A Theory of Arrays with set and copy Operations

(Extended Abstract)

Stephan Falke, Carsten Sinz and Florian Merz
Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
http://verialg.iti.kit.edu
{stephan.falke, carsten.sinz, florian.merz}@kit.edu

Abstract

The theory of arrays is widely used in order to model main memory in program analysis, software verification, bounded model checking, symbolic execution, etc. Nonetheless, the basic theory as introduced by McCarthy is not expressive enough for important practical cases, since it only supports array updates at single locations. In programs, memory is often modified using functions such as memcpy or memmove, which modify a user-specified range of locations whose size might not be known statically. In this paper we present an extension of the theory of arrays with set and copy operations which make it possible to reason about such functions. We also discuss further applications of the theory.

1 Introduction

The theory of arrays is widely used in order to model main memory in formal methods such as program analysis, software verification, bounded model checking, or symbolic execution. In the simplest case, main memory is modelled using a one-dimensional array, but the use of the theory of arrays goes beyond such flat memory models. Reasoning about arrays is thus an essential part of systems that are based on the aforementioned methods.

Since the basic theory of arrays is quite simple, it is insufficient (or at least inconvenient to use) in many application cases. While it supports storing and retrieving of data at specific locations, it does not support the functionality provided by C library functions such as memset or memcpy/memmove which operate on regions of locations. While these region-based operations can be broken down into operations on single locations in many cases (e.g., a memcpy operation of size 10 can be simulated using 10 read and 10 write operations), this approach does not scale if the involved regions are large. Even worse, the sizes of the affected regions might not be statically known, making it more complicated to break the region-based operation into operations on single locations.

These limitations of the theory of arrays also manifest themselves in tools implementing the formal methods mentioned above. In software bounded model checking tools such as CBMC and ESBMC, calls to memset and memcpy/memmove are handled by including an implementation of these methods and unrolling the loop contained in the implementations. Due to this unrolling, CBMC and ESBMC are incomplete in their treatment of memset and memcpy/memmove. Our own
software bounded model checking tool LLBMC \cite{20,22} was equally incomplete since it relied on user-provided implementations of `memset` and `memcpy/memmove` until we implemented the approach discussed in this work.

In this paper we present an extension of the theory of arrays with set and copy operations which make it possible to reason about `memset` and `memcpy/memmove`. We discuss several implementations of satisfiability solvers for this theory in the context of LLBMC and show with an empirical evaluation that these implementations outperform the unrolling based approach as used in CBMC and ESBMC in all but the most trivial cases.

2 The Theory of Arrays

The (non-extensional) theory of arrays ($T_A$), as introduced by McCarthy in his seminal paper of 1962 \cite{19}, is given by the following signature and axioms:

<table>
<thead>
<tr>
<th>Sorts</th>
<th>ELEMENT : elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INDEX : indices</td>
</tr>
<tr>
<td></td>
<td>ARRAY : arrays</td>
</tr>
<tr>
<td>Functions</td>
<td>read : ARRAY × INDEX → ELEMENT</td>
</tr>
<tr>
<td></td>
<td>write : ARRAY × INDEX × ELEMENT → ARRAY</td>
</tr>
</tbody>
</table>

Objects of sort ARRAY denote arrays, i.e., maps from indices of sort INDEX to elements of sort ELEMENT. The write function is used to store an element in an array. Its counter-part, the read function, is used to retrieve an element from an array.

The semantics of read and write is given be the following read-over-write axioms:\footnote{Here and in the following, all variables are implicitly universally quantified. Also, variables $a, b$ range over arrays, variables $p, q, r, s$ range over indices, and the variable $v$ ranges over elements.}

\begin{align*}
  p = r & \implies \text{read}(\text{write}(a, p, v), r) = v \tag{1} \\
  \neg(p = r) & \implies \text{read}(\text{write}(a, p, v), r) = \text{read}(a, r) \tag{2}
\end{align*}

These axioms state that storing the value $v$ into an array $a$ at index $p$ and subsequently reading $a$’s value at index $q$ results in the value $v$ if the indices $p$ and $q$ are identical. Otherwise, the write operation does not influence the result of the read operation.

