Expressive and Efficient Memory Representation for Bounded Model Checking of C programs

by

Xiang Zhou

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some figures and formulations are taken from Bounded Model Checking for LLVM paper [29] with contributions from Siddharth Priya, Yusen Su, Prof. Yakir Vizel, Dr. Yuyan Bao and Prof. Arie Gurfinkel.

Some of the features in SeaBmc, which SeaM depends on were implemented by Siddharth Priya and Prof. Arie Gurfinkel.

The benchmark Verify-C-Common project [28] contains contributions from Siddharth Priya, Yusen Su, Prof. Yakir Vizel, Dr. Yuyan Bao and Prof. Arie Gurfinkel.

Abstract

Ensuring memory safety in programs has been an important yet difficult topic of research. Most static analysis approaches rely on the theory of arrays to model memory access. The limitation of the theory of arrays in terms of scalability and compatibility with SAT/SMT solvers is well-known, and there has been many attempts at optimizing either the theory itself or memory encodings based on theory of arrays.

In this thesis, we demonstrate that existing arrays-based memory encodings miss potential optimization opportunities by omitting language specific properties such as alignment and pointer arithmetic in C. We present SeaM, a new memory representation for C programs built around a more expressive First-order Theory: the Theory of Memory. We show that by preserving more C language specific rules and properties, the Theory of Memory allows for more thorough optimization methods during eager rewriting of sequences of stores. We introduce two such optimization methods in this thesis. First, we over-approximate pointer comparison with an abstract interpretation-like approach called AddressRangeMap. Second, we compress sequences of stores with Store-Map for faster address offset look-ups.

The new memory representation is implemented in SeaBmc, a new BMC tool for LLVM. We evaluate our approach on real-world bounded model checking tasks from the aws-c-common library and Sv-Comp benchmarks and compare it against two existing memory representations in SeaBmc. Our results show that SeaM outperforms the theory of array based representation and is comparable with the λ based representation.

Acknowledgements

I would like to thank my supervisor Professor Arie Gurfinkel for guiding me in every step of my research projects. I would like to thank my colleagues for aiding me with their domain specific expertise. Lastly I would like to thank my readers Prof. Mahesh Tripunitara and Prof. Meng Xu for their insightful feedback.

Dedication

This is dedicated to my family, friends and Lulu the cat, who are with me during my MASc journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ensuring memory safety has always been one of the most crucial yet tricky aspects of any programming language that manipulates memory. Some high-level programming languages like Java [20] assist programmers with built-in runtime checks against certain memory faults like null pointer dereference and out-of-bound memory access. More recent development in programming language like Rust [23] attempt to achieve balance between high-level memory safety and low-level control with its *ownership* type system, where pointer aliasing is tracked *statically* to prevent direct mutation of aliased state. However, low-level languages often lack built-in memory safety mechanisms like those possessed by Java and Rust. The C language, for example, gives programmers low-level control of memory operations, yet features like arbitrary pointer arithmetic and manual memory management can take years to master, and can lead to catastrophic memory errors if used incorrectly. The lack of built-in compile-time or runtime checking means severe security exploits like *Heartbleed* [21] can go unnoticed for years in some of the most trusted software libraries like OpenSSL.

Over the years, C programmers have gained access to a plethora of 3rd-party tools for avoiding and detecting memory errors. Dynamic Binary Instrumentation (DBI) tools like Valgrind [26] check memory errors at runtime by executing instrumented binary under emulated environment. DBI tools allow for analysis of programs without the source code, yet often suffer from heavy overhead in terms of running time. Memory checks instrumented at compile-time alleviate running-time overhead by executing natively instead of under emulation. One of the most commonly used compile-time instrumentation tools is LLVM's **AddressSanitizer** [32]. The drawback of compile-time instrumentation tools is that they require the source code of program under analysis in order to re-compile under special configurations. The main drawback of all dynamic analysis tools is that their effectiveness rely on the quality of test suites executed. Even with a great test suite, it is still nearly

impossible for dynamic analysis to explore all program states of any complex program. An alternative approach to ensure memory safety is through static analysis techniques like Bounded Model Checking (BMC) and Symbolic Execution (SE) [7]. Many static analysis approaches reduce the problem of program verification to Satisfiability Modulo Theories (SMT) [2] and then eventually to satisfiability (SAT) solving, which allows them to explore more of the program states without handcrafting as many test cases. To verify programs manipulating memory then, we need an expressive SMT theory that can model memory operations as well as efficient decision procedures that reduces the theory to simpler form.

The theory of arrays ($\mathcal{T}_\mathcal{A}$) ([24], [1], [22]) is a common choice for modeling memory. An *Array* sort is defined by its *index* sort σ_I and *element* sort σ_E. An array can be seen as an infinite map from indices of σ_I to elements of σ_E. A *write* to memory is modeled by the $\mathcal{T}_\mathcal{A}$ *write* (store in SMT-LIB2) operation: $write : \sigma_A \times \sigma_I \times \sigma_E \rightarrow \sigma_A$; while a *read* from memory is modeled by the $\mathcal{T}_\mathcal{A}$ *read* (select in SMT-LIB2) operation: $read : \sigma_A \times \sigma_I \rightarrow \sigma_E$. The reduction of formulas with $\mathcal{T}_\mathcal{A}$ terms to equivalent formulas with only underlying theories of σ_I and σ_E along with if-then-else (*ite*) terms.

The basic form of $\mathcal{T}_\mathcal{A}$ still has a few missing pieces for sufficiently modeling real-world C programs. **First**, the transformation of long sequences of *read*-over-*write* (ROW) terms can lead to large nested case-splits. [19] introduced a variety of preprocessing techniques that eliminates ROW terms, including *lazily* delaying the transformation of *read* terms by substituting them with new variables. Similarly, [6] replaces *read*s with variables of another theory (like uninterpreted functions) and tries to solve the formula *lazily*, only adding lemmas when inconsistencies occur between indices equality and read variables equality. A more recent work [18] further reduced the overhead of ROW elimination by grouping writes with *comparable* indices together. Note that [18] inspired much of our work in Chapter 6. **Second**, common memory operations over a *range* of indices like memcpy and memset cannot be represented succinctly by $\mathcal{T}_\mathcal{A}$ *read* and *write* operations alone. Sinz et al.([33], [17], [16]) addressed this issue by extending $\mathcal{T}_\mathcal{A}$ to better model memcpy and memset operations. In their BMC tool LLBMC ([25], [15]), operations like memcpy and memset are directly modeled with succinct λ-terms instead of sequences of chained *write*s. Similar efforts have also been applied on the SMT solver side. Boolector [5], for example, matches patterns representing array operations over a range of consecutive indices and extracts λ-terms from the patterns [27].

Existing optimizations drastically improved the performance of array-based memory encodings. However, few has taken account C-specific memory rules. Specifically, C standard implies that given two *distinct* memory objects o_a and o_b, and two corresponding *legal* pointer arithmetic terms $o_a + x$ and $o_b + y$, then $o_a + x \neq o_b + y$ must hold. We hypothesize that by omitting C language specific properties like such, existing memory encodings could

miss certain simplification opportunities only applicable under C memory model.

In this thesis, we present SeaM, a more expressive memory representation preserving more features of the C language. The first component of the new memory representation is an expressive First-Order Theory called Theory of Memory $\mathcal{T}_{\mathcal{M}}$ based on ROW axioms similar to those in $\mathcal{T}_{\mathcal{A}}$. $\mathcal{T}_{\mathcal{M}}$ preserves features of C language source code such as rules for legal pointer arithmetic. $\mathcal{T}_{\mathcal{M}}$ also contains succinct encoding of batch memory operations like memcpy and memset based on the extended $\mathcal{T}_{\mathcal{A}}$ in [33]. We also present the rewrite rules for transforming formulas with $\mathcal{T}_{\mathcal{M}}$ terms present to formulas containing only *ite* and underlying theory terms.

With memory operations encoded in $\mathcal{T}_{\mathcal{M}}$, we next showcase two optimization techniques enabled by $\mathcal{T}_{\mathcal{M}}$. **First**, we observe that one of the more expensive operations during ROW simplification is the resolution of (dis-)equalities between nested pointer expressions. If left to the SMT/SAT solver, these (dis-)equalities would require expensive ite-pull and ite-push transformations (see Chapter 2 for detail). For faster resolution of pointer (dis-)equality, we over-approximate pointer expressions with an Abstract Interpretation-inspired approach called Address Range Map (ARM), and apply comparisons between abstract values where possible.

Second, we make the observation that pre-processing each ROW term is an operation with *linear* time complexity. With $\mathcal{T}_{\mathcal{M}}$, memory *write* operations into addresses with different unique base address can be safely commuted since they are guaranteed to be disjoint; conversely, *write*s with identical unique base address can be grouped together for faster lookup. We introduce the Store-Map data structure to compress consecutive store operations, and present rewrite rules for *read-over-store-map* with *logarithmic* in best-case scenario.

SeaM is implemented as part of the Verification Engine of SeaBMC, a BMC tool that targets LLVM IR compatible languages like C. We evaluate our approach against existing basic array-based memory representation and lambda-based memory representation in SeaBMC, on the BMC tasks targeting real-world C code from the Verify-C-Common benchmark. Our current results show that $\mathcal{T}_{\mathcal{M}}$-based memory representation drastically outperforms array based approach in terms of verification time. We further observe that over-approximating pointer comparisons with ARM resulted in more pointer (dis-)equality resolutions, effectively reducing formula size and verification time. We also evaluate SeaM on selected verification tasks from Sv-Comp that contain longer sequences of ROW terms. This set of experiments show that SeaM achieves similar level of simplification as λ-based encoding with lower overhead in terms of pre-processing time. The reduction in pre-processing overhead is even more pronounced with store compression using Store-Map.

Chapter 2

Background

This chapter serves as a basis for understanding the rest of the thesis. In this chapter, we first summarize rewrite rules and procedures for ite-terms, comparison terms and Boolean terms. This chapter describes the axoims and rules that establishes the basic form of the Theory of Arrays. Next, we present an extension of $\mathcal{T}_{\mathcal{A}}$ that allows for more succinct representation of batch memory operations like memcpy and memset. SeaM is implemented in SeaBmc, a BMC tools for LLVM IR. We briefly summarize the design and features of SeaBmc for modeling memory and verifying memory safety properties.

2.1 Basic rewrite rules

In this thesis, we frequently use *rewrite rules* to define semantics and simplification rules. For rest of the thesis, we use the notation $\vdash_P \varphi$ to indicate that the formula φ is deduced to be valid by some proof system P, and $\vdash \varphi$ when the proof system P is irrelevant. We write $\models \varphi$ to indicate that φ is valid (but not necessarily provable). A rewrite rule

$$\frac{\vdash a}{t_1 \rightsquigarrow t_2}$$

means if a is proved to be valid, then the term t_1 can be rewritten to t_2. This means that any formula ϕ containing t_1 is equivalent to the formula $\phi[t_1/t_2]$. For the rest of this thesis, when presenting axioms and rewrite rules, we assume that all free variables are implicitly universally quantified. For example, we write $\psi(a)$ to mean $\forall a \,.\, \psi(a)$.

$$\frac{}{f(ite(c,x,y)) \;\rightsquigarrow\; ite(c,f(x),f(y))}\text{ ITE-PUSH}$$

$$\frac{}{ite(c,f(x),f(y)) \;\rightsquigarrow\; f(ite(c,x,y))}\text{ ITE-PULL}$$

$$\frac{\vdash c}{ite(c,x,y) \rightsquigarrow x} \qquad \frac{\vdash \neg c}{ite(c,x,y) \rightsquigarrow y}$$

$$\frac{}{ite(c,x,y) = w \;\rightsquigarrow\; ite(c, x = w, y = w)}\text{ ITE-EQ-1}$$

$$\frac{}{ite(c,x,y) = ite(c,u,w) \;\rightsquigarrow\; ite(c, x = u, y = w)}\text{ ITE-EQ-2}$$

$$\frac{\vdash x = y}{ite(c,x,y) \;\rightsquigarrow\; x} \qquad \frac{\vdash x = \top \;\; \vdash y = \bot}{ite(c,x,y) \;\rightsquigarrow\; c} \qquad \frac{\vdash x = \bot \;\; \vdash y = \top}{ite(c,x,y) \;\rightsquigarrow\; \neg c}$$

Figure 2.1: Rewrite rules for *ite* terms.

In the rest of this thesis, we use *ite*-terms in the form of $ite(c, a, b)$ as a shorthand for *if* c *then* a *else* b. Figure 2.1 defines the semantics of ite-terms with rewrite rules. Given an arbitrary function f, **ITE-PUSH** pushes f into an ite-term, and **ITE-PULL** lifts f from an ite-term. While the rewrite rules are shown for unary functions only, they easily extend to functions and predicates of arbitrary arity. As an example, we show rewrite rules for equality (a binary predicate) over ite-terms in **ITE-EQ** rules.

Reducing ite-terms A common problem in compiling expressions down to satisfiability solvers is to reduce the formulas to simpler expressions that can be represented effectively in CNF. Here, we describe the procedure for ite-terms.

Let φ be a formula containing an ite-term t of the form $f(ite(c, x, y))$. Construct a formula ψ as follows: $\psi = (\varphi[t \mapsto f(w)] \wedge (c \Rightarrow w = x) \wedge (\neg c \Rightarrow w = y))$, where w is a fresh variable not used in φ. Then, φ is SAT iff ψ is SAT, and ψ is approximately same size as φ: $|\varphi| \approx |\psi|$. The encoding is similar to Tseytin transformation [34] commonly used to encode arbitrary formulas to equisatisfiable CNF. The transformation is formalized by

the following rewrite rule:

$$\frac{t = f(ite(c,x,y)) \quad w \text{ fresh in } \varphi}{\varphi[t] \;\rightsquigarrow\; (\varphi[t \mapsto f(w)] \wedge (c \Rightarrow w = x) \wedge (\neg c \Rightarrow w = y))} \tag{2.1}$$

Note that while the transformation does not increase the size of the formula, it does introduce additional variables (namely w). This, in turn, complicates potential simplifications.

Comparison $(=, \leq)$ terms are crucial for representing array operations with *ite*-terms by forming *index comparison*. The details are covered in Sec. 2.2. Figure 2.2 presents some basic rewrite rules for comparison terms.

$$\frac{\vdash a = b \quad \bowtie \in \{\leq, =\}}{(a \; op \; p) \bowtie (b \; op \; q) \;\rightsquigarrow\; p \bowtie q}$$

$$\frac{\vdash a = b}{a = b \;\rightsquigarrow\; \top} \qquad \frac{\vdash a \neq b}{a = b \;\rightsquigarrow\; \bot}$$

$$\frac{}{a \neq b \;\rightsquigarrow\; \neg(a = b)}$$

Figure 2.2: Example rewrite rules for comparison terms.

In order to perform simplifications on *ite*-terms, it is necessary to first perform simplifications on the Boolean terms forming the *condition* part. Figure 2.3 presents some example rewrite rules for Boolean terms containing negations. Note that the rule **NEG-PUSH** is a variation of **ITE-PUSH**.

$$\frac{}{\neg(\neg(a)) \;\rightsquigarrow\; a} \qquad \frac{}{\neg(ite(c,x,y)) \;\rightsquigarrow\; ite(c,\neg x,\neg y)}\,\text{NEG-PUSH}$$

$$\frac{\vdash a = \top}{\neg(a) \;\rightsquigarrow\; \bot} \qquad \frac{\vdash a = \bot}{\neg(a) \;\rightsquigarrow\; \top}$$

Figure 2.3: Example rewrite rules for Boolean terms.

Arithmetic $(+, -)$ terms are also important for representing index arithmetics. Sec. 2.2 also shows index arithmetics representing the *range* of batch array operations. Figure 2.4 shows two rewrite rules for arithmetics containing addition $(+)$ operation.

$$\frac{\vdash +(b, c) = d}{+(a, b, c) \rightsquigarrow +(a, d)} \text{ADD-COMPRESS}$$

$$\frac{}{+(a, +(b, c)) \rightsquigarrow +(a, b, c)} \text{ADD-FLATTEN}$$

Figure 2.4: Rewrite rules for addition arithmetic terms.

Note that apart from the purely syntactic rules, most of the rewrite rules presented above require certain assumption(s) to be true under some proof system. One of the more common assumptions take the form of (dis-)equality $(=, \neq)$. In Sec. 2.2 we show the importance of (dis-)equality between array indices. In Chapter 4 we show how preserving C memory semantics can serve as a proof system for deciding array-like index (dis-)equality checks. Furthermore, in Chapter 5 we show another proof system for index (dis-)equality checks based on abstract interpretation.

2.2 Theory of Arrays

Theory of arrays $(\mathcal{T}_{\mathcal{A}})$ is commonly used to represent memory operations. The classic theory of arrays was first introduced by McCarthy [24] with the following signature:

$$\Sigma_{\mathcal{A}} = \{read, write, =\}$$

$$read : \sigma_A \times \sigma_I \rightarrow \sigma_E$$

$$write : \sigma_A \times \sigma_I \times \sigma_E \rightarrow \sigma_A$$

note that the binary function *equality* $(=)$ is only defined between *element terms* and *index terms* in non-extensional $\mathcal{T}_{\mathcal{A}}$. $\mathcal{T}_{\mathcal{A}}$ defines an array sort σ_A with the element sort σ_E and the index sort σ_I of the array. The formation rules of terms in $\mathcal{T}_{\mathcal{A}}$ and syntax of *read* and *write* are defined in Figure 2.5, where the detailed formation rules of index terms $t_{\mathcal{I}}$ and element terms $t_{\mathcal{E}}$ are determined by the underlying theories $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{I}}$.

$$
\begin{array}{lcll}
t_{\mathcal{I}} & ::= & ... & \textit{index terms} \\
t_{\mathcal{E}} & ::= & ... \mid \mathit{read}(t_{\mathcal{A}}, t_{\mathcal{I}}) & \textit{element terms} \\
t_{\mathcal{A}} & ::= & a \mid \mathit{write}(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}}) & \textit{array terms}
\end{array}
$$

Figure 2.5: Term formation rules of $\mathcal{T}_{\mathcal{A}}$, a is an σ_A constant.

