Better Lemmas with Lambda Extraction

Mathias Preiner, Aina Niemetz, and Armin Biere

Johannes Kepler University, Linz, Austria

***Abstract*—In Satisfiability Modulo Theories (SMT), the theory of arrays provides operations to access and modify an array at a given index, e.g., *read* and *write*. However, common operations to modify multiple indices at once, e.g., *memset* or *memcpy* of the standard C library, are not supported. We describe algorithms to identify and extract array patterns representing such operations, including *memset* and *memcpy*. We represent these patterns in our SMT solver Boolector by means of compact and succinct lambda terms, which yields better lemmas and increases overall performance. We describe how extraction and merging of lambda terms affects lemma generation, and provide an extensive experimental evaluation of the presented techniques. It shows a considerable improvement in terms of solver performance, particularly on instances from symbolic execution.**

I. Introduction

The theory of arrays, which for instance has been axiomatized by McCarthy [7], enables reasoning about "memory" in both software and hardware verification. It provides two operations *read* and *write* for accessing and modifying arrays on *single* array indices. While these two operations can be used to capture many aspects of modeling memory, they are not sufficient to succinctly encode array operations over *multiple* indices or a range of indices, e.g., *memset* or *memcpy* from the standard C library. Such array operations can therefore only be represented verbosely by means of a constant number of read and write operations. It is further impossible to reason about a variable number of indices e.g., a memset operation of variable size (without introducing quantifiers).

To overcome these limitations, Seshia et. al. [1] introduced an approach to model arrays by means of restricted lambda terms. This also enabled their SMT solver UCLID [10] to reason about ordered data structures and partially interpreted functions. However, UCLID employs the eager SMT approach and thus eliminates all lambda terms as a rewriting step prior to bit-blasting the formula to SAT, which might result in an exponential blow-up in the size of the formula [10].

An extension to the theory of arrays by Sinz et.al. [5] uses lambda terms similarly to UCLID in order to model memset and memcpy operations as well as loop summarizations, which in essence are initialization loops for arrays. As UCLID, this approach suffers from the problem of exponential explosion through eager lambda elimination.

To avoid exponential lambda elimination, in [9] we introduced a new decision procedure, which lazily handles non-recursive and non-extensional lambda terms. That decision procedure enabled us to succinctly represent array operations such as memset and memcpy as well as other array initialization patterns by means of lambda terms within our SMT solver Boolector. Lambda terms also allow to reason about variable ranges of indices without the need for quantifiers.

Supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE).

In this paper, we continue this thread of research and describe various patterns of operations on arrays occurring in benchmarks from SMT-LIB (http://www.smtlib.org). We provide algorithms to identify these patterns, and to extract succinct lambda terms from them. Extraction leads to stronger, as well as fewer lemmas. This improves performance by orders of magnitude on certain benchmarks, particularly on instances from symbolic execution [2]. We further describe a technique called *lambda merging*. Our extensive experimental evaluation shows that both techniques considerably improve the performance of Boolector, the winner of the QF_ABV track of the SMT competition 2014.

II. Preliminaries

We assume the usual notions and terminology of first order logic and are mainly interested in many-sorted languages, where bit vectors of different bit width correspond to different sorts, and array sorts correspond to a mapping $(\tau_i \Rightarrow \tau_e)$ from index sort τ_i to element sort τ_e. We primarily focus on the *quantifier-free* theories of *fixed size bit vectors* and *arrays*. However, our approach is not restricted to the above.

In general, we refer to 0-arity function symbols as *constant* symbols. Symbols a, b, i, j, and e denote constants, where a and b are used for array constants, i and j for array indices, and e for an array element. We denote an *if-then-else* over bit vector terms with condition c, then branch t_1, and else branch t_2 as $ite(c, t_1, t_2)$, which is interpreted as $ite(\top, t_1, t_2) = t_1$ and $ite(\bot, t_1, t_2) = t_2$. We identify operations *read* and *write* as basic array operations (cf. *select* and *store* in SMT-LIBv2 notation) for accessing and modifying arrays. A *read* operation $read(a, i)$ denotes the element of array a at index i, whereas a *write* operation $write(a, i, e)$ represents the modified array a with element e written to index i. The non-extensional theory of arrays is axiomatized by the following axioms originally introduced by McCarthy in [7]:

$$i = j \rightarrow read(a, i) = read(a, j) \tag{A1}$$
$$i = j \rightarrow read(write(a, i, e), j) = e \tag{A2}$$
$$i \neq j \rightarrow read(write(a, i, e), j) = read(a, j) \tag{A3}$$

Axiom (A1) asserts that accessing array a at two indices that are equal always yields the same element. Axiom (A2) asserts that accessing a modified array on the updated index i yields the written element e, whereas axiom (A3) ensures that the unmodified element of the original array a at index j is returned if the modified index i is not accessed.

A *write sequence* of n (consecutive) write operations of the form $a_1 = write(a_0, i_1, e_1), \ldots, a_n = write(a_{n-1}, i_n, e_n)$ is denoted as $(a_k := write(a_{k-1}, i_k, e_k))_{k=1}^{n}$ with array a_0 as the *base array* of the write sequence. In the following we use $a_n = write(a, \bar{i}, \bar{e})$ as shorthand for write sequences.

In [9] we use *uninterpreted functions* (UF) and *lambda* terms to represent array variables and array operations, respectively. Consequently, a read on an array of sort $\tau_i \Rightarrow \tau_e$ is represented as a function application $f(i)$ on either an UF f or a lambda term $f := \lambda j \,.\, t$, where function f maps terms of sort τ_i to terms of sort τ_e. Furthermore, write operations $write(a, i, e)$ are represented as lambda terms $\lambda j \,.\, ite(i = j, e, a(j))$, where given an array a, a function application yields element e if j is equal to the modified index i and the unchanged element $a(j)$, otherwise. Lambda terms allow us to succinctly model array operations such as *memset* and *memcpy* from the standard C library, or arrays initialized with a constant value. For example, memset with signature $memset\ (a, i, n, e)$, which sets each element of array a to e within the range $[i, i+n[$, can be represented as $\lambda j \,.\, ite(i \leq j < i+n, e, a(j))$. In this paper, we use read operations and function applications interchangeably.