In a simple implementation of a decision procedure for the theory of arrays, the read-over-write axioms could be applied from left to right using the if-then-else operator ITE, i.e., a term $\text{read}(\text{write}(a, p, v), q)$ is replaced by $\text{ITE}(p = q, v, \text{read}(a, q))$. After this transformation has been applied exhaustively, only read operations remain. These can then be treated as uninterpreted functions. Alternatively the resulting formula can be further transformed into pure equality logic using Ackermann’s construction: for all array variables $a$, let $P_a$ be the set of all index arguments that occur in a read operation for $a$. Then, each occurrence of $\text{read}(a, p)$ is replaced by a fresh variable $A_p$, and consistency constraints of the form $p_1 = p_2 \implies A_{p_1} = A_{p_2}$ for all $p_1, p_2 \in P_a$ are added as constraints to the formula.

Instead of this eager and expensive approach, modern SMT-solvers use abstraction refinement loops applying lazy axiom instantiation or lemma-on-demand techniques, see, e.g., \cite{12,4}, to efficiently support the theory of arrays.
3 The Theory of Arrays with set and copy

In order to extend $\mathcal{T}_A$ with set and copy operations we restrict INDEX to a linear arithmetical theory containing $+$, $-$, $\leq$, and $\prec$. Similar to [13], this section is based on the assumption that $\text{INDEX} = \mathbb{N}$. Furthermore, $+$, $-$, $\leq$, and $\prec$ are interpreted in the expected way. Necessary changes in order to replace $\mathbb{N}$ by fixed-width bitvectors are discussed in Section 4.

The theory of arrays with set and copy operations ($\mathcal{T}_{ASC}$) extends $\mathcal{T}_A$ by the following sorts, functions and axioms. The sort size is used for the sizes of set and copy operations. It is introduced as an alias for the sort index since the axioms contain arithmetical operations that combine terms of sort index with terms of sort size.

<table>
<thead>
<tr>
<th>Sorts</th>
<th>size = index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>ARRAY x INDEX x ELEMENT x SIZE -&gt; ARRAY</td>
</tr>
<tr>
<td>set$_\infty$</td>
<td>ARRAY x INDEX x ELEMENT -&gt; ARRAY</td>
</tr>
<tr>
<td>copy</td>
<td>ARRAY x INDEX x ARRAY x INDEX x SIZE -&gt; ARRAY</td>
</tr>
<tr>
<td>copy$_\infty$</td>
<td>ARRAY x INDEX x ARRAY x INDEX -&gt; ARRAY</td>
</tr>
</tbody>
</table>

The informal semantics of $\text{set}(a, p, v, s)$ is to set all entries of $a$ that are between $p$ and $p + s - 1$ to the value $v$. Similarly, $\text{set}_\infty(a, p, v)$ sets all entries from $p$ on upwards to $v$.

Formally, the semantics of set is given by the following read-over-set axioms:

$$p \leq r < p + s \implies \text{read}(\text{set}(a, p, v, s), r) = v \quad (3)$$

$$\neg(p \leq r < p + s) \implies \text{read}(\text{set}(a, p, v, s), r) = \text{read}(a, r) \quad (4)$$

Similarly, the read-over-set$_\infty$ axioms give a semantics to set$_\infty$:

$$p \leq r \implies \text{read}(\text{set}_\infty(a, p, v), r) = v \quad (5)$$

$$\neg(p \leq r) \implies \text{read}(\text{set}_\infty(a, p, v), r) = \text{read}(a, r) \quad (6)$$

The axioms have some similarity to the read-over-write axioms [1] and [2], but the simple equality comparison in the read-over-write axioms has been replaced by a more complex check for containment in a region of indices.

Informally, copy and copy$_\infty$ are used in order to copy a range of values from one array to another array. This copying process may also involve a shift, i.e., the ranges in the arrays do not need to start at the same index position. See Figure 1 for a graphical illustration of copy.
The semantics of \texttt{copy} is given by the \textit{read-over-copy} axioms
\begin{align*}
  p \leq r < p + s & \implies \text{read}(\text{copy}(a, p, b, q, s), r) = \text{read}(b, q + (r - p)) \quad (7) \\
  \neg(p \leq r < p + s) & \implies \text{read}(\text{copy}(a, p, b, q, s), r) = \text{read}(a, r) \quad (8)
\end{align*}
and the semantics of \texttt{copy}_\infty is specified using the \textit{read-over-copy}_\infty axioms
\begin{align*}
  p \leq r & \implies \text{read}(\text{copy}_\infty(a, p, b, q, r) = \text{read}(b, q + (r - p)) \quad (9) \\
  \neg(p \leq r) & \implies \text{read}(\text{copy}_\infty(a, p, b, q, r) = \text{read}(a, r) \quad (10)
\end{align*}

