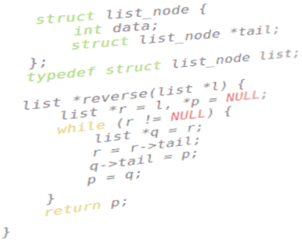
Karlsruhe Institute of Technology

A Theory of C-Style Memory Allocation

Stephan Falke, Florian Merz, and Carsten Sinz

Research Group “Verification meets Algorithm Engineering”



Introduction

- Modeling the memory content is important in program analysis
  - Formalized using McCarthy's theory of arrays

Introduction

- Modeling the memory content is important in program analysis
  - Formalized using McCarthy's theory of arrays
- Equally important: memory protection
  - Often only handled in an ad-hoc and informal way

Introduction

- Modeling the memory content is important in program analysis
  - Formalized using McCarthy's theory of arrays
- Equally important: memory protection
  - Often only handled in an ad-hoc and informal way
- Solution:

An SMT theory for reasoning about memory access correctness

- Precise formalization of `malloc` and `free`
- Opportunities for simplification of formulas using rewrite rules
- Modular and local reasoning

LLBMC — Low-Level Bounded Model Checking

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Embedded devices, systems code, ...
  - Software: Programs written in C/C++
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: bit-precise static analysis

LLBMC — Low-Level Bounded Model Checking

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Embedded devices, systems code, . . .
  - Software: Programs written in C/C++
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: bit-precise static analysis
- Properties checked:
  - Built-in properties: invalid memory access, use-after-free, double free, arithmetic overflow, division by zero, . . .
  - User-supplied properties: `assert` statements

Software Bounded Model Checking

- Property checking of programs is undecidable in general

Software Bounded Model Checking

- Property checking of programs is undecidable in general
- Bugs manifest themselves in finite runs of the program

Software Bounded Model Checking

- Property checking of programs is undecidable in general
- Bugs manifest themselves in finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
    - Restrict number of nested function calls and inline functions
    - Restrict number of loop iterations and unroll loops
  - Property checking becomes decidable using an SMT solver

The LLBMC Approach

- Fully supporting real-life programming languages is cumbersome

The LLBMC Approach

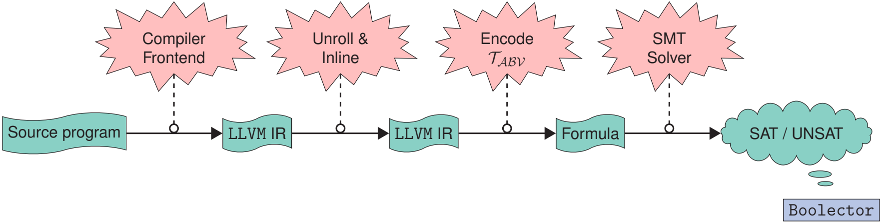
- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++ due to their complex semantics

The `LLBMC` Approach

- Fully supporting real-life programming languages is cumbersome
- Particularly true for `C`/`C++` due to their complex semantics
- Solution: Do not operate on the source code, but use a compiler IR
  - Well-defined, simple semantics
  - Closer to the program that is actually run on the machine
  - Support for wide range of programming languages comes “for free”

The LLBMC Approach

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++ due to their complex semantics
- Solution: Do not operate on the source code, but use a compiler IR
  - Well-defined, simple semantics
  - Closer to the program that is actually run on the machine
  - Support for wide range of programming languages comes “for free”



LLBMC and SMT

- Theory of bitvectors ($\mathcal{T}_{\mathcal{BV}}$) for register operations (ALU):
  - Mostly straightforward, not discussed in this talk

LLBMC and SMT

- Theory of bitvectors ($\mathcal{T}_{\mathcal{BV}}$) for register operations (ALU):
  - Mostly straightforward, not discussed in this talk
- Theory of arrays ($\mathcal{T}_{\mathcal{A}}$) for memory operations:
  - Memory is one big array of bytes
  - Translate `load` and `store` instructions
    - Introduce explicit memory states: $a, a', \ldots$