III. Extracting Lambdas

Currently, the SMT-LIBv2 standard only supports write operations for modifying the contents of an array at one index at a time. Hence, quasi-parallel array operations like memset or memcpy usually have to be represented as a *fixed* sequence of consecutive write operations, where copying or setting n indices always requires n write operations. Further, modeling such array operations with a *variable* range is not possible (without quantifiers), since it would require a variable number of write operations. Lambda terms, however, provide means to succinctly represent parallel array operations, and further allow to model these operations with *variable* ranges. For example, modeling $memset(a, i, n, e)$ with a sequence of writes for some fixed n produces n nested write operations $write(write(\ldots(write(a, i, e), i{+}1, e)\ldots), i{+}n{-}1, e)$ which could be represented in a more compact way by means of a *single* lambda term $\lambda j \,.\, ite(i \leq j < i+n, e, a(j))$.

In the following, we describe several array operation patterns we identified by analyzing QF_ABV benchmarks in the SMT-LIB benchmark library. These patterns can not be captured compactly by means of write and read operations alone, but they can be succinctly represented using lambda terms. For each pattern identified in a formula, lambda terms are extracted and used instead of the original array operations, which are defined as follows.

A. Memset Pattern

The probably most common pattern is the *memset pattern* modeling the $memset\ (a, i, n, e)$ operation, which updates n elements of array a within range $[i, i+n[$ to a value e starting from address i. This is the pattern already described above, and it is represented by the lambda term

$$\lambda_{mset} := \lambda j \,.\, ite(i \leq j < i+n, e, a(j)).$$

Lambda term λ_{mset} yields value e if index j is within the range $[i, i+n[$, and the unmodified value from array a at position j otherwise. Note that in actual benchmarks, e.g., those from SMT-LIB, the upper bound n is constant, while indices, as well as values are usually symbolic.

B. Memcpy Pattern

The *memcpy pattern* models the $memcpy(a, b, i, k, n)$ operation, which copies n elements from source array a starting at address i to destination array b at address k. If arrays a and b are syntactically distinct, or if the source and destination addresses do not overlap, i.e., $(i+n < k)$ or $(k+n < i)$, *memcpy* can be represented as

$$\lambda_{mcpy} := \lambda j \,.\, ite(k \leq j < k+n, a(i+j-k), b(j)).$$

Lambda term λ_{mcpy} returns the value copied from source array a if it is accessed within the copied range $[k, k+n[$, and the value from destination array b at position j otherwise.

Assume arrays a and b are syntactically equal, then aliasing occurs. Writing to array b at overlapping memory regions modifies elements in a to be copied to the destination address. This is not captured by lambda term λ_{mcpy}, since λ_{mcpy} behaves like a *memmove* operation. It ensures that elements of a at the overlapping memory region are copied before being overwritten. The following lambda term λ_{mcpyo} can be used to model *memcpy* applied to potentially overlapping memory regions.

$$\begin{aligned}\lambda_{mcpyo} := \lambda j \,.\, ite(&k \leq j < k+n,\\ &ite(i \leq k < i+n,\\ &\quad a(i + ((j-k) \bmod (k-i))),\\ &\quad a(i+j-k)),\\ &b(j)).\end{aligned}$$

If condition $i \leq k < i+n$ holds, source and destination memory regions overlap and consequently, the elements of the overlapping memory region always contain the repeated sequence of the elements of array a in range $[i, k[$. This corresponds to the value $a(i + (j-k) \bmod (k-i))$, where $k-i$ represents the size of the non-overlapping memory region and thus, the number of elements that occur repeatedly. If the memory regions do not overlap, the behavior of lambda terms λ_{mcpyo} and λ_{mcpy} is equivalent. For the rest of this paper, we focus on *memcpy* with non-overlapping memory regions.

C. Loop Initialization Pattern

The *loop initialization pattern* models array initialization operations that can be expressed with the following loop

$$\textbf{for}\ (j = i;\ j < i+n;\ j = j + inc)\ \{a[j] = e;\},$$

where, starting from index i, the loop counter is incremented by a constant inc greater than one. Consequently, every inc-th element of an array a is modified within the range $[i, i+n[$. The above loop pattern corresponds to the lambda term

$$\lambda_{i\to e} := \lambda j \,.\, ite(i \leq j \wedge j < i+n \;\wedge (inc \mid (j-i)), e, a(j)).$$

The memset pattern is actually a special case of this pattern with $inc = 1$. Further, the divisibility condition $inc \mid (j-i)$ makes sure that there exists a c such that index $j = i + c \cdot inc$ or equivalently $((j-i) \bmod inc = 0)$.

It is also possible that the value written on an index i depends on i itself. We found two such patterns in benchmarks. They can be expressed with the following loops

$$\begin{aligned}&\textbf{for } (j = i;\ j < i+n;\ j = j + inc)\ \{a[j] = j\},\\ &\textbf{for } (j = i;\ j < i+n;\ j = j + inc)\ \{a[j] = j+1\}\end{aligned}$$

or equivalently with the following lambda terms

$$\begin{aligned}\lambda_{i\to i} &:= \lambda j \,.\, ite(i \leq j \wedge j < i+n \;\wedge (inc \mid (j-i)),\\ &\qquad j,\ a(j))\\ \lambda_{i\to i+1} &:= \lambda j \,.\, ite(i \leq j \wedge j < i+n \;\wedge (inc \mid (j-i)),\\ &\qquad j+1,\ a(j)).\end{aligned}$$

Note that with $inc = 1$, the condition $inc \mid (j-i)$ is redundant and can be omitted. Further, this set of patterns is of course just a subset of all possible structures in benchmarks for which lambdas can be extracted. The ones discussed in this paper are those that we observed in actual benchmarks, and which turn out to be useful in our experiments.

