compute closed forms for arrays, given closed forms for all relevant lvalues, i.e., for all $\ell \in \mathcal{L}$ (as defined in Def. 11). The following theorem is an important step towards closed forms for arrays.

Lemma 24 (Towards Closed Forms for Arrays). *Let $\mathcal{T}_{loop}$ be a-solvable and let $x \in \mathcal{V}$. For each $x[\mathbf{r}] \in \vec{\ell}$, we define the following auxiliary predicates:*

$$\mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) \iff \mathbf{r}^{(m-1)} = \mathbf{c}$$

$$!\mathsf{written}_x(m, n, \mathbf{c}) \iff \bigwedge_{x[\mathbf{r}] \in \vec{\ell}} \forall m' \in [m\,..\,n].\ \neg\mathsf{written}_{\mathbf{r}}(m', \mathbf{c})$$

$$\mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) \iff \mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) \land !\mathsf{written}_x(m+1, n, \mathbf{c})$$

Then the following holds for all $x[\mathbf{r}] \in \vec{\ell}$ and all $\mathbf{c} \in \mathbb{Z}^{\mathsf{arity}(x)}$:

$$\forall m \in [1\,..\,n].\ \mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) \implies \mathsf{up}^n(x)[\mathbf{c}] = \mathsf{rhs}(x[\mathbf{r}])^{(m-1)} \qquad \boxed{\text{QUANT}}$$

Intuitively, $\mathsf{written}_{\mathbf{r}}(m, \mathbf{c})$ checks whether the write access $x[\mathbf{r}] \leftarrow \ldots$ modifies the cell $x[\mathbf{c}]$ in the m^{th} iteration. The predicate $!\mathsf{written}_x(m, n, \mathbf{c})$ is true if $x[\mathbf{c}]$ does not change between the m^{th} and the n^{th} iteration. The predicate $\mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c})$ holds if the last write access to $x[\mathbf{c}]$ (up until the n^{th} iteration) takes place in the m^{th} iteration via the assignment $x[\mathbf{r}] \leftarrow \ldots$. Note that we have $x[\mathbf{r}] \in \vec{\ell}$, so $\mathsf{rhs}(x[\mathbf{r}])$ is well defined. As the final value of $x[\mathbf{c}]$ is written by $x[\mathbf{r}] \leftarrow \mathsf{rhs}(x[\mathbf{r}])$ in the m^{th} iteration, the final value of $x[\mathbf{c}]$ is equal to the value of the right-hand side after the $(m-1)^{th}$ iteration, i.e., $\mathsf{rhs}(x[\mathbf{r}])^{(m-1)}$.

Example 25 (Towards Closed Forms for Arrays). Continuing Ex. 23, we obtain the following definitions for written and !written due to the lvalues $a[i]$ and $a[i+1]$:

$$\mathsf{written}_i(m,c) \iff i+m-1=c \qquad \mathsf{written}_{i+1}(m,c) \iff i+m=c$$
$$!\mathsf{written}_a(m,n,c) \iff \forall m' \in [m\,..\,n].\ \neg\mathsf{written}_i(m',c) \wedge \neg\mathsf{written}_{i+1}(m',c)$$

Here, we combined two universal quantifiers to simplify !written. The definition of last_write is as in Lemma 24. Then by Lemma 24, for all $c \in \mathbb{Z}$ we have:

$$\begin{aligned} \forall m \in [1\,..\,n].\ \mathsf{last\_write}_{a[i]}(m,n,c) &\implies \mathsf{up}^n(a)[c] = (a[i+1])^{(m-1)} = a[i+m] \\ \forall m \in [1\,..\,n].\ \mathsf{last\_write}_{a[i+1]}(m,n,c) &\implies \mathsf{up}^n(a)[c] = (a[i])^{(m-1)} = a[i] \end{aligned} \quad (2)$$

Lemma 24 indicates that a closed form for x might have the form

$$\begin{aligned} x^{(n)} := \lambda\mathbf{c}.\ \mathbf{if}\ (1 \leq m \leq n \wedge \mathsf{last\_write}_{x[\mathbf{r}_1]}(m,n,\mathbf{c}))\ \mathsf{rhs}(x[\mathbf{r}_1])^{(m-1)}\ \mathbf{else\ if}\ \ldots \\ \mathbf{else\ if}\ (1 \leq m \leq n \wedge \mathsf{last\_write}_{x[\mathbf{r}_k]}(m,n,\mathbf{c}))\ \mathsf{rhs}(x[\mathbf{r}_k])^{(m-1)}\ \mathbf{else}\ x[\mathbf{c}] \end{aligned}$$

where the value of m is uniquely determined by last_write. However, due to the additional free variable m, such an expression is not a closed form in the sense of Def. 7, and thus it is not suitable for acceleration. Moreover, last_write is defined in terms of !written, and the latter contains universal quantifiers, which complicates subsequent automated reasoning.

Thus, we now show how to eliminate both the variable m and the universal quantifiers in !written in the important special case that for every $x[\mathbf{r}] \in \vec{\ell}$, $\mathsf{up}(\mathbf{r}) - \mathbf{r} = \mathbf{d}$ is a vector of constants (which may differ for each $x[\mathbf{r}] \in \vec{\ell}$). Note that if $\mathbf{d}$ is a vector of constants, then we have $\mathbf{r}^{(n)} = \mathbf{r} + \mathbf{d} \cdot n$.

Eliminating the Quantifier from !written: For !written, we get:

$$\begin{aligned} & \forall m' \in [m\,..\,n].\ \neg\mathsf{written}_{\mathbf{r}}(m',\mathbf{c}) \\ \iff & \forall m' \in [m\,..\,n].\ \mathbf{r}^{(m'-1)} \neq \mathbf{c} && \text{(by def. of } \mathsf{written}) \\ \iff & \forall m' \in [m\,..\,n].\ \mathbf{r} + \mathbf{d} \cdot (m'-1) \neq \mathbf{c} && (\text{as } \mathbf{r}^{(n)} = \mathbf{r} + \mathbf{d} \cdot n) \\ \iff & \forall m' \in [m\,..\,n].\ \bigvee_{i \in [1..|\mathbf{r}|]} r_i + d_i \cdot (m'-1) \neq c_i \\ \iff & \bigvee_{\substack{i \in [1..|\mathbf{r}|] \\ d_i \geq 0}} c_i < r_i + d_i \cdot (m-1) \vee r_i + d_i \cdot (n-1) < c_i \quad \vee \\ & \bigvee_{\substack{i \in [1..|\mathbf{r}|] \\ d_i < 0}} c_i < r_i + d_i \cdot (n-1) \vee r_i + d_i \cdot (m-1) < c_i \quad \vee \bigvee_{\substack{i \in [1..|\mathbf{r}|] \\ |d_i| > 1}} d_i \nmid (c_i - r_i) && \text{(qe)} \end{aligned}$$

Again, "|" denotes divisibility, so $d_i \nmid (c_i - r_i)$ is true iff d_i does not divide $c_i - r_i$. In the last step, the first two sub-formulas $\bigvee \ldots$ cover the case that

$$c_i \notin \left[\min_{m' \in [m,n]}(r_i + d_i \cdot (m'-1)), \max_{m' \in [m,n]}(r_i + d_i \cdot (m'-1))\right], \quad (3)$$

which immediately implies $r_i + d_i \cdot (m'-1) \neq c_i$. Here, we consider the cases $d_i \geq 0$ (first $\bigvee \ldots$) and $d_i < 0$ (second $\bigvee \ldots$) separately. If (3) does not hold, then we have $\forall m' \in [m\,..\,n].\ r_i + d_i \cdot (m'-1) \neq c_i$ iff $c_i - r_i$ is not divisible by d_i.

