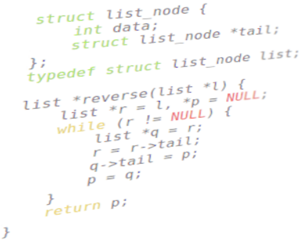
Karlsruhe Institute of Technology

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

Stephan Falke, **Florian Merz, and Carsten Sinz**

INSTITUTE FOR THEORETICAL COMPUTER SCIENCE (ITI)



Motivation

- SMT-solvers are routinely used in program analysis:
  - Deductive program verification
  - Symbolic execution
  - Software bounded model checking
  - ...

Motivation

- SMT-solvers are routinely used in program analysis:
  - Deductive program verification
  - Symbolic execution
  - Software bounded model checking
  - ...
- Prominent theory: $\mathcal{T}_{\mathcal{A}}$ (theory of arrays)
  - Model arrays/structures/objects in the program
  - Model main memory

$\mathcal{T}_{\mathcal{A}}$: The Theory of Arrays

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

$\mathcal{T}_{\mathcal{A}}$: The Theory of Arrays

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

$$p = r \quad \Longrightarrow \quad \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

$$\neg(p = r) \quad \Longrightarrow \quad \mathtt{read}(\mathtt{write}(a, p, v), r) = \mathtt{read}(a, r)$$

$\mathcal{T}_{\mathcal{A}}$: The Theory of Arrays

index terms	$t_I ::= \ldots$
element terms	$t_E ::= \ldots \mid \mathtt{read}(t_A, t_I)$
array terms	$t_A ::= a \mid \mathtt{write}(t_A, t_I, t_E)$

a write modifies the position written to ...

$$p = r \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

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

$\mathcal{T}_\mathcal{A}$: The Theory of Arrays

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

a write modifies the position written to ...

$$p = r \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

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

... and nothing else

Motivation

How to model standard library functions such as `memset` and `memcpy`?

```
void *memset(void *dst, int c, size_t n);
```

```
void *memcpy(void *dst, const void *src, size_t n);
```

Motivation

How to model standard library functions such as `memset` and `memcpy`?

might not be constant!

```
void *memset(void *dst, int c, size_t n);
```

might not be constant!

```
void *memcpy(void *dst, const void *src, size_t n);
```

Motivation

Karlsruhe Institute of Technology

```
...
memcpy(a, b, 4);
...
```

Motivation

Karlsruhe Institute of Technology

$$a_1 = \texttt{write}(a, 0, \texttt{read}(b, 0))$$

```
...
memcpy(a, b, 4);
...
```

Motivation

Karlsruhe Institute of Technology

```
...
memcpy(a, b, 4);
...
```

$$a_1 = \mathtt{write}(a, 0, \mathtt{read}(b, 0))$$

$$a_2 = \mathtt{write}(a_1, 1, \mathtt{read}(b, 1))$$

Motivation

Karlsruhe Institute of Technology

```
...
memcpy(a, b, 4);
...
```

$$a_1 = \mathtt{write}(a, 0, \mathtt{read}(b, 0))$$
$$a_2 = \mathtt{write}(a_1, 1, \mathtt{read}(b, 1))$$
$$a_3 = \mathtt{write}(a_2, 2, \mathtt{read}(b, 2))$$

Motivation

```
...
memcpy(a, b, 4);
...
```

$$a_1 = \mathtt{write}(a, 0, \mathtt{read}(b, 0))$$
$$a_2 = \mathtt{write}(a_1, 1, \mathtt{read}(b, 1))$$
$$a_3 = \mathtt{write}(a_2, 2, \mathtt{read}(b, 2))$$
$$a' = \mathtt{write}(a_3, 3, \mathtt{read}(b, 3))$$

Motivation

KIT
Karlsruhe Institute of Technology

```
...
memcpy(a, b, 4);
...
```

$$a_1 = \texttt{write}(a, 0, \texttt{read}(b, 0))$$
$$a_2 = \texttt{write}(a_1, 1, \texttt{read}(b, 1))$$
$$a_3 = \texttt{write}(a_2, 2, \texttt{read}(b, 2))$$
$$a' = \texttt{write}(a_3, 3, \texttt{read}(b, 3))$$

Does not scale well for large constants

Motivation

Karlsruhe Institute of Technology

```
...
memcpy(a, b, n);
...
```

Motivation

```
...
memcpy(a, b, n);
...
```

???