D. Lemma Generation

Extracting lambda terms from write sequences does not only yield more compact array representations but improves the lemmas generated during search. As an example, consider a memset operation with range $[i, i+n[$ and value e, which is represented as a sequence of write operations

$$b := write(write(\ldots(write(a, i, e), i+1, e)\ldots), i+n-1, e).$$

A read operation on array b at index j may produce a conflict on index i, where $read(b, j) \neq e$. As a consequence, the following lemma is generated.

$$(\bigwedge_{k=1}^{n-1} j \neq i+k) \wedge j = i \rightarrow read(b, j) = e$$

In the worst case, this might be repeated for all the indices $i+k$ with $k \in [1, n[$, which also results in n lemmas of the above form. However, if we use a lambda term to represent memset, then a conflict produces a single lemma of the form

$$i \leq j \wedge j < i+n \rightarrow read(b, j) = e,$$

which is more succinct and stronger as it covers an index range instead of single indices. This effect can be observed in our experiments in Sect. IV-B as well. If applicable, the number of generated lemmas is reduced. This improves runtime and more instances are solved.

E. Algorithms

Figure 1 depicts the main *lambda extraction* algorithm `extract_lambdas`. The purpose of this procedure is to initially identify and extract array patterns from each sequence of write operations in formula ϕ (lines 5-7). The identified patterns are then used to create lambda terms on top of each other resulting in a new lambda term b, which is equisatisfiable to the original write sequence a_n (lines 8-16), and is used to substitute a_n in ϕ. Figures 2, 3, and 4 depict the algorithms for identifying and extracting the actual array patterns. In essence, they all can be split into the following three steps. Given a sequence of write operations,

1) group *write indices* w.r.t. the corresponding pattern,
2) identify *index sequences* in these grouped write indices,
3) and create new pattern for identified *index sequence*.

In the following we describe the algorithms for identifying and extracting array patterns in more detail.

A high level view of the main lambda extraction algorithm `extract_lambdas` is given in Fig. 1. Given a formula ϕ, for any write sequence $a_n = write(a, \bar{i}, \bar{e})$ with distinct indices $i_1, \ldots, i_n$, `extract_lambdas` initially generates a map $\rho_{i\to e}$, which maps indices $i_1, \ldots, i_n$ to values $e_1, \ldots, e_n$ (line 4), and is then used to extract memset (p_{set}), memcpy (p_{cpy}), and loop initialization patterns (p_{loop}) (lines 5-7). Note that procedures `find_mset_patterns`, `find_mcopy_patterns`, and `find_lp_patterns` remove all index/value pairs included in extracted patterns from $\rho_{i\to e}$. As a consequence, at line 8, map $\rho_{i\to e}$ contains all index/value pairs for which no pattern was extracted. The actual memset, memcpy, and loop initialization lambda terms are then created on top of each other with base array a_0 of write sequence a_n as the initial base array (lines 8-14). For the remaining index/value pairs in $\rho_{i\to e}$, lambda terms representing write operations are created on top of the previously generated lambda terms, and the resulting term b is then used to substitute the original write sequence a_n.

Note that indices $i_1, \ldots, i_n$ are required to be distinct constants (line 3) as otherwise, reordering write sequence a_n does not result in an equisatisfiable sequence. As an example, assume indices i and j are equal and values e_i and e_j are distinct. Accessing sequence $a_{ij} := write(write(a, i, e_i), j, e_j)$ at index j yields value e_j. However, accessing sequence $a_{ji} := write(write(a, j, e_j), i, e_i)$ at index j yields e_i since $i = j$. Thus, a_{ji} is not equisatisfiable to a_{ij}.

Figures 2, 3, and 4 illustrate the algorithms for the actual pattern extraction, which we describe in more detail in the following. Procedure `find_mset_patterns` as in Fig. 2 extracts *memset* patterns, i.e., in essence, it identifies index sequences that map to the same value. Given map $\rho_{i\to e}$, the procedure initially generates a reverse map $\rho_{e\to i}$, which maps values to indices and therefore groups indices that map to the same value (lines 3-4). For each index group *indices* in $\rho_{e\to i}$, `find_mset_patterns` sorts *indices* in ascending order (line 6) and identifies index sequences $s := (i_k)_{k=l}^{u}$ with $i_k := i_{k-1} + 1$ within lower bound l ($i_l := indices[l]$) and

```
1  procedure extract_lambdas(φ)
2    for write sequence a_n := write(a, ī, ē) in φ \
3     and i_1, ..., i_n are distinct
4      ρ_{i→e} := index_value_map(a_n)
5      p_set := find_mset_patterns(ρ_{i→e})
6      p_cpy := find_mcopy_patterns(ρ_{i→e})
7      p_loop := find_lp_patterns(ρ_{i→e})
8      b := a_0
9      for p in p_set
10       b := mk_memset(b, p.i, p.n, p.e)
11     for p in p_cpy
12       b := mk_memcopy(p.a, b, p.i, p.k, p.n)
13     for p in p_loop
14       b := mk_loop_init(b, p.i, p.n, p.inc)
15     for i, e in ρ_{i→e}
16       b := mk_write(b, i, e)
17     φ := φ[a_n/b]
```

Fig. 1. Main lambda extraction algorithm in pseudo-code.

```
1  procedure find_mset_patterns(ρ_{i→e})
2    patterns := [], ρ_{e→i} := {}
3    for index, value in ρ_{i→e}
4      ρ_{e→i}[value].add(index)
5    for value, indices in ρ_{e→i}
6      indices := sort(indices)
7      l, u := 0
8      while u < len(indices)
9        while u + 1 < len(indices) \
10        and indices[u + 1] - indices[u] = 1
11         u += 1
12       if l ≠ u
13         Pattern p
14         p.i := indices[l]
15         p.n := indices[u] - indices[l] + 1
16         p.e := value
17         patterns.add(p)
18         ρ_{i→e} := ρ_{i→e} \ {indices[i] | ∀i ∈ [l, u]}
19         l := u + 1  /* next sequence */
20       u += 1
21   return patterns
```

Fig. 2. Memset pattern extraction algorithm in pseudo-code.

upper bound u ($i_u := indices[u]$) (lines 7-19). If sequence s includes at least two indices (i.e., $u \neq l$), a new memset pattern p with start address $p.i$, size $p.n$ and value $p.e$ is created and added to list $patterns$ (lines 13-17). All indices included in sequence s are removed from map $\rho_{i\to e}$ (line 18), since these indices are covered by a detected pattern. If all index groups have been processed, procedure `find_mset_patterns` returns the list of detected memset patterns $patterns$.

