Extending the Theory of Arrays: \texttt{memset}, \texttt{memcpy}, and Beyond

\textit{Stephan Falke}, Florian Merz, and Carsten Sinz
Motivation

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

Prominent theory: $\mathcal{T}_A$ (theory of arrays)
- Model arrays/structures/objects in the program
- Model main memory
$\mathcal{T}_A$: The Theory of Arrays

| index terms | $t_i ::= \ldots$ |
| element terms | $t_E ::= \ldots \mid \text{read}(t_A, t_i)$ |
| array terms | $t_A ::= a \mid \text{write}(t_A, t_i, t_E)$ |
$\mathcal{T}_A$: The Theory of Arrays

| index terms | $t_i ::= \ldots$ |
| element terms | $t_E ::= \ldots | \text{read}(t_A, t_i)$ |
| array terms | $t_A ::= a | \text{write}(t_A, t_i, t_E)$ |

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

$\neg(p = r) \implies \text{read}(\text{write}(a, p, v), r) = \text{read}(a, r)$
\( \mathcal{T}_A: \) The Theory of Arrays

<table>
<thead>
<tr>
<th>Term Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Terms</td>
<td>( t_i ::= \ldots )</td>
</tr>
<tr>
<td>Element Terms</td>
<td>( t_E ::= \ldots \mid \text{read}(t_A, t_i) )</td>
</tr>
<tr>
<td>Array Terms</td>
<td>( t_A ::= a \mid \text{write}(t_A, t_i, t_E) )</td>
</tr>
</tbody>
</table>

A write modifies the position written to ...

\[
p = r \implies \text{read}(\text{write}(a, p, v), r) = v
\]

\[
\neg (p = r) \implies \text{read}(\text{write}(a, p, v), r) = \text{read}(a, r)
\]
\( \mathcal{T}_A \): The Theory of Arrays

| index terms | \( t_i ::= \ldots \) |
| element terms | \( t_E ::= \ldots \mid \text{read}(t_A, t_i) \) |
| array terms | \( t_A ::= a \mid \text{write}(t_A, t_i, t_E) \) |

A write modifies the position written to ...

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

... and nothing else
Motivation

How to model standard library functions such as \texttt{memset} and \texttt{memcpy}?

\begin{verbatim}
void *memset(void *dst, int c, size_t n);

void *memcpy(void *dst, const void *src, size_t n);
\end{verbatim}
Motivation

How to model standard library functions such as \texttt{memset} and \texttt{memcpy}?\footnote{might not be constant!}

\begin{verbatim}
void *memset(void *dst, int c, size_t n);
\end{verbatim}

\begin{verbatim}
void *memcpy(void *dst, const void *src, size_t n);
\end{verbatim}

\footnote{might not be constant!}
... 
memcpy(a, b, 4);  
...
Motivation

\[ a_1 = \text{write}(a, 0, \text{read}(b, 0)) \]

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

Does not scale well for large constants
Motivation

\[ a_1 = \text{write}(a, 0, \text{read}(b, 0)) \]
\[ a_2 = \text{write}(a_1, 1, \text{read}(b, 1)) \]

\[
\ldots
d \text{memcpy}(a, b, 4);
\ldots
\]
Motivation

\[
\begin{align*}
    a_1 &= \text{write}(a,0,\text{read}(b,0)) \\
    a_2 &= \text{write}(a_1,1,\text{read}(b,1)) \\
    a_3 &= \text{write}(a_2,2,\text{read}(b,2))
\end{align*}
\]

... \text{memcpy}(a, b, 4); ...

Does not scale well for large constants
Motivation

\[ a_1 = \text{write}(a,0,\text{read}(b,0)) \]
\[ a_2 = \text{write}(a_1,1,\text{read}(b,1)) \]
\[ a_3 = \text{write}(a_2,2,\text{read}(b,2)) \]
\[ a' = \text{write}(a_3,3,\text{read}(b,3)) \]
Motivation

\[
\begin{align*}
    a_1 &= \text{write}(a, 0, \text{read}(b, 0)) \\
    a_2 &= \text{write}(a_1, 1, \text{read}(b, 1)) \\
    a_3 &= \text{write}(a_2, 2, \text{read}(b, 2)) \\
    a' &= \text{write}(a_3, 3, \text{read}(b, 3))
\end{align*}
\]

Does not scale well for large constants
Motivation

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

...
Motivation

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

Motivation

... 