The semantics of *read* and *write* is defined by the following axioms:

$$\forall i, j \;.\; i = j \Rightarrow \mathit{read}(a, i) = \mathit{read}(a, j) \tag{A1}$$

$$\forall i, j \;.\; i = j \Rightarrow \mathit{read}(\mathit{write}(a, j, v), i) = v \tag{A2}$$

$$\forall i, j \;.\; i \neq j \Rightarrow \mathit{read}(\mathit{write}(a, j, v), i) = \mathit{read}(a, i) \tag{A3}$$

Axoim (A1) states that reading from the same array a at the same index shall yield the same element. Axoims (A2) and (A3) assert the rules for *read*-over-*write*. Given a modified array $\mathit{write}(a, j, v)$, (A2) states that if a *read* over the same index as the previous *write*, then the *read* yields the written element; if the *read* is over a different index as the previous modification, the *read* shall always yield the element at index i of the unmodified array a.

Extensionality Note that with the above three axioms, it is only possible to compare the equality between *elements* of arrays. To compare equality between arrays themselves, we need to add the axiom of extensionality:

$$\forall a, b, i \;.\; a = b \Leftrightarrow \mathit{read}(a, i) = \mathit{read}(b, i) \tag{A4}$$

SeaBmc, the BMC tools in which the new memory representation is implemented, uses only non-extensional theory of arrays. For the rest of this thesis, we write $\mathcal{T}_{\mathcal{A}}$ to refer to the *non-extensional* theory of arrays.

Based on the *read*-over-*write* axioms, Figure 2.6 shows rewrite rules for $\mathcal{T}_{\mathcal{A}}$ *read-over-write* expressions that replace *write* terms with *ite*s.

With the basic form of $\mathcal{T}_{\mathcal{A}}$ presented above, each *write* operation can only represent modification of array contents one index at a time. Given consecutive writes to memory or batch memory operations such as memset and memcpy over n indices then, $\mathcal{T}_{\mathcal{A}}$ represents the operations with sequences of *writes* with size growing linearly to n. This behaviour creates a significant performance bottleneck at both the simplification stage and the SMT/SAT solving stage. Furthermore, it would be impossible to accurately represent batch memory operations with n of *variable* (symbolic) value. In the next section, we present an alternative way to encode batch memory operations with λ-expressions. Chapter 6 also presents a potential approach to compress consecutive writes to memory.

$$\frac{}{read(write(a,i,v),j) \;\rightsquigarrow\; ite(i=j,v,read(a,j))} \qquad \frac{\vdash i=j}{read(write(a,i,v),j) \;\rightsquigarrow\; v}$$

$$\frac{\vdash i \neq j}{read(write(a,i,v),j) \;\rightsquigarrow\; read(a,j)}$$

Figure 2.6: Rewrite rules for $\mathcal{T}_{\mathcal{A}}$ *read*-over-*write* expressions.

2.3 Encoding set/copy memory operations

Work from Falke ([17], [16]) introduced theory of arrays with set and copy operations ($\mathcal{T}_{\mathcal{ASC}}$). $\mathcal{T}_{\mathcal{ASC}}$ extends $\mathcal{T}_{\mathcal{A}}$ with additional signatures that describe array operations over a range of size term $t_{\mathcal{S}}$ with sort σ_S:

$$\Sigma_{\mathcal{ASC}} = \Sigma_{\mathcal{A}} \cup \{copy, set, +, -, \leq, <\}$$

$$set : \sigma_A \times \sigma_I \times \sigma_E \times \sigma_S \rightarrow \sigma_A$$

$$copy : \sigma_A \times \sigma_I \times \sigma_A \times \sigma_I \times \sigma_S \rightarrow \sigma_A$$

Note the addition of arithmetic operations $(+, -)$ and comparison operations $(\leq, <)$ to the signature. They are defined between terms of σ_I and σ_S sorts to represent contiguous range of indices. The underlying theories of σ_I and σ_S are therefore numeric such as the theory of BitVectors ($\mathcal{T}_{\mathcal{BV}}$).

The term formation rule of $\mathcal{T}_{\mathcal{ASC}}$ is also expanded, shown in Figure 2.7

$t_{\mathcal{I}}$	::=	...		*index terms*
$t_{\mathcal{S}}$	::=	...		*size terms*
$t_{\mathcal{E}}$	::=	... $\mid$	$read(t_{\mathcal{A}}, t_{\mathcal{I}})$	*element terms*
$t_{\mathcal{A}}$	::=	a $\mid$	$write(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}})$	*array terms*
		$\mid$	$set(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{E}}, t_{\mathcal{S}})$	
		$\mid$	$copy(t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{A}}, t_{\mathcal{I}}, t_{\mathcal{S}})$	

Figure 2.7: Term formation rules of $\mathcal{T}_{\mathcal{ASC}}$

Informally, a $\mathsf{set}(a, p, v, s)$ operation sets all elements of a with indices in the range between p and $p + s - 1$ to the value v; a $\mathsf{copy}(a, p, b, q, s)$ writes the values of elements in

array b with indices between q and $q+s-1$ to elements in array a with indices between p and $p+s-1$. Formally, the semantics of a *read* over set/copy is defined by the following axioms:

$$\forall p, i, s \;.\; p \leq i < p+s \Rightarrow read(set(a,p,v,s),i) = v \tag{A5}$$

$$\forall p, i, s \;.\; \neg(p \leq i < p+s) \Rightarrow read(set(a,p,v,s),i) = read(a,i) \tag{A6}$$

$$\forall p, i, s \;.\; p \leq i < p+s \Rightarrow read(copy(a,p,b,q,s),i) = read(b,q+(i-p)) \tag{A7}$$

$$\forall p, i, s \;.\; \neg(p \leq i < p+s) \Rightarrow read(copy(a,p,b,q,s),i) = read(a,i) \tag{A8}$$

Based on axioms A5-A8, we can devise rewrite rules for *read*-over-set (Figure 2.8) and *read*-over-copy (Figure 2.9) that eliminate set and copy terms.

$$read(set(a,p,s,v),i) \;\rightsquigarrow\; ite(p \leq i < (p+s), v, read(a,i))$$

$$\frac{\vdash p \leq i < (p+s)}{read(set(a,p,s,v),i) \;\rightsquigarrow\; v} \qquad \frac{\vdash \neg(p \leq i < (p+s))}{read(set(a,p,s,v),i) \;\rightsquigarrow\; read(a,i)}$$

Figure 2.8: Rewrite rules for *read*-over-set

$$read(copy(a,p,b,q,s),i) \;\rightsquigarrow\; ite(p \leq i < (p+s), read(b,q+(i-p)), read(a,i))$$

$$\frac{\vdash p \leq i < (p+s)}{read(copy(a,p,b,q,s),i) \;\rightsquigarrow\; read(b,q+(i-p))}$$

$$\frac{\vdash \neg(p \leq i < (p+s))}{read(copy(a,p,b,q,s),i) \;\rightsquigarrow\; read(a,i)}$$

Figure 2.9: Rewrite rules for *read*-over-copy

$\mathcal{T}_{\mathcal{ASC}}$ enables more succinct and precise encoding of memory state of programs containing batch memory operations like set and copy. The improvement is especially crucial for applications like BMC, which alternatively treats set and copy like iterations with basic $\mathcal{T}_{\mathcal{A}}$ encoding. With a finite limit on unrolling, soundness of the verification is compromised when the size of address range is symbolic.

2.4 Encoding Arrays with λ-expressions

Instead of eagerly rewriting *read*-over-array expressions, it is also possible to simulate array operations using λ-expressions. In fact, all σ_A terms can be expressed using a λ-expression $\lambda i.s$, where i is a bound variable of sort σ_I and s is an expression of sort σ_E.

Based on *read*-over-*write* axioms, $write(a, j, v)$ can be simulated with a λ-term:

$$write(a, j, v) \;\rightsquigarrow\; \lambda i.ite(i = j, v, read(a, j))$$

Similarly, set and copy can also be rewritten:

$$set(a, p, v, s) \;\rightsquigarrow\; \lambda i.ite(p \leq i < (p + s), v, read(a, i))$$

$$copy(a, p, b, q, s) \;\rightsquigarrow\; \lambda i.ite(p \leq i < (p + s), read(b, q + (i - p)), read(a, i))$$

A *read* over σ_A term is then just applying β-reduction over $\lambda i.s$:

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

The $\lambda-$based encoding allows for aggressive simplification of memory state expressions through beta-reduction during the *creation* / *update* stage of arrays.

In our new memory representation, the new theory of memory $\mathcal{T}_{\mathcal{M}}$ handles set and copy operations in C programs (memcpy, memset) largely the same as $\mathcal{T}_{\mathcal{ASC}}$. Improving upon $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{ASC}}$, however, $\mathcal{T}_{\mathcal{M}}$ preserves C memory semantics to enable more conclusive index comparisons and creates possibility for techniques like ARM and Store-Map.

2.5 SeaBmc

In this section, we provide an overview of SeaBmc, with a focus on how it models memory operations.

SEA-IR SeaBmc is a BMC tool developed for LLVM-based languages; input programs under verification are first translated into the intermediate representation (IR) of LLVM called *bitcode*. Next, SeaBmc extends raw LLVM *bitcode* input into an IR with explicit dependency information between memory operations called SEA-IR. A simplified subset of the SEA-IR grammar is defined in Figure 2.10.

A well-formed SEA-IR program (PR) must contain a single entry function named `main`. A `main` function must contain one or more basic blocks (BB). Each BB consists of a label (L), zero or more `PHI`-statements which represents control flow, one or more statements (S) and ends with a branch (`br`) statement or `halt`. The last BB must end with `halt` marking program termination. Registers are separated into three types: scalar registers `R`, pointer registers `P` and memory registers `M`. Scalar registers store primitive data type values like integers. Pointer registers store pointer values returned by memory allocations or by reading from memory (pointer to pointer). Memory registers represent memory states returned by either fresh allocations or writing to existing memory. For encoding C programs, SEA-IR also includes the signatures `memcpy` : $\mathbf{P}, \mathbf{P}, \mathbf{R}, \mathbf{M}, \mathbf{M} \rightarrow \mathbf{M}$ and `memset` : $\mathbf{P}, \mathbf{R}, \mathbf{R}, \mathbf{M} \rightarrow \mathbf{M}$ for representing C `memcpy`, `memset` functions. Verification users can define constraints on scalar values with `assume` and express assertions with `assert`.

Memory model in SeaBmc It is common practice for software verification tools to use a single *flat* array to represent the memory state of a program. However, many language semantics treat memory objects from different allocation site as disjoint regions in memory. Instead of the *flat* memory model, SeaBmc partitions all memory objects into disjoint sets of memory regions using alias analysis. To aid program analysis, SEA-IR follows the MemorySSA [8] scheme in which memory operations are *pure*. Allocating a fresh object (`alloca` / `malloc`) or writing to existing memory (`store`) defines new memory registers; in other words, memory write operations have no *side effects* on existing defined memory registers. Read statements (`load`) are also annoted with the source memory registers.

Verification Condition Generation (VCGen) After performing a series of further transformations on the input SEA-IR program SeaBmc produces program with following features:

- single assertion: only contains one `assert` statement at the end of the last basic block;
- single assumption: only contains one `assume` statement before the sole `assert` statement;
- pure data flow: `PHI` branching statements are replaced with `select` statements (semantic same as ITE), the program contains a single basic block with all branching paths merged with `select` (ITE) statements

$$
\begin{array}{lll}
\textbf{PR} & ::= & \texttt{fun main()}\{\textbf{BB}^+\} \\
\textbf{BB} & ::= & \textbf{L} \ : \ \textbf{PHI}^* \ \ \textbf{S}^+ \ (\textbf{BR} \ \mid \ \texttt{halt}) \\
\textbf{BR} & ::= & \texttt{br}\ \textbf{E},\ \textbf{L},\ \textbf{L} \ \mid \ \texttt{br}\ \textbf{L} \\
\textbf{PHI} & ::= & \textbf{R} = \texttt{phi}\ [\textbf{R},\ \textbf{L}](,\ [\textbf{R},\ \textbf{L}])^* \\
 & \mid & \textbf{M} = \texttt{phi}\ [\textbf{M},\ \textbf{L}](,\ [\textbf{M},\ \textbf{L}])^* \\
 & \mid & \textbf{P} = \texttt{phi}\ [\textbf{P},\ \textbf{L}](,\ [\textbf{P},\ \textbf{L}])^* \\
\textbf{S} & ::= & \textbf{RDEF} \ \mid \ \textbf{MDEF} \ \mid \ \textbf{VS} \\
\textbf{RDEF} & ::= & \textbf{R} = \textbf{E} \ \mid \ \textbf{P}, \textbf{M} = \texttt{alloca}\ \textbf{R}, \textbf{M} \\
 & \mid & \textbf{P}, \textbf{M} = \texttt{malloc}\ \textbf{R},\ \textbf{M} \ \mid \ \textbf{R} = \texttt{load}\ \textbf{P},\ \textbf{M} \\
 & \mid & \textbf{P} = \texttt{load}\ \textbf{P},\ \textbf{M} \ \mid \ \textbf{M} = \texttt{free}\ \textbf{P},\ \textbf{M} \\
\textbf{MDEF} & ::= & \textbf{M} = \texttt{store}\ \textbf{R},\ \textbf{P},\ \textbf{M} \ \mid \ \textbf{M} = \texttt{store}\ \textbf{P},\ \textbf{P},\ \textbf{M} \\
 & \mid & \textbf{M} = \texttt{memset}\ \textbf{P},\ \textbf{R},\ \textbf{R},\ \textbf{M} \\
 & \mid & \textbf{M} = \texttt{memset}\ \textbf{P},\ \textbf{R},\ \textbf{P},\ \textbf{M} \\
 & \mid & \textbf{M} = \texttt{memcpy}\ \textbf{P},\ \textbf{P},\ \textbf{R},\ \textbf{M},\ \textbf{M} \\
\textbf{VS} & ::= & \texttt{assume}\ \textbf{R} \ \mid \ \texttt{asssert}\ \textbf{R}
\end{array}
$$

Figure 2.10: Simplified grammar of SEA-IR, where `E`, `L` `R`, `P` and `M` are expressions, labels, scalar registers, pointer registers and memory registers, respectively. Credit to Siddharth Priya.

The transformed SEA-IR program is then translated into verification conditions (VC) in SMT-LIB2 with a function $sym : \texttt{IR} \rightarrow \text{VC}$. We use *scalr* and *ptrs* to represent the sorts of variables translated from SEA-IR scalar and pointer registers. The underlying sort of *scalr* and *ptrs* are usually BitVectors (bv). The sort of translated memory registers is defined as $mems : ptrs \rightarrow bv(n)$, where n is the same as underlying BitVector of *ptrs*. *sym* encodes the SEA-IR AST in bottom up fashion. First, SEA-IR registers `M`n, `R`n or `P`n are translated to variables or constants of corresponding sort m_n, r_n or p_n. Second, each expression `E` is translated into a SMT-LIB2 expression $sym(\texttt{E})$. The details of expression-wise translation are omitted since most of them are trivial. Each statement `S` is translated into a Boolean SMT-LIB2 formula, the most common form being an equality(=) predicate. Finally, a single SMT-LIB2 formula is produced from the conjunction of *sym* results of all statements. Some of the statement-wise semantics of *sym* is presented in Figure 2.11.

Memory allocation The *sym* translation of stack allocation (`alloca`) and heap allocation (`malloc`) returns a conjunction of equality predicates each defining contraints on new pointer and new memory. The constraint on the new pointer depends on the implementation of *memory allocator* denoted by function *alloc*, which takes allocation type

$$sym(\texttt{R} = \texttt{E}) \triangleq r = e \qquad sym(\texttt{assume R}) \triangleq r \qquad sym(\texttt{assert R}) \triangleq \neg r$$

$$sym(\texttt{M1} = \texttt{store R1, P2, M0}) \triangleq m_1 = write(m_0,\ r_1,\ p_2)$$

$$sym(\texttt{M1} = \texttt{memset P1, R1, R2, M0}) \triangleq m_1 = memset(m_0,\ p_1,\ r_1,\ r_2)$$

$$sym(\texttt{M1} = \texttt{memset P1, R1, P2, M0}) \triangleq m_1 = memset(m_0,\ p_1,\ r_1,\ p_2)$$

$$sym(\texttt{M2} = \texttt{memcpy P1, P2, R1, M0, M1}) \triangleq m_2 = memcpy(m_0,\ p_1,\ m_1,\ p_2,\ r_1)$$

$$sym(\texttt{R1} = \texttt{load P0, M}) \triangleq p_1 = read(m,\ p_0)$$

$$sym(\texttt{P1, M1} = \texttt{alloca R0, M0}) \triangleq p_1 = alloc(\texttt{alloca R0, M0}) \wedge m_1 = m_0$$

$$sym(\texttt{P1, M1} = \texttt{malloc R0, M0}) \triangleq p_1 = alloc(\texttt{malloc R0, M0}) \wedge m_1 = m_0$$

Figure 2.11: Definition of some *sym* semantics for translating parts of **VC**, **RDEF** and **MDEF**.

(heap, stack), allocation size and source memory register and returns a pointer. *Memory allocator* places important language specific constraints on freshly allocated pointers. For example, pointer addresses are aligned to word bit width; heap addresses and stack addresses are disjoint, etc.

Memory representations in SeaBmc Note that in Figure 2.11, the detail semantics of *sym* on memory operations `store`, `select` , `memset` and `memcpy` are abstracted in functions *read*, *write*, *memset* and *memcpy* respectively. The detailed semantics of *sym* on these memory operations are encapsulated in *memory representations*. In SeaBmc, currently there are two memory representations:

1. **Arrays**: Memories are modeled after $\mathcal{T}_{\mathcal{A}}$ as described in Sec. 2.2. SEA-IR expressions memory operations are translated to $\mathcal{T}_{\mathcal{A}}$ expressions following signatures of ArrayEx[1]. Arrays handles wide memory operations like *memset* and *memcpy*, with nested Array `store`s as shown in Figure 2.12. This is not sound when BMC unroll bound is insufficient or when range of operation is *symbolic*.