Eliminating the Quantifier from (Quant): Next, we eliminate m from the negated matrix of (Quant), i.e., from

$$m \in [1..n] \wedge \mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) \wedge \mathsf{up}^n(x)[\mathbf{c}] \neq \mathsf{rhs}(x[\mathbf{r}])^{(m-1)}, \qquad \boxed{!\text{QUANT}}$$

by deriving an instantiation $e_{x[\mathbf{r}]}$ so that $\exists m. \boxed{!\text{QUANT}} \iff \boxed{!\text{QUANT}}[m/e_{x[\mathbf{r}]}]$ and thus $\boxed{\text{QUANT}} \iff \neg\boxed{!\text{QUANT}}[m/e_{x[\mathbf{r}]}]$ holds. We get:

$$\begin{aligned}
\mathsf{last_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) &\iff \mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) \wedge \ldots && \text{(by def. of } \mathsf{last_write}\text{)}\\
&\iff \mathbf{r}^{(m-1)} = \mathbf{c} \wedge \ldots && \text{(by def. of } \mathsf{written}\text{)}\\
&\iff \mathbf{r} + \mathbf{d} \cdot (m-1) = \mathbf{c} \wedge \ldots && \text{(by def. of } \mathbf{r}^{(n)}\text{)}\\
&\iff \bigwedge_{\substack{i \in [1..|\mathbf{r}|]\\ d_i = 0}} r_i = c_i \wedge \bigwedge_{\substack{i \in [1..|\mathbf{r}|]\\ d_i \neq 0}} d_i | (c_i - r_i) \wedge m = (c_i - r_i) \mathrel{\mathsf{div}} d_i + 1 \wedge \ldots
\end{aligned}$$

The last step results from solving the equation $\mathbf{r} + \mathbf{d} \cdot (m-1) = \mathbf{c}$ for m. More explicitly, this equation means $\bigwedge_{i=1}^{|\mathbf{r}|} r_i + d_i \cdot (m-1) = c_i$. So for each conjunct, we obtain $m = \frac{c_i - r_i}{d_i} + 1$ if $d_i \neq 0$, or $r_i = c_i$ if $d_i = 0$. As m is an integer, the former is equivalent to $m = (c_i - r_i) \mathrel{\mathsf{div}} d_i + 1 \wedge d_i | (c_i - r_i)$. So if there is a $d_i \neq 0$, then m can be eliminated by propagating the equality $m = (c_i - r_i) \mathrel{\mathsf{div}} d_i + 1$.

If *all* elements of $\mathbf{d}$ are 0, then $\mathbf{r}$ does not change at all, and we obtain:

$$\begin{aligned}
\mathsf{last_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) &\iff \mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) \wedge !\mathsf{written}_x(m+1, n, \mathbf{c}) && \text{(def. of } \mathsf{last_write}\text{)}\\
&\iff \mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) \wedge m \geq n
\end{aligned}$$

The reason for the last step is that, in this case, the write access $x[\mathbf{r}] \leftarrow \ldots$ overwrites the same cell in every iteration. So $\mathsf{written}_{\mathbf{r}}(m, \mathbf{c})$ implies $\mathsf{written}_{\mathbf{r}}(m+1, \mathbf{c})$, which contradicts $!\mathsf{written}_x(m+1, n, \mathbf{c})$, unless we have $m + 1 > n$. In the latter case, $!\mathsf{written}_x(m+1, n, \mathbf{c})$ is trivially true. As m ranges over $[1, n]$ in $\boxed{!\text{QUANT}}$, $m+1 > n$ implies $m = n$, so we can instantiate m with n if $\mathbf{d} = [0, \ldots, 0]$.

Hence, if the indices of all array accesses only change by constants in every iteration, then all quantifiers that occur in Lemma 24 can be eliminated.

Example 26 (Quantifier Elimination). For $\mathcal{T}_{swap}$, we have $\mathbf{d} = 1$ for both $a[i]$ and $a[i+1]$. As explained above, we can simplify $!\mathsf{written}_a(m, n, c)$ from Ex. 25:

$$(c < i + m - 1 \vee i + n - 1 < c) \wedge (c < i + m \vee i + n < c) \qquad (4)$$

Moreover, the final formulas (2) from Ex. 25 can be simplified to:

$$\begin{aligned}
1 \leq c - i + 1 \leq n \wedge \mathsf{last_write}_{a[i]}(c - i + 1, n, c) &\implies \mathsf{up}^n(a)[c] = a[c+1]\\
1 \leq c - i \leq n \wedge \mathsf{last_write}_{a[i+1]}(c - i, n, c) &\implies \mathsf{up}^n(a)[c] = a[i]
\end{aligned} \qquad (5)$$

Here, we propagated the equality $m = c - i + 1$ (implied by $\mathsf{last\_write}_{a[i]}(m, n, c)$) in the first line, and the equality $m = c - i$ (implied by $\mathsf{last\_write}_{a[i+1]}(m, n, c)$) in the second line. For the first occurrence of $\mathsf{last\_write}$ in (5), we get:

$$\begin{aligned}
&\mathsf{last_write}_{a[i]}(c - i + 1, n, c)\\
\iff\ & i^{(c-i)} = c \wedge !\mathsf{written}_a(c - i + 2, n, c) && \text{(by def. of } \mathsf{last_write} \text{ and } \mathsf{written}\text{)}\\
\iff\ & i + c - i = c \wedge !\mathsf{written}_a(c - i + 2, n, c) \iff !\mathsf{written}_a(c - i + 2, n, c) \overset{(4)}{\iff} \mathsf{true}
\end{aligned}$$

Similarly, for the second occurrence of $\mathsf{last\_write}$ in (5), we get:

$$\mathsf{last\_write}_{a[i+1]}(c - i, n, c) \iff !\mathsf{written}_a(c - i + 1, n, c) \overset{(4)}{\iff} i + n - 1 < c$$

Thus, (5) simplifies to:

$$0 \leq c - i < n \implies \mathsf{up}^n(a)[c] = a[c+1] \qquad 1 \leq c - i = n \implies \mathsf{up}^n(a)[c] = a[i] \qquad (6)$$

The left implication states that $a[i+1], \ldots, a[i+n]$ are shifted to the left by one position, and the right implication states that $a[i]$ gets moved to index $i+n$.

