Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing

Benjamin Farinier¹, Robin David², Sébastien Bardin¹, and Matthieu Lemerre¹

¹ CEA, LIST, Software Safety and Security Lab, Université Paris-Saclay, France
firstname.lastname@cea.fr
² Quarkslab, Paris, France
rdavid@quarkslab.com

Abstract

The theory of arrays has a central place in software verification due to its ability to model memory or data structures. Yet, this theory is known to be hard to solve in both theory and practice, especially in the case of very long formulas coming from unrolling-based verification methods. Standard simplification techniques à la read-over-write suffer from two main drawbacks: they do not scale on very long sequences of stores and they miss many simplification opportunities because of a crude syntactic (dis-)equality reasoning. We propose a new approach to array formula simplification based on a new dedicated data structure together with original simplifications and low-cost reasoning. The technique is efficient, scalable and it yields significant simplification. The impact on formula resolution is always positive, and it can be dramatic on some specific classes of problems of interest, e.g. very long formula or binary-level symbolic execution. While currently implemented as a preprocessing, the approach would benefit from a deeper integration in an array solver.

1 Introduction

Context. Automatic decision procedures for Satisfiability Modulo Theory [4] are at the heart of almost all recent formal verification methods [11, 6, 10, 26]. Especially, the theory of arrays enjoys a central position in software verification as it allows to model memory or essential data structures such as maps, vectors and hash tables.

Intuitively, given a set $I$ of indexes and a set $E$ of elements, the theory of arrays describes the set $\text{Array } I \times E$ of all arrays mapping each index $i \in I$ to an element $e \in E$. Actually, logical arrays can be seen as infinite updatable maps implicitly defined by a succession of writes from an initial map. These arrays are defined by the two operations read (select) and write (store), whose semantic is given in Fig. 1 by so-called read-over-write axioms (ROW-axioms).

Despite its simplicity, the satisfiability problem for the theory of arrays is NP-complete¹. Indeed, it implies deciding (dis-)equalities between read and written indexes on read-over-write

¹Reduction of the program equivalence problem in presence of arrays (sequential, boolean case) to the equivalence case without arrays but with if-then-else operators, to SAT [17].
Arrays Made Simpler

B. Farinier, R. David, S. Bardin, M. Lemerre

select: Array $\mathcal{I} \times \mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{E}$
\[ \forall aie. \text{select}(\text{store } aie) i = e \]

store: Array $\mathcal{I} \times \mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{E} \rightarrow \text{Array } \mathcal{I} \times \mathcal{E}$
\[ \forall ai j e. (i \neq j) \Rightarrow \text{select}(\text{store } aie) j = \text{select } aj \]

Figure 1: The theory of arrays (ROW-axioms)

terms (ROW) of the form $\text{select}(\text{store } aie) j$, potentially yielding nested case-splits. Standard
decision procedures for arrays consist in eliminating as much ROW as possible through a
preprocessing step [22], using axioms from Fig. 1 as rewriting rules, and then enumerating
all possible (dis-)equalities in ROW, yielding a potentially huge search space — the remaining
ROW-axioms can be introduced lazily to mitigate this issue [9].

Problem and challenge. Yet, this is not satisfactory when considering very long chains
of writes, as can be encountered in unfolding-based verification techniques such as Symbolic
Execution (SE) [10] or Bounded Model checking [11] — the case of Deductive Verification is
different since user-defined invariants prevent the unfolding. The theory of arrays can then
quickly become a bottleneck of constraint solving. Especially, the ROW-simplification step is
very often limited, for two reasons. First, exploring for every read in a backward manner the
corresponding list of all writes yields a quadratic time cost (in the number of array operations) and
therefore it does not scale to very long formulas. This is a major issue in practice as, for example,
Symbolic Execution over malware or obfuscated programs [27, 1, 29] may have to consider
execution traces of several millions of instructions, yielding formulas with several hundreds of
thousands of array operations. Note also that bounding the backward exploration misses too
many ROW-simplifications. Second, (dis)-equalities can be rarely decided during preprocessing
as standard methods rely on efficient but crude approximate equality checks (typically, syntactic
term equality), limiting again the power of these approaches. With such checks, index equality
may be sometimes proven, but disequality can never be — except in the very restricted case of
constant-value indexes.

Our proposal. We present a novel approach to ROW-simplification named FAS (Fast Array
Simplification), allowing to scale and to simplify much more ROW than previous approaches.
The technique is based on three key components:

- A re-encoding of write sequences (total order) as sequences of packs of independent writes
  (partial order), together with a dedicated data structure (map list) ensuring scalability;
- A new simple normalization step (base normalization) allowing to amplify the efficiency of
  syntactic (dis-)equality checks;
- A lightweight integration of domain-based reasoning over packs yielding even more successful
  (dis-)equality checks for only a slight overhead.

Experimental results demonstrate that FAS scales over very large formulas (several hundreds of
thousands of ROW) typically coming from Symbolic Execution and can yield very significant
gains in terms of runtime — possibly passing from hours to seconds.
**Contribution.** Our contribution is two-fold:

- We present in detail the new **fas** preprocessing step for scalable and thorough array constraint simplification (Sec. 4), along with its three key components: dedicated data structure (Sec. 4.1), base normalization (Sec. 4.2) and domain reasoning (Sec. 4.4);

- We experimentally evaluate **fas** in different settings for three leading SMT solvers (Sec. 5).

The technique is fast and scalable, it yields a significant reduction of the number of rows with always a positive impact on resolution time. This impact is even dramatic for some key usage scenarios such as SE-like formulas with small **timeout** or very large size.

**Discussion.** In our view, **fas** reaches a sweet spot between efficiency and impact on resolution. Experiments demonstrate that even major solvers benefit from it, with gains ranging from slight to very high depending on the setting. While presented as a preprocessing, **fas** would clearly benefit from a deeper integration inside an array solver, in order to take advantage of more simplification opportunities along the resolution process.