**Example 1.** Consider the formula
\begin{align*}
s > 0 & \land b = \text{copy}(a, p, a, q, s) \land o < s \land \text{read}(b, p + o) \neq \text{read}(a, q + o)
\end{align*}
in \mathcal{T}_{ASC}. It is intuitively unsatisfiable since the \texttt{copy} operation ensures that \text{read}(b, p + o) and \text{read}(a, q + o) are identical for all \(o < s\). The unsatisfiability can formally be shown using reasoning w.r.t. the axioms of \mathcal{T}_{ASC}.

## 4 Using Bitvectors as Indices

In principle, the natural numbers used as indices in Section 3 can readily be replaced by fixed-width bitvectors if \(+\) and \(-\) are replaced by bitvector addition (\texttt{bvadd}) and bitvector subtraction (\texttt{bvsub}) and \(\leq\) and \(<\) are replaced by unsigned less-than-or-equal (\texttt{bvule}) and less-than (\texttt{bvult}) operations, respectively. In the following discussion, we only consider \texttt{set} and \texttt{copy}.

For the \textit{read-over-set} axioms 3 this results in the following axioms for the \texttt{set} operation:
\begin{align*}
bvule(p, r) \land bvult(r, bvadd(p, s)) & \implies \text{read}(\text{set}(a, p, v, s), r) = v \\
\neg(bvule(p, r) \land bvult(r, bvadd(p, s))) & \implies \text{read}(\text{set}(a, p, v, s), r) = \text{read}(a, r)
\end{align*}

The \textit{read-over-copy} axioms for \texttt{copy} now read as follows:
\begin{align*}
bvule(p, r) \land bvult(r, bvadd(p, s)) & \implies \text{read}(\text{copy}(a, p, b, q, s), r) = \text{read}(b, bvadd(q, bvsub(r, p))) \\
\neg(bvule(p, r) \land bvult(r, bvadd(p, s))) & \implies \text{read}(\text{copy}(a, p, b, q, s), r) = \text{read}(a, r)
\end{align*}

Notice that the wrap-around behavior of bitvector arithmetic that is caused by overflows leads to subtle semantic differences compared to the case of natural numbers. There are two cases in which a wrap-around can occur: (a) in the target range of a \texttt{set} or \texttt{copy} operation, i.e., in the computation of \texttt{bvadd}(p, s), or (b) in the source range of a \texttt{copy} operation, i.e., in the computation of \texttt{bvadd}(q, \texttt{bvsub}(r, p)).

A target range overflow \(6\) causes the left-hand side of the implication to be always false for the first axiom and always true for the second axiom for both \texttt{set} and \texttt{copy}. This means that an overflow in the target range of any of the operations turns it into a no-op.

In contrast to target range overflows, source range overflows \(7\) of a \texttt{copy} operation behave as expected, i.e., the indices from which the values are copied wrap around.\(^3\)

\(^3\)If consistency of the two overflow cases is desired, an overflow in the source destination could be ruled out by adding an overflow check \(\neg\texttt{bvadd}(q, s)\) to the left-hand side of the implication in the second \textit{read-over-copy} axiom. This turns the \texttt{copy} operation into a no-op, just like a target overflow does.
5 Applications

As already mentioned in Section 1, the most immediate use of $T_{ASC}$ is to precisely and succinctly model library functions such as `memset` or `memcpy/memmove` in C. There are, however, several further applications of the theory for reasoning about programs which we would like to explore in future work:

- Zero-initialization of global variables (as required by the C standard) can be achieved using the set operation.
- Operating systems zero-initialize new memory pages before handing them to a process. This can again be achieved using an appropriate set operation.
- If a memory region should be set to unconstrained values (havocked), this can be done using a copy operation from a fresh array variable. Similarly, the same approach could be used in order to model memory-mapped I/O.
- Tracking metadata for memory addresses. For instance, allocation information can be modeled using an array containing information on the allocation state of the locations. Memory allocation and memory de-allocation can then be modelled using the set operation. This makes it possible to develop an alternative to the SMT theory of memory allocation presented in [11] and the memory model presented in [21].