LLBMC and SMT

- Theory of bitvectors ($\mathcal{T}_{\mathcal{BV}}$) for register operations (ALU):
  - Mostly straightforward, not discussed in this talk
- Theory of arrays ($\mathcal{T}_{\mathcal{A}}$) for memory operations:
  - Memory is one big array of bytes
  - Translate `load` and `store` instructions
    - Introduce explicit memory states: $a, a', \ldots$

```
%x = load i8* %p
store i8* %p, i8 %x
```

$\rightsquigarrow$

$$x = \mathit{read}(a, p)$$
$$\widehat{a} = \mathit{write}(a, p, x)$$

LLBMC and SMT

- Theory of bitvectors ($\mathcal{T}_{\mathcal{BV}}$) for register operations (ALU):
  - Mostly straightforward, not discussed in this talk
- Theory of arrays ($\mathcal{T}_{\mathcal{A}}$) for memory operations:
  - Memory is one big array of bytes
  - Translate `load` and `store` instructions
    - Introduce explicit memory states: $a, a', \ldots$

```
store i8* %p, i8 %x
%x = load i8* %p
```

$\rightsquigarrow$

$$\begin{aligned} \widehat{a} &= \mathit{write}(a, p, x) \\ x &= \mathit{read}(\widehat{a}, p) \end{aligned}$$

Memory Related Program Bugs

- Kinds of memory related bugs:
  - `load` from a non-allocated region of memory
  - `store` to a non-allocated region of memory
  - `free` of a non-allocated region of memory
  - `free` of an already `freed` region of memory
  - ...

Memory Related Program Bugs

- Kinds of memory related bugs:
  - `load` from a non-allocated region of memory
  - `store` to a non-allocated region of memory
  - `free` of a non-allocated region of memory
  - `free` of an already `freed` region of memory
  - ...
- Not handled by $\mathcal{T}_{\mathcal{ABV}}$, but should be detected by `LLBMC`

Memory Related Program Bugs

- Kinds of memory related bugs:
  - `load` from a non-allocated region of memory
  - `store` to a non-allocated region of memory
  - `free` of a non-allocated region of memory
  - `free` of an already `freed` region of memory
  - ...
- Not handled by $\mathcal{T}_{\mathcal{ABV}}$, but should be detected by `LLBMC`
- Need to be handled atop of $\mathcal{T}_{\mathcal{ABV}}$

Encoding Memory Access Correctness in SMT

Karlsruhe Institute of Technology

- Current approach in `LLBMC` (SSV '10):
  - Introduce explicit heap states: $h, h', \ldots$
  - Memory access correctness constraints explicitly encoded in $\mathcal{T}_{\mathcal{ABV}}$
  - Creates huge subformulas for each memory access operation
  - Needs knowledge of the complete heap state history

Encoding Memory Access Correctness in SMT

Karlsruhe Institute of Technology

- Current approach in LLBMC (SSV '10):
  - Introduce explicit heap states: $h, h', \ldots$
  - Memory access correctness constraints explicitly encoded in $\mathcal{T}_{\mathcal{ABV}}$
  - Creates huge subformulas for each memory access operation
  - Needs knowledge of the complete heap state history

$$\texttt{accessible}(h, q, t) \quad \equiv \bigvee_{\substack{h' \preceq h \\ I:\, h' = \texttt{malloc}(\widehat{h'}, p, s)}} c_{\texttt{exec}}(I) \;\wedge\; \neg\texttt{deallocated}(h', h, p) \;\wedge\; \texttt{contained}(p, s, q, t)$$

$$\texttt{deallocated}(h, h', p) \quad \equiv \bigvee_{\substack{h \preceq h^* \preceq h' \\ I:\, h^* = \texttt{free}(\widehat{h^*}, q)}} c_{\texttt{exec}}(I) \;\wedge\; p = q$$

Encoding Memory Access Correctness in SMT

- Current approach in LLBMC (SSV '10):
  - Introduce explicit heap states: $h, h', \ldots$
  - Memory access correctness constraints explicitly encoded in $\mathcal{T}_{\mathcal{ABV}}$
  - Creates huge subformulas for each memory access operation
  - Needs knowledge of the complete heap state history

- Goals of this work:
  - Precise formalization of `malloc` and `free` as an SMT theory
  - Opportunities for simplification of formulas using rewrite rules
  - Modular and local reasoning: partial heap state history suffices