## 2 Motivation

Let us detail how the formula in the left part of Fig. 2 can be simplified into the formula in the right part using our new **fas** simplification procedure for arrays. We focus on the last assertion which involves a read on \( \text{mem}_1 \) at index \( j \). According to arrays semantics (Fig. 1), we must try to decide whether \( i \) and \( j \) (resp. \( \text{eax}_0 \) and \( i \)) are equal or different. The standard syntactic equality check is not conclusive here. But \( \text{esp}_1 \equiv \text{esp}_0 - 64 \), therefore \( j \) can be rewritten into \( \text{esp}_0 \) (base normalization in **fas**), which is exactly \( i \). Hence \( i = j \) is proven. By applying array axioms, we deduce that \( \text{eax}_0 \equiv 1415 \), and the last assertion becomes \( \text{select mem}_1 1415 = 9265 \). We now try to decide whether \( i \) and \( 1415 \) are equal or different. Again, the standard syntactic equality check fails. Yet, by the first assertion we deduce that \( i > 61424 \) (domain propagation in **fas**), leading to \( i \neq 1415 \). Therefore \( \text{mem}_1 \) is safely replaced by \( \text{mem}_0 \) in the last assertion which becomes \( \text{select mem}_0 1415 = 9265 \). Finally, as assertions in the formula now only refer to \( \text{esp}_0 \) and \( \text{mem}_0 \), we erase all the intermediate definitions to obtain the simplified formula.

This little mental gymnastic emphasises two important aspects of ROW-simplifications. First, simplifications often require (dis-)equality reasoning beyond pure syntactic equality. Second, simplifications involve a backward reasoning through the formula which may become prohibitive on large formulas if not treated with care (not shown here, up to 1h simplification time in Fig. 10). Our proposal focuses especially on these two aspects.
3 Background

The theory of arrays has been introduced in Fig. 1 (Sec. 1). As already stated, the main difficulty for reasoning over arrays comes from terms of the form \( \text{select} (\text{store} \ a \ i \ e) \ j \), called read-over-write (ROW), since depending on whether \( i = j \) holds or not, the term evaluates to \( e \) (SELECT-HIT) or to \( \text{select} \ a \ j \) (SELECT-MISS). Array (formula) simplification consists in removing as many ROW as possible before resolution by proving (when possible) the validity of the (dis-) equality of such pairs of indexes \((i, j)\) and rewrite the term accordingly. Such simplification procedures critically depend on two factors: 1. the equality check procedure, and 2. the underlying representation of an array and its revisions arising from successive writes.

The equality check must be both efficient — simplifying a formula must be cheaper than solving it, and correct — all proven (dis-)equalities must indeed hold. It can thus only be approximated, i.e. it is incomplete and may miss some valid (dis-)equalities. The standard solution is to rely on syntactic term equality checking. Obviously this is a crude approximation: disequality can never be proven (but for constant-value indexes), and as exemplified in Sec. 2, small syntactic variations of the same value can hinder proving equalities.

We present now two (unsatisfactory) standard array representations, coming either from the decision procedure community (the list representation: generic but slow) or from the symbolic execution community (the map representation: efficient but restricted).

Arrays represented as lists. The standard representation of an array and its subsequent revisions is basically a “store-chain”, the linked list of all successive writes in the array. Hence a fresh array is simply an empty list, while the array obtained by writing an element \( e \) at index \( i \) in array \( A \) is represented by a node containing \((i, e)\) and pointing to the list representing \( A \). Fig. 3 illustrates this encoding. This approach is very generic — it can cope with symbolic indexes, and it is the one implicitly used inside array solvers. In order to simplify a read at index \( j \) on array \( A \), one must decide whether \( i = j \) is valid for the pair \((i, e)\) inside the head of the list representing \( A \). If we succeed, then we can apply the ROW axiom and replace the read by value \( e \). Otherwise, we try to decide whether \( i \neq j \) is valid. If this is the case, then we use the second ROW axiom and move backward along the linked list. If not, the simplification process stops.

An inherent problem with this representation is the increase in the simplification cost as the number of writes rises. As mentioned in Sec. 2, this cost becomes prohibitive when dealing with large formulas. Indeed, one might be forced for each read to fully explore the write-list backward, yielding a quadratic worst case time cost. This is especially unfortunate because this worst case arises in situations where the simplification could perform the best, e.g. when all disequalities between indexes hold so that all reads could be replaced with accesses to the initial array (no more ROW). A workaround is to bound the backward exploration of the write-list, which reduces the worst case time cost to linear, but at the expense of limited simplifications (Fig. 10, Sec. 5.4).
Arrays represented as maps. In the restricted case where all indexes of reads and writes are constant values, a persistent map with logarithmic lookup and insertion can be used to simplify all row occurrences — yielding fast and scalable simplification. This representation is used in symbolic execution tools [10] with strong concretization policy [23, 14] during the formula generation step in order to limit the introduction of arrays, but it is not suited to general purpose array solvers as it cannot cope with symbolic indexes.

Here, a freshly declared array is represented by an empty map where indexes and elements sorts correspond to those of the array, and the array obtained after a write of element \( e \) at index \( i \) is simply represented by the map of the written array in which \( e \) is added at index \( i \), as illustrated in Fig. 4. Then the simplification of a read at index \( j \) becomes its substitution by the element mapped to \( j \). In the case where no such element is found, the read occurs on the initial array. Therefore, we can either replace the array by the initial one or replace the read by a fresh symbol. In the latter case, we have to ensure that two reads are replaced by the same symbol if and only if they occur at the same index.

4 Efficient simplification for read-over-write

We now present FAS (Fast Array Simplification), an efficient approach to read-over-write simplification. FAS combines three key ingredients: a new representation for arrays as a list of maps to ensure scalability, a dedicated rewriting step (base normalization) geared at improving the conclusiveness of syntactic (dis-)equality checks between indexes, and lightweight domain reasoning to go beyond purely syntactic checks.

4.1 Dedicated data structure: arrays represented as lists of maps

We look here for an array representation combining the advantages of the list representation (genericity) and the map representation (efficiency) presented in Sec. 3. As a preliminary remark, we can note that the map representation can be extended from the constant-indexes case to the case where all indexes of reads and writes are pairwise comparable. By comparable we mean that a binary comparison operator \( \prec \) is defined and decidable for every pair of indexes in the formula. Yet, if such a hypothesis might sometimes be satisfied, it is not necessary the case, for example when indexes contain uninterpreted symbols.

The representation of arrays we propose, lists of maps (denoted map lists), aims precisely at combining advantages of maps when all indexes are pairwise comparable while being as general as lists in other situations. Our array representation can be thought of as a list of packs of independent writes. The idea is that sets of comparable (and proven different) indexes can be packed together into map-like data structures, allowing efficient (i.e. logarithmic) search on these packs of indexes during the application of row-like simplification rules. While the idea is presented here in general, we instantiate it in Sec. 4.3, Fig. 7, and in Sec. 4.4, Fig. 8.