Motivation

```
...
memcpy(a, b, n);
...
```

$$a' = \text{copy}(a, 0, b, 0, n)$$

Motivation

Karlsruhe Institute of Technology

```
...
memcpy(a, b, n);
...
```

$$a' = \lambda i.\, \mathsf{ITE}(0 \leq i < n,\ \mathtt{read}(b, i),\ \mathtt{read}(a, i))$$

Motivation

Karlsruhe Institute of Technology

```
...
memcpy(a, b, n);
...
```

$$a' = \lambda i.\ \mathrm{ITE}(0 \leq i < n,\ \mathtt{read}(b, i),\ \mathtt{read}(a, i))$$

$\Longrightarrow$ Extend $\mathcal{T}_{\mathcal{A}}$ by λ-terms that describe arrays

Motivation

```
...
memset(a, v, n);
...
```

Motivation

Karlsruhe Institute of Technology

```
...
memset(a, v, n);
...
```

$$a' = \lambda i.\, \mathrm{ITE}(0 \leq i < n,\ v,\ \mathtt{read}(a, i))$$

Motivation

Karlsruhe Institute of Technology

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

Motivation

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

$$a' = \lambda i.\, \mathsf{ITE}(0 \leq i < n,\ i + 1,\ \mathtt{read}(a, i))$$

Motivation

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

$$a' = \lambda i.\, \mathsf{ITE}(0 \leq i < n,\ i + 1,\ \mathtt{read}(a, i))$$
$$a'' = \lambda j.\, \mathsf{ITE}(n \leq j < 2 * n,\ 2 * j,\ \mathtt{read}(a', j))$$

Contributions

Karlsruhe Institute of Technology

1. $\mathcal{T}_{\lambda\mathcal{A}}$: an extension of $\mathcal{T}_{\mathcal{A}}$ with λ-terms

Contributions

1. $\mathcal{T}_{\lambda\mathcal{A}}$: an extension of $\mathcal{T}_{\mathcal{A}}$ with λ-terms
2. Satisfiability checking for $\mathcal{T}_{\lambda\mathcal{A}}$

$\mathcal{T}_{\lambda\mathcal{A}}$: The Theory of Arrays with λ-Terms

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

$$p = r \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

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

$\mathcal{T}_{\lambda\mathcal{A}}$: The Theory of Arrays with λ-Terms

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

$$p = r \quad \Longrightarrow \quad \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

$$\neg(p = r) \quad \Longrightarrow \quad \mathtt{read}(\mathtt{write}(a, p, v), r) = \mathtt{read}(a, r)$$

$\mathcal{T}_{\lambda\mathcal{A}}$: The Theory of Arrays with λ-Terms

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

$$p = r \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

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

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

$\mathcal{T}_{\lambda\mathcal{A}}$: The Theory of Arrays with λ-Terms

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

$$p = r \quad \Longrightarrow \quad \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

$$\neg(p = r) \quad \Longrightarrow \quad \mathtt{read}(\mathtt{write}(a, p, v), r) = \mathtt{read}(a, r)$$

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

β-reduction

$\mathcal{T}_{\lambda\mathcal{A}}$: The Theory of Arrays with λ-Terms

index terms	$t_I ::= \ldots$
element terms	$t_E ::= \ldots \mid \texttt{read}(t_A, t_I)$
array terms	$t_A ::= a \mid \texttt{write}(t_A, t_I, t_E) \mid \lambda i.\ t_E$

$$p = r \quad \Longrightarrow \quad \texttt{read}(\texttt{write}(a, p, v), r) = v$$

$$\neg(p = r) \quad \Longrightarrow \quad \texttt{read}(\texttt{write}(a, p, v), r) = \texttt{read}(a, r)$$

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

β-reduction

$\texttt{write}(a, p, v)$ could be simulated using $\lambda i.\ \mathrm{ITE}(p = i,\ v,\ \texttt{read}(a, i))$

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`
- Summarize loops

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables
- Zero initialization of fresh memory pages

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables
- Zero initialization of fresh memory pages
- “Havoc” memory regions (volatile variables)

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables
- Zero initialization of fresh memory pages
- “Havoc” memory regions (volatile variables)
- Model memory mapped I/O

Uses of $\mathcal{T}_{\lambda\mathcal{A}}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables
- Zero initialization of fresh memory pages
- "Havoc" memory regions (volatile variables)
- Model memory mapped I/O
- Attaching metadata to memory regions (allocated, de-allocated, . . .)