Figure 3 illustrates procedure `find_mcopy_patterns` for extracting *memcpy* patterns. Assume that write operation $write(b, dst + o, a(src + o))$ represents a single memcpy operation $memcpy(a, b, src, dst, n)$ with offset o and $src \leq o < src + n$, which copies one element from source address $src + o$ of array a to destination address $dst + o$ of array b. Consequently, $\rho_{i\to e}$ maps indices of the form $dst + o$ to values of the form $a(src + o)$. Initially, the procedure

```
1  procedure find_mcopy_patterns(ρ_{i→e})
2    patterns := [], offset_groups := {}
3    for index, value in ρ_{i→e} \
4     and index = dst + o \
5     and value = a(src + o)
6      offset_groups[dst, a, src].add(o)
7    for dst, a, src in offset_groups
8      indices := sort(offset_groups[dst, a, src])
9      u, l := 0
10     while u < len(indices)
11       while u + 1 < len(indices) \
12        and indices[u + 1] - indices[u] = 1
13         u += 1
14       if l ≠ u
15         Pattern p
16         p.a := a
17         p.i := src + indices[l]
18         p.k := dst + indices[l]
19         p.n := indices[u] - indices[l] + 1
20         patterns.add(p)
21         ρ_{i→e} := ρ_{i→e} \ {indices[i] | ∀i ∈ [l, u]}
22         l := u + 1 /* next sequence */
23       u += 1
24   return patterns
```

Fig. 3. Memcopy pattern extraction algorithm in pseudo-code.

collects all offsets o from the indices in $\rho_{i\to e}$ and groups them by destination address dst, source array a, and source address src (lines 3-6). Note that a group of offsets corresponds to the memory regions copied from source address of array a to destination address of array b. For each offset group $indices$ in $offset\_groups$, `find_mcopy_patterns` identifies index sequences $s := (i_k)_{k=l}^{u}$ similar to procedure `find_mset_patterns` (lines 11-13). If a sequence with at least two indices is found, a new memcpy pattern with source array $p.a$, source address $p.i$, destination address $p.k$, and size $p.n$ is created and added to the $patterns$ list (lines 15-20). As for `find_mset_patterns`, indices included in a sequence s are removed from $\rho_{i\to e}$ (line 21). If all offset groups have been processed, procedure `find_mcopy_patterns` returns the list of detected memcpy patterns $patterns$.

Figure 4 illustrates procedure `find_lp_patterns` for extracting *loop initialization* patterns. Initially, all indices in map $\rho_{i\to e}$ are categorized w.r.t. the three loop initialization patterns defined above, which correspond to the map $\rho_{e\to i}$, and the lists $\rho_{i\to i}$ and $\rho_{i\to i+1}$. Map $\rho_{e\to i}$ groups indices that map to the same value, list $\rho_{i\to i}$ contains indices that map to themselves, and list $\rho_{i\to i+1}$ contains all indices i that map to $i+1$ (lines 4-9). For index groups $\rho_{i\to i+1}$ and $\rho_{i\to i+1}$, and for each index group in $\rho_{e\to i}$, procedure `find_lpp_aux` identifies sequences $s := (i_k)_{k=l}^{u}$ with $i_k := i_{k-1} + inc$ and $inc \geq 1$ within lower bound l ($i_l = indices[l]$) and upper bound u ($i_u := indices[u]$) (lines 10-13). Identifying index sequences in `find_lpp_aux` is similar to `find_mset_patterns`, except that increment inc can be greater than one. For each sequence, inc is initially set to $indices[u+1] - indices[u]$ (lines 21-22), which

```
1  procedure find_lp_patterns(ρ_{i→e})
2    patterns := [], ρ_{e→i} := {}
3    ρ_{i→i} := [], ρ_{i→i+1} := []
4    for index, value in ρ_{i→e}
5      ρ_{e→i}[value].add(index)
6      if value = index
7        ρ_{i→i}.add(index)
8      elif value = index + 1
9        ρ_{i→i+1}.add(index)
10   for value, indices in ρ_{e→i}
11     patterns.add(find_lpp_aux(ρ_{i→e}, ρ_{e→i}))
12   patterns.add(find_lpp_aux(ρ_{i→e}, ρ_{i→i}))
13   patterns.add(find_lpp_aux(ρ_{i→e}, ρ_{i→i+1}))
14   return patterns
15
16 procedure find_lpp_aux(ρ_{i→e}, indices)
17     patterns := []
18     indices := sort(indices)
19     l, u := 0
20     while u < len(indices)
21       if u + 1 < len(indices)
22         inc := indices[u + 1] - indices[u]
23       while u + 1 < len(indices) \
24        and indices[u + 1] - indices[u] = inc
25         u += 1
26       if l ≠ u
27         Pattern p
28         p.i := indices[l]
29         p.n := indices[u] - indices[l] + 1
30         p.e := value
31         p.inc := inc
32         patterns.add(p)
33         ρ_{i→e} := ρ_{i→e} \ {indices[i] | ∀i ∈ [l,u]}
34         l := u + 1  /* next sequence */
35       u += 1
36   return patterns
```

Fig. 4. Loop initialization pattern extraction algorithm in pseudo-code.

defines the increment value between neighbouring indices, e.g., $(l, l + inc, l + 2 \cdot inc, l + 3 \cdot inc, \ldots, u)$. If a sequence with at least two indices is found, a new *loop initialization* pattern with lower bound $p.i$, size $p.n$, and increment $p.inc$ is created and added to the $patterns$ list. Index sequences found in $\rho_{e \to i}$ correspond to $\lambda_{i \to e}$ patterns. These require a $p.e$ value, which is saved in addition (but remains unused for sequences $\rho_{i \to i}$ and $\rho_{i \to i+1}$). As before, indices included in a detected sequence s are removed from map $\rho_{i \to e}$ (line 33). If index group $indices$ has been processed, procedure `find_lpp_aux` returns the list of detected loop initialization patterns.