In this representation, nodes of the list are maps from pairwise-comparable indexes to written elements, as illustrated in Fig. 5. A fresh array is represented as an empty list (of maps). The array obtained after the write of element \( e \) at index \( i \) is defined by:
• If \( i \) is comparable with all other indexes of elements already inserted in the map at head position, then we add the element \( e \) at index \( i \) into this map (\text{STORE-HIT});
• Else we add to the list a fresh node containing the singleton map of index \( i \) to element \( e \) (\text{STORE-MISS}).

For a read at index \( j \), the simplification of \textsc{row} is done as follows:
• If indexes in the head position map of the list representing the array are all comparable with \( j \), then if \( j \) belongs to this map we substitute the read by the associated element (\text{SELECT-HIT}), else we re-iterate on the following node in the list (\text{SELECT-MISS});
• Else, we abort (\text{SELECT-ABORT}).

A first version of the dedicated (dis-)equality checks we use is presented in Sec. 4.2. The whole \textsc{fas} procedure, together with the associated notion of comparable, is formally described in Sec. 4.3, and a refinement using more semantic checks is presented in Sec. 4.4.

Intuitively, the benefit of this representation is that its behavior varies between the one of the list representation and the one of the map representation, depending on the proportion of indexes pairwise comparable. Indeed, when all indexes are pairwise comparable, the list only contains a single map of all indexes, which is equivalent to the map representation. And when none of the index pairs are comparable, the list is composed of singleton maps, which is equivalent to the list representation.

From a technical point of view, map lists enjoys several good properties:

**Property 1** (Compactness). By construction, all indexes in any map of a map list are pairwise comparable, while indexes from adjacent maps are never comparable.

**Property 2** (Complexity). Assuming that 1. we can decide efficiently (constant or logarithmic time) whether an index is comparable to all the other indexes of a given map, 2. that \(<\) between comparable terms can also be efficiently decided (constant or logarithmic time), and 3. a decent implementation of maps (logarithmic time insertion and lookup), then:
• Array writes are computed in logarithmic time (map insertion) — where the standard list approach requires only constant time;
• Array reads are also computed in logarithmic time (map lookup) as \text{SELECT-MISS} can only led to \text{SELECT-ABORT} (Prop. 1) — where the standard list approach requires linear time.

In the case where all indexes are pairwise comparable, our representation contains a single map and simplification cost for \( r \) reads and \( w \) writes is bounded by \( r \cdot \ln(w) \), while the list approach requires a quadratic \( r \cdot w \) time.

Finally, map lists allow to easily take into account some cases of write-over-write (a write masked by a later write at the same index can be ignored if no read happens in-between), while it requires a dedicated and expensive \( (w^2) \) treatment with lists.
4.2 Approximated equality check and dedicated rewriting

We consider as equality check a variation of syntactic term equality, namely **syntactic base/off-set equality**, which is regarding two terms $t_1$ and $t_2$ defined as follows:

- If $t_1 \triangleq \beta_1 + \iota_1$ and $t_2 \triangleq \beta_2 + \iota_2$ — where $\beta_1, \beta_2$ are arbitrary terms (bases) and $\iota_1, \iota_2$ are constant values (offsets), and $\beta_1 = \beta_2$ (syntactically) then return the result of $\iota_1 = \iota_2$,
- Otherwise the check is not conclusive.

This equality check is correct and efficient, and it strictly extends syntactic term equality — the result is more often conclusive. Actually, in practice it turns out that this extension is significant. Indeed, a common pattern in array formulas coming from software analysis is reads or writes at indexes defined as the sum of a base and an offset (think of C or assembly programming idioms). Hence, dealing with such terms is particularly interesting for verification-oriented formulas.

**Dedicated rewriting: base normalization.** Yet, this equality check still suffers from the rigidity of syntactic approaches. Therefore it is worthwhile to normalizes indexes as much as possible by applying a dedicated set of rewriting rules called **base normalization** (rebase), cf. Fig. 6. These rules are essentially based on limited inlining of variables together with associativity and commutativity rules of $+/-$ operators, the goal being to minimize the number of possible bases in order to increase the “conclusiveness” of our equality check, as done in example Sec. 2.

```
if $u \triangleq v$ then $u + k \leadsto v + k$ alias inlining
if $u \triangleq v + l$ then $u + k \leadsto v + (k + l)$ base/offset inlining
    $- (x + k) \leadsto (-k) - x$ constant negation
    $(x + k) + l \leadsto x + (k + l)$ constant packing
    $(x + k) + y \leadsto (x + y) + k$ constant lifting
    $(x + k) - (y + l) \leadsto (x + y) + (k + l)$ base/offset addition
    $(x + k) - (y + l) \leadsto (x - y) + (k - l)$ base/offset subtraction
...
```

Figure 6: Example of base normalization rules. $u, v$ are variables, $k, l$ are constant values and $x, y$ are terms. Non-inlining rules reduce either the number of operators or the depth of constant values, ensuring termination. Note that $(-k), (k + l), (k - l)$ are constant values, not terms.

**Optimization: sub-term sharing.** Sharing of sub-terms consists in giving a common name to two syntactically equal terms. This improvement is not new, but has an original implication in this context. Besides easing the decision of equality between terms, it remedies to an issue induced by the simplification of ROW. Indeed, the simplification of ROW can be seen as a kind of “inlining” stage, which may in some cases lead to terms size explosion. This problem arises when after a write of element $e$ at index $i$, several reads at index $i$ are simplified. It may result in numerous copies of term $e$, term which may contain itself other reads to simplify. By naming and sharing terms read and written in arrays, the sub-term sharing phase prevents this issue. **Experiments in Sec. 5.4 demonstrate the practical interest on very large formulas.**

4.3 The FAS procedure

Using the generic algorithm of Sec. 4.1 with equality check and normalization from Sec. 4.2, we formalize FAS as the set of inference rules presented in Fig. 7. Two terms will be said comparable when they share the same base $\beta$. **STORE-HIT and STORE-MISS** rules explain how
to update the representation of an array on writes, and select-hit and select-miss rules explain how to simplify reads. Store rules are presented as triples \( \{ \Lambda \} \) \( \text{store} \ a \ i \ e \ \{ \Lambda' \} \) where \( \Lambda' \) is the representation for \( \text{store} \ a \ i \ e \) when \( \Lambda \) is the representation for \( a \). Select rules are presented as triples \( \{ \Lambda \} \vdash \text{select} \ a \ i \sim e \) meaning that \( \text{select} \ a \ i \) can be rewritten in \( e \) when \( \Lambda \) is the representation for \( a \).