Putting it all together: Now we can finally compute closed forms for arrays. Recall that the quantified variable m from (QUANT) was eliminated via instantiation. For each $x[\mathbf{r}] \in \vec{\ell}$, let $e_{x[\mathbf{r}]}$ be the corresponding instantiation of m, i.e., the expression such that

$$\forall m \in [1 \mathinner{..} n].\ \mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) \implies \mathsf{up}^n(x)[\mathbf{c}] = \mathsf{rhs}(x[\mathbf{r}])^{(m-1)}$$

$$\text{iff} \quad 1 \leq e_{x[\mathbf{r}]} \leq n \wedge \mathsf{last\_write}_{x[\mathbf{r}]}(e_{x[\mathbf{r}]}, n, \mathbf{c}) \implies \mathsf{up}^n(x)[\mathbf{c}] = \mathsf{rhs}(x[\mathbf{r}])^{(e_{x[\mathbf{r}]}-1)}.$$

Then we obtain our main theorem:

Theorem 27 (Closed Forms for Arrays). *Let $x \in \mathcal{V}$, let $\mathcal{T}_{loop}$ be a-solvable, assume that* Rec *can be solved, and let $x[\mathbf{r}_1], \ldots, x[\mathbf{r}_k]$ be all elements of $\vec{\ell}$ of the form $x[\ldots]$. If $\mathsf{up}(\mathbf{r}_i) - \mathbf{r}_i$ is a vector of constants for every $i \in [1 \mathinner{..} k]$, then*

$$x^{(n)} := \lambda \mathbf{c}.$$

$$\mathbf{if}\ (1 \leq e_{x[\mathbf{r}_1]} \leq n \wedge \mathsf{last\_write}_{x[\mathbf{r}_1]}(e_{x[\mathbf{r}_1]}, n, \mathbf{c}))\ \mathsf{rhs}(x[\mathbf{r}_1])^{(e_{x[\mathbf{r}_1]}-1)}\ \mathbf{else\ if} \ldots \tag{7}$$

$$\mathbf{else\ if}\ (1 \leq e_{x[\mathbf{r}_k]} \leq n \wedge \mathsf{last\_write}_{x[\mathbf{r}_k]}(e_{x[\mathbf{r}_k]}, n, \mathbf{c}))\ \mathsf{rhs}(x[\mathbf{r}_k])^{(e_{x[\mathbf{r}_k]}-1)}\ \mathbf{else}\ x[\mathbf{c}]$$

is a closed form for x.

So the closed form $x^{(n)}$ is the array that maps every index $\mathbf{c}$ to the updated right-hand side $\mathsf{rhs}(x[\mathbf{r}_i])^{(e_{x[\mathbf{r}_i]}-1)}$ (which is constructed using the closed forms for lvalues from Sect. 4), where the last write access to the cell $x[\mathbf{c}]$ took place in the ${e_{x[\mathbf{r}_i]}}^{th}$ iteration via an assignment to $x[\mathbf{r}_i]$. As shown when deriving (qe), all quantifiers can be eliminated from the definition of !written (and thus last_write), so Thm. 27 gives rise to *quantifier-free* closed forms for arrays.

Example 28. In Ex. 26, we instantiated m with $e_{a[i]} := c-i+1$ and $e_{a[i+1]} := c-i$, respectively. Thus, similar to (5), we obtain the closed form

$$\begin{aligned} a^{(n)} = \lambda c.\ &\mathbf{if}\ (1 \leq c-i+1 \leq n \wedge \mathsf{last\_write}_{a[i]}(c-i+1, n, c))\ a[c+1] && \mathbf{else} \\ &\mathbf{if}\ (1 \leq c-i \leq n \wedge \mathsf{last\_write}_{a[i+1]}(c-i, n, c))\ a[i] && \mathbf{else}\ a[c] \end{aligned}$$

since $(a[i+1])^{(c-i+1-1)} = a[c+1]$ and $(a[i])^{(c-i-1)} = a[i]$. By eliminating all remaining quantifiers and applying simplifications, analogously to (6), we obtain:

$$a^{(n)} = \lambda c.\ \mathbf{if}\ (0 \leq c-i < n)\ a[c+1]\ \mathbf{else\ if}\ (1 \leq c-i = n)\ a[i]\ \mathbf{else}\ a[c] \tag{8}$$

At this point, we obtained closed form expressions in the sense of Def. 7, i.e., they are suitable for acceleration. As they are quantifier-free, they are also suitable for automation via SMT by handling λs with *lemmas on demand*, see App. A.

6 Related Work

Acceleration techniques for arrays were already studied in [1,28,33], where [28] is restricted to loops where arrays are partitioned into read- and write-only arrays.

The technique from [33] does not require such a partitioning, but if the same array cell is written more than once (as in $\mathcal{T}_{swap}$, where $a[i] \leftarrow \ldots$ overwrites a cell that has already been written by $a[i+1] \leftarrow \ldots$ in the previous iteration), then [33] cannot capture the effect of the loop precisely. Moreover, to compute a closed form for an array a, [33] requires that the closed forms for right-hand sides of updates of the form $a[i] \leftarrow \ldots$ are already known. In particular, this means that a itself must not occur on the right-hand side.

In [1], it is shown how to accelerate *local ground assignments*. Their definition imposes certain restrictions that are not required in our setting. In particular, the loop body must only update each array a at a single index i. So their approach does not apply to, e.g., $\mathcal{T}_{swap}$. In contrast to [33], restricted forms of occurrences of the array a are possible on the right-hand side of $a[i] \leftarrow \ldots$

A similar approach to our elimination of the quantifiers from !written in Sect. 5 is also used in [1], but our technique for eliminating the quantifier from (Quant) is specific to our approach. Moreover, [1] also uses λs for the result of acceleration, but only to simplify the presentation. In their implementation, they use quantifiers instead, whereas we use SMT solving with λs.

Subsequently, the results from [1] have been used in [2] to show decidability of safety for a certain class of array programs. In an orthogonal line of research, [3] describes a framework to combine acceleration techniques.

7 Implementation and Evaluation

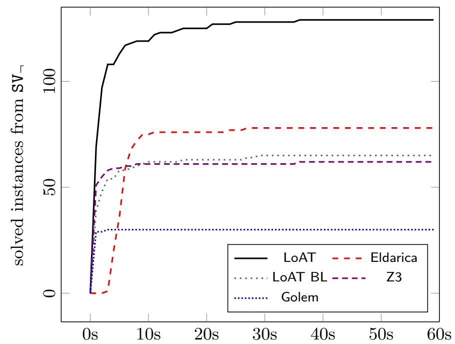
We implemented our approach in LoAT [20], where we use it in our model checking algorithm ABMC [23], which integrates acceleration into Bounded Model Checking (BMC). The implementation uses our SMT solver SwInE [22], which is built on top of Z3 [36]. The input format of LoAT are Constrained Horn Clauses (CHCs) [15], where (un)safe verification tasks correspond to (un)satisfiable CHCs.

We used our novel tool HornKlaus [24] to transform the C benchmarks from the category `ReachSafety-Arrays` of the Software Verification Competition (SV-COMP [7]) to CHCs. HornKlaus does not (yet) aim to preserve (un)safety in all cases, as it does not take all intricacies of the C standard into account. Instead, its purpose is to generate CHCs that mirror the given *algorithm* as closely as possible. Thus, HornKlaus cannot yet be used to compare CHC solvers with C verifiers, but it has certain advantages over other tools that convert C to CHCs: SeaHorn [27] converts multi-dimensional arrays to one-dimensional arrays, resulting in a larger gap between the C program and the CHCs, and Korn [17] often yields non-linear CHCs (which are not yet supported by LoAT) when HornKlaus does not.