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

... 

\[ a' = \text{copy}(a, 0, b, 0, n) \]
Motivation

\[
\ldots \text{memcpy}(a, b, n); \ldots
\]

\[
a' = \lambda i. \ \text{ITE}(0 \leq i < n, \ \text{read}(b, i), \ \text{read}(a, i))
\]
Motivation

\[
\ldots \text{memcpy}(a, b, n); \ldots
\]

\[
a' = \lambda i. \text{ITE}(0 \leq i < n, \text{read}(b, i), \text{read}(a, i))
\]

\[\Rightarrow\text{Extend } \mathcal{T}_A \text{ by } \lambda\text{-terms that describe arrays}\]
Motivation

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

\[ a' = \lambda i. \text{ITE}(0 \leq i < n, v, \text{read}(a, i)) \]
Motivation

```c
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;
}
```
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;
}

\[ a' = \lambda i. \text{ITE}(0 \leq i < n, i + 1, \text{read}(a, i)) \]
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;
}

\[
a' = \lambda i. \text{ITE}(0 \leq i < n, i + 1, \text{read}(a, i))
\]
\[
a'' = \lambda j. \text{ITE}(n \leq j < 2 \times n, 2 \times j, \text{read}(a', j))
\]
Contributions

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

1. $\mathcal{T}_{\lambda A}$: an extension of $\mathcal{T}_A$ with $\lambda$-terms
2. Satisfiability checking for $\mathcal{T}_{\lambda A}$
$\mathcal{T}_{\lambda A}$: The Theory of Arrays with $\lambda$-Terms

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

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

$\neg(p = r) \implies \text{read} (\text{write}(a, p, v), r) = \text{read}(a, r)$
\[ T_{\lambda A} : \text{The Theory of Arrays with } \lambda \text{-Terms} \]

<table>
<thead>
<tr>
<th>index terms</th>
<th>( t_i ::= \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>element terms</td>
<td>( t_E ::= \ldots \mid \text{read}(t_A, t_i) )</td>
</tr>
<tr>
<td>array terms</td>
<td>( t_A ::= a \mid \text{write}(t_A, t_i, t_E) \mid \lambda i. t_E )</td>
</tr>
</tbody>
</table>

\[
p = r \quad \implies \quad \text{read}(\text{write}(a, p, v), r) = v
\]

\[
\neg(p = r) \quad \implies \quad \text{read}(\text{write}(a, p, v), r) = \text{read}(a, r)
\]

\[
\beta \text{-reduction}
\]

write \((a, p, v)\) could be simulated using \(\lambda i. \text{ITE}(p = i, v, \text{read}(a, i))\)
$\mathcal{T}_{\lambda A}$: The Theory of Arrays with $\lambda$-Terms

<table>
<thead>
<tr>
<th>index terms</th>
<th>$t_1 ::= \ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>element terms</td>
<td>$t_E ::= \ldots \mid \text{read}(t_A, t_1)$</td>
</tr>
<tr>
<td>array terms</td>
<td>$t_A ::= a \mid \text{write}(t_A, t_1, t_E) \mid \lambda i. t_E$</td>
</tr>
</tbody>
</table>

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

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

$\text{read}(\lambda i. s, r) = s[i/r]$
$\mathcal{T}_{\lambda A}$: The Theory of Arrays with $\lambda$-Terms

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

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

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

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

$\beta$-reduction
\( T_{\lambda A} \): The Theory of Arrays with \( \lambda \)-Terms

<table>
<thead>
<tr>
<th>index terms</th>
<th>( t_1 ::= \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>element terms</td>
<td>( t_E ::= \ldots \</td>
</tr>
<tr>
<td>array terms</td>
<td>( t_A ::= a \</td>
</tr>
</tbody>
</table>

\[ p = r \implies read(write(a, p, v), r) = v \]

\[ \neg(p = r) \implies read(write(a, p, v), r) = read(a, r) \]

\[ \beta\text{-reduction} \]

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

Write \( (a, p, v) \) could be simulated using \( \lambda i. \text{ITE}(p = i, v, \text{read}(a, i)) \)
Precisely model \texttt{memset} and \texttt{memcpy}
Uses of $T_{\lambda A}$

- Precisely model \texttt{memset} and \texttt{memcpy}
- Summarize loops
Uses of $\mathcal{T}_{\lambda A}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables
Uses of $\mathcal{T}_{\lambda A}$

- Precisely model `memset` and `memcpy`
- Summarize loops
- Zero initialization of global variables
- Zero initialization of fresh memory pages
Uses of $\mathcal{T}_{\lambda A}$

- 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 $\mathcal{T}_{\lambda A}$

- 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 $T_{\lambda A}$

- 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 $\mathcal{T}_{\lambda A}$

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

Loops can often be automatically transformed into loops that satisfy these requirements.
Loop Summarization Using $\mathcal{T}_{LA}$

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

- 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

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

to the formula

Requires an SMT-solver that supports quantifiers

Does not provide a decision procedure in general
Quantifier-Based Approach

- Replace $\lambda i. s$ by a fresh constant $a_s$
- Add the constraint
  \[
  \forall r. \text{read}(a_s, r) = s[i/r]
  \]
  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

