Extending the Theory of Arrays: `memset`, `memcpy`, and Beyond

Stephan Falke, Florian Merz, and Carsten Sinz

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{stephan.falke, florian.merz, carsten.sinz}@kit.edu

Abstract. The theory of arrays is widely used in program analysis, (deductive) software verification, bounded model checking, and symbolic execution to model arrays in programs or the computer's main memory. Nonetheless, the theory as introduced by McCarthy is not expressive enough in many cases since it only supports array updates at single locations. In programs, memory is often modified at multiple locations at once using functions such as `memset` or `memcpy`. Furthermore, initialization loops that store loop-counter-dependent values in an array are commonly used. This paper presents an extension of the theory of arrays with λ-terms which makes it possible to reason about such cases. We also discuss how loops can be automatically summarized using such λ-terms.

1 Introduction

The theory of arrays is widely used in formal methods such as program analysis, (deductive) software verification, bounded model checking, or symbolic execution. In the most simple case, the computer's 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 theory of arrays is quite basic, it is insufficient (or at least inconvenient to use) in many application cases. While it supports storing and loading of data at specific locations, it does not support the functionality provided by C library functions such as `memset` or `memcpy` which operate on regions of locations. While these region-based operations can be broken down into operations on single locations in some 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 down region-based operation into operations on single locations.

Apart from library functions, a further construct that often occurs in real-life programs are initialization loops such as

```
1 for (i = 0; i < n; ++i) {
2     a[i] = 2 * i + 1;
3 }
```

which sets the array entry $a[i]$ to the value $2 * i + 1$ for all indices between 0 and $n - 1$. Representing the array a after these initializations is not easily possible in the theory of arrays if n is a large constant or not statically known.

In software bounded model checking tools such as `CBMC` [9] or `ESBMC` [11], calls to `memset` and `memcpy` 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` if the number of loop iterations cannot be bounded by a constant.[1] Our own software bounded model checking tool `LLBMC` [24] was equally incomplete since it relied on user-provided implementations of `memset` and `memcpy` until we implemented the approach discussed in the preliminary version of this work [14].

In this paper, we present an extension of the theory of arrays with λ-terms which makes it possible to reason about `memset`, `memcpy`, initialization loops as discussed above, etc. We show that satisfiability of quantifier-free formulas in this theory is decidable by presenting three reductions to decidable theories supported by SMT solvers. An evaluation shows that using this new theory in `LLBMC` outperforms the unrolling based approach as used in `CBMC` and `ESBMC`.

Example 1. Consider the following program fragment:

```
1  int i, j, n = ...;
2  int *a = malloc(2 * n * sizeof(int));
3  for (i = 0; i < n; ++i) {
4      a[i] = i + 1;
5  }
6  for (j = n; j < 2 * n; ++j) {
7      a[j] = 2 * j;
8  }
```

Using the theory of arrays with λ-terms, the array a after executing line 2 can be described using a fresh constant a_2 since nothing is known about the content of the array. The array a after executing the loop in lines 3–5 can be described using the λ-term $a_5 = \lambda i.\ \mathsf{ITE}(0 \le i < n,\ i+1,\ \mathsf{read}(a_2, i))$ which represents the array containing as entry $a[i]$ the value $i+1$ whenever $0 \le i < n$, and the original value of a at index i (i.e., $\mathsf{read}(a_2, i)$) otherwise. Here, ITE is the *if-then-else* operator. Similarly, the array a after executing the loop in lines 6–8 can be described using the λ-term $a_8 = \lambda j.\ \mathsf{ITE}(n \le j < 2*n,\ 2*j,\ \mathsf{read}(a_5, j))$, which could be simplified to get $a'_8 = \lambda j.\ \mathsf{ITE}(n \le j < 2*n,\ 2*j,\ \mathsf{ITE}(0 \le j < n,\ j+1,\ \mathsf{read}(a_2, j)))$. ◊

A preliminary version of this work has appeared as an extended abstract in [14]. This paper extends that preliminary version in two important directions:

- The previous version was restricted to `memset` and `memcpy` and did not support any other extension of the theory of arrays. As shown in this paper, the use of λ-terms makes it possible to simulate `memset` and `memcpy`, as well as many kinds of initialization loops. Furthermore, we discuss how such loop can be summarized *automatically* using λ-terms.

[1] The situation is similar in symbolic execution tools such as `EXE` [8] or `KLEE` [7].

– While [14] discusses decidability of the extended theory of arrays, soundness and correctness proofs were missing. In contrast, the aforementioned reductions are formally shown to be sound and complete in this paper.

The present paper is structured as follows: Sect. 2 presents preliminaries and fixes notation. Sect. 3 first recalls the theory $\mathcal{T}_{\mathcal{A}}$ of arrays and then introduces our generalization $\mathcal{T}_{\lambda\mathcal{A}}$. Several uses of $\mathcal{T}_{\lambda\mathcal{A}}$, including loop summarization, are discussed in Sect. 4. Reductions that establish the decidability of satisfiability for quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formulas are presented in Sect. 5. Sect. 6 describes the implementation within LLBMC and contains the results of an evaluation of the different reductions. Related work is surveyed in Sect. 7, while Sect. 8 concludes.

2 Preliminaries

In many-sorted logic, a *signature* Σ is a triple $(\Sigma^S, \Sigma^F, \Sigma^P)$ where Σ^S is a set of sorts, Σ^F is a set of function symbols, and Σ^P is a set of predicate symbols. Σ-*terms*, Σ-*formulas*, and Σ-*sentences* are defined in the usual way.

We use the standard definition of a Σ-*structure* $\mathfrak{M}$. It contains non-empty, pairwise disjoint sets M_σ for every sort $\sigma \in \Sigma^S$ and an interpretation of the function symbols in Σ^F and the predicate symbols in Σ^P that respects sorts and arities. We use $\mathfrak{M}(f)$ to denote the interpretation of $f \in \Sigma^F$ in $\mathfrak{M}$ and $\mathfrak{M}(P)$ to denote the interpretation of $P \in \Sigma^P$ in $\mathfrak{M}$. The interpretation of an arbitrary term t in $\mathfrak{M}$ is denoted $[\![t]\!]^{\mathfrak{M}}$ and defined in the standard way. Similarly, $[\![\varphi]\!]^{\mathfrak{M}} \in \{\top, \bot\}$ denotes the truth value of a formula φ in $\mathfrak{M}$. Finally, a structure $\mathfrak{M}$ is a model of a formula φ if $[\![\varphi]\!]^{\mathfrak{M}} = \top$.

A (first-order) Σ-*theory* $\mathcal{T}$ is a set of Σ-sentences, its axioms. An *empty theory* is a theory not containing any axioms. A Σ-theory is *single-sorted* if $|\Sigma^S| = 1$. For a single-sorted theory $\mathcal{T}_i$, its only sort is usually denoted by σ_i.

Two signatures Σ_1 and Σ_2 are *disjoint* if $F_1 \cap F_2 = \varnothing$ and $P_1 \cap P_2 = \varnothing$. A Σ_1-theory $\mathcal{T}_1$ and a Σ_2-theory $\mathcal{T}_2$ are disjoint if Σ_1 and Σ_2 are disjoint. The *combined theory* $\mathcal{T}_1 \oplus \mathcal{T}_2$ of two disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$ is the $(\Sigma_1 \cup \Sigma_2)$-theory containing the union of $\mathcal{T}_1$'s and $\mathcal{T}_2$'s axioms. Theory combination of a theory with itself is defined to be the same theory again: $\mathcal{T}_1 \oplus \mathcal{T}_1 = \mathcal{T}_1$.

The symbol $=_\sigma$ is implicitly defined for most sorts σ. It is not part of any signature Σ and is always interpreted as the identity relation over σ. For brevity, its subscript is usually omitted.

If x, t_1, t_2 are terms, then $t_1[x/t_2]$ stands for the term obtained from t_1 by substituting all occurrences of x by t_2. A substitution is applied to a formula by applying it to all terms in the formula.

For two terms t_1, t_2, writing $t_1 \hookrightarrow t_2$ indicates that the term t_1 can be *simulated* by the term t_2. This means that for any formula φ containing t_1, the formula $\varphi[t_1/t_2]$ is equivalent to φ. Thus, t_1 can be rewritten to t_2.

For any formula ψ and terms t_1, t_2 with the same sort σ, the meta-symbol $\mathsf{ITE}(\psi,\ t_1,\ t_2)$ stands for an *if-then-else* expression. Conceptually, for any formula φ containing the term $t \equiv \mathsf{ITE}(\psi,\ t_1,\ t_2)$, an equisatisfiable formula φ' not

containing t can be constructed as follows. If the identity relation $=_\sigma$ is available, then φ' can be defined as

$$\varphi[t/t_3] \ \wedge\ (\psi \implies t_3 = t_1) \ \wedge\ (\neg\psi \implies t_3 = t_2)$$

where t_3 is a fresh constant. If $=_\sigma$ is not available, then φ' can be defined as

$$(\psi \ \wedge\ \varphi[t/t_1]) \ \vee\ (\neg\psi \ \wedge\ \varphi[t/t_2])$$

Note that most SMT solvers natively support the ITE construct, i.e., φ' does not need to be constructed up front.