Motivation: McCarthy's Theory of Arrays

McCarthy's *read*-over-*write* axioms:

$$
\begin{aligned}
i = j \quad &\Rightarrow \quad read(write(a, i, x), j) = x \\
i \neq j \quad &\Rightarrow \quad read(write(a, i, x), j) = read(a, j)
\end{aligned}
$$

Motivation: McCarthy's Theory of Arrays

McCarthy's *read*-over-*write* axioms:

$$\begin{aligned} i = j \quad &\Rightarrow \quad \mathit{read}(\mathit{write}(a, i, x), j) = x \\ i \neq j \quad &\Rightarrow \quad \mathit{read}(\mathit{write}(a, i, x), j) = \mathit{read}(a, j) \end{aligned}$$

Reduction to equality logic:

1. Apply axioms in the form of the rewrite rule

$$\mathit{read}(\mathit{write}(a, i, x), j) \quad \rightarrow \quad \mathrm{ITE}(\; i = j,\; x,\; \mathit{read}(a, j)\;)$$

Motivation: McCarthy's Theory of Arrays

McCarthy's *read*-over-*write* axioms:

$$i = j \quad \Rightarrow \quad read(write(a, i, x), j) = x$$
$$i \neq j \quad \Rightarrow \quad read(write(a, i, x), j) = read(a, j)$$

Reduction to equality logic:

1. Apply axioms in the form of the rewrite rule

$$read(write(a, i, x), j) \quad \rightarrow \quad \mathrm{ITE}(\ i = j,\ x,\ read(a, j)\ )$$

2. Replace *reads* by fresh variables, add Ackermann constraints

$$i = j \quad \Rightarrow \quad v_{read(a,i)} = v_{read(a,j)}$$

Motivation: McCarthy's Theory of Arrays

McCarthy's *read*-over-*write* axioms:

$$
\begin{aligned}
i = j \quad &\Rightarrow \quad read(write(a, i, x), j) = x \\
i \neq j \quad &\Rightarrow \quad read(write(a, i, x), j) = read(a, j)
\end{aligned}
$$

Reduction to equality logic:

1. Apply axioms in the form of the rewrite rule

$$read(write(a, i, x), j) \quad \rightarrow \quad \mathrm{ITE}(\ i = j,\ x,\ read(a, j)\ )$$

2. Replace *reads* by fresh variables, add Ackermann constraints

$$i = j \quad \Rightarrow \quad v_{read(a,i)} = v_{read(a,j)}$$

More efficient approaches exist, e.g., `Boolector`'s lemmas-on-demand

Introducing $\mathcal{T}_{\mathcal{H}}$

- Objects of type H encode information about the heap state
  - Contain the whole history of `malloc` and `free` operations
  - Do not encode information about the data stored in the heap
  - Are used to determine memory access correctness

Introducing $\mathcal{T}_{\mathcal{H}}$

- Objects of type H encode information about the heap state
  - Contain the whole history of `malloc` and `free` operations
  - Do not encode information about the data stored in the heap
  - Are used to determine memory access correctness

Functions of type H:

- ε — Heap without any allocation
- $\mathtt{malloc}(h, p, s)$ — Allocate region $[p, p + s)$ (if possible)
- $\mathtt{free}(h, p)$ — De-allocate region starting at p (if currently allocated)

Introducing $\mathcal{T}_{\mathcal{H}}$

- Objects of type H encode information about the heap state
  - Contain the whole history of `malloc` and `free` operations
  - Do not encode information about the data stored in the heap
  - Are used to determine memory access correctness

Functions of type H:

ε	Heap without any allocation
$\texttt{malloc}(h, p, s)$	Allocate region $[p, p + s)$ (if possible)
$\texttt{free}(h, p)$	De-allocate region starting at p (if currently allocated)

Auxilliary functions:

$\texttt{mallocsize}(h, p)$	Size of the memory region starting at p (if currently allocated)

Predicates of $\mathcal{T}_{\mathcal{H}}$

Predicates:

- $\texttt{accessible}(h, p, s)$ Is $[p, p + s)$ contained in an allocated region?
- $\texttt{freeable}(h, p)$ Is p the start of an allocated region?
- $\texttt{mallocable}(h, p, s)$ Can $[p, p + s)$ be allocated?