6 Satisfiability Solving

Current SMT-solvers do not support $T_{ASC}$ yet. There are, however, several possibilities to reduce the satisfiability problem for quantifier-free $T_{ASC}$ formulas to the satisfiability problem for a theory that is supported by existing SMT-solvers. In the following we only consider the set and copy operations, the set$_\infty$ and copy$_\infty$ operations are similar.

Eager Encoding. In this setting, the axioms are applied in a pre-processing step before the resulting formula is passed to an SMT-solver. This pre-processing is done similarly to the case of $T_A$ as discussed in Section 2 by applying the axioms from left to right using the ITE operator. The main disadvantage of this approach is that the read-over-write axioms have to be applied eagerly as well in the pre-processing in order to eliminate writes that are “between” a read operation and a set or copy operation. The main advantage of the eager encoding is that any SMT-solver that can handle $T_A$ in combination with the index theory can be used.

Example 2. For the formula from Example 1 the eager encoding produces the formula

$$s > 0 \land o < s \land ITE(p < p + o < p + s, \text{read}(a, q + o), \text{read}(a, p + o)) \neq \text{read}(a, q + o)$$

which can easily be seen to be unsatisfiable.

Fully Quantified Axioms. In this approach, set and copy are added as interpreted function symbols whose semantics is given by explicitly stating the quantified axioms. As an example, the fully quantified axiom for set has the form

$$\forall a: \text{ARRAY}, p: \text{INDEX}, v: \text{ELEMENT}, s: \text{SIZE}, r: \text{INDEX}. \text{read(set}(a, p, v, s), r) =$$

$$\text{ITE}(p \leq r < p + s, v, \text{read}(a, r))$$
The use of quantifiers makes the satisfiability problem hard, but it remains (in principle) decidable if bitvectors are used as indices.

Quantified Axioms Instantiated for Arrays. As in the previous approach, \texttt{set} and \texttt{copy} are added as interpreted function symbols but the fully quantified axiom is partially instantiated for those arrays to which the operators are applied in the formula. Furthermore, each occurrence of, e.g., \texttt{set} is replaced by a fresh interpreted function symbol whose arity coincides with the number of arguments that are still quantified. Then, suitable constraints on this function symbol are added. If the formula contains, e.g., a term of the form \texttt{set}(a', p', v', s'), then the constraint

\[
\forall p: \text{INDEX}, v: \text{ELEMENT}, s: \text{SIZE}, r: \text{INDEX}. \ \text{read}_{a'}(p, v, s) = \text{ITE}(p \leq r < p + s, v, \text{read}(a', r))
\]

is added to the formula.

Quantified Axioms Instantiated for all Arguments. Here, the fully quantified axiom is instantiated even further using all arguments occurring in applications to \texttt{set} and \texttt{copy}. Concretely, if the formula contains a term of the form \texttt{set}(a', p', v', s'), then the following constraint is added to the formula:

\[
\forall r: \text{INDEX}. \ \text{read}_{a', p', v', s'}(r) = \text{ITE}(p' \leq r < p' + s', v', \text{read}(a', r))
\]

Notice that the interpreted function symbol for \texttt{set}_{a', p', v', s'} is now just a constant symbol.

Instantiation-Based Approach. As in the previous approach, each occurrence of \texttt{set} and \texttt{copy} in the formula is replaced by a fresh constant symbol and constraints on this constant are added. In contrast to the three aforementioned approaches, these constraints do not contain any quantifier. Instead, an instantiation of the quantified constraint from the previous approach is added for all needed read indices. Here, the needed read indices are the (potentially) observed entries of the array produced by the \texttt{set} or \texttt{copy} operation (i.e., entries of the array that are read later on). They are determined by the following algorithm, in which we assume formulas to be represented as terms. Moreover, \texttt{root(x)} denotes the root symbol of a term \(x\) and \(y \text{array} x\) denotes that \(x\) is a (non-strict) subterm of \(y\) such that the path leading from the root of \(y\) to the root of \(x\) contains only function symbols of sort \texttt{array}:

```plaintext
/* x is a subterm of the complete formula \(\varphi\) with \texttt{root}(x) = \texttt{copy} or \texttt{root}(x) = \texttt{set} */
function \texttt{GetNeededReadIndices(}\varphi, x) \n\begin{align*}
N &= \emptyset \\
C &= \{ y \mid y \text{array} x \land \text{root}(y) = \texttt{copy} \land \neg \exists z. (y \text{array} z \text{array} x \land \text{root}(z) = \texttt{copy}) \} \\
\text{for } c := \texttt{copy}(a, p, b, q, s) \in C \text{ do} \\
\text{if } a \text{array} x \text{ then} \\
\quad \text{/* add recursively computed indices */} \\
\quad N = N \cup \text{GetNeededReadIndices}(\varphi, c) \\
\text{end if} \\
\text{if } b \text{array} x \text{ then} \\
\quad \text{/* apply “pointer arithmetic” to the recursively computed indices */} \\
\quad N = N \cup \{ q + (r - p) \mid p \in \text{GetNeededReadIndices}(\varphi, c) \} \\
\text{end if}
\end{align*}
```