3 The Theory $\mathcal{T}_{\lambda\mathcal{A}}$

The theory $\mathcal{T}_{\lambda\mathcal{A}}$ is an extension of the non-extensional theory of arrays $\mathcal{T}_{\mathcal{A}}$ that was introduced by McCarthy in his seminal paper [23] in 1962. The theory $\mathcal{T}_{\mathcal{A}}$ is parameterized by the *index theory* $\mathcal{T}_{\mathcal{I}}$ and the *element theory* $\mathcal{T}_{\mathcal{E}}$. Here, both $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{E}}$ are single-sorted theories of sort $\sigma_{\mathcal{I}}$ and $\sigma_{\mathcal{E}}$, respectively. Note that $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{E}}$ may coincide. In the most simple case, both $\sigma_{\mathcal{I}}$ and $\sigma_{\mathcal{E}}$ are uninterpreted sorts and $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{I}}$ are both empty. In practice, $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{E}}$ are often the theory of linear integer arithmetic ($\mathcal{T}_{\mathcal{LIA}}$) or the theory of bit-vectors ($\mathcal{T}_{\mathcal{BV}}$). $\mathcal{T}_{\mathcal{A}}$ now adds the sort $\sigma_{\mathcal{A}}$ and function symbols $\mathsf{read} : \sigma_{\mathcal{A}} \times \sigma_{\mathcal{I}} \to \sigma_{\mathcal{E}}$ and $\mathsf{write} : \sigma_{\mathcal{A}} \times \sigma_{\mathcal{I}} \times \sigma_{\mathcal{E}} \to \sigma_{\mathcal{A}}$ to the combination $\mathcal{T}_{\mathcal{I}} \oplus \mathcal{T}_{\mathcal{E}}$. Due to non-extensionality, $=_{\sigma_{\mathcal{A}}}$ is not available. *Terms* in $\mathcal{T}_{\mathcal{A}}$ are built according to the following grammar, where the detailed definitions of $t_{\mathcal{I}}$ and $t_{\mathcal{E}}$ depend on the theories $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{E}}$:

index terms	$t_{\mathcal{I}} ::= \ldots$
element terms	$t_{\mathcal{E}} ::= \ldots \mid \mathsf{read}(t_{\mathcal{A}}, t_{\mathcal{I}})$
array terms	$t_{\mathcal{A}} ::= a \mid \mathsf{write}(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}})$

Here, a stands for a constant of sort $\sigma_{\mathcal{A}}$.

Objects of sort $\sigma_{\mathcal{A}}$ denote arrays, i.e., maps from indices to elements. The write function is used to store an element in an array, and the read function is used to retrieve an element from an array. Formally, the semantics of these functions is given by the following read-*over*-write axioms:[2]

$$p = r \implies \mathsf{read}(\mathsf{write}(a, p, v), r) = v \quad (1)$$

$$\neg(p = r) \implies \mathsf{read}(\mathsf{write}(a, p, v), r) = \mathsf{read}(a, r) \quad (2)$$

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

[2] Here and in the following, all variables in axioms 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.

In a simple implementation of a decision procedure for $\mathcal{T}_{\mathcal{A}}$ based on the *reduction approach* [20], the read-over-write axioms are applied from left to right using the *if-then-else* operator ITE, i.e., a term $\mathsf{read}(\mathsf{write}(a, p, v), q)$ is replaced by $\mathsf{ITE}(p = q,\ v,\ \mathsf{read}(a, q))$. After this transformation has been applied exhaustively, only read operations where the first argument is a constant remain. The read symbol can then be treated as an uninterpreted function, and a decision procedure for the combination $\mathcal{T}_{\mathcal{I}} \oplus \mathcal{T}_{\mathcal{E}} \oplus \mathcal{T}_{\mathcal{EUF}}$ can be used, where $\mathcal{T}_{\mathcal{EUF}}$ denotes the theory of equality with uninterpreted functions.

Instead of this eager approach, modern SMT solvers use abstraction refinement. For this, they apply techniques such as lazy axiom instantiation or lemmas-on-demand (see, e.g., [5, 15]) to efficiently support the theory of arrays.

The theory $\mathcal{T}_{\lambda\mathcal{A}}$ extends the theory of arrays by anonymous arrays that are built using λ-expressions, i.e., the term formation rules are extended as follows:

index terms	$t_{\mathcal{I}} ::= \ldots$
element terms	$t_{\mathcal{E}} ::= \ldots \mid \mathsf{read}(t_{\mathcal{A}}, t_{\mathcal{I}})$
array terms	$t_{\mathcal{A}} ::= a \mid \mathsf{write}(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}}) \mid \lambda i.\ t_{\mathcal{E}}$

Here, the "i" occurring in the λ-expression $\lambda i.\ t_{\mathcal{E}}$ is a bound variable of sort $\sigma_{\mathcal{I}}$. In $\mathcal{T}_{\lambda\mathcal{A}}$, the bound variable i may not occur below any further λ-binder (i.e., each occurrence of i has De Bruijn index 1).[3]

Intuitively, $\lambda i.\ s$ denotes the anonymous array that maps each index i to the element denoted by the term s. Formally, this is captured by the following read-*over*-λ axiom scheme:

$$\mathsf{read}(\lambda i.\ s, r) = s[i/r] \tag{3}$$

Here, variables bound by λ-terms within s are first suitably renamed in order to be different from i. This axiom scheme is essentially the well-known β-reduction from λ-calculus.

Note that array terms of the form $\mathsf{write}(a, p, v)$ can be simulated using λ-terms as follows:

$$\mathsf{write}(a, p, v) \quad \hookrightarrow \quad \lambda i.\ \mathsf{ITE}(i = p,\ v,\ \mathsf{read}(a, i))$$

It is, however, advantageous to keep the write operation since this makes it possible to reduce $\mathcal{T}_{\lambda\mathcal{A}}$ to $\mathcal{T}_{\mathcal{A}}$ instead of the combination $\mathcal{T}_{\mathcal{I}} \oplus \mathcal{T}_{\mathcal{E}} \oplus \mathcal{T}_{\mathcal{EUF}}$. Thus, the efficient techniques employed by modern SMT solvers for $\mathcal{T}_{\mathcal{A}}$ can be applied (see Sect. 5 for details).

In [14], we have presented the theory $\mathcal{T}_{\mathcal{ASC}}$, which generalizes $\mathcal{T}_{\mathcal{A}}$ by introducing set, set_∞, copy, and copy_∞ operations. In $\mathcal{T}_{\mathcal{ASC}}$, the term formation rules of $\mathcal{T}_{\mathcal{A}}$ are extended as follows:

index terms	$t_{\mathcal{I}} ::= \ldots$
element terms	$t_{\mathcal{E}} ::= \ldots \mid \mathsf{read}(t_{\mathcal{A}}, t_{\mathcal{I}})$
array terms	$t_{\mathcal{A}} ::= a \mid \mathsf{write}(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}})$
	$\mid \mathsf{set}(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}}, t_{\mathcal{I}}) \mid \mathsf{set}_\infty(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}})$
	$\mid \mathsf{copy}(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{I}}) \mid \mathsf{copy}_\infty(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{A}}, t_{\mathcal{I}})$

[3] This is not a restriction when modeling programs where an array at a given point in the program does not depend on arrays at a *later* point in the program.

For $\mathcal{T}_{\mathcal{ASC}}$, the index theory $\mathcal{T}_{\mathcal{I}}$ needs to be a linear arithmetical theory containing $+$, $-$, $\leq$, and $<$ (e.g., linear integer arithmetic or bit-vectors). Intuitively, $\mathsf{set}(a, p, v, s)$ denotes the array obtained from a by setting the entries in the range $[p, p+s)$ to v and $\mathsf{set}_\infty(a, p, v)$ denotes the array obtained from a by setting all entries starting from p to v. Furthermore, $\mathsf{copy}(a, p, b, q, s)$ denotes the array obtained from a by setting the entries in the range $[p, p+s)$ to the values contained in b in the range $[q, q+s)$ and $\mathsf{copy}_\infty(a, p, b, q)$ denotes the array obtained from a by setting the entries starting from p to the values contained in b starting from q. Formally, the semantics of the operations is given by the following axioms:[4]

$$\begin{aligned}
p \leq r < p+s \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{set}(a,p,v,s),r) = v \\
\neg(p \leq r < p+s) \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{set}(a,p,v,s),r) = \mathsf{read}(a,r) \\
r \geq p \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{set}_\infty(a,p,v),r) = v \\
\neg(r \geq p) \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{set}_\infty(a,p,v),r) = \mathsf{read}(a,r) \\
p \leq r < p+s \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{copy}(a,p,b,q,s),r) = \mathsf{read}(b,q+(r-p)) \\
\neg(p \leq r < p+s) \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{copy}(a,p,b,q,s),r) = \mathsf{read}(a,r) \\
r \geq p \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{copy}_\infty(a,p,b,q),r) = \mathsf{read}(b,q+(r-p)) \\
\neg(r \geq p) \quad &\Longrightarrow \quad \mathsf{read}(\mathsf{copy}_\infty(a,p,b,q),r) = \mathsf{read}(a,r)
\end{aligned}$$