[1] http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml

	ARRAYS	LAMBDAS
$read(m, p_0)$	`select` m p_0	$m(p_0)$
$write(m_0, r_1, p_2)$	`store` m_0 r_1 p_2	$\lambda x.ite(x = p_2, r_1, m_0(x))$

$memset(m_0,\ p_1,\ r_1,\ r_2)$	`(store`
	`  (store`
	`  ...`
	`  (store` m_0 r_2
	`  (+` p_1 `(-` r_1 `1)))`
	`  ...`
	`  ` r_2 `(+` p_1 `1))`
	r_2 `)` p_1
		$\lambda x.ite(p_1 \leq x < (p_1 + r_1),$
		$r_2, m_0(x))$

$memcpy(m_0,\ p_1,\ m_1,\ p_2,\ r_1)$	`(store`
	`  (store`
	`  ...`
	`  (store` m_0
	`  (select` m_1 `(+` p_2 `(-` r_1 `1)))`
	`  (+` p_1 `(-` r_1 `1)))`
	`  ...`
	`  (select` m_1 `(+` p_2 `1))` `(+` p_1 `1))`
	`(select` m_1 p_2`)` p_1
		$\lambda x.ite(p_1 \leq x < (p_1 + r_1),$
		$m_1(p_2 + (x - p_1)), m_0(x))$

Figure 2.12: *sym* translation of memory operations in LAMBDAS and ARRAYS.

2. **LAMBDAS**: Memories are modeled with λ-functions as described in Sec. 2.4. As shown in Figure 2.12, writing to memory $write(m_0, r_1, p_2)$ is simulated with a λ function $\lambda x.ite(x = p_2, r_1, m_0(x))$; reading from memory $read(m, p_0)$ is translated by applying a pointer value as argument to a λ function $m(p_0)$. Formulas can be simplified with β-reduction, resulting in final VC containing *ite*s only, eliminating the need of ArrayEx support in SMT solver. λ-functions provide *sound* encoding for *memset* and *memcpy*.

SEABMC also applies techniques like shadow memory and fat pointers to verify both temporal and spatial memory safety, detailed in [29]. These techniques are applied at a higher level than memory representation and are thus out of the scope of this thesis. Neither of the existing memory representations in SEABMC retain much information from C memory semantics, as detailed in Chapter 3 and Chapter 4. The contributions in this thesis presents an alternative memory representation SEAM for SEABMC. SEAM explores possibilities for improving VC simplification performance by preserving C semantics and implementing custom optimization techniques during VCGen.

Chapter 3

Overview

This chapter provides an overview of the approaches used in SEAM using a series of examples. Consider sample C program P_1 in Figure 3.1.

```
typedef struct { int a; int b; } s;
int main() {
  int *num = malloc(sizeof(int)); // obj8
  *num = 4;
  s *st = malloc(sizeof(s));      // obj16
  st->a = 21, st->b = 42;
  int *p = nd_bool() ? &(st->a) : &(st->b);
  *p = 4;
  assert(*num == st->b); // read-over-write
  return 0;
}
```

Figure 3.1: Sample C program containing memory operations P_1.

The program initializes an integer num and a structure st containing two integer fields. It non-deterministically writes the same value in num into one of the fields in st. Figure 3.2 shows the memory state of P_1 denoted by $\texttt{mem}_1$ encoded with $\mathcal{T}_\mathcal{A}$ after the execution of line 17. For demonstration, we use a fresh variable obj_n to represent a pointer to allocated memory objects.

```
mem1 = (store
          (store
            (store
              (store mem0 obj8 4)
              obj16 21)
            (+ obj16 8) 42)
          (ite bool_a obj16 (+ obj16 8)) 4)
```

Figure 3.2: Memory state of P_1 after line 17 encoded in $\mathcal{T}_{\mathcal{A}}$.

Note that the final assertion on line 18 depends on two reads on $\texttt{mem}_1$: first read ($*\texttt{num}$) is on the address $\texttt{obj}_8$; second read is on the address $\texttt{obj}_{16} + 8$. According to the *read*-over-*write* axioms (A2), (A3) and rewrite rules in Figure 2.6, the two *read*-over-*write* expressions can be eagerly rewritten into *ite*-expressions in Figure 3.3a and Figure 3.4a.

```
(select mem1 obj8) =
(ite
  (=
    obj8
    (ite bool_a obj16 (+ obj16 8)))
  4
  (ite
    (= obj8 (+ obj16 8))
    42
    (ite
      (= obj8 obj16)
      21
      (ite
        (= obj8 obj8)
        4
        (select mem0 obj8)
        )
      )
    )
  )
```

(a) Full expression of after rewrite into *ite*.

```
(select mem1 obj8) =
(ite false 4 4) = 4
```

(b) Simplified expression.

Figure 3.3: $\mathcal{T}_{\mathcal{A}}$ expression of *read*-over-*write* operation $*\texttt{num}$.

```
(select mem_1 (+obj_16 8))
=
(ite
  (=
    (+obj_16 8)
    (ite bool_a obj_16 (+ obj_16 8)))
  4
  (ite
    (= (+obj_16 8) (+ obj_16 8))
    42
    (ite
      (= (+obj_16 8) obj_16)
      21
      (ite
        (= (+obj_16 8) obj_8)
        4
        (select mem_0 (+obj_16 8))
        )
      )
    )
  )
```

(a) Full expression of after rewrite into *ite*.

```
(select mem_1 (+obj_16 8))
=
(ite
  (ite
    bool_a
    (= (+obj_16 8) obj_16)
    (= (+obj_16 8) (+obj_16 8)))
  4
  42)
=
(ite
  (ite bool_a false true)
  4
  42)
= (ite (not bool_a) 4 42)
```

(b) Simplified expression.

Figure 3.4: $\mathcal{T}_{\mathcal{A}}$ expression of *read*-over-*write* operation `st->b`.

Any reasoning or simplification on the rewritten *ite*-expressions depend heavily on (dis-)equality checks between each pair of *read* (select) and *write* (store) indices. Certain checks can be resolved with trivial *syntactic* check, for example $\texttt{obj}_8 = \texttt{obj}_8$ is *True* and $\texttt{obj}_1\texttt{6}+8 = \texttt{obj}_{16} + \texttt{8}$ is *False*. However, (dis-)equality checks like $\texttt{obj}_8 = (\texttt{obj}_{16}+8)$ would be *inconclusive* if address bases like $\texttt{obj}_8$ and $\texttt{obj}_{16}$ are *symbolic* variables. SeaBmc optionally uses concrete numbers to represent address bases instead, which can result in more conclusive (dis-)equality checks. Still, this approach would fail to resolve (dis-)equalities where *offsets* are *symbolic*: consider the (dis-)equality check $(\texttt{obj}_8+x) = (\texttt{obj}_{16}+y)$. Also, using concrete numbers for address bases result in loss of generality in terms of allocation schemes. However, such (dis-)equality check should be conclusive if C semantics is taken into consideration: any *legal* pointer arithmetic operations on pointers from different objects will *never* be equal. Hence $(\texttt{obj}_8 + x) = (\texttt{obj}_{16} + y)$ is always *false*. In Chapter 4, we introduce Theory of Memory ($\mathcal{T}_{\mathcal{M}}$) which preserves key C semantics to enable more simplification opportunities.

Next, consider (dis-)equality checks involving *ite*-expressions such as $\texttt{obj}_8 = \texttt{ite}(\texttt{bool}_\texttt{a}, \texttt{obj}_{16}, (\texttt{obj}_{16}+8))$. A typical strategy for reasoning over such expressions is to perform an **ITE-PUSH** rewrite: $\texttt{ite}(\texttt{bool}_\texttt{a}, \texttt{obj}_{16} = \texttt{obj}_8, (\texttt{obj}_{16} + 8) = \texttt{obj}_8)$, and

try to resolve equalities at leave nodes of *ite*-expressions. This approach could be expensive as the size of rewritten expression grows significantly, especially when both *lhs* and *rhs* of the equality contain nested *ite*-expressions. However, from the example equality check one can make the observation that the *lhs* of the equality can possibly resolve to the set of addresses $A_L = \{\texttt{obj}_8\}$, while the *rhs* can resolve to $A_R = \{\texttt{obj}_{16}, \texttt{obj}_{16} + 8\}$. With C memory semantics in mind, we can prove $A_R \cap A_L = \emptyset$ since they do not contain any address from the same memory object, hence the equality check is conclusively *False*. In the above demonstration, notice that A_L and A_R can be extracted individually from the *lhs* and *rhs* parts without performing **ITE-PUSH**. Similarly, in Chapter 5, we show that certain pointer dis-equalities can be resolved more efficiently by reasoning over the *abstract value* of the pointer expressions. We show that the *abstract values* can be computed *in place* without performing expensive **ITE-PUSH** rewrites.

Applying all aforementioned simplifications, it is possible to simplify the two *read*-over-*write* expressions into shorter versions in Figure 3.3b and Figure 3.4b. However, notice the number of *read*-over-*write* simplifications for each *read* grows linearly to the number of *write*s. When analysing larger programs with large number of *write*s, simplification at pre-process could become a bottleneck. Consider another C program P_2 in Figure 3.5 with multiple memory writes to same memory locations.

```
1  typedef struct { int a; int b; } s;
2  int main() {
3    int *num = malloc(sizeof(int)); // obj8
4    s *st = malloc(sizeof(s));      // obj16
5    st->a = 21;
6    *num = 4;
7    st->b = 42;
8    st->a = 42;
9    *num = 8;
10   st->b = 21;
11   assert(*num < st->b); // read-over-write
12   return 0;
13 }
```

Figure 3.5: Sample C program P_2 containing multiple memory operations on same addresses.

Neglecting compiler optimizations, the memory state of P_2 is shown in Figure 3.6a.

```
mem2 =
(store
  (store
    (store
      (store
        (store
          (store mem0 obj16 21)
          obj8 4)
        (+ obj16 8) 42)
      obj16 42)
    obj8 8)
  (+ obj16 8) 21)
```

(a) Unsimplified full expression.

```
mem2 =
(store
  (store
    (store mem0 obj16 42)
    (+ obj16 8) 21)
  obj8 8)
```

(b) Simplified expression.

Figure 3.6: Memory state of P_2 after line 10 encoded in $\mathcal{T}_{\mathcal{A}}$.

Considering C memory semantics once again, it is easy to see that after line 10 of P_2, although there have been 6 memory writes, only 3 distinct addresses have been written to. The first 3 writes are effectively overwritten and can be erased from the memory state. Figure 3.6b shows the simplified expression representing memory state after line 10. Also note that the addresses $\texttt{obj}_{16}$ and $\texttt{obj}_{16} + 8$ are *comparable*: the predicate $<$ is defined between them. Addresses like such can be grouped in a sorted fashion such that *binary search* can be done on them instead of *linear search* during *read*-over-*write* simplifications.

In Chapter 6, we introduce an approach that compresses *write*s in a fashion similar to the example shown in Figure 3.6. Repeating writes to the same address are compressed; writes to addresses with the *same base* are grouped in dedicated data structure in order to improve the time complexity of *read*-over-*write* simplification from linear to logarithmic in best case scenario.

Chapter 4

Theory of Memory

In Figure 2.12, we introduced how SeaBmc translates statements in SEA-IR into VCs with *sym* under currently implemented memory representations. The Arrays memory representation essentially models memory with arrays of BitVectors, and leverages existing syntax and semantics of ArrayEx theory, which is widely supported by most modern SMT-solvers. The Lambdas also models memory with arrays of BitVectors, but simulates array operations with λ-functions. This enables more efficient and precise representation of *ranged* memory operations like `memcpy` and `memset` in C, as well as enabling eager simplification through β reduction. Note that while there are careful distinctions between *memory*, *pointer* and *scalar* registers in SEA-IR, *sym* with Lambdas and Arrays loses such distinctions in translation. For example, pointer registers and scalar registers would both be translated into BitVectors in final VC. Language specific semantics are also not properly modeled by existing memory representations. As demonstrated in Chapter 3, we believe such loss of language-specific information would lead to loss of simplification opportunities, thus worsening the burden of SMT solvers.

In this chapter, we introduce the theory of *memory* $\mathcal{T}_{\mathcal{M}}$ with signatures

$\Sigma_{\mathcal{M}} = \{\circ_s, \circ_p, \bowtie_s, \bowtie_p, \textit{write-word}, \textit{read-word}, \textit{write-byte}, \textit{read-byte}, \textit{memset-words}, \textit{memcpy-words}\}$

$\textit{ptr} : {\sigma_S}^{word} \times {\sigma_S}^{word} \to \sigma_P$

$\textit{write-word} : \sigma_M \times \sigma_P \times {\sigma_S}^{word} \to \sigma_M$

$\textit{read-word} : \sigma_M \times \sigma_P \to {\sigma_S}^{word}$

$\textit{write-byte} : \sigma_M \times \sigma_P \times {\sigma_S}^{byte} \to \sigma_M$

$\textit{read-byte} : \sigma_M \times \sigma_P \to {\sigma_S}^{byte}$

$\textit{memset-words} : \sigma_M \times \sigma_P \times {\sigma_S}^{word} \times {\sigma_S}^{word} \to \sigma_M$

$\textit{memcpy-words} : \sigma_M \times \sigma_P \times \sigma_M \times \sigma_P \times {\sigma_S}^{word} \to \sigma_M$

$\mathcal{T}_{\mathcal{M}}$ aims to present a precise and efficient encoding of C programs that deal with memory by preserving certain C memory semantics in its axioms. $\mathcal{T}_{\mathcal{M}}$ is parameterized by *scalar theory* $\mathcal{T}_{\mathcal{S}}$, *pointer theory* $\mathcal{T}_{\mathcal{P}}$ and *condition theory* $\mathcal{T}_{\mathcal{C}}$. $\mathcal{T}_{\mathcal{S}}$ and $\mathcal{T}_{\mathcal{C}}$ are both single-sorted theories of sort σ_S and σ_C respectively. $\mathcal{T}_{\mathcal{S}}$ models scalar values and needs to support linear arithmetic operators $+, -, =, \leq, <$; $\mathcal{T}_{\mathcal{C}}$ models Boolean values and needs to support Boolean operators and ($\wedge$), or ($\vee$). In practice, both $\mathcal{T}_{\mathcal{S}}$ and $\mathcal{T}_{\mathcal{C}}$ are implemented with theory of BitVectors ($\mathcal{T}_{\mathcal{BV}}$). Note that in order to represent the bit width difference between *bytes* and *words*, σ_S needs to be parameterized by bit width. This is also handled by $\mathcal{T}_{\mathcal{BV}}$. We use ${\sigma_S}^n$ to denote a scalar sort of bit width n, and ${t_{\mathcal{S}}}^n$ to denote an n-bit scalar term. The term formation rules of $\mathcal{T}_{\mathcal{M}}$ are defined in Figure 4.1.

A pointer term $t_{\mathcal{P}}$ is defined by the function $\textit{ptr} : \sigma_S \times \sigma_S \to \sigma_P$. The first scalar term b represents *base* and the second scalar term o represents *offset*. A *base* represents the starting address of a C memory object. Note a similar set of *compare* operations are defined for scalar terms and pointer terms. We use different subscripts differentiate the two sets of operations ($\bowtie_s$ and $\bowtie_p$). The semantics of pointer comparison in $\mathcal{T}_{\mathcal{M}}$ are defined by axioms in Figure 4.2. Two $t_{\mathcal{P}}$s with different *base* are not equal. This preserves the C semantics that two pointers from two different *object*s cannot be equal. Note that two $t_{\mathcal{P}}$s are only comparable (operators $\leq_p$, $<_p$ defined) if and only if their *base*s are identical. Semantics of pointer arithmetics in $\mathcal{T}_{\mathcal{M}}$ are defined by axioms in Figure 4.3. Pointer term can also be formed by performing pointer addition $+_p$ between a pointer term and a scalar offset; performing pointer subtraction $-_p$ between pointers with same base would yield the scalar difference in offsets between them.