In case that write expressions in a write sequence are *shared*, i.e., they also appear in the formula outside of the sequence, we still extract patterns for the whole sequence. This may duplicate parts, which is not a problem since the extracted lambda terms are succinct and the "duplication" only affects the index range check of a lambda and is therefore negligible.

There are two common approaches for representing the initialization of an array variable a with n concrete values: with (1) *write* sequences of size n with array a as base array, or (2) n *read* operations on array a by asserting for each index $i \in i_1, \ldots, i_n$ that $read(a, i) = e$. In case (1), we are able to directly represent such array initializations by means of lambda terms. However, in case (2), we first have to translate the read operations into sequences of write operations. For example, given an array a that is initialized with some values e on indices $1 - 4$, we could either represent this as a sequence of write operations with a fresh array variable b as base array

$$a := write(write(write(write(b, 1, e), 2, e), 3, e), 4, e),$$

or with the following four equalities asserted to be true

$$read(a, 1) = e, read(a, 2) = e,$$
$$read(a, 3) = e, read(a, 4) = e.$$

However, the initialization with read/value equalities can also be represented as lambda term

$$a := \lambda j \,.\, ite(1 \leq j \leq 4, e, b(j)),$$

where array b is a fresh array variable. In order to extract lambda terms from these equalities, we translate them into sequences of write operations and apply the lambda extraction algorithms to it. The only requirement is that, for the same reason as for the write sequence case, the read indices have to be distinct.

IV. Merging Lambdas

Lambda terms extracted from a sequence of write operations often do not cover all indices in the sequence. Some might be left over. In order to preserve equisatisfiability, we use the uncovered write operations to create a new write sequence on top of the extracted lambda terms (cf. lines 15-16 in Fig. 2). Note that as we represent write operations as lambda terms, we actually generate a sequence of lambda terms (representing write operations) on top of the extracted terms. Given a sequence of lambda terms of size n, however, we can apply a rewriting technique we refer to as *lambda merging*, which inlines the function bodies of lambda terms $\lambda_1, \ldots, \lambda_{n-1}$. The result is a single lambda term with a function body consisting of the function bodies of lambda terms $\lambda_1, \ldots, \lambda_n$. This technique may not yield representations as compact as lambda extraction, but merging function bodies of consecutive lambdas often enables additional simplifications. As an example consider write sequence $a_n := write(a, \bar{i}, \bar{e})$ of size n, where $e_1, \ldots, e_n$ are equal. It corresponds to the following lambda sequence.

$$\begin{aligned} \lambda_n &:= \lambda j_n \,.\, ite(j_n = i_n, e, \lambda_{n-1}(j_n)), \\ &\vdots \\ \lambda_1 &:= \lambda j_1 \,.\, ite(j_1 = i_1, e, a_0(j_1)) \end{aligned}$$

If we apply lambda merging to $\lambda_n, \ldots, \lambda_1$ and inline function bodies, we obtain the following lambda term

$$\begin{aligned} \lambda_{n'} := \lambda j_n \,.\, & ite(j_n = i_n, e, \\ & \quad \ddots \\ & \qquad ite(j_n = i_1, e, a_0(j_n))) \ldots) \end{aligned}$$

```
1  procedure merge_lambdas(φ)
2    for write sequence λ_n := write(a, ī, ē) in φ
3      b := rec_merge(n, j_n, λ_n)
4      φ := φ[λ_n/b]
5
6  procedure rec_merge(n, j_n, λ_i)
7    /* base array a_0 */
8    if i = 0 return a_0(j_n)
9    /* λ_i = λj_i . ite(j_i = i_i, e_i, λ_{i-1}(j_i)) */
10   t_{i-1} := rec_merge(n, j_n, λ_{i-1})
11   if i < n
12     t_i := ite(j_i = i_i, e_i, λ_{i-1}(j_i))[j_i/j_n]
13     return t_i[λ_{i-1}(j_n)/t_{i-1}]
14   /* top-most lambda a_n */
15   return λ_n[λ_{i-1}(j_n)/t_{i-1}]
```

Fig. 5. Merge lambdas algorithm in pseudo-code.

Note that $\lambda_{n'}$ can be further simplified by merging the if-then-else terms into one (since the if-branch of each if-then-else contains value e), which results in lambda term $\lambda_{n''}$.

$$\lambda_{n''} := \lambda j_n \,.\, ite(j_n = i_n \vee \ldots \vee j_n = i_1, e, a_0(j_n))$$

A. Lemma Generation

Merged lambdas can be more compact than write sequences and may even be beneficial for lemma generation. For example, a read operation on $\lambda_{n''}$ at index j may produce a conflict on index i_1, where $read(\lambda_{n''}, j) \neq e$. As a consequence, the following lemma is generated.

$$j = i_n \vee \ldots \vee j = i_1 \rightarrow read(\lambda_{n''}, j) = e.$$

The resulting lemma covers all cases where $read(\lambda_{n''}, j)$ could produce a conflict on indices $i_2, \ldots, i_n$. In the original write sequence version, however, it might need n lemmas.