Now it is easy to see that $\mathcal{T}_{\mathcal{ASC}}$ can be simulated within $\mathcal{T}_{\lambda\mathcal{A}}$:

$$\begin{aligned}
\mathsf{set}(a,p,v,s) \quad &\hookrightarrow \quad \lambda i.\ \mathsf{ITE}(p \leq i < p+s,\ v,\ \mathsf{read}(a,i)) \\
\mathsf{set}_\infty(a,p,v) \quad &\hookrightarrow \quad \lambda i.\ \mathsf{ITE}(i \geq p,\ v,\ \mathsf{read}(a,i)) \\
\mathsf{copy}(a,p,b,q,s) \quad &\hookrightarrow \quad \lambda i.\ \mathsf{ITE}(p \leq i < p+s,\ \mathsf{read}(b,q+(i-p)),\ \mathsf{read}(a,i)) \\
\mathsf{copy}_\infty(a,p,b,q) \quad &\hookrightarrow \quad \lambda i.\ \mathsf{ITE}(i \geq p,\ \mathsf{read}(b,q+(i-p)),\ \mathsf{read}(a,i))
\end{aligned}$$

4 Applications of $\mathcal{T}_{\lambda\mathcal{A}}$

As already noted in [14], the operations set and copy, and therefore also $\mathcal{T}_{\lambda\mathcal{A}}$'s λ-terms, can be used to model the C standard library functions `memset` and `memcpy`. Intuitively, this is done by summarizing the loops that implement these functions, thereby modelling a simultaneous execution of all loop iterations.[5]

But the theory $\mathcal{T}_{\lambda\mathcal{A}}$ goes beyond what is possible with $\mathcal{T}_{\mathcal{ASC}}$ in that it can be used to summarize a wider range of loops than the particular loops in those specific library functions.

4.1 Loop Summarization Using $\mathcal{T}_{\lambda\mathcal{A}}$

Broadly speaking, $\mathcal{T}_{\lambda\mathcal{A}}$ can be used to summarize loops with data independent loop iterations where consecutive loop iterations only write to consecutive array positions. More precisely, loops need to satisfy the following requirements:

[4] Similar formulas could be used as postconditions for `memset` and `memcpy` in deductive verification tools such as `VCC` [10] and `Frama-C` [12].

[5] Because of this, copy's semantics is actually closer to `memmove` than to `memcpy`.

– The loop does not contain nested loops.
– The induction variable is incremented by one in each iteration.
– For an array a declared outside the loop, each iteration of the loop unconditionally modifies only the ith element of a, where i is the induction variable.
– All other variables declared outside the loop are not modified by the loop.
– Any iteration of the loop may not use elements of a that have been modified in earlier iterations of the loop.

In many cases, these requirements can be fulfilled by applying code transformations that are similar to standard compiler optimizations.

Example 2. Consider the following program fragment implementing part of the Sieve of Eratosthenes:

```
1  void filter_multiples(int p, int n)
2  {
3      for (int j = p*p; j <= n; j += p) {
4          a[j] = 0;
5      }
6  }
```

The loop can easily be transformed into functionally equivalent code that increments the induction variable by one, thereby making it λ-summarizable:

```
1  void filter_multiples(int p, int n)
2  {
3      for (int j = p*p; j <= n; ++j) {
4          a[j] = ((j - p*p) % p == 0 ? 0 : a[j]);
5      }
6  }
```

Note that such transformations can be performed automatically. ◊

4.2 Further Uses

While this is already useful by itself, applicability of $\mathcal{T}_{\lambda\mathcal{A}}$ goes beyond summarization of loops and calls to `memset` and `memcpy`. Some applications that we would like to explore in future work include the following:

– Zero-initialization of global variables (as required by the C standard) can be achieved using a λ-term corresponding to a set operation.
– Zero-initialization of new memory pages before the operating system hands them to a process can similarly be modelled using a λ-term.
– If certain memory locations should be set to unconstrained values (*havocked*), then this can be done using a λ-term $\lambda i.\ \mathsf{ITE}(\psi,\ \mathsf{read}(h, i),\ \mathsf{read}(a, i))$, where ψ describes the memory locations that are to be havocked and h is a fresh array constant. Similarly, memory-mapped I/O can be modelled.

– 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 λ-terms corresponding to a set operation. This makes it possible to develop an alternative to the SMT theory of memory allocation presented in [13] and to the memory model presented in [28].

5 Deciding $\mathcal{T}_{\lambda\mathcal{A}}$

In this section, we discuss several possibilities for deciding whether a quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula is satisfiable. All approaches work by a reduction to a theory that is already supported by current SMT solvers.

In order to ease presentation, it is advantageous to represent formulas in *flattened form* (similar to [27]). For this, a formula φ is represented using a pair $(\Delta_\varphi, c_\varphi)$, where Δ_φ is a list of definitions of the form

$$\begin{array}{lll} v \equiv f(v_1, \ldots, v_n) & c \equiv P(v_1, \ldots, v_n) & \\ v \equiv \lambda i.\ s & c \equiv c_1 \star c_2 & \text{for } \star \in \{\wedge, \vee\} \\ v \equiv \mathsf{ITE}(c,\ v_1,\ v_2) & c \equiv \neg c_1 & \end{array}$$

and c_φ is one of the c's denoting the root proposition of the formula. Here, f is a function symbol, P is a predicate symbol, $v, v_1, \ldots, v_n, s$ are constants, c, c_1, c_2 are propositions, and each v and c is defined *before* it is used (we assume in the following that adding definitions to a formula ensures that this property is preserved). Constants occurring in the left-hand side of a definition need to be fresh, uninterpreted constants. Thus, the v's and c's should be seen as *names* for terms and formulas, respectively. We use $v \prec w$ to denote that the definition of w uses v (def-use-relation) and $v \prec_{\mathcal{A}} w$ to denote that $v \prec w$ and v is of sort $\sigma_{\mathcal{A}}$. The transitive closures of $\prec$ and $\prec_{\mathcal{A}}$ are denoted $\prec^+$ and $\prec_{\mathcal{A}}^+$, respectively, and $\prec^*$ denotes the reflexive-transitive closure of $\prec$. Note that a definition for v such that $v \not\prec^* c_\varphi$ can be deleted from Δ_φ (*clean-up*) without affecting the satisfiability status of the formula.

Example 3. Consider the following $\mathcal{T}_{\mathcal{ASC}}$ formula (with $\mathcal{T}_{\mathcal{I}} = \mathcal{T}_{\mathcal{E}} = \mathcal{T}_{\mathcal{LIA}}$):

$$\begin{gathered} \mathsf{read}(\mathsf{write}(\mathsf{copy}_\infty(\mathsf{copy}_\infty(a,0,a,1),1,\mathsf{copy}_\infty(a,0,a,1),0),0,\mathsf{read}(a,0)),k) \\ \neq \\ \mathsf{read}(a,k) \end{gathered}$$

This formula states that the array obtained from a by first moving all array elements at indices ≥ 1 down by one positions, then all elements at indices ≥ 0 up by one positions, and afterwards replacing the element at index 0 by the original element $\mathsf{read}(a,0)$ differs at index k from the initial array a. This formula is clearly unsatisfiable.