With new pointer syntax and semantics, we now also need to modify the semantics of allocation in *sym*. The behaviour of *memory allocator* remains unchanged: *alloc* now

$$
\begin{array}{lcl}
\circ_s & ::= & +_s \;\mid\; -_s \\
\circ_p & ::= & +_p \;\mid\; -_p \\
\bowtie_s & ::= & <_s \;\mid\; \leq_s \;\mid\; =_s \\
\bowtie_p & ::= & <_p \;\mid\; \leq_p \;\mid\; =_p \\
t_{\mathcal{C}} & ::= & \top \;\mid\; \bot \;\mid\; \neg t_{\mathcal{C}} \;\mid\; t_{\mathcal{C}} \wedge t_{\mathcal{C}} \;\mid\; t_{\mathcal{C}} \vee t_{\mathcal{C}} \\
 & \mid & t_{\mathcal{S}} \bowtie_s t_{\mathcal{S}} \;\mid\; t_{\mathcal{P}} \bowtie_p t_{\mathcal{P}} \\
 & \mid & \mathit{ite}(t_{\mathcal{C}}, t_{\mathcal{C}}, t_{\mathcal{C}}) \\
{t_{\mathcal{S}}}^8 & ::= & \ldots \;\mid\; {t_{\mathcal{S}}}^8 \circ_s {t_{\mathcal{S}}}^8 \;\mid\; \mathit{read\text{-}byte}(t_{\mathcal{M}}, t_{\mathcal{P}}) \;\mid\; \mathit{ite}(t_{\mathcal{C}}, {t_{\mathcal{S}}}^8, {t_{\mathcal{S}}}^8) \\
{t_{\mathcal{S}}}^{32} & ::= & \ldots \;\mid\; {t_{\mathcal{S}}}^{32} \circ_s {t_{\mathcal{S}}}^{32} \;\mid\; \mathit{read\text{-}word}(t_{\mathcal{M}}, t_{\mathcal{P}}) \\
 & \mid & \mathit{ite}(t_{\mathcal{C}}, {t_{\mathcal{S}}}^{32}, {t_{\mathcal{S}}}^{32}) \;\mid\; t_{\mathcal{P}} -_p t_{\mathcal{P}} \\
t_{\mathcal{P}} & ::= & \mathit{ptr}({t_{\mathcal{S}}}^{32}, {t_{\mathcal{S}}}^{32}) \;\mid\; t_{\mathcal{P}} +_p {t_{\mathcal{S}}}^{32} \;\mid\; \mathit{ite}(t_{\mathcal{C}}, t_{\mathcal{P}}, t_{\mathcal{P}}) \\
t_{\mathcal{M}} & ::= & \mathit{mem\text{-}identifier} \\
 & \mid & \mathit{ite}(t_{\mathcal{C}}, t_{\mathcal{M}}, t_{\mathcal{M}}) \\
 & \mid & \mathit{write\text{-}byte}(t_{\mathcal{M}}, t_{\mathcal{P}}, {t_{\mathcal{S}}}^8) \;\mid\; \mathit{write\text{-}word}(t_{\mathcal{M}}, t_{\mathcal{P}}, {t_{\mathcal{S}}}^{32}) \\
 & \mid & \mathit{memset\text{-}words}(t_{\mathcal{M}}, t_{\mathcal{P}}, {t_{\mathcal{S}}}^{32}, {t_{\mathcal{S}}}^{32}) \\
 & \mid & \mathit{memcpy\text{-}words}(t_{\mathcal{M}}, t_{\mathcal{P}}, t_{\mathcal{M}}, t_{\mathcal{P}}, {t_{\mathcal{S}}}^{32})
\end{array}
$$

Figure 4.1: Term formation rules of $\mathcal{T}_{\mathcal{M}}$ for programs with 8-bit byte size, 32-bit word size. Operations between pointer and scalar terms are differentiated with subscripts p and s.

places constraints on newly created $t_{\mathcal{P}}$ terms as side conditions.

$$
\begin{aligned}
\mathit{sym}(\texttt{P1},\ \texttt{M1} = \texttt{alloca R0},\ \texttt{M0}) \triangleq\ & p_1 = \mathit{ptr}(\mathit{obj}_n, 0) \wedge \\
& \mathit{ptr}(\mathit{obj}_n, 0) = \mathit{alloc}(\texttt{alloca R0},\ \texttt{M0}) \wedge m_1 = m_0 \\
\mathit{sym}(\texttt{P1},\ \texttt{M1} = \texttt{malloc R0},\ \texttt{M0}) \triangleq\ & p_1 = \mathit{ptr}(\mathit{obj}_n, 0) \wedge \\
& \mathit{ptr}(\mathit{obj}_n, 0) = \mathit{alloc}(\texttt{malloc R0},\ \texttt{M0}) \wedge m_1 = m_0
\end{aligned}
$$

The new base obj_n created at each allocation site is a fresh scalar variable that is greater than all previous allocation bases to denote *temporal* order of allocation. This would be useful later in Chapter 6.

mem-identifier denotes a symbol or constant of *memory sort* σ_M. A memory term $t_{\mathcal{M}}$ represent a mapping from a pointer to either a byte or a word. Writing a byte or a word to memory is represented by *write-byte* : $\sigma_M \times \sigma_P \times {\sigma_S}^b \to \sigma_M$ and *write-word* : $\sigma_M \times \sigma_P \times {\sigma_S}^w \to \sigma_M$ respectively, where b and w are the bit widths of a *byte* and a *word*. Reading a byte or a word from memory is represented by *read-byte* : $\sigma_M \times \sigma_P \to {\sigma_S}^b$ and *read-word* : $\sigma_M \times \sigma_P \to {\sigma_S}^w$. Without loss of generality, we discuss word-sized memory operations only in the rest of this chapter. The semantics of word-sized memory operations are similar to $\mathcal{T}_{\mathcal{A}}$ ROW axioms, as shown in Figure 4.4.

$$\forall b_1, b_2 \,.\, b_1 \neq_s b_2 \Rightarrow ptr(b_1, o_1) \neq_p ptr(b_2, o_2)$$
$$\forall b_1, b_2, o_1, o_2 \,.\, b_1 =_s b_2 \wedge o_1 =_s o_2 \iff ptr(b_1, o_1) =_p ptr(b_2, o_2)$$
$$\forall b_1, b_2, o_1, o_2 \,.\, b_1 =_s b_2 \wedge o_1 <_s o_2 \iff ptr(b_1, o_1) <_p ptr(b_2, o_2)$$
$$\forall b_1, b_2, o_1, o_2 \,.\, b_1 =_s b_2 \wedge o_1 \leq_s o_2 \iff ptr(b_1, o_1) \leq_p ptr(b_2, o_2)$$

Figure 4.2: Axioms for $\mathcal{T}_{\mathcal{M}}$ pointer comparison.

$$\forall b, o_1, o_2 \,.\, ptr(b, o_1) +_p o_2 = ptr(b, o_1 +_s o_2)$$
$$\forall b_1, b_2 \,.\, b_1 =_s b_2 \Rightarrow ptr(b_1, o_1) -_p ptr(b_2, o_2) = o_1 -_s o_2$$

Figure 4.3: Axioms for $\mathcal{T}_{\mathcal{M}}$ pointer arithmetics.

Ranged memory operations are represented by *memset-words* : $\sigma_M \times \sigma_P \times {\sigma_S}^w \times {\sigma_S}^w \to \sigma_M$ and *memcpy-words* : $\sigma_M \times \sigma_P \times \sigma_M \times \sigma_P \times {\sigma_S}^w \to \sigma_M$. The semantics of word-sized *ranged* memory operations are similar to the semantics of *set* and *copy* in $\mathcal{T}_{\mathcal{ASC}}$, as shown in Figure 4.5.

With semantics of word-sized memory operations defined in Figure 4.4 and Figure 4.5, we define rules for eagerly rewriting word-sized *reads* over $t_{\mathcal{M}}$ into formulas with only *ite*-terms and terminal *mem-identifier* symbols. The rewrite rules are shown in Figure 4.6 and Figure 4.7 for *read* over word-sized `store`s and word-sized `memset`, `memcpy` respectively.

$\mathcal{T}_{\mathcal{M}}$ defines *sym* VCGen semantics for SeaM memory representation. We expand *sym* memory operations translation semantics (previous shown in Figure 2.12) with SeaM shown in Figure 4.8. Note that VC in SeaBmc is non-extensional, so final VC from SeaM would not contain any `write-word`, `memset-words` and `memcpy-words` after eager rewriting.

In the rewrite rules of $\mathcal{T}_{\mathcal{M}}$ memory operations, notice that most of the simplifications depend on resolution of pointer term (dis)-equalities. In the next chapter, we introduce an efficient technique for pointer comparison based on Abstract Interpretation.

$$\forall i, j \;.\; i =_p j \Rightarrow \mathit{read\text{-}word}(a, i) = \mathit{read\text{-}word}(a, j)$$
$$\forall i, j \;.\; i =_p j \Rightarrow \mathit{read\text{-}word}(\mathit{write\text{-}word}(a, j, v), i) = v$$
$$\forall i, j \;.\; i \neq_p j \Rightarrow \mathit{read\text{-}word}(\mathit{write\text{-}word}(a, j, v), i) = \mathit{read\text{-}word}(a, i)$$

Figure 4.4: $\mathcal{T}_{\mathcal{M}}$ word-sized *read*-over-*write* axioms.

$$\forall p, i, s \;.\; p \leq_p i <_p p +_p s \Rightarrow \mathit{read\text{-}word}(\mathit{memset\text{-}words}(a, p, v, s), i) = v$$
$$\forall p, i, s \;.\; \neg(p \leq_p i <_p p +_p s) \Rightarrow \mathit{read\text{-}word}(\mathit{memset\text{-}words}(a, p, v, s), i) = \mathit{read\text{-}word}(a, i)$$
$$\forall p, i, s \;.\; p \leq_p i <_p p +_p s \Rightarrow \mathit{read\text{-}word}(\mathit{memcpy\text{-}words}(a, p, b, q, s), i) = \mathit{read\text{-}word}(b, q +_p (i -_p p))$$
$$\forall p, i, s \;.\; \neg(p \leq_p i <_p p +_p s) \Rightarrow \mathit{read\text{-}word}(\mathit{memcpy\text{-}words}(a, p, b, q, s), i) = \mathit{read\text{-}word}(a, i)$$

Figure 4.5: $\mathcal{T}_{\mathcal{M}}$ read over word-sized *copy*, *set* axioms.

$$\frac{}{\mathit{read\text{-}word}(\mathit{write\text{-}word}(a, i, v),\ j) \ \rightsquigarrow\ \mathit{ite}(i =_p j, v, \mathit{read\text{-}word}(a, j))}\ \text{ROW}$$

$$\frac{\vdash i =_p j}{\mathit{read\text{-}word}(\mathit{write\text{-}word}(a, i, v), j) \ \rightsquigarrow\ v}\ \text{ROW-HIT}$$

$$\frac{\vdash i \neq_p j}{\mathit{read\text{-}word}(\mathit{write\text{-}word}(a, i, v),\ j) \ \rightsquigarrow\ \mathit{read\text{-}word}(a, j)}\ \text{ROW-SKIP}$$

Figure 4.6: *read*-over-*write* rewrite rules for word-sized memory `stores`.

$$\frac{}{\textit{read-word}(\textit{memset-words}(a,p,s,v),i) \rightsquigarrow \textit{ite}(p \leq_s i <_s (p +_p s), v, \textit{read-word}(a,\ i))}$$

$$\frac{\vdash p \leq_p i <_p (p +_p s)}{\textit{read-word}(\textit{memset-words}(a,p,s,v),i) \rightsquigarrow v}$$

$$\frac{\vdash \neg(p \leq_p i <_p (p +_p s))}{\textit{read-word}(\textit{memset-words}(a,p,s,v),\ i) \rightsquigarrow \textit{read-word}(a,\ i)}$$

$$\frac{}{\textit{read-word}(\textit{memcpy-words}(a,p,b,q,s),\ i) \rightsquigarrow \textit{ite}(p \leq_p i <_p p +_p s, \textit{read-word}(b,\ q +_p (i -_p p)), \textit{read-word}(a,i))}$$

$$\frac{\vdash p \leq_p i <_p p +_p s}{\textit{read-word}(\textit{memcpy-words}(a,p,b,q,s),i) \rightsquigarrow \textit{read-word}(b, q +_p (i -_p p))}$$

$$\frac{\vdash \neg(p \leq_p i <_p p +_p s)}{\textit{read-word}(\textit{memcpy-words}(a,p,b,q,s),i) \rightsquigarrow \textit{read-word}(a,i)}$$

Figure 4.7: *read*-over-*write* rewrite rules for word-sized `memset` and `memcpy`.

	SeaM
$\textit{write}(m_0, r_1, p_2)$	`write-word` $m_0\ r_1\ p_2$
$\textit{memset}(m_0,\ p_1,\ r_1,\ r_2)$	`memset-words`$(m_0,\ p_1,\ r_1,\ r_2)$
$\textit{memcpy}(m_0,\ p_1,\ m_1,\ p_2,\ r_1)$	`memcpy-words`$(m_0,\ p_1,\ m_1,\ p_2,\ r_1)$
$\textit{read}(\textit{write}(m_0,r_1,p_2),p_1)$	$\textit{ite}(p_1 =_p p_2, r_1,$ `read-word`$(m_0,\ p_1))$
$\textit{read}(\textit{memset}(m_0,\ p_1,\ r_1,\ r_2),p_2)$	$\textit{ite}(p_1 \leq_p p_2 <_p (p_1 +_p r_1), r_2,$ `read-word`$(m_0,\ p_2))$
$\textit{read}(\textit{memcpy}(m_0,\ p_1,\ m_1,\ p_2,\ r_1),p_3)$	$\textit{ite}(p_1 \leq_p p_3 <_p (p_1 +_p r_1),$ `read-word`$(m_1, p_2 +_p (p_3 -_p p_1)),$ `read-word`$(m_0,\ p_3))$

Figure 4.8: *sym* translation of memory operations in SeaM.

Chapter 5

Address Range Map

In C programs translated to SEA-IR, it is common to represent control flow with `PHI` statements, which is further translated to *ite*-terms during VCGen to represent different possible values based on condition. As a result, it is possible for pointer terms $t_{\mathcal{P}}$ in $\mathcal{T}_{\mathcal{M}}$ expressions to contain *ite*-expressions, as described in Figure 4.1. Precise comparison between $t_{\mathcal{P}}$ terms containing nested *ite*s can be exponential when done with **ITE-PULL** and **ITE-PUSH** rewrites. In this chapter, we introduce Address Range Map (ARM), an Abstract Interpretation [11] based technique that results in more efficient pointer comparisons by over-approximating $t_{\mathcal{P}}$ pointer terms.

Abstract Interpretation Abstract Interpretation is a powerful technique for approximation by representing large space of states with small number of abstract states. Given a concrete domain $\mathcal{D}$, an abstract domain is defined by a complete lattice $\langle \mathcal{D}^{\#}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$. Sets of concrete elements from $\mathcal{D}$ are mapped to abstract values from $\mathcal{D}^{\#}$ by abstraction function $\alpha : \mathcal{P}(\mathcal{D}) \mapsto \mathcal{D}^{\#}$, and abstract values are mapped back to sets of concrete elements by concretization function $\gamma : \mathcal{D}^{\#} \mapsto \mathcal{P}(\mathcal{D})$ such that $\gamma(\top) = \mathcal{D}$ and $\gamma(\bot) = \emptyset$. Given the operational semantics of the statement transformers in a concrete domain, the corresponding abstract domain also defines *abstract semantics* that transforms abstract values as concrete elements transform. The connection between a concrete transformer F and corresponding abstract transformer $\hat{F}$ is denoted by $F \hookrightarrow \hat{F}$.

The abstract semantics of $\mathcal{D}^{\#}$ is required to be **sound** with regards to the concrete semantics of $\mathcal{D}$. Given a concrete transformer F and corresponding abstract transformer $F^{\#}$, $F^{\#}$ is sound wrt. F if

$$\forall x \in \mathcal{D}, \forall x^{\#} \in \mathcal{D}^{\#} \,.\, \alpha(x) \sqsubseteq x^{\#} \Rightarrow \alpha(F(x)) \sqsubseteq F^{\#}(x^{\#})$$

$$
\begin{aligned}
nums(a) &= \begin{cases} \{a\} & \text{If } a \text{ is scalar concrete constant.} \\ \mathcal{D}_s & \text{If } a \text{ is scalar symbolic value.} \end{cases} \\
nums(ite(c,a,b)) &= ite(c, nums(a), nums(b)) \\
&= \begin{cases} nums(a) & \text{If } c = True \\ nums(b) & \text{If } c = False \end{cases} \\
nums(a +_s b) &= \{x +_s y \mid x \in nums(a), y \in nums(b)\}
\end{aligned}
$$

Figure 5.1: Concrete semantics in domain $\mathcal{D}_s$.

5.1 Offset Intervals

Given a $t_{\mathcal{P}}$ $ptr(b, o)$, the offset o is a scalar term $t_{\mathcal{S}}$ representing a set of scalar offset values in concrete domain denoted by $\mathcal{D}_s$. We define function $nums : \sigma_S \to \mathcal{P}(\mathcal{D}_s)$ that returns the set of all concrete scalar *offset*s represented by $t_{\mathcal{S}}$. The concrete semantics $ite : \sigma_C \times \sigma_S \times \sigma_S \to \sigma_S$ and $+_s : \sigma_S \times \sigma_S \to \sigma_S$ is defined by $nums(t_{\mathcal{S}})$ in Figure 5.1. The concrete scalar set of sum of a and b contains set of sums between cartesian product of $nums(a)$ and $nums(b)$; the concrete set of $ite(i, a, b)$ is $nums(a)$ if i is True, otherwise $nums(b)$. For example, the scalar term s_1 shown in Figure 5.4a represents the set of scalars shown in Figure 5.4b. We can over-approximate $\mathcal{D}_s$ elements with the abstract domain of intervals [10] (itv) denoted as $\hat{\mathcal{D}}_s$. An abstract value of the interval domain takes the form $[l, h]$, where l represents the lower bound, with concrete values $l \in \mathbb{Z} \cup -\infty$; h represents the upper bound with concrete values $h \in \mathbb{Z} \cup \infty$.

The abstraction function α and concretization function γ between scalar offset concrete domain and interval abstract domain in Figure 5.2. A concrete offset with scalar value o maps to a itv with lower bound and higher bound o; a symbolic offset maps to $\top$. Note that the scalar addition operator ($+_s$) corresponds to the abstract addition operator ($\hat{+}$) between abstract itv values.

The abstract version of scalar add $+_s$ is itv abstract add $\hat{+}$; the abstract version of *ite* is $\sqcup$. The semantic of abstract *union* ($\sqcup$), *join* ($\sqcap$), *subset of* ($\sqsubseteq$) and itv *addition* $\hat{+}$ operators between itv abstract values are defined in Figure 5.3. Note that we assume overflow does not happen on $+_s$ since we are focusing on the subset of scalar values representing pointer offsets. *Legal* pointer arithmetics should not overflow. Figure 5.4c shows an example abstraction process of scalar term s_1.