Predicates of $\mathcal{T}_{\mathcal{H}}$

Predicates:

- $\texttt{accessible}(h, p, s)$ Is $[p, p + s)$ contained in an allocated region?
- $\texttt{freeable}(h, p)$ Is p the start of an allocated region?
- $\texttt{mallocable}(h, p, s)$ Can $[p, p + s)$ be allocated?

The semantics of the predicates is specified using axioms

Axioms for `mallocable`

$\texttt{mallocable}(h, p, s)$ Can $[p, p + s)$ be allocated?

Axioms for `mallocable`

`mallocable`(h, p, s) Can $[p, p+s)$ be allocated?

- $[p, p+s)$ can be allocated if it does not overlap with any allocated memory region

Axioms for `mallocable`

$\texttt{mallocable}(h, p, s)$ Can $[p, p + s)$ be allocated?

- $[p, p + s)$ can be allocated if it does not overlap with any allocated memory region
- Various possible axiomatizations:
  - Stack-like: p needs to be “on the right” of all regions that have been allocated before
  - Axiomatize non-overlap precisely
  - …

Axioms for `freeable`

`freeable`(h, q) Is q the start of an allocated region?

Axioms for `freeable`

$\texttt{freeable}(h, q)$ Is q the start of an allocated region?

$$\texttt{freeable}(\varepsilon, q) \Leftrightarrow \perp$$

Axioms for `freeable`

$\texttt{freeable}(h, q)$ Is q the start of an allocated region?

$$\texttt{freeable}(\varepsilon, q) \Leftrightarrow \bot$$

$$\texttt{mallocable}(h, p, s) \wedge p = q \Rightarrow \texttt{freeable}(\texttt{malloc}(h, p, s), q) \Leftrightarrow \top$$

$$\neg\texttt{mallocable}(h, p, s) \vee p \neq q \Rightarrow \texttt{freeable}(\texttt{malloc}(h, p, s), q) \Leftrightarrow \texttt{freeable}(h, q)$$

Axioms for `freeable`

$\texttt{freeable}(h, q)$ Is q the start of an allocated region?

$$\texttt{freeable}(\varepsilon, q) \Leftrightarrow \bot$$

$$\begin{aligned}
\texttt{mallocable}(h, p, s) \wedge p = q &\Rightarrow \texttt{freeable}(\texttt{malloc}(h, p, s), q) \Leftrightarrow \top \\
\neg\texttt{mallocable}(h, p, s) \vee p \neq q &\Rightarrow \texttt{freeable}(\texttt{malloc}(h, p, s), q) \Leftrightarrow \texttt{freeable}(h, q) \\
\texttt{freeable}(h, p) \wedge p = q &\Rightarrow \texttt{freeable}(\texttt{free}(h, p), q) \Leftrightarrow \bot \\
\neg\texttt{freeable}(h, p) \vee p \neq q &\Rightarrow \texttt{freeable}(\texttt{free}(h, p), q) \Leftrightarrow \texttt{freeable}(h, q)
\end{aligned}$$

Axioms for `mallocsize`

$\texttt{mallocsize}(h, q)$	Size of the memory region starting at q (if currently allocated)

Axioms for `mallocsize`

$\texttt{mallocsize}(h, q)$ Size of the memory region starting at q (if currently allocated)

$$\texttt{mallocsize}(\varepsilon, q) = 0$$

Axioms for `mallocsize`

$\texttt{mallocsize}(h, q)$ Size of the memory region starting at q (if currently allocated)

$$\texttt{mallocsize}(\varepsilon, q) = 0$$

$$\texttt{mallocable}(h, p, s) \wedge p = q \Rightarrow \texttt{mallocsize}(\texttt{malloc}(h, p, s), q) = s$$

$$\neg\texttt{mallocable}(h, p, s) \vee p \neq q \Rightarrow \texttt{mallocsize}(\texttt{malloc}(h, p, s), q) = \texttt{mallocsize}(h, q)$$