The formula can be converted into the following $\mathcal{T}_{\lambda\mathcal{A}}$ formula φ:

$$\begin{aligned}&\mathsf{read}(\mathsf{write}(\lambda j.\ \mathsf{ITE}(j \geq 1,\ \mathsf{read}(\lambda i.\ \mathsf{ITE}(i \geq 0,\ \mathsf{read}(a, i+1),\ \mathsf{read}(a, i)), j-1),\\ &\qquad\qquad\qquad\qquad \mathsf{read}(\lambda i.\ \mathsf{ITE}(i \geq 0,\ \mathsf{read}(a, i+1),\ \mathsf{read}(a, i)), j)),\\ &\qquad 0, \mathsf{read}(a, 0)),\\ &\quad k) \qquad\qquad\qquad \neq\\ &\qquad\qquad\qquad\qquad \mathsf{read}(a, k)\end{aligned}$$

The flattened form is then given as $(\Delta_\varphi, c_\varphi)$ where Δ_φ contains

$v_1 \equiv \mathsf{read}(a, 0)$	$a_1 \equiv \lambda i.\ s_1$	$a_2 \equiv \lambda j.\ s_2$
$c_1 \equiv i \geq 0$	$c_2 \equiv j \geq 1$	$a_3 \equiv \mathsf{write}(a_2, 0, v_1)$
$v_2 \equiv i + 1$	$v_5 \equiv j - 1$	$v_8 \equiv \mathsf{read}(a, k)$
$v_3 \equiv \mathsf{read}(a, v_2)$	$v_6 \equiv \mathsf{read}(a_1, v_5)$	$v_9 \equiv \mathsf{read}(a_3, k)$
$v_4 \equiv \mathsf{read}(a, i)$	$v_7 \equiv \mathsf{read}(a_1, j)$	$c_3 \equiv v_8 \neq v_9$
$s_1 \equiv \mathsf{ITE}(c_1,\ v_3,\ v_4)$	$s_2 \equiv \mathsf{ITE}(c_2,\ v_6,\ v_7)$	

and $c_\varphi = c_3$. Note that the subterm $\lambda i.\ \mathsf{ITE}(i \geq 0,\ \mathsf{read}(a, i+1),\ \mathsf{read}(a, i))$ is shared in the flattened form. ◊

5.1 Eager Reduction

The first reduction reduces satisfiability of a quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula to satisfiability of a quantifier-free $\mathcal{T}_\mathcal{I} \oplus \mathcal{T}_\mathcal{E} \oplus \mathcal{T}_{\mathcal{EUF}}$ formula. This reduction is based on exhaustively applying the read-over-write and read-over-λ axioms in order to eliminate all array terms except for constants. Note that this reduction establishes decidability of satisfiability for quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formulas in the case where satisfiability of quantifier-free $\mathcal{T}_\mathcal{I} \oplus \mathcal{T}_\mathcal{E} \oplus \mathcal{T}_{\mathcal{EUF}}$ formulas is decidable.

Theorem 1. *Each quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula φ can effectively be converted into an equisatisfiable quantifier-free $\mathcal{T}_\mathcal{I} \oplus \mathcal{T}_\mathcal{E} \oplus \mathcal{T}_{\mathcal{EUF}}$ formula φ'.*

Proof. The reduction is similar to the reduction from $\mathcal{T}_\mathcal{A}$ to $\mathcal{T}_\mathcal{I} \oplus \mathcal{T}_\mathcal{E} \oplus \mathcal{T}_{\mathcal{EUF}}$ described in Sect. 3, i.e., the read-over-write axioms (1) and (2) and the read-over-λ axiom scheme (3) are applied exhaustively as rewrite rules using the innermost strategy.[6] Thus, if Δ_φ contains definitions $a_k \equiv \mathsf{write}(a_l, p_l, v_l)$ and $v_m \equiv \mathsf{read}(a_k, p_n)$, then v_m is replaced by v'_m with the definition $v'_m \equiv \mathsf{ITE}(c,\ v_l,\ v)$, where the new definitions $c \equiv p_l = p_n$ and $v \equiv \mathsf{read}(a_l, p_n)$ are added as well. Similarly, if Δ_φ contains definitions $a_k \equiv \lambda i.\ s$ and $v_m \equiv \mathsf{read}(a_k, p_n)$, then v_m is replaced by v'_m, where v'_m names the flattened form of $s[i/p_n]$ and all definitions needed for this flattened form are added as well.

[6] The innermost reduction strategy is obeyed if the list of definitions is processed from front to back and new definitions are added after the definition they replace.

In order to show that this rewrite process is terminating, recursively define the function ρ by letting

$$\begin{aligned}
\rho(\mathsf{write}(a,p,v)) &= 1 + \rho(a) + \rho(p) + \rho(v) \\
\rho(f(v_1,\ldots,v_n)) &= \rho(v_1) + \ldots + \rho(v_n) && \text{if } f \neq \mathsf{write} \\
\rho(\lambda i.\ s) &= 1 + \rho(s) \\
\rho(\mathsf{ITE}(c,\ v_1,\ v_2)) &= \rho(c) + \rho(v_1) + \rho(v_2) \\
\rho(P(v_1,\ldots,v_n) &= \rho(v_1) + \ldots + \rho(v_n) \\
\rho(c_1 \star c_2) &= \rho(c_1) + \rho(c_2) && \text{for } \star \in \{\wedge, \vee\} \\
\rho(\neg c_1) &= \rho(c_1)
\end{aligned}$$

Thus, ρ counts occurrences of write and λ, where multiple uses of the same definitions are counted multiple times. Since the rewriting process is triggered by read-definitions, it now suffices to show that each transformation step replaces a definition of the form $v_m \equiv \mathsf{read}(a_k, p_k)$ by finitely many new definitions of the form $v' \equiv \mathsf{read}(a', p')$ with $\rho(v_m) > \rho(v')$ and does not increase the ρ-number of any remaining read-definition. For both of these properties, it is sufficient to show that $\rho(v_m) > \rho(v'_m)$ when v_m is replaced by v'_m.

In the first case, $a_k \equiv \mathsf{write}(a_l, p_l, v_l) \in \Delta_\varphi$ and the new definition $v \equiv \mathsf{read}(a_l, p_n)$ is introduced. First, note that $\rho(p_l) = \rho(v_l) = \rho(p_n) = \rho(c) = 0$ due to the innermost reduction strategy since the rewrite rules suffice to eliminate all occurrences of write and λ in terms of sort $\sigma_{\mathcal{I}}$ or $\sigma_{\mathcal{E}}$ (and thus also in propositions since $\mathcal{T}_{\lambda\mathcal{A}}$ is non-extensional). Then the desired $\rho(v_m) > \rho(v'_m)$ easily follows since $\rho(v_m) = \rho(a_k) = 1 + \rho(a_l) > \rho(a_l) = \rho(v'_m)$.

In the second case, $a_k \equiv \lambda i.\ s \in \Delta_\varphi$ and new read-definitions are only introduced in the construction of $s[i/p_n]$. As in the first case, $\rho(p_n) = 0$ due to the innermost reduction strategy. Thus, $\rho(v'_m) = \rho(s)$ and therefore $\rho(v_m) = \rho(a_k) = 1 + \rho(s) > \rho(s) = \rho(v'_m)$.

After exhaustive application of the rewrite rules, a clean-up produces a quantifier-free $\mathcal{T}_{\mathcal{I}} \oplus \mathcal{T}_{\mathcal{E}} \oplus \mathcal{T}_{\mathcal{EUF}}$ formula φ'. Equisatisfiability of φ and φ' follows since the conversion only applies axioms of $\mathcal{T}_{\lambda\mathcal{A}}$. □

Example 4. Continuing Ex. 3, the definition of v_9 is first replaced, by an application of the read-over-write axioms (1) and (2), by the definitions

$$c_7 \equiv k = 0 \qquad v_{18} \equiv \mathsf{read}(a_2, k) \qquad v'_9 \equiv \mathsf{ITE}(c_7,\ v_1,\ v_{18})$$

Then, the definition of v_{18} is replaced, by an application of the read-over-λ axiom scheme (3), by the definitions

$$\begin{aligned}
&c_6 \equiv k \geq 1 && v_{14} \equiv \mathsf{read}(a_1, v_{10}) && v'_{18} \equiv \mathsf{ITE}(c_6,\ v_{14},\ v_{17}) \\
&v_{10} \equiv k - 1 && v_{17} \equiv \mathsf{read}(a_1, k)
\end{aligned}$$

obtained from $c_2, v_5, v_6, v_7,$ and s_2 when constructing $s_2[j/k]$.

Next, the definitions of v_{14} and v_{17} are replaced, again by applications of the read-over-λ axiom scheme (3), by the definitions

$$
\begin{array}{lll}
c_4 \equiv v_{10} \geq 0 & v_{13} \equiv \mathsf{read}(a, v_{10}) & v_{15} \equiv k+1 \\
v_{11} \equiv v_{10}+1 & v'_{14} \equiv \mathsf{ITE}(c_4,\ v_{12},\ v_{13}) & v_{16} \equiv \mathsf{read}(a, v_{15}) \\
v_{12} \equiv \mathsf{read}(a, v_{11}) & c_5 \equiv k \geq 0 & v'_{17} \equiv \mathsf{ITE}(c_5,\ v_{16},\ v_8)
\end{array}
$$

obtained when constructing $s_1[i/v_{10}]$ and $s_1[i/k]$.