Soundness of Abstract Semantics The abstract transformers $\sqcup$ and $\hat{+}$ should be sound wrt. their concrete counterpart *ite* and $+_s$.

$$\alpha(O) = \begin{cases} [\min O_{i\in\mathbb{N}}, \max O_{i\in\mathbb{N}}] & O \text{ is non-empty set of scalars.} \\ \top & O = \mathcal{D}_s \\ \bot & O = \emptyset \end{cases}$$
$$\gamma([l,h]) = \{x \in \mathbb{Z} \mid l \leq_s x \leq_s h\}$$

Figure 5.2: α and γ between $\mathcal{D}_s$ and $\hat{\mathcal{D}}_s$. min $S_{i\in\mathbb{N}}$ and max $S_{i\in\mathbb{N}}$ returns the maximum and minimum element of a set S respectively.

$$\begin{array}{rr}
a +_s b \hookrightarrow \alpha(nums(a)) \mathbin{\hat{+}} \alpha(nums(b)) & [a,b] \mathbin{\hat{+}} [a',b'] = [a +_s a', b +_s b'] \\
ite(i,a,b) \hookrightarrow \alpha(nums(a)) \sqcup \alpha(nums(b)) & [a,b] \sqcup [a',b'] = [min(a,a'), max(b,b')] \\
[a,b] \sqcup \top = \top & [a,b] \sqcup \bot = [a,b] \\
[a,b] \sqcap \top = [a,b] & [a,b] \sqcap \bot = \bot \\
[a,b] \mathbin{\hat{+}} \top = \top & [a,b] \mathbin{\hat{+}} \bot = [a,b] \\
\forall a,b\,.\,[a,b] \sqsubseteq \top & \forall a,b\,.\,\bot \sqsubseteq [a,b]
\end{array}$$
$$[a,b] \sqcap [a',b'] = ite(a \leq_s a' \leq_s b \vee a' \leq_s a \leq_s b', [max(a,a'), min(b,b')],\ \bot)$$
$$[a,b] \sqsubseteq [a',b'] \iff [a,b] \subseteq [a',b'] \iff a' \leq_s a \wedge b \leq_s b'$$

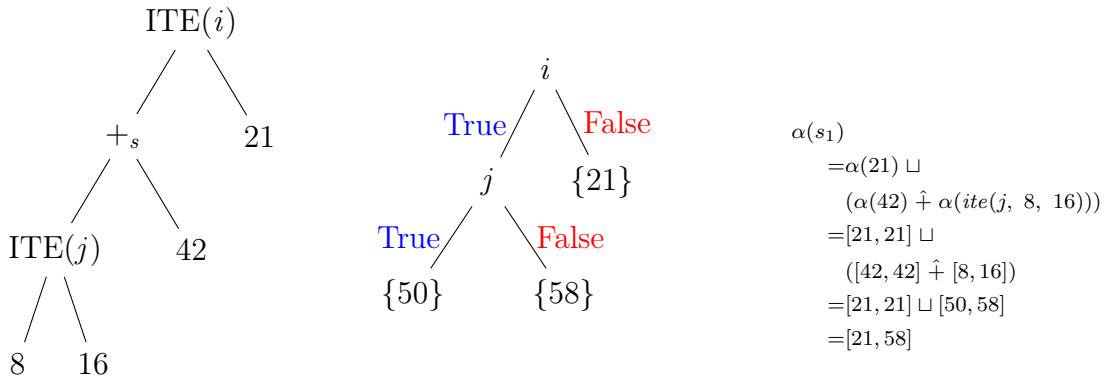
Figure 5.3: Abstract semantics of itv abstract operators $\sqcup, \sqcap$ and $\mathbin{\hat{+}}$. $min(a,b)$, $max(a,b)$ returns the smaller and larger of a and b; $ite(c,a,b)$ returns a if c is True and b otherwise.

ITE Take $t_{\mathcal{S}}$ a and b representing sets of scalar offsets $A = nums(a), B = nums(b)$; then the itv abstract values of them are $\hat{a} = \alpha(A) = [\min A_{i\in\mathbb{N}}, \max A_{i\in\mathbb{N}}]$, $\hat{b} = \alpha(B) = [\min B_{i\in\mathbb{N}}, \max B_{i\in\mathbb{N}}]$. The abstract value $\hat{k}$ of abstract version of *ite* ($\sqcup$) between $\hat{a}$ and $\hat{b}$ is

$$\hat{k} = \hat{a} \sqcup \hat{b} = [min(\min A_{i\in\mathbb{N}}, \min B_{i\in\mathbb{N}}), max(\max A_{i\in\mathbb{N}}, \max B_{i\in\mathbb{N}})]$$

After forming new $t_{\mathcal{S}}$ k with *ite*,

$$\begin{aligned} k &= ite(i,a,b) \\ K &= nums(k) \\ &= nums(ite(i,a,b)) \\ &= ite(i,A,B) \quad \textbf{ITE-PUSH} \end{aligned}$$



(a) s_1 AST (b) $nums(s_1)$ (c) $\alpha(nums(s_1))$

Figure 5.4: Example $t_{\mathcal{S}}$ s_1 representing a set of scalar numbers $nums(s_1)$, with abstract value $\alpha(nums(s_1))$. ITE(i) node represents expression $ite(i, ...)$.

K resolves to A if i is True otherwise B. If i is True:

$$\begin{aligned}\alpha(K) &= \alpha(A)\\ &= [\min A_{i\in\mathbb{N}}, \max A_{i\in\mathbb{N}}]\\ &\sqsubseteq [min(\min A_{i\in\mathbb{N}}, \min B_{i\in\mathbb{N}}), max(\max A_{i\in\mathbb{N}}, \max B_{i\in\mathbb{N}})]\\ &\sqsubseteq \hat{k} = \hat{a} \sqcup \hat{b}\end{aligned}$$

If i is False:

$$\begin{aligned}\alpha(K) &= \alpha(B)\\ &= [\min B_{i\in\mathbb{N}}, \max B_{i\in\mathbb{N}}]\\ &\sqsubseteq [min(\min A_{i\in\mathbb{N}}, \min B_{i\in\mathbb{N}}), max(\max A_{i\in\mathbb{N}}, \max B_{i\in\mathbb{N}})]\\ &\sqsubseteq \hat{k} = \hat{a} \sqcup \hat{b}\end{aligned}$$

Regardless of value of i,

$$\forall \hat{a}, \hat{b} \in \mathsf{itv}, \forall A, B \in \mathcal{D}_s \,.\, \alpha(A) \sqsubseteq \hat{a} \wedge \alpha(B) \sqsubseteq \hat{b} \Rightarrow \alpha(ite(i, A, B)) \sqsubseteq \hat{a} \sqcup \hat{b} \tag{5.1}$$

is True.

Addition Take $t_{\mathcal{S}}$ a and b representing sets of scalar offsets $A = nums(a), B = nums(b)$; then the itv abstract values of them are $\hat{a} = [\min A_{i\in\mathbb{N}}, \max A_{i\in\mathbb{N}}]$, $\hat{b} = [\min B_{i\in\mathbb{N}}, \max B_{i\in\mathbb{N}}]$.

The abstract sum of $\hat{a}$ and $\hat{b}$ is

$$\hat{a} \mathbin{\hat{+}} \hat{b} = [\min\ A_{i\in\mathbb{N}} + \min\ B_{i\in\mathbb{N}}, \max\ A_{i\in\mathbb{N}} + \max\ B_{i\in\mathbb{N}}]$$

. After forming new $t_{\mathcal{S}}\ k$ with addition $+_s$,

$$\begin{aligned} k &= a +_s b \\ K &= nums(k) \\ &= nums(a +_s b) \\ &= \{x +_s y \mid x \in A, y \in B\} \end{aligned}$$

The minimum element in K is the sum of minimum elements in A and B; the maximum element in K is the sum of maximum elements in A and B. Therefore the abstract value of K is:

$$\begin{aligned} \alpha(K) &= [\min\ K_{i\in\mathbb{N}}, \max\ K_{i\in\mathbb{N}}] \\ &= [\min\ A_{i\in\mathbb{N}} + \min\ B_{i\in\mathbb{N}}, \max\ A_{i\in\mathbb{N}} + \max\ B_{i\in\mathbb{N}}] \\ &\sqsubseteq \hat{a} \mathbin{\hat{+}} \hat{b} \end{aligned}$$

In other words,

$$\forall \hat{a}, \hat{b} \in \mathsf{itv}, \forall A, B \in \mathcal{D}_s\ .\ \alpha(A) \sqsubseteq \hat{a} \wedge \alpha(B) \sqsubseteq \hat{b} \Rightarrow \alpha(\{x +_s y \mid x \in A, y \in B\}) \sqsubseteq \hat{a} \mathbin{\hat{+}} \hat{b} \quad (5.2)$$

is True.

Theorem 1. *The abstract semantics of* $\mathsf{itv}\ \sqcup$ *and* $\hat{+}$ *is* ***sound*** *wrt. concrete semantics of* $\mathcal{D}_s$ *ite and* $+_s$.

Proof. We have established soundness of $\sqcup$ wrt. *ite* in (5.1). Intuitively, given scalar term $ite(i, a, b)$, $\sqcup$ joins the *then* and *else* branches of *ite* regardless of the condition i, therefore $\hat{a} \sqcup \hat{b} \sqsupseteq \alpha(ite(i, nums(a), nums(b)))$ is always true. The soundness of $\hat{+}$ wrt. scalar $+$ is established in (5.2). Given two sets of scalar numbers $nums(a)$ and $nums(b)$, concrete addition creates a new set ab from sums of cartesian product of $nums(a)$ and $nums(b)$. The itv of ab is $[\min\ nums(a)_{i\in\mathbb{N}} + \min\ nums(b)_{i\in\mathbb{N}}, \max\ nums(a)_{i\in\mathbb{N}} + \max\ nums(b)_{i\in\mathbb{N}}]$, which is the same as $\alpha(nums(a)) \mathbin{\hat{+}} \alpha(nums(b))$. □

Note that although lacking a concrete counterpart in $\mathcal{D}_s$, abstract intersection $\sqcap$ is also sound: given two sets of scalar values over-approximated by $[a, b]$ and $[a', b']$ the new interval between $max(a, a')$ and $min(b, b')$ over-approximates the set intersection. We omit the proof for itv $\sqcap$ soundness, which is already elaborated in [10].

Given a concrete offset element o, we define the inference rules for inclusion ($\in$) of o in a itv $[l, h]$ in Figure 5.5.

$$\frac{\vdash l \leq_s o \leq_s h}{\forall o \,.\, o \in [l,\, h]} \qquad \forall o \,.\, o \in \top \qquad \forall o \,.\, o \notin \bot$$

Figure 5.5: Inference rules for numeric offset inclusion in Offset itv.

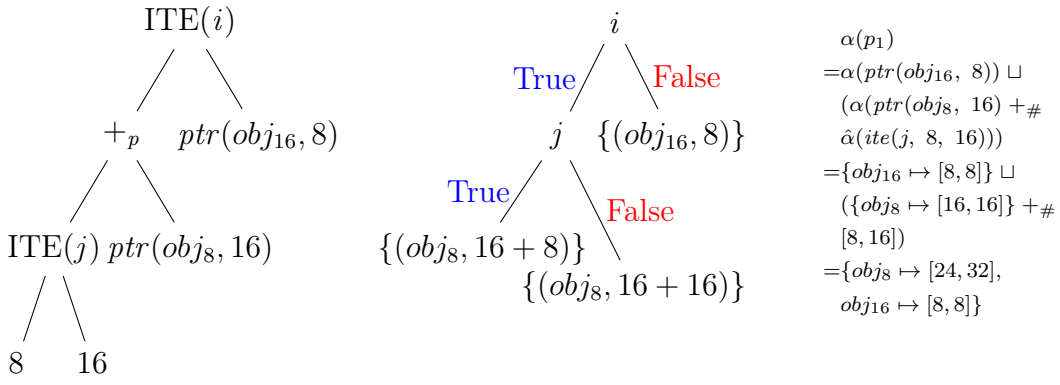
5.2 Address Range Map

With *ite*s, a $t_{\mathcal{P}}$ represent a set of pointer base-offset pairs: (b, o). We define the concrete domain of pointers as $\mathcal{D}_p = \{(b, o) \mid b \in \mathcal{AP}, o \in \mathbb{Z}_{\geq 0}\}$, where $\mathcal{AP}$ denotes architecture-specific address space of non-negative integers. We define $ptrs : \sigma_P \to \mathcal{P}(\mathcal{D}_p)$ to return the set of all pointers represented by a pointer term $t_{\mathcal{P}}$. The concrete semantics of ite : $\sigma_C \times \sigma_P \times \sigma_P \to \sigma_P$ and $+_p : \sigma_P \times \sigma_S \to \sigma_P$ are defined by $ptrs(t_{\mathcal{P}})$ in Figure 5.6. For example, p_1 in Figure 5.7a represents different sets of base-offset pairs depending on the values of conditional terms i and j, as shown in Figure 5.7b. Instead of directly perform comparison between elements of $\mathcal{D}_p$, we propose comparisons within the abstract domain ${\mathcal{D}_p}^{\#}$ instead. The abstract values in ${\mathcal{D}_p}^{\#}$ are maps between scalar *pointer base* and offset itv abstractions $m : \sigma_S \mapsto [\sigma_S, \sigma_S]$. We use $\top$ and $\bot$ to represent *all* pointers and *none* possible pointers. We name the abstract values **Address Range Map** (ARM). An ARM $\phi^{\#}$ in which $\forall (b \mapsto \hat{o}) \in \phi^{\#} \,.\, \hat{o} = \bot$ is $\bot$ since all bases map to none possible offsets. For convenience, we define function *bases* that returns the set of *bases* in an ARM $\phi^{\#}$: $bases(\phi^{\#}) = \{b \mid (b \mapsto o) \in \phi^{\#}\}$. Given a *base* b in an ARM $\phi^{\#}$, we write $\phi^{\#}[b]$ to refer to the mapped itv by b.

The concretization function (γ) and abstraction function (α) between $\mathcal{D}_p$ and ${\mathcal{D}_p}^{\#}$ are defined in Figure 5.8. Concretization of ARM $\gamma(\phi^{\#})$ is the set of pointers with all *pointer bases* in $\phi^{\#}$ keys and offsets between the mapped intervals per *base*.

$$
\begin{aligned}
ptrs(ptr(b,o)) &= \{(b,o') \mid o' \in nums(o)\} \\
ptrs(ite(c,a,b)) &= ite(c, ptrs(a), ptrs(b)) \\
&= \begin{cases} ptrs(a) & \text{If } c = True \\ ptrs(b) & \text{If } c = False \end{cases} \\
ptrs(p +_p i) &= \{(b, o +_s o') \mid (b,o) \in ptrs(p), o' \in nums(i)\}
\end{aligned}
$$

Figure 5.6: Concrete semantics in domain $\mathcal{D}_p$. Note *nums* from itv domain.



(a) p_1 AST (b) $ptrs(p_1)$ (c) $\alpha(ptrs(p_1))$

Figure 5.7: Example $t_{\mathcal{P}}$ p_1 representing set of pointers $ptrs(p_1)$, with abstract value $\alpha(ptrs(p_1))$. ITE(i) node represents expression $ite(i, ...)$.

Figure 5.9 defines the abstract semantics for $\sqcup$, $\sqcap$ and abstract version of $+_p$ ($+_\# : \text{ARM} \times \mathsf{itv} \to \text{ARM}$) , *ite* ($\sqcup : \text{ARM} \times \text{ARM} \to \text{ARM}$) between **ARM**s. As shown in the abstraction process of the *ite* pointer term $ptrs(p_1)$ in Figure 5.7c, for the pointer addition in the *then* branch, ARM abstraction performs itv $\hat{+}$ operation between the itv value of $ite(j, 8, 16)$ and all mapped itvs in the ARM of $ptr(obj_8, 16)$:

$$\{obj_8 \mapsto [16,16]\} +_\# [8,16] = \{obj_8 \mapsto [16+8, 16+16]\} = \{obj_8 \mapsto [24,32]\}$$

ARM over-approximates $ite(i,a,b)$ by ignoring the value of condition term i and performing $\sqcup$ operation between $\alpha(a)$ and $\alpha(b)$: the ARM of *ite* p_1 is calculated from $\sqcup$ between the ARM of *then* branch and the *else* branch

$$\{obj_{16} \mapsto [8,8]\} \sqcup \{obj_8 \mapsto [24,32]\} = \{obj_{16} \mapsto [8,8], obj_8 \mapsto [24,32]\}$$

Between ARMs with common bases, $\sqcup$ operation would perform itv $\hat{\sqcup}$ for the mapped offsets of common bases. For example, $\sqcup$ between $\alpha(ptrs(p_1))$ and ARM $\{obj_8 \mapsto [8,16]\}$

$$\alpha(\phi) = \{b \mapsto \hat{\alpha}(\{o \mid (b,o) \in \phi\}) \mid (b,o) \in \phi\}$$
$$\gamma(\phi^{\#}) = \{(b,o) \mid o \in \phi^{\#}[b], b \in \mathit{bases}(\phi^{\#})\}$$

Figure 5.8: γ and α functions between ARM and pointers.

$$\mathit{ite}(c,p,p') \hookrightarrow \alpha(\phi) \sqcup \alpha(\psi), \quad \text{where } \phi = \mathit{ptrs}(p) \text{ and } \psi = \mathit{ptrs}(p')$$
$$\phi^{\#} \hat{\sqcup} \psi^{\#} = \{b \mapsto \hat{o} \mid \ b \in \mathit{bases}(\phi^{\#}) \cup \mathit{bases}(\psi^{\#}),$$
$$\hat{o} = \begin{cases} \phi^{\#}[b] \hat{\sqcup} \psi^{\#}[b] & \text{if } b \in \mathit{bases}(\phi^{\#}) \wedge b \in \mathit{bases}(\psi^{\#}) \\ \phi^{\#}[b] & \text{if } b \in \mathit{bases}(\phi^{\#}) \wedge b \notin \mathit{bases}(\psi^{\#}) \\ \psi^{\#}[b] & \text{if } b \notin \mathit{bases}(\phi^{\#}) \wedge b \in \mathit{bases}(\psi^{\#}) \} \end{cases}$$
$$\phi^{\#} \sqcap \psi^{\#} = \{b \mapsto \phi^{\#}[b] \hat{\sqcap} \psi^{\#}[b] \mid b \in \mathit{bases}(\phi^{\#}) \cap \mathit{bases}(\psi^{\#})\}$$
$$p +_p o \hookrightarrow \alpha(\phi) +_{\#} \hat{\alpha}(A) \quad \text{where } \phi = \mathit{ptrs}(p) \wedge A = \mathit{nums}(o)$$
$$\phi^{\#} +_{\#} [l,h] = \{b \mapsto \phi^{\#}[b] \hat{+} [l,h] \mid b \in \mathit{bases}(\phi^{\#})\}$$
$$\mathit{bases}(\phi^{\#}) \subseteq \mathit{bases}(\psi^{\#}) \wedge \forall b \in \mathit{bases}(\phi^{\#} \sqcup \psi^{\#}) \cdot \phi^{\#}[b] \hat{\sqsubseteq} \psi^{\#}[b] \iff \phi^{\#} \sqsubseteq \psi^{\#}$$

Figure 5.9: Abstract semantics for **ARM** $\sqcup$, $\sqcap$, $\sqsubseteq$ and $+_{\#}$. Accent $\hat{}$ denotes values and operators of itv.

would yield

$$\begin{aligned} &\{obj_{16} \mapsto [8,8], obj_8 \mapsto [24,32]\} \sqcup \{obj_8 \mapsto [8,16]\} \\ =&\{obj_{16} \mapsto [8,8], obj_8 \mapsto ([24,32] \hat{\sqcup} [8,16])\} \\ =&\{obj_{16} \mapsto [8,8], obj_8 \mapsto [8,32]\} \end{aligned}$$

Soundness of Abstract semantics The abstract transformers $\sqcup$ and $+_{\#}$ should be sound wrt. their concrete counterpart *ite* and $+_p$.