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

- Broadly speaking:
  - loop iterations do not depend on earlier iterations
  - consecutive iterations update consecutive array locations

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

- Broadly speaking:
  - loop iterations do not depend on earlier iterations
  - consecutive iterations update consecutive array locations
- More precisely:
  - Induction variable i is incremented by one in each iteration
  - i^{th} iteration unconditionally updates only $a[i]$
  - No other variable declared outside the loop is modified
  - i^{th} iteration of the loop may not use elements of a that have been modified in earlier iterations

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

- Broadly speaking:
  - loop iterations do not depend on earlier iterations
  - consecutive iterations update consecutive array locations
- More precisely:
  - Induction variable i is incremented by one in each iteration
  - i^{th} iteration unconditionally updates only $a[i]$
  - No other variable declared outside the loop is modified
  - i^{th} iteration of the loop may not use elements of a that have been modified in earlier iterations
- Loops can often be automatically transformed into loops that satisfy these requirements

Satisfiability Checking

- Based on reductions to theories supported by SMT-solvers

Satisfiability Checking

- Based on reductions to theories supported by SMT-solvers
- One quantifier-based approach

Satisfiability Checking

- Based on reductions to theories supported by SMT-solvers
- One quantifier-based approach
- Two quantifier-free approaches
  - Eager reduction
  - Instantiation-based approach

Quantifier-Based Approach

- Replace $\lambda i.\, s$ by a fresh constant a_s

Quantifier-Based Approach

- Replace $\lambda i.\, s$ by a fresh constant a_s
- Add the constraint

$$\forall r.\, \mathtt{read}(a_s, r) = s[i/r]$$

  to the formula

Quantifier-Based Approach

- Replace $\lambda i.\ s$ by a fresh constant a_s
- Add the constraint

$$\forall r.\ \texttt{read}(a_s, r) = s[i/r]$$

  to the formula
- Requires an SMT-solver that supports quantifiers

Quantifier-Based Approach

- Replace $\lambda i.\, s$ by a fresh constant a_s
- Add the constraint

$$\forall r.\, \mathtt{read}(a_s, r) = s[i/r]$$

  to the formula
- Requires an SMT-solver that supports quantifiers
- Does not provide a decision procedure in general

Eager Reduction

- Replace $\texttt{read}(\texttt{write}(a, p, v), r)$ by

$$\text{ITE}(p = r,\ v,\ \texttt{read}(a, r))$$

Eager Reduction

- Replace $\texttt{read}(\texttt{write}(a, p, v), r)$ by

$$\mathsf{ITE}(p = r,\ v,\ \texttt{read}(a, r))$$

- Replace $\texttt{read}(\lambda i.\ s, r)$ by

$$s[i / r]$$

Eager Reduction

- Replace $\texttt{read}(\texttt{write}(a, p, v), r)$ by

$$\text{ITE}(p = r,\ v,\ \texttt{read}(a, r))$$

- Replace $\texttt{read}(\lambda i.\ s, r)$ by

$$s[i / r]$$

- $\mathcal{T}_{\lambda\mathcal{A}}$ axioms are applied eagerly

Eager Reduction

- Replace $\texttt{read}(\texttt{write}(a, p, v), r)$ by

$$\text{ITE}(p = r,\ v,\ \texttt{read}(a, r))$$

- Replace $\texttt{read}(\lambda i.\ s, r)$ by

$$s[i/r]$$

- $\mathcal{T}_{\lambda\mathcal{A}}$ axioms are applied eagerly
- Can be used in combination with any solver that supports $\mathcal{T}_{\mathcal{A}}$ and the index and element theories

Instantiation-Based Approach

- Replace $\lambda i.\ s$ by a fresh constant a_s

Instantiation-Based Approach

Karlsruhe Institute of Technology

- Replace $\lambda i.\ s$ by a fresh constant a_s
- Add needed instantiations of

$$\forall r.\ \mathtt{read}(a_s, r) = s[i/r]$$

to the formula

Instantiation-Based Approach

Karlsruhe Institute of Technology

- Replace $\lambda i.\, s$ by a fresh constant a_s
- Add needed instantiations of

$$\forall r.\, \mathtt{read}(a_s, r) = s[i/r]$$

  to the formula
- Needed instantiations are determined by `reads` that "depend" on a_s

Instantiation-Based Approach

- Replace $\lambda i.\, s$ by a fresh constant a_s
- Add needed instantiations of

$$\forall r.\, \texttt{read}(a_s, r) = s[i/r]$$

  to the formula
- Needed instantiations are determined by `read`s that “depend” on a_s
- Can be used in combination with any solver that supports $\mathcal{T}_{\mathcal{A}}$ and the index and element theories

Evaluation

- Done in the software bounded model checker `LLBMC`

Evaluation

- Done in the software bounded model checker `LLBMC`
- Uses bitvectors as index and element theories

Evaluation

- Done in the software bounded model checker LLBMC
- Uses bitvectors as index and element theories
- Applied on 81 benchmark programs
  - 67 programs produce λ-terms obtained from `memset` or `memcpy`
  - 14 program contain loops that can be summarized using λ-terms