\[
i = \beta + \iota \quad \iota \text{ a constant}
\]

\[
\{ (\Gamma, \beta, b) :: \Lambda \} \text{store} \ a \ i \ e \ \{ (\Gamma \ [i \leftarrow e], \beta, b) :: \Lambda \} \quad \text{STORE-HIT}
\]

\[
i = \alpha + \iota \quad \alpha \neq \beta
\]

\[
\{ (\Gamma, \beta, b) :: \Lambda \} \text{store} \ a \ i \ e \ \{ (\emptyset [i \leftarrow e], \alpha, a) :: (\Gamma, \beta, b) :: \Lambda \} \quad \text{STORE-MISS}
\]

\[
\Gamma \ [i] = e \quad i = \beta + \iota
\]

\[
\{ (\Gamma, \beta, b) :: \Lambda \} \vdash \text{select} \ a \ i \sim e \quad \text{SELECT-HIT}
\]

\[
\{ \Lambda \} \vdash \text{select} \ b \ i \sim e \quad \Gamma \ [i] = \emptyset \quad i = \beta + \iota
\]

\[
\{ (\Gamma, \beta, b) :: \Lambda \} \vdash \text{select} \ a \ i \sim e \quad \text{SELECT-MISS}
\]

Figure 7: Inference rules for \text{select} and \text{store} using the map list representation

The representation \( (\Gamma, \beta, b) :: \Lambda \) we use is a specialized version of the map list representation that we just defined, where \( \Gamma \) is a map, \( \beta \) is the common base of indexes present in \( \Gamma \), \( b \) the last revision of the array written at a different index than \( \beta \), and where \( \Lambda \) is the tail of the list. Assuming that all indexes have been normalized, if the base of the write index is equal to \( \beta \), then the store-hit rule applies and we add the written element into \( \Gamma \). If the base of the write index is not equal to \( \beta \), then the store-miss rule applies. We add as a new node of the list a singleton map containing only the written element, the new base and the written array. For row-simplification, the select-hit rule states that if the base of the read index is equal to \( \beta \), and if there is an element in \( \Gamma \) mapped to this index, then we return this element. Finally the select-miss rule states that if there is no such element, then we return the simplified read on \( b \) at the same index, using \( \Lambda \) as the representation.

4.4 Refinement: adding domain-based reasoning

While our equality check performs well for deciding (dis-)equalities between indexes with a same base, it behaves poorly with different bases. So we extend FAS in Fig. 8 with domain-based reasoning abilities. Basically, maps are now equipped with abstract domains over-approximating their sets of (possible) concrete indexes, and the data structure is now a list of sets of maps, all maps in a set having different bases but disjoint sets of concrete indexes. When syntactic base/offset equality check is not conclusive, domain intersection may be used to prove disequality.

We borrow ideas from Abstract Interpretation [13]. Given a concrete domain \( D \), an abstract domain is a complete lattice \( \langle D^\sharp, \sqsubseteq, \sqcup, \sqcap, T, \bot \rangle \) coming with a monotonic concretization function \( \gamma : D^\sharp \rightarrow \mathcal{P}(D) \) such that \( \gamma(T) = D \) and \( \gamma(\bot) = \emptyset \). An element of an abstract domain is called an abstract value. In the following the concrete domain is the set of array indexes.

The representation is now a list of sets of tuples \( (\Gamma, \beta, b, \Gamma^\sharp) \) where \( \Gamma \), \( \beta \) and \( b \) are a map, a base and an array as previously described, and where \( \Gamma^\sharp \) is the joined abstract value of indexes.
We append first with congruence (e.g. \(\text{Arrays Made Simpler}\) B. Farinier, R. David, S. Bardin, M. Lemerre good candidate for refining our method at an affordable cost. The general difficulty is to find a sweet spot between the potential gain (more checks become conclusive) and the overhead of propagation. As a rule of thumb, non-relational domains should be tractable and useful. Especially, combining multi-intervals with congruence (e.g. \(x \equiv 5 \mod 8\)) or bit-level information (e.g. the second bit of \(x\) is 1) \([3]\) is a good candidate for refining our method at an affordable cost.

**Domain propagation.** So far, we did not explained how abstract values are computed. The literature on abstract domains is plentiful \([28]\). Nevertheless we present in Fig. 9 propagation rules for a specific abstract domain, the well-known domain of (multi-)intervals — used in our implementation (note that operations are performed over bitvectors of a known size \(N\), and \(+\) is the wraparound addition). The general difficulty is to find a sweet spot between the potential gain (more checks become conclusive) and the overhead of propagation. As a rule of thumb, non-relational domains should be tractable and useful. Especially, combining multi-intervals with congruence (e.g. \(x \equiv 5 \mod 8\)) or bit-level information (e.g. the second bit of \(x\) is 1) \([3]\) is a good candidate for refining our method at an affordable cost.

\[
i = \beta + i \\
i \text{ a constant}
\]

\[
\Theta = \left\{ \langle \Sigma, \sigma, \Sigma^\|^i \mid \sigma \neq \beta \land \Sigma^\|^i \cap i^\# = \bot \right\}
\]

\[
\Xi = \left\{ \langle \Sigma, \sigma, \Sigma^\|^i \mid \sigma \neq \alpha \land \Sigma^\|^i \cap i^\# \neq \bot \right\}
\]

\[
\left\langle \left( \langle \Gamma, \beta, \Gamma^\# \rangle \uplus \Theta \uplus \Xi \right) :: \Lambda \right\rangle \left\rangle \text{ store } a \ i \ e \right\rangle \left\langle \left( \langle \Gamma \uparrow e \downarrow, \beta, b, \Gamma^\# \uplus \Theta \right) :: \Xi :: \Lambda \right\rangle
\]

\[
i = \alpha + i \\
i \text{ a constant}
\]

\[
\Theta = \left\{ \langle \Sigma, \sigma, \Sigma^\|^i \mid \sigma \neq \alpha \land \Sigma^\|^i \cap i^\# = \bot \right\}
\]

\[
\Xi = \left\{ \langle \Sigma, \sigma, \Sigma^\|^i \mid \sigma \neq \alpha \land \Sigma^\|^i \cap i^\# \neq \bot \right\}
\]