ITE Given pointer term $t_{\mathcal{P}}$ in *ite* form $k = \mathit{ite}(i,p,q)$, the concrete sets of the children are $P = \mathit{ptrs}(p)$, $Q = \mathit{ptrs}(q)$. k represents a set of pointer base-offset pairs K:

$$K = \mathit{ite}(i,P,Q)$$

which resolves to P if i is *True* and Q otherwise. Abstract values of P and Q are $p^{\#} = \alpha(P), q^{\#} = \alpha(Q)$. The result of abstract version of *ite* ($\sqcup$) between $p^{\#}$ and $q^{\#}$ is

$$p^{\#} \sqcup q^{\#} = \{b \mapsto \hat{o} \mid b \in p^{\#} \vee b \in q^{\#}\}$$

where each offset interval $\hat{o}$ in entries are defined according to Figure 5.9.

Without loss of generality, we discuss the case if $i = True$ only. If $i = True$, the ARM of k is just the ARM of p:

$$\alpha(K) = ite(True, P, Q) = \alpha(P) = p^{\#}$$

Lemma 1. *For all pointer terms p and q, $bases(p^{\#}) \subseteq bases(p^{\#} \sqcup q^{\#})$, where $p^{\#} = \alpha(ptrs(p))$ and $q^{\#} = \alpha(ptrs(q))$.*

Proof. From abstract semantics of $\sqcup_{\#}$ in Figure 5.9, $bases(p^{\#} \sqcup q^{\#}) = bases(p^{\#}) \cup bases(q^{\#}) \supseteq bases(p^{\#})$. □

Lemma 2. *Given pointer terms p and q, and their ARMs $p^{\#}$ and $q^{\#}$, $p^{\#}[b] \hat{\sqsubseteq} (p^{\#} \sqcup q^{\#})[b]$ for all b where $b \in p^{\#} \wedge b \in q^{\#}$.*

Proof. For each base b that exist in both $p^{\#}$ and $q^{\#}$, their mapped itvs $\hat{o}$ are re-calculated with $\hat{o}_p \hat{\sqcup} \hat{o}_q$, where $\hat{o}_p = p^{\#}[b] = [l, h], \hat{o}_q = q^{\#}[b] = [l', h']$.

$$\begin{aligned}\hat{o} &= (p^{\#} \sqcup_{\#} q^{\#})[b] = \hat{o}_p \hat{\sqcup} \hat{o}_q \\ &= [l, h] \hat{\sqcup} [l', h'] \\ &= [min(l, l'), max(h, h')]\end{aligned}$$

From above and semantics of $\hat{\sqsubseteq}$ in Figure 5.3:

$$\begin{aligned}\hat{o}_p &= [l, h] \\ &\hat{\sqsubseteq} [min(l, l'), max(h, h')] = \hat{o}\end{aligned}$$

□

From Lemma 1, Lemma 2 and the semantics of ARM $\sqsubseteq$ in Figure 5.9,

$$\alpha(ite(i, P, Q)) \sqsubseteq (\alpha(P) \sqcup \alpha(Q))$$

The above proves that

$$\forall P, Q \in \mathcal{D}_p, \forall p^{\#}, p^{\#} \in \mathcal{D}_p{}^{\#} \,.\, \alpha(P) \sqsubseteq p^{\#} \wedge \alpha(Q) \sqsubseteq q^{\#} \Rightarrow \alpha(ite(i, P, Q)) \sqsubseteq p^{\#} \sqcup q^{\#}$$

Pointer addition Given pointer term $t_{\mathcal{P}}$ formed by pointer arithmetic $k = p +_p r$, the original pointer term p represents a set of pointer base-offset pairs $P = ptrs(p)$, while the scalar term o represents an set of scalar offsets $R = nums(r)$. The abstract value of P is ARM

$$p^{\#} = \alpha(P) = \{b \mapsto \hat{\alpha}(\{o \mid (b, o) \in P\}) \mid (b, o) \in P\} \tag{5.3}$$

we denote the abstract itv of all offsets with same base b $\hat{\alpha}(\{o \mid (b, o) \in P\})$ with $\hat{o}$. The abstract value of R is itv

$$\hat{r} = \hat{\alpha}(R) = [l', h'] = [\min R_{i \in \mathbb{N}}, \max R_{i \in \mathbb{N}}] \tag{5.4}$$

According to ARM abstract semantics in Figure 5.9, the result of the abstract version of $+_p$ ($+_{\#}$) between $p^{\#}$ and $\hat{r}$ is

$$p^{\#} +_{\#} \hat{r} = \{b \mapsto p^{\#}[b] \hat{+} \hat{r} \mid b \in p^{\#}\} \tag{5.5}$$

For convenience, we write $P +_{\mathcal{D}_p} R$ as shorthand of $ptrs(p +_p r)$. According to concrete semantics of $+_p$ in Figure 5.6, the set of pointer pairs K represent by new pointer term k after addition is

$$\begin{aligned} K = ptrs(k) &= P +_{\mathcal{D}_p} R \\ &= \{(b, o +_s o') \mid (b, o) \in P, o' \in R\} \end{aligned}$$

Thus the abstract value of K is

$$\begin{aligned} k^{\#} = \alpha(K) &= \alpha(\{(b, o +_s o') \mid (b, o) \in P, o' \in R\}) \\ &= \{b \mapsto \hat{\alpha}(\{o +_s o' \mid (b, o) \in P, o' \in R\}) | \ (b, o) \in P, o' \in R\} \end{aligned}$$

Lemma 3. *For all pointer term p and $k = p +_p r$ with ARMs $p^{\#}$ and $k^{\#}$, $bases(p^{\#}) \subseteq bases(k^{\#})$, where $p^{\#} = \alpha(ptrs(p))$ and $k^{\#} = \alpha(ptrs(p +_p r))$.*

Proof. From (5.5), $k^{\#}$ has the same set of keys (bases) as $p^{\#}$ and $p^{\#} +_{\#} \hat{r}$. □

Lemma 4. *For all pointer term p and $k = p +_p r$ with ARMs $p^{\#}$ and $k^{\#}$, $\alpha(ptrs(k))[b] \hat{\sqsubseteq} k^{\#}[b]$ is True for all base b in p and k.*

Proof. For each unique base b in K, the set of all unique offset values is

$$O' = \{o +_s o' \mid (b, o) \in P, o' \in R\}$$

with abstract itv

$$\begin{aligned}\hat{\alpha}(O') &= [\min\ O'_{i\in\mathbb{N}}, \max\ O'_{i\in\mathbb{N}}] \\ &= [\min\ O_{i\in\mathbb{N}} + \min\ R_{i\in\mathbb{N}}, \max\ O_{i\in\mathbb{N}} + \max\ R_{i\in\mathbb{N}}]\end{aligned}$$

where O denotes the set of offsets $\{o|(b,o) \in P\}$ per unique base b. From the semantics of $\hat{+}$ in Figure 5.3, Equation (5.3) and Equation (5.5), we can see that per unique base b

$$\alpha(K)[b] = \hat{\alpha}(O') \sqsubseteq \hat{o} \hat{+} \hat{r} = (p^{\#} +_{\#} \hat{r})[b]$$

□

With Lemma 3, Lemma 4 and semantics of ARM $\sqsubseteq$ in Figure 5.9, we prove that

$$\forall P \in \mathcal{D}_p, \forall R \in \mathcal{D}_s, \forall p^{\#} \in {\mathcal{D}_p}^{\#}, \forall \hat{r} \in \mathsf{itv}\ .\ \alpha(P) \sqsubseteq p^{\#} \wedge \hat{\alpha}(R) \sqsubseteq \hat{r} \Rightarrow \alpha(P +_{\mathcal{D}_p} R) \sqsubseteq p^{\#} +_{\#} \hat{r}$$

is True.

Theorem 2. *The abstract semantics of ARM* $\sqcup$ *and* $+_{\#}$ *is* ***sound*** *wrt. concrete semantics of* $\mathcal{D}_p$ *ite and* $+_p$ *.*

Proof. Recall for all ARMs $p^{\#}$ and $q^{\#}$, $p^{\#} \sqsubseteq q^{\#}$ is True iff 1. the set of bases in $p^{\#}$ B_p and set of bases in $q^{\#}$ B_q satisfy $B_p \sqsubseteq B_q$ and 2. for all $b \in B_p \sqcap B_q$, the mapped itvs satisfy $p^{\#}[b] \sqsubseteq q^{\#}[b]$.

For $\sqcup$ wrt. *ite*, constraint 1 is established by Lemma 1. and constraint 2 is established by Lemma 2.

For $+_{\#}$ wrt. *ite*, constraint 1 is established by Lemma 3 and constraint 2 is established by Lemma 4. □

With soundness of abstract transformers $\sqcup$ and $+_{\#}$ established in Theorem 2, we have established that **ARM** is a *sound* over-approximation of $t_{\mathcal{P}}$: given a set of pointers ϕ represented by a pointer term $t_{\mathcal{P}}$ p and the **ARM** of p $\phi^{\#} = \alpha(ptrs(p))$, then $\phi \sqsubseteq \gamma(\phi^{\#})$.

Given two pointer expressions i and j that represent sets of pointers ϕ and ψ respectively, empty intersection between the sets implies dis-equality between i and j.

$$\phi \sqcap \psi = \emptyset \iff i \neq_p j$$

Theorem 3. *The abstract intersection transformer* $\sqcap$ *for ARM is sound wrt. concrete set intersection between sets of pointers base-offset pairs:*

$$\forall P, Q \in \mathcal{D}_p, \forall P^{\#}, Q^{\#} \in {\mathcal{D}_p}^{\#}\ .\ \alpha(P) \sqsubseteq P^{\#} \wedge \alpha(Q) \sqsubseteq Q^{\#} \Rightarrow \alpha(P \sqcap Q) \sqsubseteq P^{\#} \sqcap Q^{\#}$$

Proof. Two sets of pointer pairs ϕ and ψ are over-approximated by their ARM abstractions $\phi^{\#}$ and $\psi^{\#}$. The intersection between ϕ and ψ is

$$\rho = \phi \sqcap \psi$$

and the ARM approximation of ρ is

$$\alpha(\rho) = \{b \mapsto \hat{\alpha}(\{o \mid (b, o) \in \phi \sqcap \psi\}) \mid (b, o) \in \phi \sqcap \psi\}$$

while the abstract intersection between $\phi^{\#}$ and $\psi^{\#}$ is

$$\phi^{\#} \ \sqcap \ \psi^{\#} = \{b \mapsto \phi^{\#}[b] \ \hat{\sqcap} \ \psi^{\#}[b] \mid b \in \phi^{\#} \wedge b \in \psi^{\#}\}$$

First we can observe that $\alpha(\rho)$ share the same set of *keys* (base) with $\phi^{\#} \ \sqcap \ \psi^{\#}$. Next, we need to establish the relationship between mapped *values* (offset) for each *key* (base). For every unique base b in both ϕ and ψ, $\phi^{\#}[b]$ and $\phi^{\#}[b]$ intervals over-approximate corresponding sets of scalar offsets. Recall that itv abstract $\hat{\sqcap}$ is sound wrt. set intersection. We can now see that for each unique b in both ϕ and ψ, $\alpha(\{o \mid (b, o) \in \phi \sqcap \psi\}) \ \sqsubseteq \ \phi^{\#}[b] \ \hat{\sqcap} \ \psi^{\#}[b]$. With *keys* and *values* both satisfying the condition for abstract $\sqsubseteq$ in Figure 5.9, we have established $\alpha(\rho) \sqsubseteq \phi^{\#} \ \sqcap \ \psi^{\#}$. □

Theorem 4. *Two pointer terms p and q are not equal if $\alpha(ptrs(p)) \sqcap \alpha(ptrs(q)) = \bot$.*

Proof. From the soundness of abstract $\sqcap$ operation, we can use results of $\sqcap$ between ARMs to impliy dis-equalities between $t_{\mathcal{P}}$s. Assume two $t_{\mathcal{P}}$s p and q represent two sets of pointer pairs ϕ and ψ; ϕ and ψ are over-approximated by their ARM abstractions $\phi^{\#}$ and $\psi^{\#}$. If intersection between $\phi^{\#}$ and $\phi^{\#}$ is $\bot$, then we know $\alpha(\phi \sqcap \psi) = \bot$ must be *True* since $\alpha(\phi \sqcap \psi) \sqsubseteq \phi^{\#} \ \sqcap \ \psi^{\#}$. We can then deduce $\phi \sqcap \psi$ is *emtpy*, and $p \neq_p q$. □

Take the example $t_{\mathcal{P}}\ p_1$ from Figure 5.7 again. $t_{\mathcal{P}}\ p_2 = ite(k, obj_8, obj_1 6)$ has abstract value ARM $\{obj_8 \mapsto [0, 0], obj_1 6 \mapsto [0, 0]\}$. The abstract value of $\sqcap$ between the ARMs of p_1 and p_2 is

$$\begin{aligned} \alpha(ptrs(p_1)) \sqcap \alpha(ptrs(p_2)) &= \{obj_8 \mapsto ([0, 0] \ \hat{\sqcap} \ [24, 36]), obj_{16} \mapsto ([0, 0] \ \hat{\sqcap} \ [8, 8])\} \\ &= \{obj_8 \mapsto \bot, obj_{16} \mapsto \bot\} \\ &= \bot \end{aligned}$$

The $\bot$ result implies that $p_1 \neq_p p_2$: regardless of the values of condition terms i, j and k, the set pointers represented by p_1 does not overlap with the set represented by p_2. Note that the converse of Theorem 4 is not True. Take a $t_{\mathcal{P}}\ p_1' = ite(\neg i, ptrOf(obj_8, 16) +_p ite(j, 8, 16))$, which is the same as p_1 other than inverted condition term i. Clearly $p_1 \neq_p p_1'$, but p_1' and p_1 have identical ARMs.

Time and space complexity of ARM Given two $t_{\mathcal{P}}$s with syntactic size n, the space complexity of of corresponding ARMs is $O(n)$ with a hash map based implementation since each node in $t_{\mathcal{P}}$ AST corresponds to at most 1 *key-value* pair in an ARM. In order to over-approximate pointer dis-equality using Theorem 4, we first need to construct two ARMs by traversing the AST of each $t_{\mathcal{P}}$ and applying the abstract semantics from Figure 5.9. The abstract semantics of $+_{\#}$ and $\sqcup$ both perform linear time-complexity operations on children of current node. Therefore the time complexity during the ARM construction stage is $n \cdot O(n) = O(n^2)$. Finally, the abstract semantics of $\sqcap$ between two constructed ARMs is linear to the size of ARMs ($O(n)$). Overall the time complexity is

$$T(n) = T_{\text{constr}} + T_{\text{inter}} = O(n^2) + O(n) = O(n^2)$$

which is better than the *exponential* complexity of pointer term comparisons with **ITE-PUSH** and **ITE-PULL** applied on terms with nested ITEs. In practice, SEAM without ARM skips comparisons between $t_{\mathcal{P}}$s containing nested ITEs: if the comparison is conclusive, cost in terms of time is too expensive; if the comparison is inconclusive, the time cost is spent in vain since rewritten VC would explode in syntactic size instead of being simplified. With ARM, more pointer comparisons can be performed without rewriting of the $t_{\mathcal{P}}$s at much smaller cost.

Chapter 6

Compressing *write* with Store-Map

In previous chapters we have introduced the VC translation semantics of SeaM with Theory of Memory $\mathcal{T}_{\mathcal{M}}$, which preserves C memory semantics and allows for more conclusive pointer comparison during simplification. We have also introduced ARM, which provides more efficient pointer comparison by over-approximating complex pointer expressions containing nested ITEs. To apply the simplifications rules detailed in Figure 4.6 and Figure 4.7, SeaM needs to traverse the data structure representing current *sequence* of memory writes during the VCGen process of a ROW with pointer index i, and check pointer dis-(equality) between i and each *write* index j. There are two data structures commonly used for representing array/memory write sequences in static analysis.

Linked-list Array/memory write sequences are commonly represented by *nested write* expressions. The term $write(a, idx, val)$ is analogous to a *linked-list* node storing a pair of values (idx, val), where the nested array/memory term a can be seen as the "pointer" to *next* node. For example, the write sequence from Figure 3.2 can be viewed as a linked-list of *write*s as illustrated in Figure 6.1. The "head" of the linked-list provides access to the *latest write*, while the linked-list always ends with an emtpy memory symbol representing the initial state. The linked-list data structure is *generic*, supporting both constant indices and symbolic ones like the *ite* term at the head of Figure 6.1. The *temporal* order of *write* operations is also perfectly maintained. Note that *temporal* order refers to the order in which instructions are executed. For example, in Figure 3.2, the write to obj_8 is *temporally* earlier than the write to obj_{16}. However, like all linked-list based data structures, *nested write*s does not support *random access*. ROW simplifications on *nested write*s require linear traversal of the linked-list. Therefore, time-complexity of ROW simplifications on

linked-list *write* sequence is **linear** to the number of *write*s. Given n *write*s and m *read*s, overall simplification time-complexity is $O(m \cdot n)$ with only ROW simplifications.

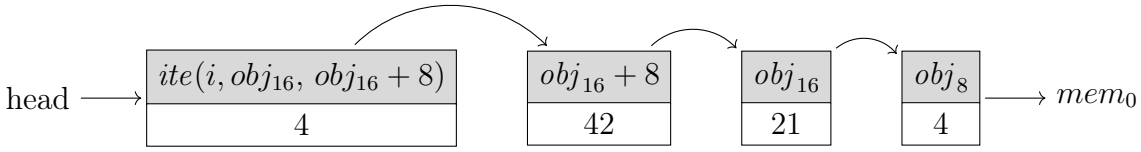


Figure 6.1: Memory write sequence represented as a linked-list.