Currently, HornKlaus supports a fragment of C that corresponds to the CHCs handled by LoAT (e.g., it does not support pointers, structs, or bit-wise operations). Thus, it can only transform 201 of the 439 SV-COMP benchmarks into CHCs, resulting in our first collection of benchmarks, called SV. However, most of these benchmarks are satisfiable, but ABMC's main purpose is proving unsat. To obtain unsatisfiable instances, we modified HornKlaus so that it negates the assertions that characterize the error states, resulting in our second set of benchmarks $\mathsf{SV}_{\neg}$.

We compared LoAT with the state-of-the-art CHC solvers Eldarica [30], Golem [8], and Z3 [36]. We tested several configurations, and used the most powerful ones for proving unsat: The default configuration of Eldarica 2.2.1, the symbolic

		✓	unsat	sat
SV (201 ex.)	LoAT	25	24	1
	Eldarica	2	0	2
	LoAT BL	2	2	0
	Z3	1	0	1
	Golem	0	0	0
$\mathtt{SV}_{\neg}$ (201 ex.)	LoAT	129	120	9
	Eldarica	78	68	10
	LoAT BL	65	65	0
	Z3	62	62	0
	Golem	30	30	0



execution engine of Golem 0.9.0, and the BMC implementation of Z3 4.15.4. Additionally, we compared with a baseline variant LoAT BL without the contributions from Sections 4 and 5, i.e., it can only accelerate loops without arrays.

Note that some SV-COMP benchmarks contain non-linear arithmetic, which is not supported by Golem, and sometimes prevents Eldarica from computing interpolants. Moreover, each of our benchmark collections contains 13 examples with "curried" arrays (see Sect. 3), which are not supported by LoAT.

We could not compare with MCMT [1–3], a state-of-the-art model checker that supports array acceleration. The reason is that MCMT's array acceleration introduces quantifiers, which are then handled via over-approximations (see [26] for details), and thus they cannot be used for proving unsafety. This shows the importance of our quantifier elimination technique from Sect. 5 and the use of λs, as these two ingredients allow us to use acceleration for proving unsatisfiability.

We used CLAIX-2023-HPC nodes of the RWTH University High Performance Computing Cluster with 10560 MiB memory limit and 60 s timeout per example.

The results of our experiments are shown in the table above. On SV, all tools perform quite badly, which confirms the observation from Sect. 1 that model checking array programs is notoriously difficult. LoAT is the only tool that can prove unsat in some cases. On $\mathtt{SV}_{\neg}$, all tools can prove unsat sometimes, but LoAT is significantly ahead of its closest competitors. Whenever another tool can prove unsatisfiability, LoAT can do so as well (apart from 3 of the examples with "curried" arrays). The fact that LoAT outperforms LoAT BL shows that its good results are indeed due to our novel contributions. For $\mathtt{SV}_{\neg}$, the plot on the right depicts the runtimes for solved instances. It shows that LoAT solves most instances quickly, i.e., our approach is not only competitive in terms of solved instances, but it is also quite efficient. See [18, 25] for more details on LoAT and our evaluation.

Conclusion: We presented a new loop acceleration technique for array loops. By using the novel notion of *inductive lvalues*, it can accelerate more classes of loops than earlier techniques, and it handles arrays and scalars uniformly. Due to the use of λ-expressions and our technique for eliminating quantifiers from Sect. 5, it improves over previous SMT-based model checking algorithms for array programs, in particular for proving unsafety, as demonstrated by our evaluation.

Acknowledgments: We thank Matthias Heizmann for vital initial discussions.

References

1. Alberti, F., Ghilardi, S., Sharygina, N.: Definability of accelerated relations in a theory of arrays and its applications. In: FroCoS 13. pp. 23–39. LNCS 8152 (2013). https://doi.org/10.1007/978-3-642-40885-4_3
2. Alberti, F., Ghilardi, S., Sharygina, N.: Decision procedures for flat array properties. J. Autom. Reason. **54**(4), 327–352 (2015). https://doi.org/10.1007/S10817-015-9323-7
3. Alberti, F., Ghilardi, S., Sharygina, N.: A new acceleration-based combination framework for array properties. In: FroCoS 15. pp. 169–185. LNCS 9322 (2015). https://doi.org/10.1007/978-3-319-24246-0_11
4. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. CoRR **abs/cs/0512056** (2005). https://doi.org/10.48550/arXiv.cs/0512056
5. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Acceleration from theory to practice. Int. J. Softw. Tools Technol. Transf. **10**(5), 401–424 (2008). https://doi.org/10.1007/s10009-008-0064-3
6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016), https://smt-lib.org/
7. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: TACAS '25. pp. 151–186. LNCS 15698 (2025). https://doi.org/10.1007/978-3-031-90660-2_9, website of SV-COMP: https://sv-comp.sosy-lab.org
8. Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn solver. In: CAV '23. pp. 209–223. LNCS 13965 (2023). https://doi.org/10.1007/978-3-031-37703-7_10
9. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. Theor. Comput. Sci. **309**(1-3), 413–468 (2003). https://doi.org/10.1016/S0304-3975(03)00314-1
10. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS '09. pp. 337–351. LNCS 5505 (2009). https://doi.org/10.1007/978-3-642-00768-2_29
11. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: CAV '10. pp. 227–242. LNCS 6174 (2010). https://doi.org/10.1007/978-3-642-14295-6_23
12. Bozga, M., Iosif, R., Konečný, F.: Relational analysis of integer programs. Tech. Rep. TR-2012-10, VERIMAG (2012), https://www-verimag.imag.fr/TR/TR-2012-10.pdf
13. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI '06. pp. 427–442. LNCS 3855 (2006). https://doi.org/10.1007/11609773_28
14. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. J. Satisf. Boolean Model. Comput. **6**(1-3), 165–201 (2009). https://doi.org/10.3233/SAT190067
15. CHC Competition Input Format, https://chc-comp.github.io/format.html
16. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and Presburger arithmetic. In: CAV '98. pp. 268–279. LNCS 1427 (1998). https://doi.org/10.1007/BFb0028751
17. Ernst, G.: Loop verification with invariants and contracts. In: VMCAI '22. pp. 69–92. LNCS 13182 (2022). https://doi.org/10.1007/978-3-030-94583-1_4
18. Frohn, F.: LoAT website (2018), https://loat-developers.github.io/LoAT/
19. Frohn, F.: A calculus for modular loop acceleration. In: TACAS '20. pp. 58–76. LNCS 12078 (2020). https://doi.org/10.1007/978-3-030-45190-5_4

20. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (system description). In: IJCAR '22. pp. 712–722. LNCS 13385 (2022). https://doi.org/10.1007/978-3-031-10769-6_41
21. Frohn, F., Giesl, J.: ADCL: Acceleration Driven Clause Learning for constrained Horn clauses. In: SAS '23. pp. 259–285. LNCS 14284 (2023). https://doi.org/10.1007/978-3-031-44245-2_13
22. Frohn, F., Giesl, J.: Satisfiability modulo exponential integer arithmetic. In: IJCAR '24. pp. 344–365. LNCS 14739 (2024). https://doi.org/10.1007/978-3-031-63498-7_21
23. Frohn, F., Giesl, J.: Integrating loop acceleration into bounded model checking. In: FM '24. pp. 73–91. LNCS 14933 (2024). https://doi.org/10.1007/978-3-031-71162-6_4
24. Frohn, F.: HornKlaus (2026), https://github.com/LoAT-developers/HornKlaus
25. Frohn, F., Giesl, J.: Evaluation of "Accelerating Loops with Arrays" (2026), https://loat-developers.github.io/arrays-eval/
26. Ghilardi, S.: MCMT v. 3.0 - User Manual (2020), http://users.mat.unimi.it/users/ghilardi/mcmt/UM_MCMT_3.0.pdf
27. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV '15. pp. 343–361. LNCS 9206 (2015). https://doi.org/10.1007/978-3-319-21690-4_20
28. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for arrays (tool paper). In: LPAR 10. pp. 348–356. LNCS 6397 (2010). https://doi.org/10.1007/978-3-642-16242-8_25
29. Herrmann, R., Rümmer, P.: What's decidable about arrays with sums? In: CADE '25. pp. 56–74. LNCS 15943 (2025). https://doi.org/10.1007/978-3-031-99984-0_4
30. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD '18. pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013
31. Konečný, F.: PTIME computation of transitive closures of octagonal relations. In: TACAS '16. pp. 645–661. LNCS 9636 (2016). https://doi.org/10.1007/978-3-662-49674-9_42
32. Kovács, L.: Reasoning algebraically about p-solvable loops. In: TACAS '08. pp. 249–264. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_18
33. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods Syst. Des. **47**(1), 75–92 (2015). https://doi.org/10.1007/s10703-015-0228-1
34. McCarthy, J.: Towards a mathematical science of computation. In: Program Verification - Fundamental Issues in Computer Science, Studies in Cognitive Systems, vol. 14, pp. 35–56. Springer Netherlands (1993). https://doi.org/10.1007/978-94-011-1793-7_2
35. Niemetz, A., Preiner, M.: Bitwuzla. In: CAV '23. pp. 3–17. LNCS 13965 (2023). https://doi.org/10.1007/978-3-031-37703-7_1
36. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS '08. pp. 337–340. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
37. Preiner, M., Niemetz, A., Biere, A.: Lemmas on demand for lambdas. In: Proc. 2nd Int. Workshop on Design and Implementation of Formal Tools and Systems. CEUR Workshop Proceedings 1130 (2013), https://ceur-ws.org/Vol-1130/paper_7.pdf
38. Preiner, M.: Lambdas, Arrays and Quantifiers. Ph.D. thesis, Johannes Kepler Universität Linz (2017), https://epub.jku.at/obvulihs/content/titleinfo/1915381

A Reasoning about Lambdas

Using the approach from Sect. 5, we obtain quantifier-free closed forms for arrays, which allows us to encode many verification problems as SMT problems.

Example 29 (Encoding Verification as SMT). Consider our running example $\mathcal{T}_{swap}$ once more, and assume that we want to verify the Hoare triple

$$\{\varphi \land i < k\}\mathcal{T}_{swap}\{\psi\},$$

where φ and ψ are quantifier-free first-order formulas over $\mathcal{V}$. Together with the closed form (8), we can use an acceleration techniques like [19] to derive the following precise characterization of the transitive closure of $\mathcal{T}_{swap}$:

$$n > 0 \land i + n \leq k \land i' = i + n \land a' = a^{(n)}$$

Then $\{\varphi \land i < k\}\mathcal{T}_{swap}\{\psi\}$ is a valid Hoare triple iff

$$\varphi \land i < k \land n > 0 \land i + n \leq k \land i' = i + n \land a' = a^{(n)} \land i' \geq k \land \neg\psi[a/a', i/i'] \tag{9}$$

is unsatisfiable.

However, while all major SMT solvers support λs as input (via non-standard extensions of SMT-LIB or their API), in our experience, they can rarely solve instances where λ-expressions cannot be eliminated via β-reduction, i.e., they can hardly handle cases where λs cannot be evaluated immediately. However, even in the simple example above, this is not the case, as the λ-expression $a^{(n)}$ occurs in a position that is not β-reducible (in the equation $a' = a^{(n)}$).

Thus, we now describe a simple and incomplete theory solver to deal with λs, which turns out to be very effective in our setting. More precisely, we assume that the input to our theory solver is a conjunction φ of literals that are either *arithmetic literals* that relate arithmetic expressions (via the usual arithmetic relations, including (in)divisibility), or *array literals* that relate array expressions via $=$ or $\neq$. For array literals, we additionally require that the expressions on both sides have the same arity. Here, it is important to note that, like SMT-LIB, we consider *extensional* arrays, i.e., we have

$$a = a' \iff \forall \mathbf{i} \in \mathbb{Z}^{\mathsf{arity}(a)}.\ a[\mathbf{i}] = a'[\mathbf{i}]. \tag{Ext}$$

Then our goal is to either produce a model, or to return `unsat` if no such model exists.

The first ingredient of our theory solver is *equality propagation*.

Example 30 (Equality Propagation). To see why equality propagation is useful, assume that we want to check the Hoare triple

$$\{b = a \land j = i < k\}\mathcal{T}_{swap}\{a[i] = b[j]\}. \tag{10}$$

So b and j store copies of the initial values of a and i, respectively, and the post-condition claims that the final value of $a[i]$ is equal to the initial value of $a[i]$. Instantiating (9) correspondingly yields

$$b = a \land j = i < k \land n > 0 \land i + n \leq k \land i' = i + n \land a' = a^{(n)} \land i' \geq k \land a'[i'] \neq b[j].$$

After propagating the equality $a' = a^{(n)}$, we obtain

$$b = a \land j = i < k \land n > 0 \land i + n \leq k \land i' = i + n \land i' \geq k \land a^{(n)}[i'] \neq b[j].$$

Algorithm 1: Theory Solving with λs

Input: conjunction of literals φ
Result: a model of φ, unsat, or unknown

1 $\varphi \leftarrow \varphi[(p \neq q)/(p[\mathbf{i}] \neq q[\mathbf{i}]) \mid (p \neq q) \in \varphi, \mathsf{arity}(p) > 0, \mathbf{i} \subseteq \mathcal{V}$ fresh] `// remove ≠`
2 **repeat**
3   **if** *φ contains a literal $x = p$ where $x \in \mathcal{V}$* **then** $\varphi \leftarrow \varphi[x/p]$ `// propagate eq.`
4   **if** *φ contains $(\lambda \mathbf{i}.\ e)[\mathbf{e}]$* **then** $\varphi \leftarrow \varphi[(\lambda \mathbf{i}.\ e)[\mathbf{e}]/e[\mathbf{i}/\mathbf{e}]]$ `// beta reduction`
5 **until** *φ does not change anymore*
6 $\varphi_{abs} \leftarrow \varphi[(\lambda \mathbf{i}.\ e)/x_{\mathbf{i},e} \mid (\lambda \mathbf{i}.\ e)$ occurs in $\varphi, x_{\mathbf{i},e} \in \mathcal{V}$ fresh]
7 $Idx \leftarrow \{\mathbf{e} \mid a[\mathbf{e}] \in Lval(\varphi), \mathsf{arity}(a) > 0\}$
8 **repeat**
9   **if** *φ_{abs} is* unsat **then return** unsat
10   let $\mathcal{M}$ be a model of φ_{abs}
11   **if** *$\mathcal{M}$ is a model of φ* **then return** $\mathcal{M}$
12   **foreach** $(p = q) \in \varphi$ *and* $\mathbf{e} \in Idx$ *such that $\mathcal{M}$ violates* $p[\mathbf{e}] = q[\mathbf{e}]$ **do**
13         $\varphi_{abs} \leftarrow \varphi_{abs} \wedge p[\mathbf{e}] = q[\mathbf{e}]$ `// refinement`
14 **until** *φ_{abs} does not change anymore*
15 **return** unknown `// refinement failed`

Now $a^{(n)}[i']$ can be β-reduced, and the resulting λ-free formula can be handled by off-the-shelf SMT techniques. Indeed, by taking the equality $i' = i + n$ into account, it becomes apparent that $a^{(n)}[i']$ evaluates to $a[i]$ due to the second case of (8) if $n > 0$, or due to the third case if $n = 0$, i.e., we obtain

$$b = a \wedge j = i \wedge \ldots \wedge a[i] \neq b[j],$$

which is clearly unsatisfiable. Thus, (10) is valid.