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

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

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

  \[
  \text{ITE}(p = r, v, \text{read}(a, r))
  \]
Eager Reduction

- Replace \( \text{read}(\text{write}(a, p, v), r) \) by
  \[
  \text{ITE}(p = r, v, \text{read}(a, r))
  \]

- Replace \( \text{read}(\lambda i. s, r) \) by
  \[
  s[i/r]
  \]
Eager Reduction

- Replace \( \text{read}(\text{write}(a, p, v), r) \) by
  \[
  \text{ITE}(p = r, v, \text{read}(a, r))
  \]

- Replace \( \text{read}(\lambda i. s, r) \) by
  \[
  s[i/r]
  \]

- \( \mathcal{T}_{\lambda A} \) axioms are applied eagerly
Eager Reduction

- Replace \( \text{read}(\text{write}(a, p, v), r) \) by
  \[
  \text{ITE}(p = r, v, \text{read}(a, r))
  \]

- Replace \( \text{read}(\lambda i. s, r) \) by
  \[
  s[i/r]
  \]

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

- Can be used in combination with any solver that supports \( \mathcal{T}_A \) and the index and element theories
Instantiation-Based Approach

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

- Replace \( \lambda i. s \) by a fresh constant \( a_s \)
- Add needed instantiations of 
  \[
  \forall r. \text{read}(a_s, r) = s[i/r]
  \]
  to the formula
Instantiation-Based Approach

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

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

...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

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

to the formula

- Needed instantiations are determined by reads that “depend” on \( a_s \)
- Can be used in combination with any solver that supports \( T_A \) 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 $\lambda$-terms obtained from `memset` or `memcpy`
  - 14 program contain loops that can be summarized using $\lambda$-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 $\lambda$-terms obtained from `memcpy` or `memset`
  - 14 program contain loops that can be summarized using $\lambda$-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 $\lambda$-terms obtained from `memset` or `memcpy`
  - 14 program contain loops that can be summarized using $\lambda$-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
<table>
<thead>
<tr>
<th>SMT solver</th>
<th>Approach</th>
<th>Total Time</th>
<th># Solved Formulas</th>
<th># Timeouts</th>
<th># Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>STP</td>
<td>Instantiation</td>
<td>206.034</td>
<td>80</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>STP</td>
<td>Eager</td>
<td>779.544</td>
<td>70</td>
<td>11</td>
<td>–</td>
</tr>
<tr>
<td>STP</td>
<td>Loops</td>
<td>670.526</td>
<td>70</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Boolector</td>
<td>Instantiation</td>
<td>818.782</td>
<td>71</td>
<td>10</td>
<td>–</td>
</tr>
<tr>
<td>Boolector</td>
<td>Eager</td>
<td>986.751</td>
<td>70</td>
<td>11</td>
<td>–</td>
</tr>
<tr>
<td>Boolector</td>
<td>Loops</td>
<td>1139.483</td>
<td>61</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>Z3</td>
<td>Instantiation</td>
<td>948.365</td>
<td>67</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Z3</td>
<td>Eager</td>
<td>1043.632</td>
<td>66</td>
<td>15</td>
<td>–</td>
</tr>
<tr>
<td>Z3</td>
<td>Quantifiers</td>
<td>1122.489</td>
<td>65</td>
<td>16</td>
<td>–</td>
</tr>
<tr>
<td>Z3</td>
<td>Loops</td>
<td>1619.583</td>
<td>53</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>CVC4</td>
<td>Instantiation</td>
<td>928.079</td>
<td>67</td>
<td>14</td>
<td>–</td>
</tr>
<tr>
<td>CVC4</td>
<td>Eager</td>
<td>1119.748</td>
<td>65</td>
<td>16</td>
<td>–</td>
</tr>
<tr>
<td>CVC4</td>
<td>Quantifiers</td>
<td>1407.118</td>
<td>54</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>CVC4</td>
<td>Loops</td>
<td>1552.698</td>
<td>56</td>
<td>19</td>
<td>6</td>
</tr>
</tbody>
</table>
Results

Instantiation (STP) ●
Eager (STP) ▲
Loops (STP) □
Quantifiers (Z3) ▼
Conclusion and Future Work

- $\mathcal{T}_{\lambda A}$ is a useful, decidable extension of $\mathcal{T}_A$
Conclusion and Future Work

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

Quantifier-free reductions perform better than $\mathcal{Z3}$'s and $\mathcal{CVC4}$'s reasoning involving quantifiers. Integration into an SMT-solver using "Lemmas-on-demand"/"lazy instantiation" is the next step.
Conclusion and Future Work

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

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