Map In special cases where indices of all *write* operations are *pairwise comparible*, ie. linear arithmetic $\leq$ operator is defined between two indices, it is sound to ignore the *temporal* order of *write* operations with *different* indices and store *write* sequences in a *map* with one key-value pairs per unique index. A *write* would replace the value written by a previous *write* with identical index in the map, so the *temporal* ordering between *write*s with same index is still maintained. ROW simplifications are performed with a map lookup, which will always result in *full* simplification: only the value written to the index being read will be returned. ROW simplifications is *efficient* with running-time **logrithmic** to the number of *write*s. However, indices that are not *pairwise comparible* with the rest cannot be represented by such map-like data structure. For example, the map in Figure 6.2 can only represent the memory state before the last write with an *ite* index. A purely map-based data structure is therefore not *generic* enough for application in BMC.

obj_8	$\mapsto 4$
obj_{16}	$\mapsto 21$
$obj_{16} + 8$	$\mapsto 42$

Figure 6.2: Part of a memory write sequence represented as a map.

6.1 Store-Map data structure

SeaM represents memory write history with nested $\mathcal{T}_{\mathcal{M}}$ memory terms, which is essentially a Linked-list data structure. A recent work in [18] proposed a new data structure *map list*

for representing array writes. *Map lists* groups **array** *write*s into a list of maps, where each map contains a group of *writes* with **pairwise comparable** indices. In theory, *map list* is both **generic** like a linked-list-based data structure and **efficient** like a map-based data structure. Previously, $\mathcal{T}_{\mathcal{M}}$ introduced C-based semantics for **pointer pairwise comparison** in Figure 4.2, which states that $\leq_p$ operator is defined between two pointer terms if the *base*s of the pointers are identical, and the *offsets* are **pairwise comparable**. $\mathcal{T}_{\mathcal{M}}$ creates the basis for grouping $t_{\mathcal{P}}$s in a similar manner as *map list* nodes, which supports **logarithmic**-time ROW simplifications within each group.

We expand the syntax of $\mathcal{T}_{\mathcal{M}}$ with a new σ_M signature representing a memory term named *store-map* : $\sigma_M \times \sigma_P \times \{\sigma_S \mapsto \sigma_S\} \to \sigma_M$:

$$t_{\mathcal{M}} \quad ::= \quad ... \quad | \quad \textit{store-map}(t_{\mathcal{M}}, t_{\mathcal{P}}, {t_{\mathcal{S}}}^{32} \mapsto {t_{\mathcal{S}}}^{32})$$

A *store-map* term represents a group of memory writes within the same memory object obj_n — each index is comparable with $\leq_p$ with the rest. The base address of the memory object is represented by the second $t_{\mathcal{P}}$ argument of *store-map*, always taking the form $ptr(obj_n, 0)$. The first $t_{\mathcal{M}}$ argument represents the state of memory with writes that are either: (1) *temporally* earlier in execution order, and with addresses that are not pairwise comparable to any of the pointers in *store-map*, or (2) with addresses that are provable to be in different objects which are allocated earlier than obj_n. The third argument of a *store-map* is a map between offsets o from obj_n base to scalar values written to $ptr(obj_n, o)$. The same memory state previously represented in *Linked-list* and *map* representations can be represent in STORE-MAP, as illustrated in Figure 6.3. The two writes to obj_{16} and $obj_{16} + 8$ are compressed into a *store-map* node.

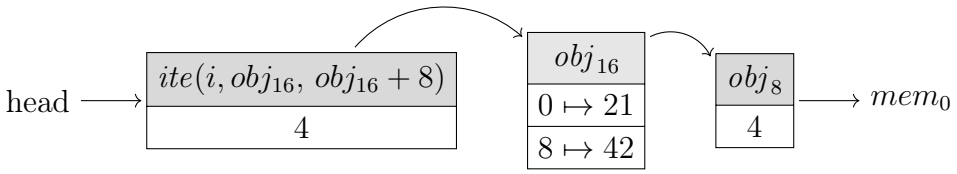


Figure 6.3: Memory write sequence represented as a linked list with a STORE-MAP node.

To construct and update *store-map* during VC, new rewrite rules are applied during *sym* translation of *write*-over-*write* (WOW) as shown in Figure 6.4. A *write* over an existing $t_{\mathcal{M}}$ a at address i is rewritten as follows:

SMAP-NEW If a is another *write* at address j, and i and j share the common object base β, the two *write*s are merged into a *store-map*.

$$\frac{\vdash i =_p ptr(\beta, o_1) \quad \vdash j =_p ptr(\beta, o_2)}{\begin{array}{c} write\text{-}word(write\text{-}word(a, i, v_1), j, v_2) \rightsquigarrow \\ store\text{-}map(a, ptr(\beta, 0), \{o_1 \mapsto v_1, o_2 \mapsto v_2\}) \end{array}} \text{SMAP-NEW}$$

$$\frac{\vdash i =_p ptr(\beta, o_1)}{\begin{array}{c} write\text{-}word(store\text{-}map(a, ptr(\beta, 0), m), i, v) \rightsquigarrow \\ store\text{-}map(a, ptr(\beta, 0), m[o_1 \mapsto v]) \end{array}} \text{SMAP-HIT}$$

$$\frac{\vdash j <_t i}{\begin{array}{c} write\text{-}word(write\text{-}word(a, i, w), j, v) \rightsquigarrow \\ write\text{-}word(write\text{-}word(a, j, v), i, w) \end{array}} \text{WOW-COMMUTE}$$

$$\frac{\vdash i =_p ptr(\alpha, o) \quad \vdash ptr(\alpha, 0) <_t ptr(\beta, 0)}{\begin{array}{c} write\text{-}word(store\text{-}map(a, \beta, m), i, v) \rightsquigarrow \\ store\text{-}map(write\text{-}word(a, i, v), \beta, m) \end{array}} \text{WOSMAP-COMMUTE}$$

Figure 6.4: *write*-over-*write* rewrite rules related to creating and updating store-map.

SMAP-HIT If a is a *store-map* with object base β, which is the same base of i, then offset-value map of a is updated with the offset and value of the outer *write*.

We also add the rules WOW-COMMUTE, and WOSMAP-COMMUTE that commutes two consecutive $t_{\mathcal{M}}$s if the outer $t_{\mathcal{M}}$ pointer belongs to an object allocated *temporally* earlier (denoted by $<_t$: $\sigma_P \times \sigma_P \rightarrow \sigma_C$) than the inner $t_{\mathcal{M}}$. The *commute* rewrites help merge writes to a same object separated by *write*(s) to other objects. However, note that allocation *temporal* order $<_t$ between objects does not imply pairwise comparability $\leq_p$ between pointers belonging to the objects. The relationship $\leq_p$, in line with C semantics, is *spatial* and strictly defined between pointers in a same object.

ROW rewrite rules are also expanded to accommodate *read*s over *store-map*s. A *read* over *store-map* a with object base β at address i is rewritten following the rules:

R-HIT If β is proven equal to the base of i, **and** the offset-value map of a contains the offset of i, return the mapped value;

R-MISS If β is proven equal to the base of i, but the offset-value map of a does not contain the offset of i, recursively rewrite *read* over nested $t_{\mathcal{M}}$ of a at i;

R-SKIP If β is proven not equal to the base of i, recursively rewrite *read* over nested $t_{\mathcal{M}}$ of a at i;

$$\frac{\vdash i =_p ptr(\beta, o) \quad \vdash m[o] =_s e}{read\text{-}word(store\text{-}map(a, ptr(\beta, 0), m),\ i) \ \rightsquigarrow\ e}\ \text{R-HIT}$$

$$\frac{\vdash i =_p ptr(\beta, o) \quad \vdash o \notin m}{read\text{-}word(store\text{-}map(a, ptr(\beta, 0), m),\ i) \ \rightsquigarrow\ read\text{-}word(a, i)}\ \text{R-MISS}$$

$$\frac{\vdash i =_p ptr(\alpha, o) \quad \vdash \alpha \neq_s \beta}{read\text{-}word(store\text{-}map(a, ptr(\beta, 0), m),\ i) \ \rightsquigarrow\ read\text{-}word(a, i)}\ \text{R-SKIP}$$

$$\frac{\vdash m[o_1] =_s v_1 \quad \nvdash i =_p \beta \quad \nvdash i \neq_p \beta}{\begin{array}{c} read\text{-}word(store\text{-}map(a, ptr(\beta, 0), m),\ i) \ \rightsquigarrow \\ read\text{-}word(write\text{-}word(store\text{-}map(a, \beta, m \setminus o_1), ptr(\beta, o_1), v_1), i) \end{array}}\ \text{R-ABORT}$$

Figure 6.5: Rewrite rules for reading store-map

R-ABORT If neither equality nor dis-equality can be proven between the base of i and β, recursively rebuild *read* over nested *write*s with offset-value pairs in a.

SEAM with *store-map* is still *generic*: the *temporal* order of *write*s at symbolic addresses not *comparable* to others is maintained since they are neither grouped into *store-map*s nor *commuted* with neighbouring memory *write*s. Similar to *map list* from [18], the simplification performance of SEAM memory representation with *store-map* varies from that of a Linked-list data structure to that of a map data structure depending on programs under verification. The time-complexity of SEAM simplification with *store-map* can be analysis in two scenarios, with number of memory *write*s denoted by n and number of memory *read*s denoted by m:

Best-case scenario the program maintains a single memory object, the memory write sequence would be compressed into a single *store-map*; each WOW simplification inserts or updates an entry in an ordered map, so time-complexity of WOW simplifications is $T_{wow} = O(n \cdot log(n))$; each ROW simplification perform a lookup from the map, so overall ROW time complexity $T_{row} = O(m \cdot log(n))$. Overall time complexity is then $T = T_{row} + T_{wow} = O(n \cdot log(n)) + O(m \cdot log(n))$.

Worst-case scenario all memory writes in the program are at disjoint memory objects and the memory write sequence is a Linked-list sorted by allocation *temporal* order; WOW simplifications push writes linearly down the write sequence, performing essentially *bubble sort*, so time complexity of WOW simplifications is $T_{wow} = O(n^2)$;

each ROW simplification performs a linear search in the Linked-list, so overall *read*-over-*write* time complexity is $T_{row} = O(m \cdot n)$. Overall time complexity is then $T = T_{row} + T_{wow} = O(n^2) + O(m \cdot n)$.

If number of *write* and *read* are approximately the same ($m \approx n$), in a program close to ones in base-case scenario , SEAM with *store-map* can in theory out-perform configuration with linear data structure in terms of simplification time: $T = O(n \cdot log(n))$ is better than $T = O(n^2)$. Under the same assumption that $m \approx n$, programs close to the ones in worst-case scenario has overall simplifications time-complexity of $O(n^2)$, which is the same as linear configuration, meaning simplification time overhead would not be significant.

During implementation, integrating a ordered map data structure with logarithmic lookup and insertion into VC AST proves to be challenging. The VC AST wrapper data structure `Expr` was originally designed to be **immutable**. Sec. 7.2 describes the implementation details of STORE-MAP.

Chapter 7

Implementation

We have implemented SeaM as a memory representation in SeaBmc. This chapter describes two of the more interesting implementation details: 1. multi-step rewriter for custom VC simplification and 2. cached ordered map for Store-Map.

7.1 Multi-step rewriter

The VC ASTs are represented by the `Expr` data structure. `Expr` nodes are **immutable** and supports shared children, which essentially make `Expr` a DAG data structure. The *sym* VCGen process described in Figure 4.8 and ROW simplifications described in Chapter 4 and Chapter 5 all require an efficient and configurable rewriter for `Expr`. The rewriter should also support *multi-step* rewriting. For example, after a **ITE-PUSH** rewrite

$$ite(i, t, e) = x \;\rightsquigarrow\; ite(i, t = x, e = x)$$

the rewritten formula could be further simplified if the sub-formulas $t = x$ and $e = x$ are rewritten to *True* or *False*. Multi-step rewriting is particularly important for ROW simplifications. For example, after applying ROW rewrite rule in Figure 4.4

$$\textit{read-word}(\textit{write-word}(a, i, v),\, j) \;\rightsquigarrow\; ite(i =_p j, v, \textit{read-word}(a, j))$$

conclusive rewrite result on the sub-formula $i =_p$ opens opportunities for further applying ROW-HIT rule or ROW-SKIP rule. We have implemented a rewriter for VC ASTs supporting multi-step rewriting based on iterative post-order tree traversal with caching. The

rewriter design is similar to that implemented in Z3 [12]. A rewriter implementation is a template class containing the overall AST traversal procedure parameterized by a rewriter configuration class: `Rewriter<RewriterConfig>`. A `RewriterConfig` class defines the following procedures:

- `shouldRewrite(Expr e)` $\rightarrow$ `bool`: takes an `Expr` e as argument and returns Boolean dictating whether any rewrite rules are applicable to the expression;
- `applyRewriteRules(Expr e)` $\rightarrow$ (`Expr`, `rewrite_status`): applies a rewrite rule on `e` and returns a tuple containing the rewritten expression and a `rewrite_status`; `rewrite_status` serves as instruction for rewriter on next actions for the rewritten expression:

  `RW_DONE`: simplification is final;

  `RW_<n>`: rewrite expression bounded by n;

  `RW_FULL`: rewrite fully without bound;

  `RW_SKIP`: future visits on the rewritten expression should be skipped; should only return this status if `applyRewriteRules` would return `False` on the rewritten expression.

- `applyAfterRewriteActions(Expr oldE, Expr newE)` $\rightarrow$ *void*: given original expression `oldE` and rewritten expression `newE` from applying a rewrite rule, apply actions like update cache or update logging.

The rewriter performs post-order traversal of a `Expr` iteratively with the aid of two stacks. The first `rewrite_stack` holds data structures named `RewriteFrame`, which contains an expression under rewrite and fields recording the progress of its rewrite

```
RewriteFrame {
  Expr m_exp;        // the expression under rewrite
  size_t m_depth;    // number of levels to rewrite from this node
  size_t m_i;        // up to m_i th children have been rewritten
  bool m_rewriting; // this frame is currently under further rewrite
}
```

As shown in Alg. 1, the procedure `visit` tries visiting a `Expr` node in the expression and returns a Boolean result indicating whether rewrite should halt at the node. To prevent repeated visits on same `Expr` nodes, a cache from original `Expr` to fully rewritten version is maintained by the rewriter. If the node is not in the cache, depth limit has not been reached

```
In  : e : Expr to visit; depth : current depth limit
Out: True if e has been fully rewritten before or depth is 0, otherwise False
Procedure visit(e: int, depth: int):
    if depth = 0 or not config.shouldRewrite(e) then
        result_stack.push(e);
        return True;
    end
    if e is in cache then
        result_stack.push(cache[e]);
        return True;
    end
    if depth ≠ FULL then
        depth := depth −1;
    end
    F:= RewriteFrame(e, depth, 0, False);
    rewrite_stack.push(F);
    return False;
end
```

Algorithm 1: Procedure for trying to visit an `Expr` node.

and the `shouldRewrite` function under current config returns True, a new `RewriteFrame` is created with the node and pushed to the top of `rewrite_stack`.

The second stack maintained by the rewriter is `result_stack`. At the start of a rewrite, the rewriter calls `visit` on the *root* node, and pops the first element of `rewrite_stack` and processes it with procedure `processFrame` in Alg. 2. Lines 2-8 tries to visit each of the arguments, if any argument node needs to be visited, it is pushed to `rewrite_stack` and current `processFrame` exits. By line 9, all arguments of f are fully rewritten and are the top-most elements in `result_stack`. A new `Expr` is formed with rewritten arguments and rewrite rules contained in `RewriterConfig` is applied to rewrite it. If returned `rewrite_status` indicates that no further rewrite is possible (`RW_DONE`) or necessary (`RW_SKIP`), the rewritten `Expr` is pushed to top of `result_stack`. If `rewrite_status` is `RW_DONE`, the rewritten result is also cached. If returned status is `RW_n`, it means rewriter should apply further rewrite up to n levels. The rewriter pushes a new `RewriteFrame` containing the intermediate rewritten value and *depth*. Note that `RW_n` is set to a integer constant of value n, so the rewriter passes `RW_n` for *depth* argument. When `rewrite_stack` is empty, the final rewritten version of *root* should be on top of `result_stack`. The full outer loop of rewrite process is summarized in Alg. 3.

```
1  Procedure processFrame(f: RewriteFrame):
2      for i = f.m_i to size of f.m_exp.args do
3          inc f.m_i;
4          k:= f.m_exp.args[i];
5          if not visit(f.m_exp.args, f.m_depth) then
6              return
7          end
8      end
9      rewrite_stack.pop();
10     n := size of f.m_exp.args;
11     args := list from top n in result_stack;        /* get rewritten args */
12     result_stack.pop(n);
13     old := f.m_exp.op(a in args) ;                 /* renew with new args */
14     new := config.applyRewriteRules(old);
15     config.applyAfterRewriteActions(old, new);
16     if res.status := RW_SKIP then
17         cache multi-step;
18         result_stack.push(res.rewritten);
19     else if res.status := RW_DONE then
20         cache multi-step;
21         cache[f.m_exp] := res.rewritten;
22         result_stack.push(res.rewritten);
23     else
24         mark multi-step start;
25         rewrite_stack.push(RewriteFrame(res.rewritten, res.status, 0, False));
26     end
27 end
```

Algorithm 2: Procedure for processing an `Expr` node.

```
e    : Root node to be rewritten.
visit(e);
while rewrite_stack not empty do
    processFrame(rewrite_stack.top);
end
return result_stack.top;
```

Algorithm 3: Full rewrite loop.

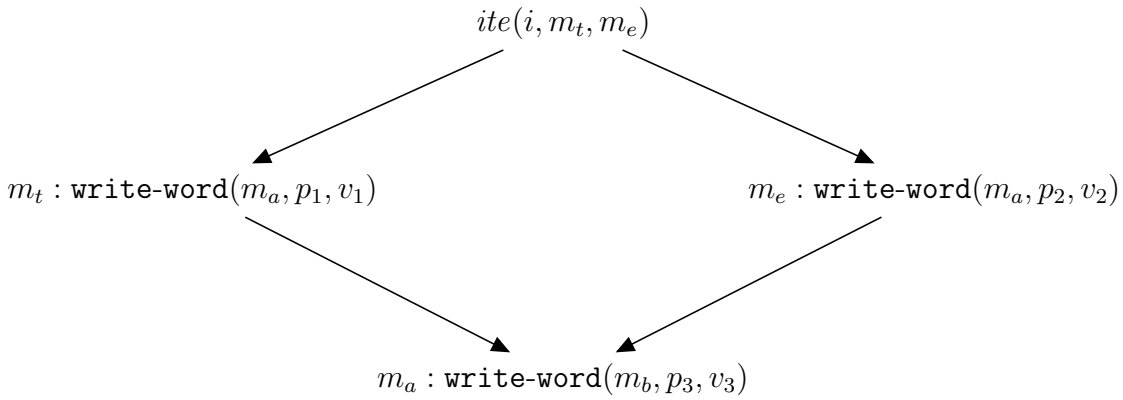


Figure 7.1: Example `Expr` of a $t_{\mathcal{M}}$ under rewrite.