Due to (Ext), an array literal $p \neq q$ can easily be converted into the arithmetic literal $p[\mathbf{i}] \neq q[\mathbf{i}]$ where $\mathbf{i}$ consists of fresh (implicitly existentially quantified) scalar variables. Using these two ingredients and β-reduction, our theory solver tries to eliminate as many λ-expressions as possible, which constitutes the first part of our theory solver, see Algorithm 1. Here, we slightly abuse notation by using "substitutions" that replace expressions or literals instead of variables, and we do not mention arity constraints explicitly (i.e., whenever we write, e.g., $p[\mathbf{i}]$, we implicitly assume $|\mathbf{i}| = \mathsf{arity}(p)$).

Then we *abstract* the SMT problem by replacing all remaining λ-expressions with fresh variables of the corresponding arity (Line 6). If the resulting SMT problem is unsatisfiable, then the original problem φ is unsatisfiable as well, so our solver returns unsat (Line 9). Otherwise, we check if the model that we obtained for the abstraction is also a model for φ, and in that case, our solver returns it (Line 11). If this is not the case, then we refine the abstraction by considering the set Idx of all vectors of expressions $\mathbf{e}$ that occur syntactically as indices[6] in parts of φ that are not below a λ, and all array literals $p = q$ that were abstracted (Lines 7 and 12 – note that β-redexes and array-inequations can *always* be eliminated): For each pair of a literal $p = q$ and an index $\mathbf{e}$ of the corresponding arity where $p[\mathbf{e}] = q[\mathbf{e}]$ is violated, we add $p[\mathbf{e}] = q[\mathbf{e}]$ to the

[6] This idea stems from the decision procedure for the *array property fragment* [13].

abstraction (Line 13). If we find at least one such pair, then the refinement was successful and we search for a model for the refined abstraction. Otherwise, the refinement fails and our solver returns unknown (Line 15).

To compute *Idx* in Line 7, we lift *Lval* to expressions as follows:

$$Lval(e) := \begin{cases} \{e\} & \text{if } e \in Lval \\ Lval(e_1) \cup Lval(e_2) & \text{if } e = e_1 \circ e_2 \\ & \quad \text{where } \circ \text{ is an arithmetic operator} \\ Lval(\mu) \cup Lval(e_1) \cup Lval(e_2) & \text{if } e = \mathbf{if}\ (\mu)\ e_1\ \mathbf{else}\ e_2 \\ \varnothing & \text{otherwise} \end{cases}$$

Moreover, for quantifier-free formulas, we define:

$$Lval(\mu) := \begin{cases} Lval(\mu_1) \cup Lval(\mu_2) & \text{if } \mu = \mu_1 \bullet \mu_2 \text{ where } \bullet \in \{\wedge, \vee, \ldots\} \\ & \quad \text{is a binary Boolean connective} \\ Lval(\mu') & \text{if } \mu = \neg\mu' \\ Lval(e_1) \cup Lval(e_2) & \text{if } \mu = e_1 \bowtie e_2 \text{ is an arithmetic literal} \\ \varnothing & \text{otherwise} \end{cases}$$

So in particular, for array literals we have $Lval(p = q) = \varnothing$, i.e., we do not consider lvalues below λs.

To see why our refinement in Line 13 may fail and Algorithm 1 may reach Line 15, consider the formula

$$(\lambda i.\ 0) = (\lambda i.\ 1)$$

which gets abstracted to

$$x_{i,0} = x_{i,1},$$

and assume that we obtain a model $\mathcal{M}$ with $\mathcal{M}(x_{i,0}) = \mathcal{M}(x_{i,1}) = \lambda i.\ 42$. Then $\mathcal{M}$ is not a model for the original formula, but as *Idx* is empty, the refinement fails and Algorithm 1 returns unknown.

Note that our approach is very similar to *lemmas on demand* [14, 37, 38]. However, their technique is more sophisticated then our approach. For example, our selection of the relevant indices *Idx* is a lot simpler. Moreover, for each array literal $a = b$, *lemmas on demand* adds the implication

$$a \neq b \implies \exists \mathbf{i}.\ a[\mathbf{i}] \neq b[\mathbf{i}]$$

to the SMT problem [38], which is required for concluding satisfiability in cases where the refinement fails. Thus, we return unknown in Line 15, but their algorithm is a decision procedure (assuming that the abstraction in Line 6 yields formulas in a decidable logic). However, to the best of our knowledge, *lemmas on demand* has only been implemented in Bitwuzla [35], which does not support integer arithmetic. Thus, we use our own, incomplete implementation. Improving it is a subject of future work.

B Proofs

We first prove the following auxiliary lemma, which shows that the value that is referenced by a displacing lvalue in the n^{th} iteration has not been changed in previous iterations.

Lemma 31 (Stability of Displacing Lvalues). *If $\mathcal{T}_{loop}$ is monotonic and $x[\mathbf{r}] \in \mathcal{L}$ is displacing, then we have*

$$x[\mathsf{up}^n(\mathbf{r})] = \mathsf{up}^m(x)[\mathsf{up}^n(\mathbf{r})]$$

for all $m \leq n$.

Proof. We use induction on m, where the case $m = 0$ is trivial. Let $m = k + 1$. Then for all $x[\mathbf{r}'] \in \vec{\ell}$, we have:

$$\begin{aligned} \mathsf{up}^k(\mathbf{r}') &< \mathsf{up}^{k+1}(\mathbf{r}) && \text{(as } x[\mathbf{r}] \text{ is displacing)} \\ &\leq \mathsf{up}^n(\mathbf{r}) && \text{(by monotonicity, as } k+1 \leq n\text{)} \end{aligned}$$

Thus,

$$\mathsf{up}^k(\mathbf{r}') < \mathsf{up}^n(\mathbf{r}) \qquad \text{whenever } x[\mathbf{r}'] \in \vec{\ell} \tag{11}$$

Hence, we get

$$\begin{aligned} \mathsf{up}^m(x)[\mathsf{up}^n(\mathbf{r})] &= \mathsf{up}^{k+1}(x)[\mathsf{up}^n(\mathbf{r})] && \text{(as } m = k+1\text{)} \\ &= \mathsf{up}^k(x)[\mathsf{up}^n(\mathbf{r})] && \text{(by } \boxed{\substack{read \\ write}} \text{ due to } \boxed{11}\text{)} \\ &= x[\mathsf{up}^n(\mathbf{r})] && \text{(IH)} \end{aligned}$$