B. Algorithm

Figure 5 illustrates procedures `merge_lambdas` and `rec_merge` for merging lambda sequences. Given formula ϕ, for every lambda sequence λ_n, procedure `merge_lambdas` recursively merges the lambda terms in λ_n into lambda term b, which is then used to substitute lambda sequence λ_n in formula ϕ (line 4). Procedure `rec_merge` recursively traverses the lambda sequence starting at the top most lambda term λ_n and substitutes every bound variable j_i by the variable j_n, which is bound by the top most lambda term λ_n. In the base case ($i = 0$), the procedure returns a fresh read operation on base array a_0 at index j_n (which substitutes variable j_1). Else, it performs a recursive call on λ_{i-1}, which yields term t_{i-1}. For every lambda term λ_i with $i < n$, `rec_merge` generates lambda term t_i by substituting all occurrences of variable j_i in the function body of λ_i by j_n, and returns the lambda term obtained by substituting all occurrences of read operation $read(\lambda_{i-1}, j_n,)$ in t_i with term t_{i-1} (line 13). For the top most lambda ($i = n$), procedure `rec_merge` returns the lambda term obtained by substituting all occurrences of read operation $read(\lambda_{i-1}, j_n,)$ in λ_n with term t_{i-1} (line 15).

V. Experimental Evaluation

We implemented lambda extraction and merging in our SMT solver Boolector and evaluated our techniques on all non-extensional benchmarks from the QF_ABV category of the SMT-LIBv2 benchmark library. Six configurations are considered: (1) $\text{Boolector}_{\text{Base}}$, (2) $\text{Boolector}_{\text{E}}$, (3) $\text{Boolector}_{\text{M}}$, (4) $\text{Boolector}_{\text{X}}$, (5) $\text{Boolector}_{\text{XME}}$, and (4) $\text{Boolector}_{\text{XM}}$. The base line $\text{Boolector}_{\text{Base}}$ is an improved version of Boolector that won the QF_ABV track of the SMT competition in 2014. For the other configurations, subscript X indicates that lambda extraction is enabled, and subscript M indicates that lambda merging is enabled. Subscript E indicates an eager solving approach by reducing the formula to QF_BV. It eliminates lambda terms with beta reduction, and the remaining read operations, i.e., applications of uninterpreted functions (UF), by Ackermann reduction. The $\text{Boolector}_{\text{E}}$ and $\text{Boolector}_{\text{XME}}$ configurations essentially simulate an eager approach similar to that of UCLID [10].

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.2 LTS. The memory and time limit for each solver/benchmark pair was set to 7GB and 1200 seconds CPU time, respectively. In case of a timeout or memory out, a penalty of 1200 seconds was added to the total CPU time. Note that the time and memory limits and the hardware used for our experiments differ from the setup used at the SMT competition 2014.

Table I depicts the overall results consisting of the number of solved benchmarks (Solved), number of timeouts (TO), number of memory outs (MO), and the CPU time (Time) of all four configurations on the QF_ABV benchmarks. Enabling either lambda extraction ($\text{Boolector}_{\text{X}}$) or lambda merging ($\text{Boolector}_{\text{M}}$) improves the number of solved benchmarks by up to 17 instances and the runtime by up to 19% compared to $\text{Boolector}_{\text{Base}}$. Combining both techniques ($\text{Boolector}_{\text{XM}}$) solves 21 more benchmarks and requires 30% less runtime compared to $\text{Boolector}_{\text{Base}}$. This suggests, that lambda extraction and merging have orthogonal effects. They complement each other and in combination improve solver performance further (most of the time). However, if the eager solving approach is employed, both configurations $\text{Boolector}_{\text{E}}$ and $\text{Boolector}_{\text{XME}}$ do not show a notable improvement in terms of solved instances (less timeouts, but more memory outs). This is due to the high memory consumption caused by eager elimination of lambda terms and UFs, where $\text{Boolector}_{\text{E}}$ in total consumes 2.6 times (397 GB), and $\text{Boolector}_{\text{XME}}$ 2.3 times (347 GB) more memory than $\text{Boolector}_{\text{Base}}$. The other four configurations require roughly the same amount of memory. Table II depicts the overall results and the number of extracted patterns grouped by QF_ABV benchmark families in more detail. On benchmark families *bmc*, *brubiere2*, *klee*, *platania*, and *stp* $\text{Boolector}_{\text{XM}}$ considerably improves in terms of runtime and number of solved instances compared to $\text{Boolector}_{\text{Base}}$. On the *brubiere2* and *platania* benchmark families, the combined use of lambda extraction and lambda

Solver	Solved	TO	MO	Time [s]
Boolector$_{\text{Base}}$	13242	68	7	122645
Boolector$_{\text{E}}$	13242	49	26	120659
Boolector$_{\text{M}}$	13259	50	8	105647
Boolector$_{\text{X}}$	13256	54	7	99834
Boolector$_{\text{XME}}$	13246	47	24	111114
Boolector$_{\text{XM}}$	**13263**	46	8	**84760**

TABLE I
OVERALL RESULTS ON QF_ABV BENCHMARKS (13317 IN TOTAL).