\[
\left\langle \left( \langle \Theta \uplus \Xi \right) :: \Lambda \right\rangle \left\rangle \text{ store } a \ i \ e \right\rangle \left\langle \left( \langle \Theta \uparrow e \downarrow \downarrow, \alpha, a, i^\# \uplus \Theta \right) :: \Xi :: \Lambda \right\rangle
\]

\[
\Gamma[i] = e \quad i = \beta + i
\]

\[
\left\langle \left( \langle \Gamma, \beta, b, \Gamma^\# \rangle \uplus \Xi \right) :: \Lambda \right\rangle \left\rangle \text{ select } a \ i \sim \ e \right\rangle
\]

\[
\{ \Lambda \} \vdash \text{ select } b \ i \sim \ e \quad \quad \Gamma[i] = \emptyset \quad i = \beta + i
\]

\[
\left\langle \left( \langle \Gamma, \beta, b, \Gamma^\# \rangle \uplus \Xi \right) :: \Lambda \right\rangle \left\rangle \text{ select } a \ i \sim \ e \right\rangle
\]

\[
\{ \Lambda \} \vdash \text{ select } b \ i \sim \ e \quad i = \beta + i \quad \Theta = \left\{ \langle \Sigma, \sigma, \Sigma^\|^i \mid \sigma \neq \beta \land \Sigma^\|^i \cap i^\# = \bot \right\}
\]

\[
\{ \Theta :: \Lambda \} \vdash \text{ select } a \ i \sim \ e
\]

Figure 8: Inference rules for select and store using domains

in \(\Gamma\). Given a write at index \(i\), the set at head position in the list is split into: 1. \(\Theta\) the set of tuples whose map abstract value does not overlap with \(i^\#\), the abstract value of \(i\), 2. \(\Xi\) the set of tuples whose map abstract value overlap with \(i^\#\), and if it exists, 3. the tuple \(\langle \Gamma, \beta, b, \Gamma^\# \rangle\) where \(\beta\) is after normalization the base of \(i\). If this tuple exists, then the store-HIT rule applies. We update \(\Gamma\) as previously and its associated abstract value becomes the join value of \(\gamma^\#\) and \(\beta^\#\). We append first \(\Xi\) alone onto the list, and then \(\Theta\) together with the updated tuple. Else, the store-MISS rule applies. Again we first append \(\Xi\) alone, then \(\Theta\) together with a new singleton map, the new base, the written array and the write index abstract value. Finally, select-HIT and select-MISS are similar to previous ones, but we add a new rule select-SKIP. This rule states that, if the read index abstract value do not overlap with maps abstract values in the set at head position, then we drop the head and reiterate on the tail of the list.

Note that if abstract values in these rules are set to \(\top\), then \(\Theta\) is always empty and we get back to the previous inference rules. Also the complexity of reads becomes linear in the list size, as domains can prove disequality at each node of the list. Yet, it is not a problem in practice, as demonstrated by experimental evaluation in Sec. 5.
Let $i, j$ two bitvectors of size $N$, with $i^\sharp = [m_i, M_i], j^\sharp = [m_j, M_j]$ where $0 \leq m_{i,j} \leq M_{i,j} \leq 2^N$,

- $c^\sharp = [c, c]$ for any constant $c$
- $v^\sharp = [m_i, M_i]$ if $i \leq v \leq j$
- $(\text{extract}_{l,h} i)^\sharp = [0, 2^{h-l+1} - 1]$ if $(M_i \gg l) - (m_i \gg l) \geq 2^{h-l+1}$
- $= [\text{extract}_{l,h} (m_i), \text{extract}_{l,h} (M_i)]$ if $\text{extract}_{l,h} (M_i) \geq \text{extract}_{l,h} (m_i)$
- $\cup [\text{extract}_{l,h} (m_i), 2^{h-l+1} - 1]$ otherwise
- $(i + j)^\sharp = [m_i + m_j, M_i + M_j]$ if $M_i + M_j < 2^N$
- $= [m_i + m_j - 2^N, M_i + M_j - 2^N]$ if $m_i + m_j \geq 2^N$
- $\cup [0, M_i + M_j - 2^N]$ otherwise

Figure 9: Examples of propagation for intervals. These propagations are extended to multi-intervals by distribution for unary operators and pairwise distribution for binary operators.

5 Implementation and experimental evaluation

5.1 Implementation

In order to evaluate the efficiency of our approach, we implemented FAS (with the different representations presented so far and the abstract domain of multi-intervals) as a preprocessor for SMT formulas belonging to the QF_ABV logic (quantifier-free formulas over the theory of bitvectors and arrays) — as typical choice in software verification. In that setting, all bitvector values and expressions have statically known sizes, arithmetic operations are performed modulo and values can “wraparound”. For reproducibility purposes source code and benchmarks are available online. The implementation comprises 6,300 lines of OCaml integrated into the TFML SMT formula preprocessing engine [19], part of the BINSEC symbolic execution tool [15]. It comprises all simplifications and optimizations described in Sec. 4, including map lists, base normalization, sub-term sharing and domain propagation (multi-intervals) over bitvectors. Note that our normalization rules (Sec. 4.2) and domain propagators (Sec. 4.4) correctly handle possible arithmetic wraparounds.

An advantage operating as a preprocessor is to be independent of the underlying solver used for formula resolution, and therefore allow us to evaluate the impact of our approach with several of them. A drawback is that we do not have access to various internal components of the solver, like accessing the model under construction, and cannot use them to refine our approach. In the long term, a deeper integration into a solver would be more suitable.

5.2 Experimental setup

We evaluated FAS performances under three criteria: 1. simplification thoroughness, measured by the reduction of the number of row terms; 2. simplification impact, measured by resolution time before and after simplification; 3. simplification cost, measured by the total time of simplification.

We devise three sets of experiments corresponding to three different scenarios: mid-sized formulas generated by the SE-tool BINSEC [15] from real executables programs — typical of test generation and vulnerability finding (cf. Sec. 5.3), very large formulas generated by BINSEC from

---

2http://benjamin.farinier.org/lpar2018/
very long traces — typical of reverse and malware analysis (cf. Sec. 5.4), and formulas taken from the SMT-LIB benchmarks (cf. Sec. 5.5). Regarding experiments over SE-generated formulas, we also consider three variants corresponding to standard concretization / symbolization policies [14] (cf. Sec. 5.3), as well as different timeout values. Experiments are carried out on an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz. We consider three of the best SMT solvers for the QF_ABV theory, namely Boolector [8], Yices [18] and Z3 [16]. Note that the impact of map lists (w.r.t. a list-based representation) and sub-term sharing will be evaluated only in Sec. 5.4, as they are interesting only on large enough formulas. Moreover, the map list representation impacts only preprocessing time, not its thoroughness: assuming preprocessing does not time out (and rebase and domains are used), FAS and FAS-list will carry out the same simplifications.