103
A Theory of Arrays with \texttt{set} and \texttt{copy} Operations

S. Falke, C. Sinz, F. Merz

end for
/* compute the \texttt{set} and write superterms “between” \texttt{x} and \texttt{C} */
\begin{verbatim}
A = \{y \mid y \not\subseteq \text{array} x \land \exists z \in C. y \not\subseteq \text{array} z \not\subseteq \text{array} x\}
\end{verbatim}
/* compute the indices used in \texttt{reads} of elements from \texttt{A} */
\begin{verbatim}
N' = \{y \mid \exists z \in A. \text{read}(z, y) \text{ occurs in } \varphi\}
\end{verbatim}
return \(N \cup N'\)
end function

Since the final formula does not contain quantifiers, any SMT-solver that can handle \(T_A\) in combination with the index theory can be applied.

Example 3. For the formula from Example 1, the instantiation-based approach produces the formula
\begin{align*}
s > 0 \land \text{read}(b, p + o) = & \text{ITE}(p < p + o < p + s, \text{read}(a, q + o), \text{read}(a, p + o)) \\
& \land o < s \land \text{read}(b, p + o) \neq \text{read}(a, q + o)
\end{align*}
which again can easily be seen to be unsatisfiable. For the construction of the formula, notice that \(p + o\) is the only read index that is needed as an instantiation.

7 Evaluation

We have experimented with all possible implementations described in Section 6 for determining the satisfiability of quantifier-free formulas in \(T_{ASC}\). Since our motivation was the application in our bounded model checking tool \textsc{LLBMC} \cite{20,22}, we have restricted attention to the case where INDEX consists of bitvectors.

The bounded model checking tool \textsc{LLBMC} is a bounded model checker for C and (to some extent) C++ programs. In order to support the complex and intricate syntax and semantics of these programming languages, \textsc{LLBMC} uses the LLVM compiler framework \cite{17} in order to translate C and C++ programs into LLVM’s intermediate representation (IR). This IR is then converted into a logical representation and simplified using an extensive set of rewrite rules. The simplified formula is finally passed to an SMT-solver. Distinguishing features of \textsc{LLBMC} in comparison with related tools such as \textsc{CBMC} \cite{7} and \textsc{ESBMC} \cite{8} are its use of a flat, bit-precise memory model, its exhaustive set of built-in checks, and its performance (see \cite{20}).

Within \textsc{LLBMC}, we evaluated the following approaches for \(T_{ASC}\) satisfiability:

1. The eager encoding and the instantiation-based approach were evaluated in combination with the SMT-solver \textsc{STP} (SVN revision 1625) \cite{12} (alternatively, any other SMT-solver for the logic QF_ABV could be applied, e.g., \textsc{Boolector} \cite{3} or \textsc{Z3} \cite{9}).

2. The approaches based on fully quantified or partially instantiated axioms have been evaluated in combination with the SMT-solver \textsc{Z3} (version 3.2) \cite{9}, because \textsc{STP} does not support quantifiers.

3. \texttt{set} and \texttt{copy} are simulated using loops on the level of LLVM’s IR. Consequently, the boundedness restriction inherent to bounded model checking applies to these loops. This approach was again evaluated in combination with \textsc{STP}.