□

B.1 Proof of Thm. 16

Theorem 16 (Closed Forms for Displacing Lvalues). *If $\mathcal{T}_{loop}$ is monotonic and $x[\mathbf{r}] \in \mathcal{L}$ is displacing, then $x[\mathbf{r}]^{(n)} := x[\mathbf{r}^{(n)}]$ is a closed form for $x[\mathbf{r}]$.*

Proof. By Def. 14, we need to show $x[\mathbf{r}]^{(n)} = \mathsf{up}^n(x[\mathbf{r}])$. We have:

$$\begin{aligned} \mathsf{up}^n(x[\mathbf{r}]) &= \mathsf{up}^n(x)[\mathsf{up}^n(\mathbf{r})] \\ &= x[\mathsf{up}^n(\mathbf{r})] && \text{(by Lemma 31)} \\ &= x[\mathbf{r}^{(n)}] && \text{(by Def. 14)} \\ &= x[\mathbf{r}]^{(n)} \end{aligned}$$

□

B.2 Proof of Thm. 22

Theorem 22 (Closed Forms for Inductive Lvalues). *Let $\mathcal{T}_{loop}$ be a-solvable and let θ be a solution for* Rec*. Then for each inductive $\ell \in \mathcal{L}$, $\ell^{(n)} := \theta(\mathsf{rec}_\ell)\sigma_{\mathsf{rec}}^{-1}$ is a closed form for ℓ. Here, $\sigma_{\mathsf{rec}}^{-1}$ is the inverse of σ_{rec}.*

Proof. We use induction on n to show $\ell^{(n)} = \mathsf{up}^n(\ell)$.
Induction Base $-$ $n = 0$**:**

$$\begin{aligned} & \ell^{(n)}[n/0] \\ &= \theta(\mathsf{rec}_\ell)\sigma_{\mathsf{rec}}^{-1}[n/0] && \text{(by def. of } \ell^{(n)}\text{)} \\ &= \theta(\mathsf{rec}_\ell)[n/0]\sigma_{\mathsf{rec}}^{-1} && \text{(as } n \notin \mathrm{dom}(\sigma_{\mathsf{rec}}^{-1}) \cup \mathcal{V}(\mathrm{img}(\sigma_{\mathsf{rec}}^{-1}))\text{)} \\ &= \sigma_{\mathsf{rec}}^{-1}(\mathsf{rec}_\ell) && \text{(by Def. 18)} \\ &= \ell && \text{(by def. of } \sigma_{\mathsf{rec}}\text{)} \end{aligned}$$

$$= \mathsf{up}^0(\ell)$$
$$= \mathsf{up}^n(\ell) \qquad \text{(as } n = 0\text{)}$$

Induction Step $-\ n = m+1$**:**

$$\begin{aligned}
& \ell^{(n)}[n/m+1] \\
&= x[\mathbf{r}]^{(n)}[n/m+1] && \text{(where } \ell = x[\mathbf{r}]\text{)} \\
&= \theta(\mathsf{rec}_{x[\mathbf{r}]})\sigma_{\mathsf{rec}}^{-1}[n/m+1] && \text{(by def. of } \ell^{(n)}\text{)} \\
&= \theta(\mathsf{rec}_{x[\mathbf{r}]})[n/m+1]\sigma_{\mathsf{rec}}^{-1} && \text{(as } n \notin \mathrm{dom}(\sigma_{\mathsf{rec}}^{-1}) \cup \mathcal{V}(\mathrm{img}(\sigma_{\mathsf{rec}}^{-1}))\text{)} \\
&= \theta(\mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])\sigma_{\mathsf{rec}})[n/m]\sigma_{\mathsf{rec}}^{-1} && \text{(by Defs. 18 and 20)} \\
&= \theta(\mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])\sigma_{\mathsf{rec}})\sigma_{\mathsf{rec}}^{-1}[n/m] && \text{(as } n \notin \mathrm{dom}(\sigma_{\mathsf{rec}}^{-1}) \cup \mathcal{V}(\mathrm{img}(\sigma_{\mathsf{rec}}^{-1}))\text{)} \\
&= \mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])[\ell'/\theta(\mathsf{rec}_{\ell'})\sigma_{\mathsf{rec}}^{-1}[n/m] \mid \ell' \in \mathit{Lval}] && \text{(by def. of } \sigma_{\mathsf{rec}}\text{)} \\
&= \mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])[\ell'/\mathsf{up}^m(\ell') \mid \ell' \in \mathit{Lval}] && \text{(IH)} \\
&= \mathsf{up}^m(\mathsf{rhs}(x[\mathsf{up}(\mathbf{r})])) \\
&= \mathsf{up}^m(\mathsf{up}(x)[\mathsf{up}(\mathbf{r})]) && \text{(by } \boxed{\substack{\mathit{read} \\ \mathit{write}}}\text{)} \\
&= \mathsf{up}^m(\mathsf{up}(x[\mathbf{r}])) \\
&= \mathsf{up}^{m+1}(x[\mathbf{r}]) \\
&= \mathsf{up}^{m+1}(\ell) && \text{(as } \ell = x[\mathbf{r}]\text{)}
\end{aligned}$$

□

B.3 Proof of Lemma 24

Lemma 24 (Towards Closed Forms for Arrays). *Let $\mathcal{T}_{loop}$ be a-solvable and let $x \in \mathcal{V}$. For each $x[\mathbf{r}] \in \vec{\ell}$, we define the following auxiliary predicates:*

$$\begin{aligned}
\mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) &\iff \mathbf{r}^{(m-1)} = \mathbf{c} \\
!\mathsf{written}_{x}(m, n, \mathbf{c}) &\iff \bigwedge_{x[\mathbf{r}] \in \vec{\ell}} \forall m' \in [m\,..\,n].\ \neg\mathsf{written}_{\mathbf{r}}(m', \mathbf{c}) \\
\mathsf{last_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) &\iff \mathsf{written}_{\mathbf{r}}(m, \mathbf{c}) \land !\mathsf{written}_{x}(m+1, n, \mathbf{c})
\end{aligned}$$

Then the following holds for all $x[\mathbf{r}] \in \vec{\ell}$ and all $\mathbf{c} \in \mathbb{Z}^{\mathsf{arity}(x)}$:

$$\forall m \in [1\,..\,n].\ \mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c}) \implies \mathsf{up}^n(x)[\mathbf{c}] = \mathsf{rhs}(x[\mathbf{r}])^{(m-1)} \qquad \boxed{\text{QUANT}}$$