Evaluation

- Done in the software bounded model checker `LLBMC`
- Uses bitvectors as index and element theories
- Applied on 81 benchmark programs
  - 67 programs produce λ-terms obtained from `memset` or `memcpy`
  - 14 program contain loops that can be summarized using λ-terms
- Of the resulting formulas, 20 are satisfiable and 61 are unsatisfiable

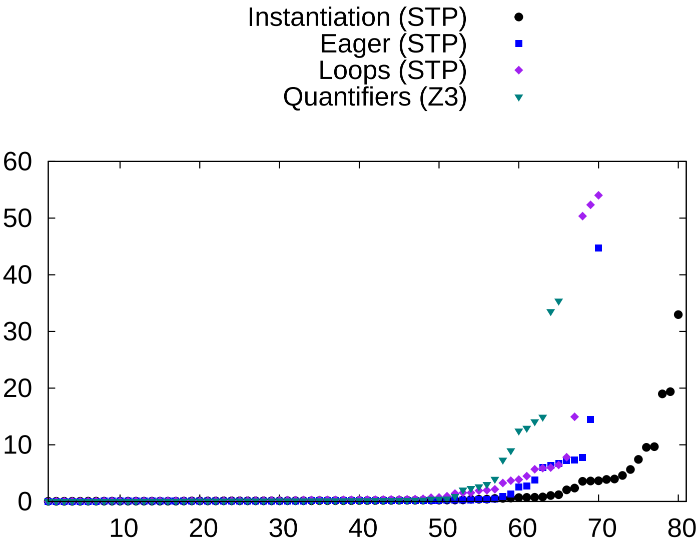
Evaluation

- Done in the software bounded model checker `LLBMC`
- Uses bitvectors as index and element theories
- Applied on 81 benchmark programs
  - 67 programs produce λ-terms obtained from `memset` or `memcpy`
  - 14 program contain loops that can be summarized using λ-terms
- Of the resulting formulas, 20 are satisfiable and 61 are unsatisfiable
- Evaluated three reductions and loop unrolling
  - Quantifier-based approach using `Z3` and `CVC4`
  - Eager reduction and instantiation-based approach using `STP`, `Boolector`, `Z3`, and `CVC4`
  - Loop unrolling approach using `STP`, `Boolector`, `Z3`, and `CVC4`

Results

SMT solver	Approach	Total Time	# Solved Formulas	# Timeouts	# Aborts
STP	Instantiation	206.034	80	1	–
STP	Eager	779.544	70	11	–
STP	Loops	670.526	70	6	5
Boolector	Instantiation	818.782	71	10	–
Boolector	Eager	986.751	70	11	–
Boolector	Loops	1139.483	61	15	5
Z3	Instantiation	948.365	67	13	1
Z3	Eager	1043.632	66	15	–
Z3	Quantifiers	1122.489	65	16	–
Z3	Loops	1619.583	53	23	5
CVC4	Instantiation	928.079	67	14	–
CVC4	Eager	1119.748	65	16	–
CVC4	Quantifiers	1407.118	54	21	6
CVC4	Loops	1552.698	56	19	6

Results



Instantiation (STP)
Eager (STP)
Loops (STP)
Quantifiers (Z3)
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80

Conclusion and Future Work

- $\mathcal{T}_{\lambda\mathcal{A}}$ is a useful, decidable extension of $\mathcal{T}_{\mathcal{A}}$

Conclusion and Future Work

- $\mathcal{T}_{\lambda\mathcal{A}}$ is a useful, decidable extension of $\mathcal{T}_{\mathcal{A}}$
- Performs better than unrolling for
  - `memset` and `memcpy`
  - summarizable loops

Conclusion and Future Work

- $\mathcal{T}_{\lambda\mathcal{A}}$ is a useful, decidable extension of $\mathcal{T}_{\mathcal{A}}$
- Performs better than unrolling for
  - `memset` and `memcpy`
  - summarizable loops
- Quantifier-free reductions perform better than `Z3`'s and `CVC4`'s reasoning involving quantifiers

Conclusion and Future Work

- $\mathcal{T}_{\lambda\mathcal{A}}$ is a useful, decidable extension of $\mathcal{T}_{\mathcal{A}}$
- Performs better than unrolling for
  - `memset` and `memcpy`
  - summarizable loops
- Quantifier-free reductions perform better than `Z3`'s and `CVC4`'s reasoning involving quantifiers
- Integration into an SMT-solver using "Lemmas-on-demand"/"lazy instantiation" is the next step

`http://llbmc.org`