A note on problem encoding. As already stated, we consider quantifier-free formulas over the theory of bitvectors and arrays coming from the encoding of low-level software verification problems. Arithmetic operations are performed modulo and values can “wraparound”. Also, since memory accesses in real hardware are performed at word-level (reading 4 or 8 bytes at once), they are modelled here by successive byte-level reads and writes — allowing to take properly into account misaligned or overlapping accesses. Finally, memory is often modelled as a single logical array of bytes (i.e., bitvector values of size 8), without any a priori distinction between stack and heap (this is the case for all examples from Binsec).

5.3 Medium-size formulas from SE

We consider here typical formulas coming from symbolic execution over executable codes. While mid-sized (max. 3.42 MB, avg. 1.40 MB), these formulas comprise quite long sequences of nested row (max. 11,368 row, avg. 4,726 row) as there is only one initial array (corresponding to the initial memory of the execution, i.e. a flat memory model). More precisely, we consider 6,590 traces generated by Binsec [15] from 10 security challenges (e.g. crackme such as Manticore or Flare-On) and vulnerability finding problems (e.g. GRUB vulnerability), and from these traces we generate 3 x 6,590 formulas depending on the concretization / symbolization policies used in Binsec to generate them: concrete (all array indexes are set to constant values), symbolic (symbolic array indexes), and interval (array indexes bound by intervals). We consider two different timeout: 1,000 seconds (close to SMT-LIB benchmarks setting) and 1 second (typical of program analysis involving a large number of solver calls, e.g. deductive verification or symbolic execution).

The whole results are presented in Table 1 (timeout 1,000 sec.) and Table 2 (timeout 1 sec.). Note that resolution time does not include timeout. Columns FAS and FAS-itv represents respectively our technique (map list, rebase and sharing) potentially improved with domain reasoning based on intervals (FAS-itv). The default column represents a minimal preprocessing step consisting of constant propagation and formula pruning, without any array simplification.

We can see that:

- Simplification time is always very low on these examples (340 sec. for 3 x 6,590 formulas, in avg. 0.017 sec. per formula). Moreover, it is also very low w.r.t. resolution time (taking timeout into account: Boolector 6%, Yices 4% and Z3 0.3%) and largely compensated by the gains in resolution, but for one case where Boolector performs especially well (concrete formulas: cost of 118% — not compensated by gains in resolution).

- Formula simplification is indeed thorough: as a whole, the number of row is reduced by a factor 5 (2.5 without interval reasoning). The simplification performs extremely well, as
Arrays Made Simpler  
B. Farinier, R. David, S. Bardin, M. Lemerre

Table 1: 6,590 x 3 medium-size formulas from SE, with \textit{timeout} = 1,000 sec.: simplification time (in seconds), number of \textit{ROW} after simplification, number of \textit{timeout} and resolution time (in seconds, without \textit{timeout})

<table>
<thead>
<tr>
<th>simpl. time</th>
<th>#timeout and resolution time</th>
<th>#ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td>default</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>FAS</td>
<td>85</td>
<td>0</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>FAS</td>
<td>99</td>
<td>0</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>118</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>FAS</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>187</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>93</td>
</tr>
<tr>
<td>FAS</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>total</td>
<td>1,230</td>
<td>730</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td></td>
<td>593</td>
<td>1,213</td>
</tr>
<tr>
<td>FAS</td>
<td>52</td>
<td>602</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>1,947</td>
<td>575</td>
</tr>
<tr>
<td>total</td>
<td>1,888</td>
<td>618</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td></td>
<td>1,597</td>
<td>647</td>
</tr>
<tr>
<td>FAS</td>
<td>1,451</td>
<td>1,273</td>
</tr>
</tbody>
</table>

Table 2: 6,590 x 3 medium-size formulas from SE, with \textit{timeout} = 1 sec.

<table>
<thead>
<tr>
<th>#timeout and resolution time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boolector</td>
</tr>
<tr>
<td>default</td>
<td>2</td>
</tr>
<tr>
<td>FAS</td>
<td>2</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>2</td>
</tr>
<tr>
<td>total</td>
<td>1,230</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
</tr>
<tr>
<td></td>
<td>593</td>
</tr>
<tr>
<td>FAS</td>
<td>52</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>1,947</td>
</tr>
<tr>
<td>total</td>
<td>1,888</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
</tr>
<tr>
<td></td>
<td>1,597</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>1,451</td>
</tr>
</tbody>
</table>

expected, on \textit{concrete} formulas, where almost all \textit{ROW} instances are solved at preprocessing time. On \textit{interval} formulas, the number of \textit{ROW} is sliced by a factor 4, and a factor 3 in the case of \textit{full symbolic} formulas.

- The \textit{impact} of the simplification over resolution time (for a 1,000 sec. \textit{timeout}) varies greatly from one solver to another, but it is always significant: factor 1.5 for Boolector, factor 1.9 for Yices with one fewer \textit{timeout}, up to a factor 3.8 and 32 fewer \textit{timeout} for Z3. Especially, on interval formulas \textit{FAS} with domain reasoning yields a 3.4 (resp. 3.3) speed factor for Boolector (resp. Yices), while Z3 on this category enjoys a 4.1 speedup together with 14 fewer \textit{timeout}. Interestingly, domain reasoning is useful also in the case of fully symbolic formulas, i.e. with no explicit introduction of domain-based constraints.

Results for a 1 sec. \textit{timeout} follows the same trend but they are much more significant
(number of timeout: Boolector -48%, Yices -47% and Z3 -21%), and they become especially dramatic on interval formulas (number of timeout: Boolector -96%, Yices -90% and Z3 -44%).

Focus on specific cases. We highlight now a few interesting scenarios where FAS performs very well, especially formulas generated from the GRUB vulnerability (Table 3, 753 formulas) and formulas representing the inversion of a crypto-like challenge (Table 4, 139 formulas). Regarding GRUB, while basic FAS does not really impact resolution time, adding domain-based reasoning does allow a significant improvement — Boolector, Yices and Z3 becoming respectively 4.1x, 4.7x and 7x faster. Regarding UNGAR, again FAS alone does not improve resolution time (for Z3, we even see worse performance), but adding interval reasoning yields dramatic improvement: Boolector becomes 18.8x faster, Yices becomes 48.2x faster (with -1 timeout) and Z3 does not time out anymore (-12 timeout).