Proof. Let $\mathbf{c} \in \mathbb{Z}^{\mathsf{arity}(x)}$ and $m \in [1\,..\,n]$ be arbitrary but fixed and assume that $\mathsf{last\_write}_{x[\mathbf{r}]}(m, n, \mathbf{c})$ holds. Then $!\mathsf{written}_{x}(m+1, n, \mathbf{c})$ implies

$$\hat{\mathbf{r}}^{(m'-1)} \neq \mathbf{c} \qquad \text{for all } m' \in (m, n] \text{ and all } x[\hat{\mathbf{r}}] \in \vec{\ell}. \qquad \boxed{12}$$

and $\mathsf{written}_{\mathbf{r}}(m, \mathbf{c})$ implies $\mathbf{r}^{(m-1)} = \mathbf{c}$. Thus,

$$\begin{aligned}
& \mathsf{up}^n(x)[\mathbf{c}] \\
&= \mathsf{up}^m(x)[\mathbf{c}] && \text{(by } \boxed{12}\text{)} \\
&= \left(\left(\mathsf{up}^{(m-1)} \circ [y/\mathsf{up}_y(\vec{\ell}) \mid y \in \mathcal{V}]\right)(x)\right)[\mathbf{c}] && \text{(by def. of } \mathsf{up}\text{)} \\
&= \mathsf{up}^{(m-1)}(x[y/\mathsf{up}_y(\vec{\ell}) \mid y \in \mathcal{V}])[\mathbf{c}] \\
&= \mathsf{up}^{(m-1)}(\mathsf{up}_x(\vec{\ell}))[\mathbf{c}]
\end{aligned}$$

$$= \mathsf{up}^{(m-1)}(\lambda \mathbf{i}.\ \mathbf{if}\ (\mathbf{i} = \mathbf{r})\ \mathsf{rhs}(x[\mathbf{r}])\ \mathbf{else}\ \ldots)[\mathbf{c}] \qquad (\dagger)$$
$$= (\lambda \mathbf{i}.\ \mathbf{if}\ (\mathbf{i} = \mathsf{up}^{m-1}(\mathbf{r}))\ \mathsf{up}^{m-1}(\mathsf{rhs}(x[\mathbf{r}]))\ \mathbf{else}\ \ldots)[\mathbf{c}]$$
$$= (\lambda \mathbf{i}.\ \mathbf{if}\ (\mathbf{i} = \mathbf{r}^{(m-1)})\ \mathsf{rhs}(x[\mathbf{r}])^{(m-1)}\ \mathbf{else}\ \ldots)[\mathbf{c}]$$
$$= (\lambda \mathbf{i}.\ \mathbf{if}\ (\mathbf{i} = \mathbf{r}^{(m-1)})\ \mathsf{rhs}(x[\mathbf{r}])^{(m-1)}\ \mathbf{else}\ \ldots)[\mathbf{r}^{(m-1)}] \qquad (\text{as } \mathbf{r}^{(m-1)} = \mathbf{c})$$
$$= \mathsf{rhs}(x[\mathbf{r}])^{(m-1)}$$

For the step marked with (†), note that the order of the cases in the definition of $\mathsf{up}_x(\vec{\ell})$ is irrelevant due to DISTINCT, and thus we may assume that the case $\mathbf{i} = \mathbf{r}$ is tested first without loss of generality. □

B.4 Proof of Thm. 27

Theorem 27 (Closed Forms for Arrays). *Let $x \in \mathcal{V}$, let $\mathcal{T}_{loop}$ be a-solvable, assume that* Rec *can be solved, and let $x[\mathbf{r}_1], \ldots, x[\mathbf{r}_k]$ be all elements of $\vec{\ell}$ of the form $x[\ldots]$. If $\mathsf{up}(\mathbf{r}_i) - \mathbf{r}_i$ is a vector of constants for every $i \in [1\,..\,k]$, then*

$$x^{(n)} := \lambda \mathbf{c}.$$
$$\mathbf{if}\ (1 \le e_{x[\mathbf{r}_1]} \le n \wedge \mathsf{last\_write}_{x[\mathbf{r}_1]}(e_{x[\mathbf{r}_1]}, n, \mathbf{c}))\ \mathsf{rhs}(x[\mathbf{r}_1])^{(e_{x[\mathbf{r}_1]}-1)}\ \mathbf{else\ if} \ldots \qquad (7)$$
$$\mathbf{else\ if}\ (1 \le e_{x[\mathbf{r}_k]} \le n \wedge \mathsf{last\_write}_{x[\mathbf{r}_k]}(e_{x[\mathbf{r}_k]}, n, \mathbf{c}))\ \mathsf{rhs}(x[\mathbf{r}_k])^{(e_{x[\mathbf{r}_k]}-1)}\ \mathbf{else}\ x[\mathbf{c}]$$

is a closed form for x.

Proof. Let $\mathbf{c} \in \mathbb{Z}^{\mathsf{arity}(x)}$ be arbitrary but fixed. If there is an $i \in [1\,..\,k]$ such that $1 \le e_{x[\mathbf{r}_i]} \le n \wedge \mathsf{last\_write}_{x[\mathbf{r}_i]}(e_{x[\mathbf{r}_i]}, n, \mathbf{c})$, then $x^{(n)}[\mathbf{c}] = \mathsf{up}^n(x)[\mathbf{c}]$ follows from Lemma 24 and correctness of our quantifier elimination technique. Otherwise, by correctness of our quantifier elimination technique, we have

$$\forall m \in [1\,..\,n].\ \neg\mathsf{last\_write}_{x[\mathbf{r}_i]}(m, n, \mathbf{c})$$

for all $i \in [1\,..\,k]$. By definition of last_write, this implies

$$\forall m \in [1\,..\,n].\ \mathbf{r}_i^{(m-1)} \ne \mathbf{c}.$$

Thus, the cell $x[\mathbf{c}]$ is not written by the loop, i.e., we have $\mathsf{up}^n(x)[\mathbf{c}] = x[\mathbf{c}] = x^{(n)}[\mathbf{c}]$. □

C Semantics of Expressions

In this section, we define the semantics of expressions as introduced in Sect. 3.

Definition 32 (Semantics of Expressions). *Let s with $\mathrm{dom}(s) \subseteq \mathcal{V}$ be a function that maps variables $x \in \mathrm{dom}(s)$ to functions of arity* arity(x)*. The* interpretation *$[\![e]\!]_s$ of an (array) expression e w.r.t. s is defined as follows:*

$$[\![e]\!]_s := \begin{cases} e & \textit{if } e \in \mathbb{Z} \\ [\![e_1]\!]_s \circ [\![e_2]\!]_s & \textit{if } e = e_1 \circ e_2 \\ [\![p]\!]_s\,([\![\mathbf{e}]\!]_s) & \textit{if } e = p[\mathbf{e}], p \textit{ is an array expression} \\ s(e) & \textit{if } e \in \mathcal{V} \cap \mathrm{dom}(s) \\ e & \textit{if } e \in \mathcal{V} \setminus \mathrm{dom}(s) \\ \lambda\mathbf{i}.\ [\![e']\!]_{s|_{\mathrm{dom}(s)\setminus\mathbf{i}}} & \textit{if } e = \lambda\mathbf{i}.\ e' \\ [\![e_1]\!]_s & \textit{if } e = \mathbf{if}\ (\mu)\ e_1\ \mathbf{else}\ e_2\ \textit{and}\ [\![\mu]\!]_s = \mathsf{true} \\ [\![e_2]\!]_s & \textit{if } e = \mathbf{if}\ (\mu)\ e_1\ \mathbf{else}\ e_2\ \textit{and}\ [\![\mu]\!]_s = \mathsf{false} \end{cases}$$

Here, we lift $[\![\_]\!]_s$ *to vectors of expressions and formulas in the obvious way, and* $s|_{\mathrm{dom}(s)\setminus\mathbf{i}}$ *denotes the restriction of* s *to the domain* $\mathrm{dom}(s) \setminus \mathbf{i}$*.*