Axioms for `mallocsize`

$\texttt{mallocsize}(h, q)$ Size of the memory region starting at q (if currently allocated)

$$\texttt{mallocsize}(\varepsilon, q) = 0$$

$$\begin{aligned}
\texttt{mallocable}(h,p,s) \wedge p = q &\Rightarrow \texttt{mallocsize}(\texttt{malloc}(h,p,s),q) = s \\
\neg\texttt{mallocable}(h,p,s) \vee p \neq q &\Rightarrow \texttt{mallocsize}(\texttt{malloc}(h,p,s),q) = \texttt{mallocsize}(h,q) \\
\\
\texttt{freeable}(h,p) \wedge p = q &\Rightarrow \texttt{mallocsize}(\texttt{free}(h,p),q) = 0 \\
\neg\texttt{freeable}(h,p) \vee p \neq q &\Rightarrow \texttt{mallocsize}(\texttt{free}(h,p),q) = \texttt{mallocsize}(h,q)
\end{aligned}$$

Axioms for `accessible`

`accessible`(h, q, t) Is $[q, q + t)$ contained in an allocated region?

Axioms for `accessible`

$\texttt{accessible}(h, q, t)$ Is $[q, q + t)$ contained in an allocated region?

$$\texttt{accessible}(\varepsilon, q, t) \Leftrightarrow \bot$$

Axioms for `accessible`

$\texttt{accessible}(h, q, t)$ Is $[q, q + t)$ contained in an allocated region?

$$\texttt{accessible}(\varepsilon, q, t) \Leftrightarrow \bot$$

$$\begin{aligned}
\texttt{mallocable}(h, p, s) \ \wedge \ \texttt{contained}(p, s, q, t) \quad &\Rightarrow \quad \texttt{accessible}(\texttt{malloc}(h, p, s), q, t) \Leftrightarrow \top \\
\neg\texttt{mallocable}(h, p, s) \ \vee \ \neg\texttt{contained}(p, s, q, t) \quad &\Rightarrow \quad \texttt{accessible}(\texttt{malloc}(h, p, s), q, t) \\
&\qquad \Leftrightarrow \texttt{accessible}(h, q, t)
\end{aligned}$$

Axioms for `accessible`

$\texttt{accessible}(h, q, t)$ Is $[q, q + t)$ contained in an allocated region?

$$\texttt{accessible}(\varepsilon, q, t) \Leftrightarrow \bot$$

$$\begin{aligned}
\texttt{mallocable}(h, p, s) \wedge \texttt{contained}(p, s, q, t) &\Rightarrow \texttt{accessible}(\texttt{malloc}(h, p, s), q, t) \Leftrightarrow \top \\
\neg\texttt{mallocable}(h, p, s) \vee \neg\texttt{contained}(p, s, q, t) &\Rightarrow \texttt{accessible}(\texttt{malloc}(h, p, s), q, t) \\
&\qquad \Leftrightarrow \texttt{accessible}(h, q, t)
\end{aligned}$$

$$\begin{aligned}
\neg\texttt{freeable}(h, p) &\Rightarrow \texttt{accessible}(\texttt{free}(h, p), q, t) \\
&\qquad \Leftrightarrow \texttt{accessible}(h, q, t) \\
\texttt{freeable}(h, p) \wedge \texttt{disjoint}(p, \texttt{mallocsize}(h, p), q, t) &\Rightarrow \texttt{accessible}(\texttt{free}(h, p), q, t) \\
&\qquad \Leftrightarrow \texttt{accessible}(h, q, t) \\
\texttt{freeable}(h, p) \wedge \neg\texttt{disjoint}(p, \texttt{mallocsize}(h, p), q, t) &\Rightarrow \texttt{accessible}(\texttt{free}(h, p), q, t) \Leftrightarrow \bot
\end{aligned}$$

Implementation

- $\mathcal{T}_{\mathcal{H}}$ is not supported by off-the-shelf SMT solvers (yet?)

Implementation

- $\mathcal{T}_{\mathcal{H}}$ is not supported by off-the-shelf SMT solvers (yet?)
- Reduction to $\mathcal{T}_{\mathcal{BV}}$: Apply axioms in the form of rewrite rules

Implementation

- $\mathcal{T}_{\mathcal{H}}$ is not supported by off-the-shelf SMT solvers (yet?)
- Reduction to $\mathcal{T}_{\mathcal{BV}}$: Apply axioms in the form of rewrite rules