Table 3: GRUB (interval), 753 formulas — Number of timeout and resolution time (in seconds, without timeout)

<table>
<thead>
<tr>
<th>GRUB</th>
<th>Boolector</th>
<th>Yices</th>
<th>Z3</th>
<th>Boolector</th>
<th>Yices</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>0</td>
<td>508</td>
<td>0</td>
<td>258</td>
<td>0</td>
<td>31,322</td>
</tr>
<tr>
<td>FAS</td>
<td>0</td>
<td>505</td>
<td>0</td>
<td>257</td>
<td>1</td>
<td>26,809</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>0</td>
<td>123</td>
<td>0</td>
<td>54</td>
<td>0</td>
<td>4,481</td>
</tr>
</tbody>
</table>

Table 4: UNGAR (symbolic), 139 formulas — Number of timeout and resolution time (in seconds, without timeout)

<table>
<thead>
<tr>
<th>UNGAR</th>
<th>Boolector</th>
<th>Yices</th>
<th>Z3</th>
<th>Boolector</th>
<th>Yices</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>0</td>
<td>359</td>
<td>3</td>
<td>627</td>
<td>12</td>
<td>926</td>
</tr>
<tr>
<td>FAS</td>
<td>0</td>
<td>373</td>
<td>3</td>
<td>624</td>
<td>12</td>
<td>1,130</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td>13</td>
<td>0</td>
<td>569</td>
</tr>
</tbody>
</table>

Conclusion. On these middle-size formulas coming from typical SE problems, we can draw the following conclusion: Speed. FAS is extremely efficient and does not yield any noticeable overhead; Thoroughness. Formula simplification is significant — even on fully symbolic formulas, and it becomes (as expected) dramatic on “concrete” formulas; Impact. The impact of FAS varies across solvers and formulas categories, yet it is always positive and it can be dramatic in some settings (low timeout, interval formulas, etc.).

5.4 Very large formulas

We now turn our attention to large formulas (max. 458 MB, avg. 45 MB) involving very long sequences of nested row (max. 510,066 row, avg. 49,850 row), as can be found for example in symbolic deobfuscation. We consider 29 benchmarks taken from a recent paper on the topic [27] representing execution traces over (mostly non crypto-) hash functions (e.g. MD5, City, Fast, Spooky, etc.) obfuscated by the Tigress tool [12]. We also consider a trace taken from the AS Pack packing tool.

Results are presented in Table 5, where FAS-list represents our simplification method where the map list is replaced by a normal list — getting an improved version of the standard list-based row-simplification (the goal being to evaluate the gain of our new data structure). Again, simplification is significant with a strong impact on the number of time outs and on resolution time, especially in the concrete case and for Z3. Impact in the symbolic case is more mixed but positive (-1 timeout for Boolector and Z3, no impact for Yices). In term of size, FAS reduces formulas to max. 86.49MB, avg. 6.98MB, and FAS-itv to max. 86.45MB, avg. 6.17MB. If sub-term sharing is disabled, formulas size jumps to max. 591.99MB, avg. 14.95MB for FAS and max. 591.71MB, avg. 16.35MB for FAS-itv. Regarding simplification time, FAS-list suffers from scalability issues on these formulas (5x slower than FAS).
Table 5: 29 x 3 very large formulas from SE, with \textsc{timeout} = 1,000 sec.: simplification time (in seconds), number of \textsc{row} after simplification, number of \textsc{timeout} and resolution time (in seconds, without \textsc{timeout})

<table>
<thead>
<tr>
<th>simpl. time</th>
<th>#\textsc{timeout} and resolution time</th>
<th>#\textsc{row}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td>\textsc{concrete}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>44</td>
<td>10</td>
</tr>
<tr>
<td>FAS-list</td>
<td>1,108</td>
<td>8</td>
</tr>
<tr>
<td>FAS</td>
<td>196</td>
<td>8</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>210</td>
<td>4</td>
</tr>
<tr>
<td>\textsc{interval}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>44</td>
<td>12</td>
</tr>
<tr>
<td>FAS-list</td>
<td>222</td>
<td>12</td>
</tr>
<tr>
<td>FAS</td>
<td>231</td>
<td>12</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>237</td>
<td>12</td>
</tr>
<tr>
<td>\textsc{symbolic}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>FAS-list</td>
<td>187</td>
<td>11</td>
</tr>
<tr>
<td>FAS</td>
<td>194</td>
<td>11</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>200</td>
<td>11</td>
</tr>
</tbody>
</table>

The \textsc{ASPack} example. We now turn our attention to the formula generated from a trace of a program protected by \textsc{ASPack} (96 MB and 363,594 \textsc{row}, \textsc{concrete} mode). Solving the formula is highly challenging: while \textsc{Yices} succeeds in a decent amount of time (69 seconds), \textsc{Z3} terminates in 2h36min while \textsc{Boolector} needs 24h. Table 6 presents our results on this particular example. \textsc{FAS} performs extremely well (Table 6), turning resolution time from hours to a few seconds (\textsc{Boolector}) or minutes (\textsc{Z3}). \textsc{Yices} also benefits from it. Especially, all \textsc{row} instances are simplified away. \textsc{FAS} and \textsc{FAS-itv} reduce the \textsc{ASPack} formula size to 3.81MB, while it jumps to 443.54MB when sub-term sharing is disabled.

Interestingly, this example clearly highlights the scalability of \textsc{FAS} \wrt a standard list-based approach, passing roughly from 1h (\textsc{list}) to 1 minute (\textsc{fas}). Fig. 10 proposes a detailed view of the performance and impact of the standard list-based simplification method (\textsc{Boolector} only), depending on the bound for backward reasoning (the standard method has no bound). For comparison, the two horizontal lines represent simplification and resolution time with \textsc{FAS}. We can see that bounding the list-based reasoning has no tangible effect here, as we need at least a 3,000 seconds (50 minutes) simplification time to get a resolution time under 3,000 seconds.