Similar replacements take place for v_7 and v_6. After a clean-up, the following definitions remain:

$$
\begin{array}{lll}
v_1 \equiv \mathsf{read}(a, 0) & v_{13} \equiv \mathsf{read}(a, v_{10}) & c_6 \equiv k \geq 1 \\
v_8 \equiv \mathsf{read}(a, k) & v'_{14} \equiv \mathsf{ITE}(c_4,\ v_{12},\ v_{13}) & v'_{18} \equiv \mathsf{ITE}(c_6,\ v'_{14},\ v'_{17}) \\
v_{10} \equiv k-1 & c_5 \equiv k \geq 0 & c_7 \equiv k = 0 \\
c_4 \equiv v_{10} \geq 0 & v_{15} \equiv k+1 & v'_9 \equiv \mathsf{ITE}(c_7,\ v_1,\ v'_{18}) \\
v_{11} \equiv v_{10}+1 & v_{16} \equiv \mathsf{read}(a, v_{15}) & c_3 \equiv v_8 \neq v'_9 \\
v_{12} \equiv \mathsf{read}(a, v_{11}) & v'_{17} \equiv \mathsf{ITE}(c_5,\ v_{16},\ v_8) &
\end{array}
$$

Unsatisfiability of this formula can easily be established using an SMT solver for $\mathcal{T_I} \oplus \mathcal{T_E} \oplus \mathcal{T_{EUF}}$. ♢

5.2 Using Quantifiers

The next approach reduces satisfiability of a quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula to satisfiability of a $\mathcal{T_A}$ formula containing quantifiers that range over the sort $\sigma_\mathcal{I}$. The idea for the reduction is to replace a λ-term $\lambda i.\ s$ by a constant a_k while adding the constraint $\forall i.\ \mathsf{read}(a_k, i) = s$ that restricts the interpretation of this constant to agree with the λ-term for all indices. Note that due to the introduced quantifiers, this reduction *does not* establish decidability of satisfiability for quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formulas even if satisfiability of quantifier-free $\mathcal{T_A}$ formulas is decidable. It is, however, illustrative for the approach in Sect. 5.3, which can be seen as a complete instantiation strategy for the introduced quantifiers.

First, the representation of formulas is extended to quantifiers by admitting definitions of the form $c \equiv \forall i.\ c_1$ for universal quantification (existential quantification could also be admitted, but this is not needed for our reduction).

Theorem 2. *Each quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula φ can effectively be converted into an equisatisfiable $\mathcal{T_A}$ formula φ′ containing universal quantifiers that range over the sort $\sigma_\mathcal{I}$.*

Proof. The reduction proceeds by repeating the following step: Let (Δ_n, c_n) be the formula in the nth iteration (i.e., $(\Delta_1, c_1) = (\Delta_\varphi, c_\varphi)$). If Δ_n contains a definition of the form $a_k \equiv \lambda i.\ s$, then this definition is deleted from Δ_n (turning a_k into an uninterpreted constant) and the definitions $v_{a_k} \equiv \mathsf{read}(a_k, i)$, $c_{a_k} \equiv v_{a_k} = s$, and $c_{\forall_{a_k}} \equiv \forall i.\ c_{a_k}$ are added instead. Furthermore, add $c_{n+1} \equiv c_n \wedge c_{\forall_{a_k}}$, resulting in the formula (Δ_{n+1}, c_{n+1}) for the next iteration. Since φ contains only finitely many λ-terms and no new λ-terms are introduced in the reduction, this process eventually terminates. Furthermore, equisatisfiability of (Δ_n, c_n) and

(Δ_{n+1}, c_{n+1}) is easily seen for all $n \geq 1$ due to the restriction on the De Bruijn indices of the occurrences of i in s. □

Example 5. Continuing Ex. 3, the reduction to a quantified $\mathcal{T}_\mathcal{A}$ formula produces the following definitions:

$$
\begin{aligned}
v_1 &\equiv \mathsf{read}(a, 0) & c_{\forall_{a_1}} &\equiv \forall i.\ c_{a_1} & c_{\forall_{a_2}} &\equiv \forall j.\ c_{a_2} \\
c_1 &\equiv i \geq 0 & c_2 &\equiv j \geq 1 & a_3 &\equiv \mathsf{write}(a_2, 0, v_1) \\
v_2 &\equiv i + 1 & v_5 &\equiv j - 1 & v_8 &\equiv \mathsf{read}(a, k) \\
v_3 &\equiv \mathsf{read}(a, v_2) & v_6 &\equiv \mathsf{read}(a_1, v_5) & v_9 &\equiv \mathsf{read}(a_3, k) \\
v_4 &\equiv \mathsf{read}(a, i) & v_7 &\equiv \mathsf{read}(a_1, j) & c_3 &\equiv v_8 \neq v_9 \\
s_1 &\equiv \mathsf{ITE}(c_1,\ v_3,\ v_4) & s_2 &\equiv \mathsf{ITE}(c_2,\ v_6,\ v_7) & c_4 &\equiv c_3 \wedge c_{\forall_{a_1}} \\
v_{a_1} &\equiv \mathsf{read}(a_1, i) & v_{a_2} &\equiv \mathsf{read}(a_2, j) & c_5 &\equiv c_4 \wedge c_{\forall_{a_2}} \\
c_{a_1} &\equiv v_{a_1} = s_1 & c_{a_2} &\equiv v_{a_2} = s_2 & &
\end{aligned}
$$

The resulting formula (with $c_\varphi = c_5$) is unsatisfiable, as can be shown, e.g., using the SMT solvers `Z3` [26] or `CVC4` [2]. ♢

5.3 Instantiating Quantifiers

Since reasoning involving quantifiers is hard for current SMT solvers, the goal of this section is to develop a method that can be seen as a sound and complete instantiation strategy for the quantifiers introduced in Sect. 5.2. For this, the quantifier introduced for the constant a_k is instantiated for all indices that occur in read operations $v_l \equiv \mathsf{read}(a_j, p_j)$ such that $a_k \prec^+_\mathcal{A} v_l$. Intuitively, these instantiations are sufficient since the elements at indices that are never read from a_k are not relevant for the satisfiability status of the formula. Note that this reduction establishes decidability of satisfiability for such quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formulas in the case where satisfiability of quantifier-free $\mathcal{T}_\mathcal{A}$ formulas is decidable.

While the approach introduced in this section can be seen as an instantiation strategy for the quantifiers, it is conceptually simpler to state it independent of these quantifiers and give a direct reduction.

Theorem 3. *Each quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula φ can effectively be converted into an equisatisfiable quantifier-free $\mathcal{T}_\mathcal{A}$ formula φ'.*

Proof. The reduction proceeds by repeating the following step: Let (Δ_n, c_n) be the formula in the nth iteration (i.e., $(\Delta_1, c_1) = (\Delta_\varphi, c_\varphi)$). If Δ_n contains a definition of the form $a_k \equiv \lambda i.\ s$ such that $a_k \not\prec^+_\mathcal{A} a_l$ for all $a_l \equiv \lambda i'.\ s'$ (the last definition of a λ-term in the list of definitions satisfies this requirement), then let $P = \{p_1, \ldots, p_n\}$ denote the set of all read indices occurring in a definition $v_l \equiv \mathsf{read}(a_j, p_j)$ with $a_k \prec^+_\mathcal{A} v_l$ (see Ex. 7 below for an explanation why it does *not* suffice to only consider definitions of the form $v_l \equiv \mathsf{read}(a_k, p_j)$). For the transformation, the definition of a_k is deleted from Δ_n (turning a_k into an uninterpreted constant) and the definitions $v_{p_j} \equiv \mathsf{read}(a_k, p_j)$ and $c_{p_j} \equiv v_{p_j} = s_{p_j}$

are added for all $p_j \in P$. Here, s_{p_j} names the flattened form of $s[i/p_j]$ and all definitions needed for this flattened form are added as well. Furthermore, add the definition $c_{n+1} \equiv c_n \wedge c$, where c names the flattened form of $c_{p_1} \wedge \ldots \wedge c_{p_n}$ and all definitions needed for this flattened form are added as well. Finally, a clean-up is performed. The resulting formula for the next iteration is then (Δ_{n+1}, c_{n+1}).

This transformation process eventually terminates since φ contains only finitely many λ-terms and flattening $s[i/p_j]$ does not introduce any new λ-terms due to the restriction on the De Bruijn indices of the occurrences of i (this assumes that the construction of the flattened form of $s[i/p_j]$ maximally shares common definitions with s).

Equisatisfiability of (Δ_n, c_n) and (Δ_{n+1}, c_{n+1}) for all all $n \geq 1$ is shown as follows. First, assume that (Δ_n, c_n) is satisfiable and let $\mathfrak{M}$ be a model of this formula. Let $\mathfrak{M}'$ be obtained from $\mathfrak{M}$ by adding the interpretation $\mathfrak{M}'(a_k) = [\![a_k]\!]^{\mathfrak{M}}$. Then $\mathfrak{M}'$ is a model of (Δ_{n+1}, c_{n+1}) since the read-over-λ axiom (3) implies that $[\![\mathsf{read}(a_k, p_j)]\!]^{\mathfrak{M}'} = [\![s[i/p_j]]\!]^{\mathfrak{M}'}$ for all $p_j \in P$.