These approaches have been evaluated on a collection of 55 C and C++ programs (19 programs from \textsc{LLBMC}’s test suite that were not specifically written for \(T_{ASC}\), 19 programs from \textsc{LLBMC}’s test suite that were specifically written for \(T_{ASC}\), and 17 programs from other sources (16 programs...
A Theory of Arrays with \texttt{set} and \texttt{copy} Operations

S. Falke, C. Sinz, F. Merz

<table>
<thead>
<tr>
<th>Approach</th>
<th>SMT-solver</th>
<th>Total Time</th>
<th># Solved Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instantiation-Based Approach</td>
<td>STP</td>
<td>25.368</td>
<td>55</td>
</tr>
<tr>
<td>Eager Encoding</td>
<td>STP</td>
<td>64.008</td>
<td>54</td>
</tr>
<tr>
<td>Built-In Loops</td>
<td>STP</td>
<td>225.673</td>
<td>50</td>
</tr>
<tr>
<td>Axioms Instantiated for all Arguments</td>
<td>Z3</td>
<td>339.676</td>
<td>45</td>
</tr>
<tr>
<td>Axioms Instantiated for Arrays</td>
<td>Z3</td>
<td>420.876</td>
<td>42</td>
</tr>
<tr>
<td>Fully Quantified Axioms</td>
<td>Z3</td>
<td>449.767</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 1: Times and success rates for the different approaches.

from [1] and one program from the \texttt{KLEE} distribution)). Notice that the logical encoding of these programs may contain a \texttt{set} or \texttt{copy} operation for one of several reasons:

- The source code contains an explicit call to \texttt{memset} or \texttt{memcpy}/\texttt{memmove}.
- The compiler may introduce a call to \texttt{memset} in order to initialize global variables.
- Compiler optimizations may detect user-written functionality that essentially constitutes a re-implementation of \texttt{memset} or \texttt{memcpy}/\texttt{memmove} and replace the code by calls to the respective library functions.
- Library-specific implementation details included through header files may result in calls to \texttt{memset} or \texttt{memcpy}/\texttt{memmove}. This is in particular true for C++ programs that use the container classes of the standard template library (STL).
- Default implementations of C++ constructors, especially the copy constructor, may make use of \texttt{memcpy} operations to initialize the memory.

For the purpose of this evaluation, also \texttt{memset} and \texttt{memcpy}/\texttt{memmove} with a statically known size were not broken into operations on single locations.

The results of LLBMC on the collection of examples are summarized in Table 1. The reported times are in seconds and only record the time needed for solving the simplified formula, i.e., the time needed for the logical encoding and the simplification of the formula are not included. A timeout of 30 seconds was imposed for each formula and timeouts are contained in the total times. The experiments where performed on an Intel® Core™ 2 Duo 2.4GHz with 4GB RAM.

These results indicate that the instantiation-based approach achieves the best performance, which can also be observed in the scatterplot contained in Figure 2. Notice that Z3 in combination with fully quantified or partially instantiated axioms is significantly slower and less powerful than the approaches that do not involve quantified formulas. Also, the approaches without quantifiers perform better than the naïve implementation using loops on the level of LLVM’s IR, where the latter is furthermore incomplete due to the bounded number of loop iterations that can be considered.  

8 Related Work

Decidable extensions of the theory of arrays have been considered before. Suzuki and Jefferson [24] have studied the extension of $\mathcal{T}_A$ with a restricted use of a permutation predicate. Mateti

\footnote{This incompleteness does not manifest itself in the evaluation since the number of loop iterations was chosen sufficiently large for each example.}
A Theory of Arrays with **set** and **copy** Operations

S. Falke, C. Sinz, F. Merz

9 Conclusions

We have presented an extension of the theory of arrays with **set** and **copy** operations that can be used in order to model library functions such as C’s **memset** and **memcpy/memmove** in formal methods such as program analysis, software verification, bounded model checking, or symbolic execution. We have discussed several possible implementations of satisfiability solvers for this theory and have reported on an evaluation in our bounded model checking tool **LLBMC** [20] [22].
In future work, we are particularly interested in adding “native” support for $\mathcal{T}_{\text{ASC}}$ in SMT-solvers such as STP [12] or Boolector [3]. For this, it is necessary to investigate lazy axiom instantiation or lemma-on-demand techniques for $\mathcal{T}_{\text{ASC}}$ since these techniques have been fundamental for the performance gain that SMT-solvers for $\mathcal{T}_{A}$ have experienced in recent years. Furthermore, we will formally prove soundness and completeness of the satisfiability solvers discussed in Section 6 in particular for the instantiation-based approach. Finally, we will investigate the complexity of satisfiability solving for quantifier-free $\mathcal{T}_{\text{ASC}}$ formulas and of the possible solvers discussed in Section 6.

References