Table 6: \textsc{ASPack} formula, without \textsc{timeout}

<table>
<thead>
<tr>
<th>\textsc{ASPack}</th>
<th>simpl. time</th>
<th>\textsc{resolution time}</th>
<th>#\textsc{row}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td>default</td>
<td>15 sec.</td>
<td>\approx 24h</td>
<td>69 sec.</td>
</tr>
<tr>
<td>FAS-list</td>
<td>53 min.</td>
<td>9.7 sec.</td>
<td>3.4 sec.</td>
</tr>
<tr>
<td>FAS</td>
<td>61 sec.</td>
<td>9.7 sec.</td>
<td>3.4 sec.</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>63 sec.</td>
<td>9.8 sec.</td>
<td>3.4 sec.</td>
</tr>
</tbody>
</table>
Conclusion. Once again FAS appears to be fast and to have a significant impact on resolution time, especially in the concrete case where the difference can be from several hours to a few seconds (total resolution + simplification: a few minutes). Moreover, it appears clearly that on very long traces FAS scales much better than the standard list-based ROW-simplification method.

5.5 SMT-LIB formulas

We consider now the impact of FAS on formulas taken from the SMT-LIB benchmarks. These formulas are notably different from the ones considered in the two previous experiments: while most of them do come from verification problems, they may involve complex Boolean structure (rather than “mostly conjunctive” formulas) and they do not necessarily exhibit very deep chains of ROW. These kinds of formulas are not our primary objective, yet we seek to evaluate how our technique performs on a “bad case”. We evaluate FAS on all the 15,016 SMT-LIB formulas from QF_ABV theory. TIMEOUT is set to 1,000 seconds. Results are reported in Table 7. Note that, again, resolution time does not include TIMEOUT.

Table 7: 15,016 formulas from SMT-LIB benchmarks, with TIMEOUT = 1.000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT)

<table>
<thead>
<tr>
<th>SMT-LIB</th>
<th>simpl. time</th>
<th>#TIMEOUT and resolution time</th>
<th>#ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Boolector</td>
<td>Yices</td>
</tr>
<tr>
<td>default</td>
<td>87</td>
<td>59, 20,126</td>
<td>151</td>
</tr>
<tr>
<td>FAS</td>
<td>378</td>
<td>54, 19,922</td>
<td>148</td>
</tr>
<tr>
<td>FAS-itv</td>
<td>378</td>
<td>55, 19,843</td>
<td>146</td>
</tr>
</tbody>
</table>

Conclusion. FAS is again very efficient on these formulas (avg. 0.025 sec. per formula), and reduces the number of ROW by -14%. Yet the impact of simplifications, while slight, is clearly positive on both TIMEOUT (Boolector -8%, Yices -2% and Z3 -7%) and resolution time (for Yices, only when taking TIMEOUT time into account). Such gains are not anecdotal as the best SMT solvers are highly tuned for SMT-LIB. Since the number of TIMEOUT is the main metric for SMT-LIB, Boolector with FAS would have won the last edition for QF_ABV theory. Finally, domain reasoning does not add anything here (but for Yices) — either the benchmark formulas do not exhibit such interval constraints, or our propagation mechanism is too crude to take advantage of it.

5.6 Conclusion

Our experiments demonstrate that our approach is efficient (the cost is almost always negligible w.r.t. resolution time) and scalable (compared to the list-based method). The simplification is thorough, removing a large fraction of ROW. The impact is always positive (both in resolution time and number of time outs), and it is dramatic for some key usage scenarios such as SE-like formulas with small TIMEOUT or very large size.

Finally, we can note that domain reasoning is usually helpful (though, not on SMT-LIB formulas) and that it shows a powerful synergy with the “interval C/S policy” in SE — yielding a new interesting sweet spot between tractability and genericity of reasoning.
6 Related work

A preliminary work in progress version of this work was published in a French workshop [20], in French (6 pages). The current article adds a much more refined description, the domain reasoning part and a much more systematic and thorough experimental evaluation (including SMT-LIB, long traces over packed hash functions, etc.).

Decision procedures for the theory of arrays. Surprisingly, there have been relatively few works on the efficient handling of the (basic) theory of arrays. Standard symbolic approaches for pure arrays complement symbolic read-over-write preprocessing [22, 5, 2] with enumeration on (dis-)equalities, yielding a potentially huge search space. New array lemmas can be added on-demand or incrementally discovered through an abstraction-refinement scheme [9]. Another possibility is to reduce the theory of arrays to the theory of equality by systematic “inlining” of the array axioms to remove all store operators, at the price of introducing many case-splits. The encoding can be eager [24] or lazy [9]. Our method generalizes previous preprocessing [22, 2] and is complementary to complete resolution methods [9, 24]. Note also that our approach could benefit from being integrated directly within such a complete resolution method, allowing incremental simplification all along the resolution process.

Decision procedures have also been developed for expressive extensions of the array theory, such as arrays with extensionality (i.e. equality over whole arrays) or the array property fragment [7], which enables limited forms of quantification over indexes and arithmetic constraints. These extensions aim at increasing expressiveness and they do not focus so much on practical efficiency. Our method can also be applied to these settings (as row are still a crucial issue), even though it will not cover all difficulties of these extensions.

Optimized handling of arrays inside tools. Many verification and program analysis tools and techniques ultimately rely on solving logical formulas involving the theory of arrays. Since the common practice is to re-use existing (SMT) solvers, these approaches suffer from the limitations of the current solvers over arrays. As a mitigation, some of these tools take into account knowledge from the application domain in order to generate relevant (but usually not equivalent) and simpler formulas [25, 21] — see also the specific case of SE over concrete indexes discussed in Sec. 3. Our method is complementary to these approaches as it operates on arbitrary formulas and the simplification keeps logical equivalence.

7 Conclusion

The theory of arrays has a central place in software verification due to its ability to model memory or data structures. Yet, this theory is known to be hard to solve because of read-over-write terms (row), especially in the case of very large formulas coming from unrolling-based verification methods. We have presented FAS, an original simplification method for the theory of arrays geared at eliminating row, based on a new dedicated data structure together with original simplifications and low-cost reasoning. The technique is efficient, scalable and it yields significant simplification. The impact on formula resolution is always positive, and it can be dramatic on some specific classes of problems of interest, e.g. very long formula or binary-level symbolic execution. These advantages have been experimentally proven both on realistic formulas coming from symbolic execution and on SMT-LIB formulas.

Future work includes a deeper integration inside a dedicated array solver in order to benefit from more simplification opportunities along the resolution process, as well as exploring the interest of adding more expressive domain reasoning.
References