merging yields significantly better results than both Boolector$_{\text{X}}$ and Boolector$_{\text{M}}$ alone. The most notable improvement in terms of runtime is achieved on the *klee* benchmark family, where all three configurations with *lambda extraction* enabled improve by orders of magnitude compared to Boolector$_{\text{Base}}$. The *klee* benchmark family consists of symbolic execution benchmarks obtained from KLEE [2], a symbolic virtual machine built on top of the LLVM compiler infrastructure. Previous versions of Boolector were shown to have rather poor performance on these benchmarks [8], which is confirmed by our experiments. This is due to the extreme version of lazy SMT in Boolector, using lemmas on demand. In our experiments, Boolector$_{\text{Base}}$ requires almost 13000 seconds to solve the 622 *klee* benchmarks, while lambda extraction improved runtime by up to a factor of 500 compared to Boolector$_{\text{Base}}$. This effect is illustrated by the scatter plot in Fig. 6, which shows that the runtime on most of the benchmarks is improved by a factor of 10 to 100. The *klee* benchmarks contain many instances of the λ_{mset} and $\lambda_{i\to e}$ patterns, where Boolector$_{\text{XM}}$ was able to extract 9373 and 10049 lambda terms with an average size of 108 and 11, respectively. On most of the benchmarks where Boolector$_{\text{XM}}$ was able to extract lambda terms, the runtime improved. The only exceptions are the two benchmarks in the *jager* benchmark family, on which Boolector$_{\text{XM}}$ still timed out even though 14028 λ_{mset} and 239 $\lambda_{i\to e}$ patterns were extracted. In total, Boolector$_{\text{XM}}$ was able to extract 29377 λ_{mset}, 13 λ_{mcpy}, 10683 $\lambda_{i\to e}$, 58 $\lambda_{i\to i}$, and 120 $\lambda_{i\to i+1}$ patterns with an average size of 40, 7, 12, 39, and 38, respectively. The overall time required by Boolector$_{\text{XM}}$ for extracting and merging the lambda terms amounts to 41 and 24 seconds, which is less than 0.01% of the total runtime and therefore negligible.

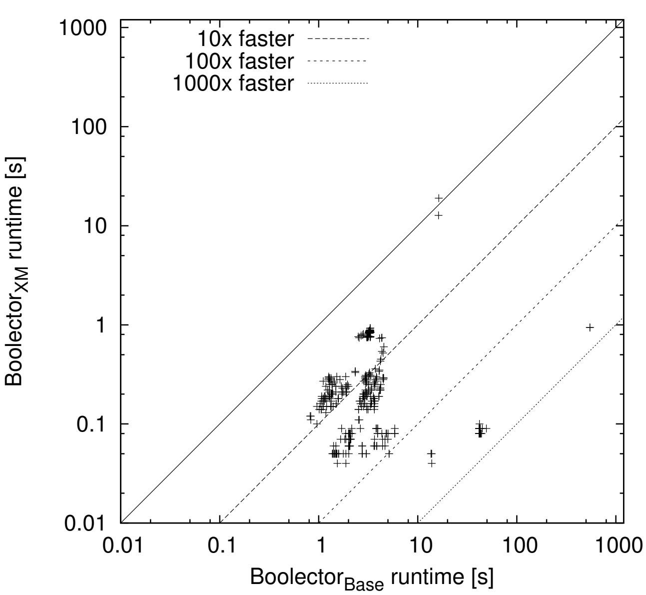

Fig. 6. Boolector$_{\text{Base}}$ vs. Boolector$_{\text{XM}}$ on *klee* benchmark family.

Benchmark family *brubiere* contains 11 benchmarks, which encode a *memcpy* operation on two non-overlapping memory regions and verify the correctness of the memcpy algorithm. The benchmarks are parameterized by the size of the copied memory region starting from size 2 up to size 12. We generated 21 additional benchmarks with size 2 to 2^{21} (i.e., 2^k with $1 \leq k \leq 21$) in order to evaluate how Boolector$_{\text{XM}}$ scales on these benchmarks. For comparison we additionally ran the top three solvers after Boolector at the SMT competition 2014, Yices [4] version 2.3.1, MathSAT [3] version 5.3.5, and SONOLAR [6] version 2014-12-04 on these benchmarks. Table III depicts the runtime of all solvers on the additional *memcpy* benchmarks of size 2 to 2^{21}, where T denotes out of time, and M denotes out of memory. Boolector$_{\text{Base}}$ and SONOLAR are able to solve these benchmarks up to size 2^5, MathSAT up to size 2^6, Yices up to size 2^9, and Boolector$_{\text{XM}}$ up to size 2^{20}. For the largest instance parsing consumes most of the runtime ($\sim$60%). For sizes greater than 2^{20}, Boolector is not able to fit the input formula into 7GB of memory, which results in a memory out.

Finally, we measured the impact of lambda extraction and lambda merging w.r.t. the number of generated lemmas. Since every lemma generated in Boolector entails an additional call to the underlying SAT solver, the number of generated lemmas usually correlates with the runtime of the solver. On the QF_ABV benchmarks commonly solved by Boolector$_{\text{Base}}$ and Boolector$_{\text{XM}}$ (13242 in total), Boolector$_{\text{Base}}$ generates 872913 lemmas, whereas Boolector$_{\text{XM}}$ generates 158175 lemmas, which is a reduction by a factor of 5.5. Consequently, the size of the CNF is reduced by 25% on average (no matter whether variables or clauses are counted). This is further illustrated in Fig. 7. On these benchmarks the reduction of the time spent in the underlying SAT solver is reduced from 59638 to 40101, i.e., an improvement of 33%.

VI. CONCLUSION

We discussed patterns of array operations occurring in actual benchmarks and presented a technique denoted as *lambda extraction*, which utilizes such patterns to extract compact and more succinct lambda terms. Another new complementary technique, called *lambda merging*, can still be exploited if lambda extraction is not applicable. These techniques allow to produce stronger and more succinct lemmas.

In the experimental analysis, based on our SMT solver Boolector, it was shown that these techniques reduce the number of generated lemmas by a factor of 5.5, and the overall size of the bit-blasted CNF by 25% on average. To summarize, we were able to considerably improve the overall performance