For the reverse direction, assume that (Δ_{n+1}, c_{n+1}) is satisfiable and consider the assignment $\mathfrak{M}(a_k)$ in a model $\mathfrak{M}$ of this formula (note that a_k is an uninterpreted constant in (Δ_{n+1}, c_{n+1})). Let $\mathfrak{M}'$ be the structure obtained from $\mathfrak{M}$ by "forgetting" the assignment $\mathfrak{M}(a_k)$. Then $\mathfrak{M}'$ is a model of (Δ_n, c_n) since an easy induction on the position in the list Δ_n shows that

1. $[\![v]\!]^{\mathfrak{M}'} = [\![v]\!]^{\mathfrak{M}}$ for all definitions $v \equiv \ldots \in \Delta_n$ of sort $\sigma_{\mathcal{I}}$ or $\sigma_{\mathcal{E}}$,
2. $[\![a]\!]^{\mathfrak{M}'} = [\![a]\!]^{\mathfrak{M}}$ for all definitions $a \equiv \ldots \in \Delta_n$ of sort $\sigma_{\mathcal{A}}$ with $a_k \not\prec^*_{\mathcal{A}} a$,
3. $[\![\mathsf{read}(a, p)]\!]^{\mathfrak{M}'} = [\![\mathsf{read}(a, p)]\!]^{\mathfrak{M}}$ for all definitions $a \equiv \ldots \in \Delta_n$ of sort $\sigma_{\mathcal{A}}$ with $a_k \prec^*_{\mathcal{A}} a$ and all $p \in P$, and
4. $[\![c]\!]^{\mathfrak{M}'} = [\![c]\!]^{\mathfrak{M}}$ for all definitions of propositions $c \equiv \ldots \in \Delta_n$.

The only non-trivial case in the induction is showing the third statement, but this is ensured by the instantiations that are added as definitions in the construction of Δ_{n+1}. □

Example 6. Continuing Ex. 3, the reduction to a quantifier-free $\mathcal{T}_{\mathcal{A}}$ proceeds as follows. First, the definition of a_2 is "forgotten". The set of read indices used for instantiation is $P_{a_2} = \{k\}$ (from the definition $v_9 \equiv \mathsf{read}(a_3, k)$). Thus, the following definitions are added to the formula:

$$
\begin{array}{lll}
v^k_{a_2} \equiv \mathsf{read}(a_2, k) & v^k_6 \equiv \mathsf{read}(a_1, v^k_5) & c^k_{a_2} \equiv v^k_{a_2} = s^k_2 \\
c^k_2 \equiv k \geq 1 & v^k_7 \equiv \mathsf{read}(a_1, k) & c_4 \equiv c_3 \wedge c^k_{a_2} \\
v^k_5 \equiv k - 1 & s^k_2 \equiv \mathsf{ITE}(c^k_2,\ v^k_6,\ v^k_7) &
\end{array}
$$

Here, definitions with superscript "^{k}" are obtained from the definitions with the same name when constructing $s_2[j/k]$. The subsequent clean-up removes the definitions of s_2, v_7, v_6, v_5, and c_2. Finally, c_φ is updated to be c_4.

Next, the definition of a_1 is "forgotten". The set of read indices for a_1 is $P_{a_1} = \{v^k_5, k\}$ (from the definitions $v^k_6 \equiv \mathsf{read}(a_1, v^k_5)$ and $v^k_7 \equiv \mathsf{read}(a_1, k)$). Thus, the following definitions are added to the formula:

$$
\begin{array}{lll}
v_{a_1}^k \equiv \mathsf{read}(a_1, k) & c_{a_1}^k \equiv v_{a_1}^k = s_1^k & v_3^{v_5^k} \equiv \mathsf{read}(a, v_2^{v_5^k}) \\
c_1^k \equiv k \geq 0 & c_5 \equiv c_4 \wedge c_{a_1}^k & v_4^{v_5^k} \equiv \mathsf{read}(a, k) \\
v_2^k \equiv k + 1 & v_{a_1}^{v_5^k} \equiv \mathsf{read}(a_1, v_5^k) & s_1^{v_5^k} \equiv \mathsf{ITE}(c_1^{v_5^k},\ v_3^{v_5^k},\ v_4^{v_5^k}) \\
v_3^k \equiv \mathsf{read}(a, v_2^k) & c_1^{v_5^k} \equiv v_5^k \geq 0 & c_{a_1}^{v_5^k} \equiv v_{a_1}^{v_5^k} = s_1^{v_5^k} \\
v_4^k \equiv \mathsf{read}(a, k) & v_2^{v_5^k} \equiv v_5^k + 1 & c_6 \equiv c_5 \wedge c_{a_1}^{v_5^k} \\
s_1^k \equiv \mathsf{ITE}(c_1^k,\ v_3^k,\ v_4^k) & &
\end{array}
$$

After performing a clean-up, the formula contains the definitions

$$
\begin{array}{lll}
v_1 \equiv \mathsf{read}(a, 0) & s_2^k \equiv \mathsf{ITE}(c_2^k,\ v_6^k,\ v_7^k) & c_5 \equiv c_4 \wedge c_{a_1}^k \\
a_3 \equiv \mathsf{write}(a_2, 0, v_1) & c_{a_2}^k \equiv v_{a_2}^k = s_2^k & v_{a_1}^{v_5^k} \equiv \mathsf{read}(a_1, v_5^k) \\
v_8 \equiv \mathsf{read}(a, k) & c_4 \equiv c_3 \wedge c_{a_2}^k & c_1^{v_5^k} \equiv v_5^k \geq 0 \\
v_9 \equiv \mathsf{read}(a_3, k) & v_{a_1}^k \equiv \mathsf{read}(a_1, k) & v_2^{v_5^k} \equiv v_5^k + 1 \\
c_3 \equiv v_8 \neq v_9 & c_1^k \equiv k \geq 0 & v_3^{v_5^k} \equiv \mathsf{read}(a, v_2^{v_5^k}) \\
v_{a_2}^k \equiv \mathsf{read}(a_2, k) & v_2^k \equiv k + 1 & v_4^{v_5^k} \equiv \mathsf{read}(a, k) \\
c_2^k \equiv k \geq 1 & v_3^k \equiv \mathsf{read}(a, v_2^k) & s_1^{v_5^k} \equiv \mathsf{ITE}(c_1^{v_5^k},\ v_3^{v_5^k},\ v_4^{v_5^k}) \\
v_5^k \equiv k - 1 & v_4^k \equiv \mathsf{read}(a, k) & c_{a_1}^{v_5^k} \equiv v_{a_1}^{v_5^k} = s_1^{v_5^k} \\
v_6^k \equiv \mathsf{read}(a_1, v_5^k) & s_1^k \equiv \mathsf{ITE}(c_1^k,\ v_3^k,\ v_4^k) & c_6 \equiv c_5 \wedge c_{a_1}^{v_5^k} \\
v_7^k \equiv \mathsf{read}(a_1, k) & c_{a_1}^k \equiv v_{a_1}^k = s_1^k &
\end{array}
$$

and $c_\varphi = c_6$. Unsatisfiability of the formula can easily be shown using SMT solvers for $\mathcal{T}_\mathcal{A}$. ◊

The following example shows why it is necessary to add instantiations for all $v_l \equiv \mathsf{read}(a_j, p_j)$ with $a_k \prec_\mathcal{A}^+ v_l$ instead of restricting attention to those $v_l \equiv \mathsf{read}(a_j, p_j)$ with $a_j = a_k$.

Example 7. Consider the $\mathcal{T}_{\lambda\mathcal{A}}$ formula $(\Delta_\varphi, c_\varphi)$ with $\mathcal{T}_\mathcal{I} = \mathcal{T}_\mathcal{E} = \mathcal{T}_{\mathcal{LIA}}$ where Δ_φ contains the definitions

$$
\begin{array}{lll}
a_1 \equiv \lambda i.\ 0 & v_1 \equiv \mathsf{read}(a_2, k) & c_2 \equiv k \neq 0 \\
a_2 \equiv \mathsf{write}(a_1, 0, 1) & c_1 \equiv v_1 \neq 0 & c_3 \equiv c_1 \wedge c_2
\end{array}
$$

and $c_\varphi = c_3$. Then this formula is clearly unsatisfiable.

If only definitions $v_l \equiv \mathsf{read}(a_j, p_j)$ with $a_j = a_k$ are considered when eliminating the definition of a_1, then no instantiations are added at all and

$$
\begin{array}{lll}
a_2 \equiv \mathsf{write}(a_1, 0, 1) & c_1 \equiv v_1 \neq 0 & c_3 \equiv c_1 \wedge c_2 \\
v_1 \equiv \mathsf{read}(a_2, k) & c_2 \equiv k \neq 0 &
\end{array}
$$

remain. This formula is satisfiable (e.g., if a_1 contains 1 for all indices).