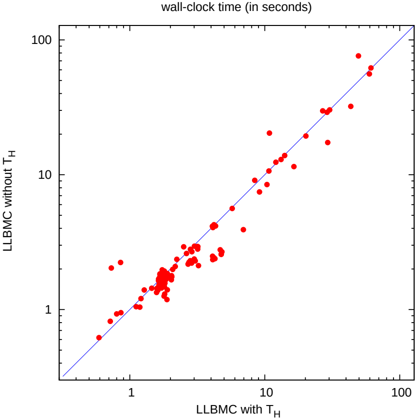
$$
\begin{aligned}
\texttt{mallocable}(h,p,s) \wedge \texttt{contained}(p,s,q,t) &\Rightarrow \texttt{accessible}(\texttt{malloc}(h,p,s),q,t) \Leftrightarrow \top \\
\neg\texttt{mallocable}(h,p,s) \vee \neg\texttt{contained}(p,s,q,t) &\Rightarrow \texttt{accessible}(\texttt{malloc}(h,p,s),q,t) \\
&\qquad \Leftrightarrow \texttt{accessible}(h,q,t)
\end{aligned}
$$

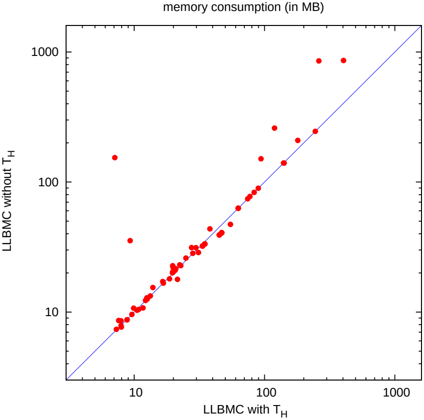
$$\rightsquigarrow$$

$$
\begin{aligned}
\texttt{accessible}(\texttt{malloc}(h,p,s),q,t) \rightarrow \mathrm{ITE}(&\\
&\texttt{mallocable}(h,p,s) \wedge \texttt{contained}(p,s,q,t),\\
&\top,\\
&\texttt{accessible}(h,q,t)\\
)&
\end{aligned}
$$

Evaluation

Comparison of the current approach (SSV '10) and $\mathcal{T}_{\mathcal{H}}$ in LLBMC on 97 C-programs





Details

	#mallocs / #frees	#accessible	time		memory	
			SSV '10	$\mathcal{T}_{\mathcal{H}}$	SSV '10	$\mathcal{T}_{\mathcal{H}}$
sparsemem	129 / 51	8374	76.2	49.5	861	404
binary-tree	127 / 127	3048	7.5	9.1	150	94
inplace-reverse	100 / 100	1800	20.4	10.8	260	119
wcet-bsort100	3 / 0	120204	12.4	12.1	246	246
wcet-statemate	106 / 0	2816	2.2	0.9	35	9

Conclusions and Future Work

Karlsruhe Institute of Technology

- We have presented an SMT theory for reasoning about memory access correctness
  - Precise formalization of `malloc` and `free`
  - Opportunities for simplification of formulas using rewrite rules
  - Modular and local reasoning

Conclusions and Future Work

- We have presented an SMT theory for reasoning about memory access correctness
  - Precise formalization of `malloc` and `free`
  - Opportunities for simplification of formulas using rewrite rules
  - Modular and local reasoning
- Performance is roughly comparable to the current approach in `LLBMC`

Conclusions and Future Work

- We have presented an SMT theory for reasoning about memory access correctness
  - Precise formalization of `malloc` and `free`
  - Opportunities for simplification of formulas using rewrite rules
  - Modular and local reasoning
- Performance is roughly comparable to the current approach in `LLBMC`
- Potential performance increases:
  - Improved rewrite engine/strategy in the reduction
  - Use of rewrite rules that are derived from the axioms

Conclusions and Future Work

Karlsruhe Institute of Technology

- We have presented an SMT theory for reasoning about memory access correctness
  - Precise formalization of `malloc` and `free`
  - Opportunities for simplification of formulas using rewrite rules
  - Modular and local reasoning
- Performance is roughly comparable to the current approach in `LLBMC`
- Potential performance increases:
  - Improved rewrite engine/strategy in the reduction
  - Use of rewrite rules that are derived from the axioms
- Open Questions:

1. Can $\mathcal{T}_{\mathcal{H}}$ be integrated into SMT solvers?
2. Is the lemmas-on-demand approach applicable?