	Boolector$_{\text{Base}}$		Boolector$_{\text{E}}$		Boolector$_{\text{M}}$		Boolector$_{\text{X}}$		Boolector$_{\text{XME}}$		Boolector$_{\text{XM}}$		Extracted Patterns				
Family	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	λ_{mset}	λ_{mcpy}	$\lambda_{i\to e}$	$\lambda_{i\to i}$	$\lambda_{i\to i+1}$
bench (119)	119	2	119	3	119	2	119	0.3	119	0.3	119	0.3	208	0	34	0	0
bmc (38)	38	1361	39	769	39	921	39	197	**39**	**88**	39	182	256	3	56	0	0
brubiere (98)	75	29455	75	30301	75	28944	75	29359	75	29167	**75**	**28854**	0	10	0	0	0
brubiere2 (22)	17	7299	21	2617	18	6927	18	7842	**22**	**2034**	20	3241	1392	0	8	0	0
brubiere3 (8)	0	9600	1	8464	1	8435	0	9600	1	8463	1	8435	0	0	0	0	0
btfnt (1)	1	134	1	144	1	134	1	134	1	146	1	134	0	0	0	0	0
calc2 (36)	36	862	36	1528	36	864	36	863	36	1527	36	863	0	0	0	0	0
dwp (4188)	4187	2668	**4188**	**2216**	4187	2090	4187	2666	4187	2235	4187	2089	42	0	0	0	0
ecc (55)	**54**	**1792**	54	1745	54	1792	54	1845	54	1808	54	1845	125	0	0	0	0
egt (7719)	7719	222	7719	544	7719	221	7719	225	7719	275	**7719**	**212**	3893	0	0	0	0
jager (2)	0	2400	0	2400	0	2400	0	2400	0	2400	0	2400	14028	0	239	0	0
klee (622)	622	12942	620	4408	622	12688	622	169	**622**	**126**	622	154	9373	0	10049	0	0
pipe (1)	1	10	1	14	1	10	1	10	1	14	1	10	0	0	0	0	0
platania (275)	247	42690	238	58807	256	35005	255	34993	240	56172	**258**	**31189**	0	0	0	58	120
sharing (40)	40	2460	40	2459	40	2459	40	2460	40	2458	40	2458	0	0	0	0	0
stp (40)	34	8749	38	4238	39	2755	38	7072	38	4200	**39**	**2695**	60	0	297	0	0
stp_sa (52)	52	0.7	52	0.5	52	0.6	52	0.6	**52**	**0.5**	52	0.7	0	0	0	0	0
totals (13317)	13242	122645	13242	120659	13259	105647	13256	99834	13246	111114	**13263**	**84760**	29377	13	10683	58	120

TABLE II

OVERALL RESULTS AND NUMBER OF EXTRACTED PATTERNS ON ALL QF_ABV BENCHMARKS GROUPED BY BENCHMARK FAMILY.

Solver	**k=1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
Boolector$_{\text{Base}}$	0.1	0.4	8	42	296	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	M
SONOLAR	0.1	0.2	2	15	201	T	T	T	T	T	T	T	T	T	T	T	T	M	M	M	M
MathSAT	0.1	0.3	2	9	70	709	T	M	T	T	T	T	T	T	T	T	T	T	M	M	M
Yices	**0.0**	**0.0**	0.1	0.6	2	8	23	93	371	T	T	T	T	T	T	T	T	M	M	M	T
Boolector$_{\text{XM}}$	0.1	0.1	0.1	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.2**	**0.2**	**0.3**	**0.5**	**1**	**2**	**6**	**14**	**44**	**140**	**463**	M

TABLE III

RUNTIME IN SECONDS ON *memcpy* BENCHMARKS OF SIZE 2^k COPIED ELEMENTS. T DENOTES OUT OF TIME, M DENOTES OUT OF MEMORY.

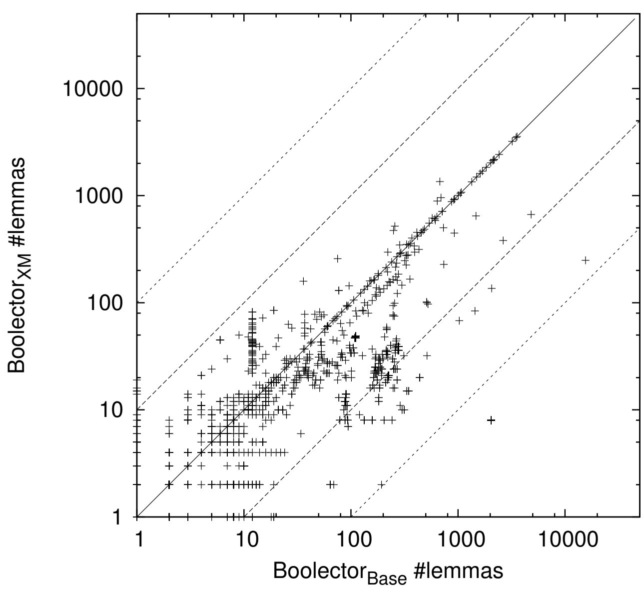

Fig. 7. Number of generated lemmas Boolector$_{\text{Base}}$ vs. Boolector$_{\text{XM}}$ on commonly solved QF_ABV benchmarks (13242 in total).

of Boolector and achieve speedups up to orders of magnitude, particularly on benchmarks from symbolic execution.

We believe, that there are additional patterns in software and hardware verification benchmarks, which can be extracted as lambdas and used to speed-up array reasoning further. Our results also suggest, that a more expressive theory of arrays might be desirable for users of SMT solvers, in order to allow more succinct encodings of common array operation patterns.

REFERENCES

[1] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proc. CAV*, volume 2404 of *LNCS*, pages 78–92. Springer, 2002.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX*, pages 209–224. USENIX Association, 2008.

[3] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proc. TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[4] B. Dutertre. Yices 2.2. In *Proc. CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[5] S. Falke, F. Merz, and C. Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Proc. VSTTE, Selected Papers*, volume 8164 of *LNCS*, pages 108–128. Springer, 2013.

[6] F. Lapschies, J. Peleska, and E. Gorbachuk. System Description: SONOLAR SMT-COMP 2014. http://smtcomp.sourceforge.net/2014/systemDescriptions/sonolar-smtcomp2014.pdf, 2014.

[7] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.

[8] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Proc. CAV*, volume 8044 of *LNCS*, pages 53–68. Springer, 2013.

[9] M. Preiner, A. Niemetz, and A. Biere. Lemmas on demand for lambdas. In *Proc. DIFTS*, volume 1130 of *CEUR Workshop Proceedings*, 2013.

[10] S. A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification.* PhD thesis, CMU, 2005.