Using all $v_l \equiv \mathsf{read}(a_j, p_j)$ with $a_k \prec^+_{\mathcal{A}} a_j$ as done in the proof of Thm. 3 adds an instantiation for k and the definitions

$$
\begin{array}{lll}
v^k_{a_1} \equiv \mathsf{read}(a_1, k) & v_1 \equiv \mathsf{read}(a_2, k) & c_3 \equiv c_1 \wedge c_2 \\
c^k_{a_1} \equiv v^k_{a_1} = 0 & c_1 \equiv v_1 \neq 0 & c_4 \equiv c_4 \wedge c^k_{a_1} \\
a_2 \equiv \mathsf{write}(a_1, 0, 1) & c_2 \equiv k \neq 0 &
\end{array}
$$

are obtained. Furthermore, c_φ is updated to c_4. As desired, the resulting formula is unsatisfiable. ♢

6 Implementation and Evaluation

We have conducted experiments with all reductions described in Sect. 5 for determining the satisfiability of quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$. Since our motivation was the application in the bounded model checking tool LLBMC [24], we have restricted attention to the case where $\mathcal{T}_{\mathcal{E}} = \mathcal{T}_{\mathcal{I}} = \mathcal{T}_{\mathcal{BV}}$ is the theory of bit-vectors.

6.1 Loop Summarization in LLBMC

The tool 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, LLBMC uses the LLVM compiler framework [21] in order to translate C and C++ programs into LLVM's intermediate representation (IR). This IR is then converted into a quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formula and simplified using an extensive set of rewrite rules. The simplified formula is finally passed to an SMT solver. Distinguishing features of LLBMC in comparison with related tools such as CBMC [9] and ESBMC [11] are its use of a flat, bit-precise memory model, its exhaustive set of built-in checks, and its performance (see [24]).

The use of the LLVM compiler framework proved itself very useful in implementing loop summarization in LLBMC, as LLVM provides passes for canonicalizing loops. Furthermore, information about a the start value, end value, and trip count of a loop's induction variable is available using LLVM' comprehensive scalar evolution analysis framework.

In our implementation, summarizable loops are transformed into λ-terms of the form $\lambda i.\ \mathsf{ITE}(g,\ s,\ r)$, where g is a guard indicating if a read at position i from the λ-term is in the summarized memory region or not, s is an encoding of the value stored in the summarized loop, and r is a read from position i of the memory state from before execution of the loop.

The implementation currently focuses on the most frequently found summarizable loops and is therefore restricted to loops with a single basic block[7], a single store instruction, and at most load instructions which are executed before the store instruction and access exactly the same memory location modified by the store instruction.

[7] This restriction can be easily relaxed in the future.

6.2 Evaluation

Within LLBMC, we have evaluated the following approaches for determining satisfiability of quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formulas:

1. The eager reduction to $\mathcal{T}_{\mathcal{I}} \oplus \mathcal{T}_{\mathcal{E}} \oplus \mathcal{T}_{\mathcal{EUF}}$ from Sect. 5.1 and the instantiation-based reduction to $\mathcal{T}_{\mathcal{A}}$ from Sect. 5.3 have been evaluated in combination with the SMT solvers STP [15] (SVN revision 1673), Boolector [4] (version 1.5.118), Z3 [26] (version 4.3.1), and CVC4 [2] (version 1.1). Here, the SMT solvers were executed using their C resp. C++ (CVC4) API.
2. The quantifier-based reduction to $\mathcal{T}_{\mathcal{A}}$ from Sect. 5.2 has been evaluated in combination with the SMT solvers Z3 and CVC4. Note that STP and Boolector do not support quantifiers. Since, according to its authors, the array solver in Z3 is optimized for quantifier-free problems [25], we have also evaluated an approach where arrays are encoded using uninterpreted functions and quantifiers (as suggested in [25]).
3. Loops that can be summarized using λ-terms can alternatively be treated like any other loop. Consequently, the boundedness restriction inherent to bounded model checking then applies. This approach was again evaluated in combination with STP, Boolector, Z3, and CVC4.

These approaches have been evaluated on a collection of 81 C and C++ programs. A total of 67 of these programs contain λ-terms corresponding to set or copy operations, where 55 programs were already used in the preliminary version of this work [14]. The set and copy operations in these programs may occur due to several reasons:

- The source code contains an explicit call to memset or memcpy.
- Library-specific implementation details included through header files may result in calls to memset or memcpy. This is in particular true for C++ programs that use the container classes of the STL.
- Default implementations of C++ constructors, especially the copy constructor, may make use of memcpy operations to initialize objects.

The remaining 14 programs contain loops that can be summarized using λ-terms as described in Sect. 4.1. Out of the 81 programs, 20 programs contain a bug and produce a satisfiable $\mathcal{T}_{\lambda\mathcal{A}}$ formula. The remaining 61 programs produce unsatisfiable $\mathcal{T}_{\lambda\mathcal{A}}$ formulas. The formulas that are produced for the different approaches are available in SMT-LIB v2 format at http://llbmc.org/.

The results of LLBMC on the collection of examples are summarized in Table 1. The reported times are in seconds and contain the time needed for the logical encoding into a $\mathcal{T}_{\lambda\mathcal{A}}$ formula, simplification of the formula, the time needed for the reductions, and the time needed by the SMT solver. A timeout of 60 seconds was imposed for each program and the experiments were performed on an Intel® Core™ 2 Duo 2.4GHz with 4GB of RAM.

The results indicate that the instantiation-based reduction achieves the best performance, regardless of the SMT solver that is used (but in particular in combination with STP). This can also be observed in the cactus plots displayed

		Satisfiable					Unsatisfiable					All				
SMT solver	Approach	Total Time	S	T	M	A	Total Time	S	T	M	A	Total Time	S	T	M	A
`STP`	Instantiation	9.908	20	–	–	–	196.126	60	1	–	–	206.034	80	1	–	–
`STP`	Eager	182.084	17	3	–	–	597.460	53	8	–	–	779.544	70	11	–	–
`STP`	Loops	114.663	17	1	2	–	555.863	53	5	3	–	670.526	70	6	5	–
`Boolector`	Instantiation	112.886	19	1	–	–	705.896	52	9	–	–	818.782	71	10	–	–
`Boolector`	Eager	189.760	17	3	–	–	796.991	53	8	–	–	986.751	70	11	–	–
`Boolector`	Loops	203.644	16	2	2	–	935.839	45	13	3	–	1139.483	61	15	5	–
`Z3`	Instantiation	126.948	18	2	–	–	821.417	49	11	1	–	948.365	67	13	1	–
`Z3`	Eager	185.436	17	3	–	–	858.196	49	12	–	–	1043.632	66	15	–	–
`Z3`	Quantifiers	288.719	16	4	–	–	833.770	49	12	–	–	1122.489	65	16	–	–
`Z3`	Quantifiers+UF	147.033	18	2	–	–	1127.661	45	16	–	–	1274.694	63	18	–	–
`Z3`	Loops	364.796	13	5	2	–	1254.787	40	18	3	–	1619.583	53	23	5	–
`CVC4`	Instantiation	196.661	17	3	–	–	731.418	50	11	–	–	928.079	67	14	–	–
`CVC4`	Eager	244.884	17	3	–	–	874.864	48	13	–	–	1119.748	65	16	–	–
`CVC4`	Quantifiers	432.676	7	7	–	6	974.442	47	14	–	–	1407.118	54	21	–	6
`CVC4`	Quantifiers+UF	452.506	7	7	–	6	1136.908	45	16	–	–	1589.414	52	23	–	6
`CVC4`	Loops	430.052	12	6	2	–	1122.646	44	13	4	–	1552.698	56	19	6	–

Table 1. Times and success rates for the different approaches on 81 benchmark problems using a timeout of 60 seconds. "S" denotes the number of solved benchmark problems, "T" denotes the number of timeouts, "M" denotes the number of times the SMT solver ran out of memory, and "A" denotes the number of times the SMT solver aborted (i.e., gave up before reaching the timeout). Total times are in seconds and include timeouts, memory exhaustions, and solver aborts with their respective runtimes.

in Fig. 1. Also note that all approaches using $\mathcal{T}_{\lambda\mathcal{A}}$ perform better than the naïve implementation using loops, where the latter is furthermore incomplete in general due to the bounded number of loop iterations that can be considered.[8]

7 Related Work

Decidable extensions of the theory of arrays have been considered before. Suzuki and Jefferson [30] have studied the extension of $\mathcal{T}_{\mathcal{A}}$ with a restricted use of a permutation predicate. Mateti [22] has described a theory of arrays where entries of an array can be exchanged. Jaffar [19] has investigated reading of array segments but does not discuss writing array segments. Ghilardi *et al.* [16] have considered the addition of axioms specifying the dimension of arrays, injectivity of arrays, arrays with domains, array prefixes, array iterators, and sorted arrays. All of these extensions are orthogonal to the theory $\mathcal{T}_{\lambda\mathcal{A}}$ considered in this paper. A theory of arrays with constant-valued arrays has been proposed by Stump *et al.* [29]. These constant-valued arrays can easily be modelled in $\mathcal{T}_{\lambda\mathcal{A}}$ using a simple λ-term of the form $\lambda i.\ k$ where k is the constant value. This theory has also been considered in [1]. De Moura and Bjørner [27] have introduced

[8] This incompleteness does not manifest itself in the evaluation since the number of loop iterations was chosen sufficiently large for each program. This causes `LLBMC` to run out of memory on some examples, though.

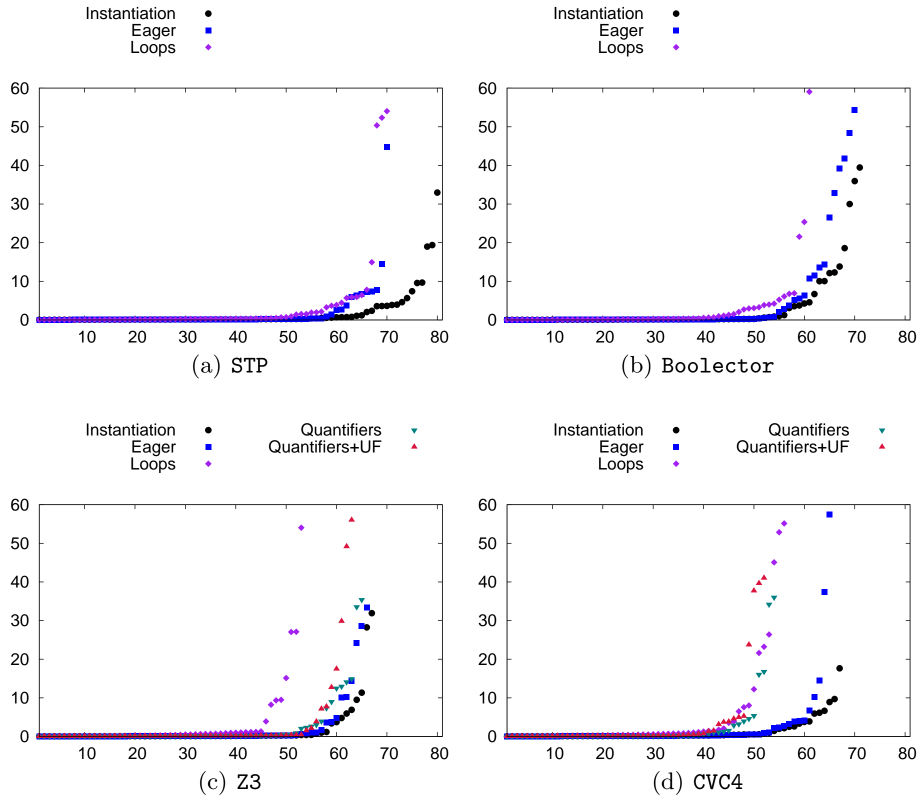

Fig. 1. Cactus plots for the 81 benchmark problems. The x-axis shows the number of solved problems and the y-axis shows the time limit for each problem in seconds. Thus, a curve that is closer to the bottom-right indicates a better performing approach.

combinatory array logic, which extends $\mathcal{T}_{\mathcal{A}}$ by constant-valued arrays as in [29] and a map combinator.

The satisfiability problem for restricted classes of quantified formulas in the theory of arrays has been investigated as well. The work by Bradley *et al.* [3] identifies the *array property fragment*, where value constraints are restricted by index guards in universally quantified subformulas. Note that already the special case of the copy operation cannot be expressed in the array property fragment due to the "pointer arithmetic" $q + (r - p)$. The conceptually simpler set operation can be defined in the array property fragment, though. The array property fragment was later extended by Habermehl *et al.* [17, 18], but the "pointer arithmetic" needed for copy is still not permitted. Finally, Zhou *et al.* [31] have investigated a theory of arrays where the elements are from a finite set.

A logic containing λ-terms has been considered by Bryant *et al.* [6], who also show that $\mathcal{T}_{\mathcal{A}}$ can be simulated using λ-terms. The key distinction of the present work in comparison to [6] is that we *extend* $\mathcal{T}_{\mathcal{A}}$ with λ-terms that denote

anonymous arrays. This makes it possible to utilize powerful and efficient SMT solvers for $\mathcal{T}_{\mathcal{A}}$ in order to decide satisfiability of quantifier-free $\mathcal{T}_{\lambda\mathcal{A}}$ formulas using the reduction based on instantiating quantifiers (Sect. 5.3). In contrast, [6] has to apply β-reduction eagerly, which corresponds to our eager reduction (Sect. 5.1). As clearly shown in Sect. 6, the instantiation-based reduction from $\mathcal{T}_{\lambda\mathcal{A}}$ to $\mathcal{T}_{\mathcal{A}}$ performs much better than the eager reduction from $\mathcal{T}_{\lambda\mathcal{A}}$ to $\mathcal{T}_{\mathcal{I}} \oplus \mathcal{T}_{\mathcal{E}} \oplus \mathcal{T}_{\mathcal{EUF}}$.

8 Conclusions and Further Work

We have presented $\mathcal{T}_{\lambda\mathcal{A}}$, an extension of the theory of arrays with λ-terms. These λ-terms can be used in order to model library functions such as `C`'s `memset` and `memcpy` in formal methods such as program analysis, (deductive) software verification, bounded model checking, or symbolic execution. Furthermore, we have shown how a class of loops can automatically be summarized using such λ-terms. We have presented three reductions from $\mathcal{T}_{\lambda\mathcal{A}}$ to theories that are supported by current SMT solvers and have reported on an evaluation in `LLBMC` [24].

For future work, we are particularly interested in adding "native" support for $\mathcal{T}_{\lambda\mathcal{A}}$ in SMT solvers such as `STP` [15], `Boolector` [4], `Z3` [26], or `CVC4` [2]. For this, it will be necessary to investigate lazy axiom instantiation or lemma-on-demand techniques for $\mathcal{T}_{\lambda\mathcal{A}}$ since these techniques have been fundamental for the performance gain that SMT solvers for $\mathcal{T}_{\mathcal{A}}$ have experienced in recent years. A first, simple idea is to add not all instantiations from Sect. 5.3 beforehand, but instead do this incrementally in a CEGAR-like loop guided by spurious models generated by the SMT solver for $\mathcal{T}_{\mathcal{A}}$. A further direction for future work is to widen the class of loops that can be summarized using λ-terms in the theory $\mathcal{T}_{\lambda\mathcal{A}}$. Finally, we are interested in adding an operation similar to `fold` as known from functional programming languages to the theory $\mathcal{T}_{\lambda\mathcal{A}}$.

References

1. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. IC 183(2), 140–164 (2003)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011. LNCS, vol. 6806, pp. 171–177 (2011)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI 2006. LNCS, vol. 3855, pp. 427–442 (2006)
4. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: TACAS 2009. LNCS, vol. 5505, pp. 174–177 (2009)
5. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. JSAT 6, 165–201 (2009)
6. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV 2002. LNCS, vol. 2404, pp. 78–92 (2002)
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008. pp. 209–224 (2008)

8. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. TISSEC 12(2), 10:1–10:38 (2008)
9. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004. LNCS, vol. 2988, pp. 168–176 (2004)
10. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs 2009. pp. 23–42 (2009)
11. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. TSE 38(4), 957–974 (2012)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: SEFM 2012. pp. 233–247 (2012)
13. Falke, S., Merz, F., Sinz, C.: A theory of C-style memory allocation. In: SMT 2011. pp. 71–80 (2011)
14. Falke, S., Sinz, C., Merz, F.: A theory of arrays with set and copy operations (extended abstract). In: SMT 2012. pp. 97–106 (2012)
15. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV 2007. LNCS, vol. 4590, pp. 519–531 (2007)
16. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. AMAI 50(3–4), 231–254 (2007)
17. Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: LPAR 2008. LNCS, vol. 5330, pp. 558–573 (2008)
18. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: FoSSaCS 2008. LNCS, vol. 4962, pp. 474–489 (2008)
19. Jaffar, J.: Presburger arithmetic with array segments. IPL 12(2), 79–82 (1981)
20. Kapur, D., Zarba, C.G.: The reduction approach to decision procedures. Tech. Rep. TR-CS-2005-44, Dept. of Computer Science, Univ. of New Mexico (2005)
21. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88 (2004)
22. Mateti, P.: A decision procedure for the correctness of a class of programs. JACM 28, 215–232 (1981)
23. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress 1962. pp. 21–28 (1962)
24. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: VSTTE 2012. LNCS, vol. 7152, pp. 146–161 (2012)
25. de Moura, L.: Answer to *Support for AUFBV?* on Stack Overflow, `http://stackoverflow.com/questions/7411995/support-for-aufbv` (2011)
26. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340 (2008)
27. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD 2009. pp. 45–52 (2009)
28. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: SSV 2010 (2010)
29. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS 2001. pp. 29–37 (2001)
30. Suzuki, N., Jefferson, D.: Verification decidability of Presburger array programs. JACM 27, 191–205 (1980)
31. Zhou, M., He, F., Wang, B.Y., Gu, M.: On array theory of bounded elements. In: CAV 2010. LNCS, vol. 6174, pp. 570–584 (2010)