Multi-step caching Consider a multi-step rewrite

$$a \rightsquigarrow b \rightsquigarrow c$$

The expected caching should be $a \mapsto c$. However, following Alg. 2, the rewrite from a to b returns a `RW_n` status, so a would not be cached. The rewrite status from b to c is `RW_DONE`, which adds to cache $b \mapsto c$. Failing to properly cache mult-step rewrite could result in higher cost in time. Take the example $t_{\mathcal{M}}$ shown in Figure 7.1, in which both m_e and and m_t contain memory state m_a. Without proper caching on m_a's multi-step rewrite, it would be rewritten twice, each time from m_t and m_e, in which case the example $t_{\mathcal{M}}$ is no longer treated as a DAG but a tree.

To properly cache such multi-step rewrites, we add procedure *mark multi-step start* as described in Alg. 4 during handling of `RW_n` status, which marks `m_rewriting` field in `RewriteFrame` f to True and pushes f back on top of `rewrite_stack`. During handling of `RW_DONE` and `RW_SKIP`, if the top element of `rewrite_stack` f' is marked with `m_rewriting`, then f' will be removed from stack and the cache will be properly updated as shown in Alg. 5.

Theorem 5. *`Expr` rewriter with multi-step caching rewrites in linear time to the syntactic size of expressions.*

Proof. Any node `e` in an `Expr` would not be visited more than once regardless of result of `applyAfterRewriteActions(e)`:

- `RW_SKIP`: `shouldRewrite(e)` must return `False`, so `e` would be skipped by next `visit` call;

f : current processed `RewriteFrame`
if `rewrite_stack` *is empty or* `rewrite_stack.top.m_rewriting` = *False* **then**
  f.m_rewriting := True;
  `rewrite_stack.push` (f);
end

Algorithm 4: Mark multi-step

res : rewrite result
if `rewrite_stack` *not empty and* `rewrite_stack.top.m_rewriting` = *True* **then**
  `cache[rewrite_stack.back.m_exp]` := res.rewritten;
  `result_stack.pop()`;
end

Algorithm 5: Cache multi-step

- `RW_DONE`: `e` and rewritten result would be cached, next `visit(e)` would directly use cached result;
- `RW_n`: `e` is marked with `m_rewriting` and pushed back to `rewrite_stack`; when `RW_DONE` is returned on intermediate `Expr`, marked `e` would be removed from `rewrite_stack` and cached.

□

Discussion During development of the first iteration of the `Expr` rewriter, we took much inspiration from Z3's rewriter[1]. Multi-step rewriting with different `rewrite_status` was one of the designs we ported, but we noticed unusual time consumption when extensive multi-step rewriting was required for Store-Map R-ABORT rewrite rule, leading to the implementation of *multi-step caching*. It is possible that Z3 rewriter does not handle *multi-step* caching or handles in very different manner, but such performance issue in a SMT solver rewriter is outside of the scope of this thesis.

7.2 Cached ordered maps for Store-Map.

Recall that the `Expr` data structure is **immutable**, ie. individual argument nodes cannot be replaced without creating a new parent node. For Store-Map, this means the offset-

[1]Publicly available at https://github.com/Z3Prover/z3/tree/master/src/ast/rewriter

value map cannot be implemented directly as binary search tree or ordered list with `Expr` nodes, since *inserting* into such data structure would require copying all existing entries to build a new instance with the inserted entry. Instead, the "offset-value map" part of a *store-map* term is represented by a nested Linked-list-like data structure that bears similarity to `cons` list in Lisp. We refer to this data structure as offsets-value `cons` list (OCL). An OCL is ordered by insertion order. The real map with keys ordered by pointer offset is an instance of C++ stl `ordered_map`. We maintain a cache from *store-map*s to stl `ordered_map`s.

Given a *store-map* s with OCL c and cached stl `ordered_map` m, during an insertion from SMAP-HIT, a new OCL c' is built in constant time with two arguments: 1. the inserted offset-value pair and 2. the previous list c. A new *store-map* s' is created with the new OCL. We also insert offset-value pair to `m` in logarithmic time, and move m from previous key s to new key s' in cache. We illustrate the process in Fig. 7.2. Note that only *store-map*s from with the latest revision will maintain a cached offset-value ordered map for logarithmic lookup. This design does mean *write* over *store-map*s that has been written to would result in *store-map cache miss* and a offset-value ordered map would not be available; ROW simplification lookup can only be done by linearly searching OCL. However, in experiments detailed in Sec. 8.3, SEAM configured with current implementation of STORE-MAP out-performs the linear configuration, indicating insignificant negative impact from *store-map cache miss*.

```
write-word(
  store-map(
    a,
    ptrOf(β, 0),
    cons((0, 21),
      cons((8, 42), nil)
      )
  ),
  ptrOf(β, 16),
  8
)
⇝
store-map(
  a,
  ptrOf(β, 0),
  cons((16, 8),
    cons((0, 21),
      cons((8, 42), nil)
    )
  )
)
```

(a) Rewrite from *store-map*$(s, ptr(\beta, 16), 8)$ to s'

```
{
  ...
  s ↦ {
    0 ↦ 21,
    8 ↦ 42
  }
  ...
}
↪
{
  ...
  s′ ↦ {
    0 ↦ 21,
    8 ↦ 42,
    16 ↦ 8
  }
  ...
}
```

(b) Updates in Expr $\mapsto$ stl map cache

Figure 7.2: Insertion process with OCL and cached stl map.

Chapter 8

Evaluations

In this chapter, we evaluate the performance of SeaM with a series of experiments. As described in Chapter 7, SeaM is implemented as part of SeaBmc. In Sec. 8.1, we compare the *overall performance* of SeaM against two existing memory representations implemented in SeaBmc: 1. **Arrays**, which encodes memory operations with basic theory of arrays; 2. **Lambdas**, which encodes memory operations with $\mathcal{T}_{\mathcal{ASC}}$ simulated with λ-expressions. We evaluate *overall performance* in terms of *soundness*, measured by number of correct results out of all verification tasks, *SMT solving time* of VC generated and *pre-processing time*, measured by the total time spent on rewriting and simplifying VC formulas. The evaluation benchmark are BMC tasks from Verify C Common project. Each task verifies the representation invariant and memory safety of a C function implemented in the `aws-c-common` library. These tasks are chosen as benchmark since they are based on C programs containing real-world patterns of memory usage and manipulation. All tasks are expected to yield result of `unsat`, so any `sat` result would be recognized as *failures*. All verification tasks are publicly available at https://github.com/seahorn/verify-c-common.

Next, we evaluate the individual impacts of the two optimization techniques - ARM and Store-Map, towards the overall performance. In Sec. 8.2, we evaluate the impact of ARM on verifying `aws-c-common` BMC tasks. The comparison is done on two configurations of SeaM: with or without ARM. As additional performance metrics, we also look at *syntactic size*: number of unique nodes in final VC DAG, and *ROW skips*: number of ROW simplifications based on axoims (A2) and (A3).

In Sec. 8.3, we evaluate the impact of SeaM on *pre-processing time*, particularly with Store-Map. For this experiment, we compare the *pre-processing time* among Lambdas and two configurations of SeaM (with and without Store-Map). The benchmark C

programs are adopted from Sv-Comp's **array-crafted** benchmarks. The reason for using Sv-Comp tasks over `aws-c-common` tasks is that pre-processing time in general are very low for all `aws-c-common` tasks. Crafted progams from Sv-Comp contain repeated access and modification to memory in loops, thus they translate to longer ROW sequences. The tasks are publicly available at a forked branch of the Verify-C-Common project[1]. To further explore the impact on scalability in terms of the correlation between pre-processing time and size of ROW sequences, we look into a case study created by isolating a single task bAnd and perform verification with increasing size of ROW sequences.

All experiments are conducted on a Linux machine with 2 Intel® Xeon® E5-2680 8-core (32 threads in total) CPUs and 64GB of memory.

8.1 Overall performance

In this section we evaluate the overall performance of **SeaM** as part of SeaBmc in terms of *soundness*, *solving time* and *pre-processing time* against the two alternative memory representations in SeaBmc: **Arrays** and **Lambdas**. For this set of experiments, we use SeaBmc to verify 157 `aws-c-common` tasks with each of the three memory representations. We repeat the experiment with two SMT solvers integrated to SeaBmc: Z3 and Yices2. Apart from memory representation and solver backend, configuration of SeaBmc is the same accross all experiments: using the *optimal* strategy described in Section 4 of [29]. Note that *pre-processing* is the same between SeaBmc configurations with different solvers: **SeaM** with a combination of custom rewriter and Z3 simplifier; **Arrays** and **Lambdas** with Z3 simplifier only. SeaBmc does not leverage stand-alone simplifier from Yices2.

We present the results with Z3 in Tab. 8.1 and the results with Yices2 in Tab. 8.2 respectively. For each configuration under different memory representation, we show the average and total pre-processing time and SMT solving time of tasks that *finished within timeout limit* per category. Note that the timeout threshold for each verification task is 1,200 seconds. Certain tasks under configurations with Arrays exceeded the timeout limit or failed with wrong results; the number of *failures* and *timeouts* per category are presented under the **fld/to** column. Total timing results of categories with *failures* and *timeouts* are highlighted in red. *Total* best results of the memory representation with no timeouts or fails are highlighted in **bold** text.

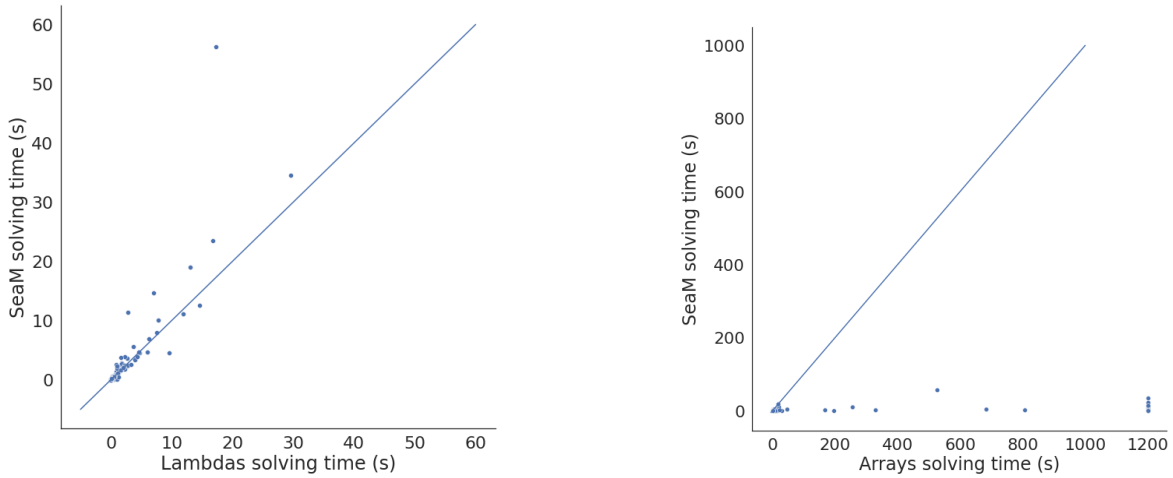
[1] Accessible at https://github.com/danblitzhou/verify-c-common/tree/xiang-thesis-bench/seahorn/jobs_bench

SeaBmc under both **Lambdas** and **SeaM** finished all verification tasks with correct results within timeout limit for both solvers, while there are 3 fails along with 29 timeouts and 2 fails for **Arrays** with Z3 and Yices2 respectively.

Tab. 8.1 shows that overall Z3 solves VCs produced by **Lambdas** the fastest. However, **SeaM** produced favourable or equal results in terms of solving time in 6 out of the 13 categories. The side-by-side comparison of solving time per task between **SeaM** and **Lambdas** is shown in Figure 8.1a. Overall, **SeaM** greatly out-performs **Arrays** in terms of solving time, producing VCs with faster or equal solving time by Z3 in 11 out of the 13 categories. In both categories (*array* and *hash_iter*) where **Arrays** beats **SeaM** in solving time, SeaBmc with **Arrays** timed out in 1 out of 4 and 4 out of 5 tasks respectively. The side-by-side comparison of solving time per task between **SeaM** and **Arrays** is shown in Figure 8.1b.

		SeaM				**Lambdas**					**Arrays**			
categories	cnt	pre-proc (s)		solving (s)		pre-proc (s)		solving (s)		fld/to	pre-proc (s)		solving (s)	
		avg	total	avg	total	avg	total	avg	total		avg	total	avg	total
arithmetic	6	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
array	4	<1	**<1**	1	4	<1	**<1**	<1	**3**	0/1	<1	<1	<1	1
array_list	24	<1	**<1**	4	99	<1	**<1**	2	**58**	0/0	<1	**<1**	31	753
byte_buf	29	<1	**<1**	<1	21	<1	**<1**	<1	**11**	1/2	<1	<1	10	280
byte_cursor	24	<1	**<1**	<1	9	<1	**<1**	<1	**8**	0/1	<1	<1	2	46
hash_callback	3	<1	**<1**	3	**9**	<1	**<1**	5	15	0/0	<1	**<1**	5	17
hash_iter	5	<1	**<1**	7	37	<1	1	6	**32**	0/4	<1	1	<1	<1
hash_table	19	<1	**<1**	4	87	1	19	3	**70**	0/6	<1	14	81	1,055
linked_list	18	<1	**<1**	2	41	<1	1	1	**21**	0/8	<1	<1	101	1,011
others	2	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
priority_queue	6	<1	**<1**	<1	**2**	<1	**<1**	<1	**2**	0/0	<1	**<1**	<1	**2**
ring_buffer	2	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
string	15	<1	**<1**	2	**32**	<1	1	2	35	2/7	<1	<1	13	104
total	157		**<1**		341		22		**255**	3/29		15		3,269

Table 8.1: Overall BMC performance of **SeaM**, **Arrays** and **Lambdas** on `aws-c-common` benchmark with **Z3**. **cnt** , **fld** and **to** measures number of **total tasks**, **failed tasks** and **timeouts** per category respectively. Timeout threshold is 1,200 seconds.



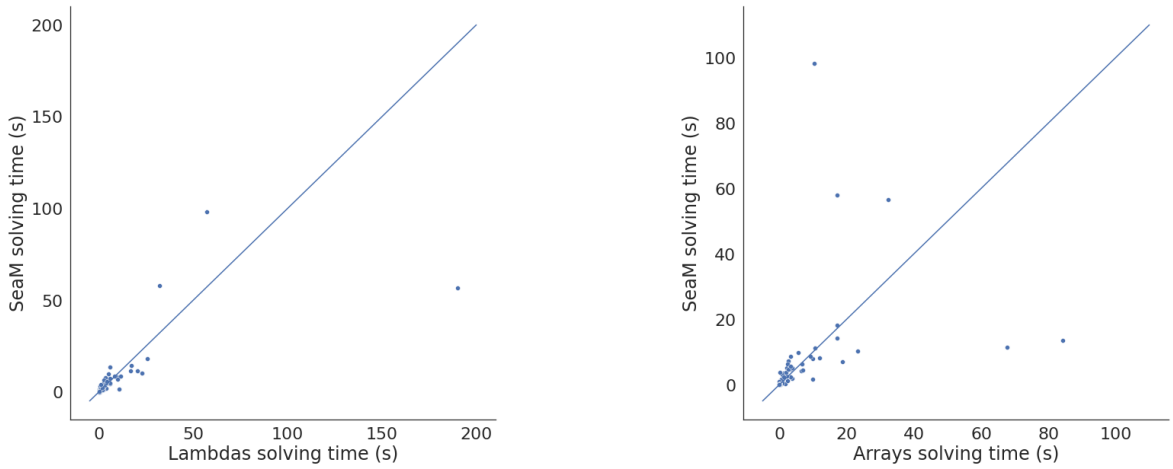
(a) SeaM vs. Lambdas (b) SeaM vs. Arrays

Figure 8.1: Per task solving time using Z3

Tab. 8.2 shows that overall Yices2 solves VCs produced by **Arrays** the fastest. However, **SeaM** produced faster or equal results in terms of solving time in 8 out of the 13 categories. Also, **Arrays** resulted in two failures in the *string* category. The side-by-side comparison of solving time per task between **SeaM** and **Arrays** is shown in Figure 8.2b. With Yices2, **SeaM** produces VCs with faster or equal solving time than **Lambdas** in 6 out of the 13 categories; total solving time of **SeaM** is shorter than **Lambdas**. The side-by-side comparison of solving time per task between **SeaM** and **Lambdas** is shown in Figure 8.2a. The plot indicates that the overall difference in solving time between **SeaM** and **Arrays** / **Lambdas** are skewed by a small number of cases; in the majority of `aws-c-common` tasks, solving time is comparable between **SeaM** and the alternative memory representations.

categories	cnt	SeaM pre-proc (s) avg	SeaM pre-proc (s) total	SeaM solving (s) avg	SeaM solving (s) total	Lambdas pre-proc (s) avg	Lambdas pre-proc (s) total	Lambdas solving (s) avg	Lambdas solving (s) total	fld/to	Arrays pre-proc (s) avg	Arrays pre-proc (s) total	Arrays solving (s) avg	Arrays solving (s) total
arithmetic	6	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
array	4	<1	**<1**	<1	2	<1	**<1**	<1	**1**	0/0	<1	**<1**	<1	**1**
array_list	24	<1	**<1**	3	**95**	<1	**<1**	5	123	0/0	<1	**<1**	7	174
byte_buf	29	<1	**<1**	<1	6	<1	**<1**	<1	**5**	0/0	<1	**<1**	<1	6
byte_cursor	24	<1	**<1**	<1	2	<1	**<1**	<1	**1**	0/0	<1	**<1**	<1	**1**
hash_callback	3	<1	**<1**	5	15	<1	**<1**	4	13	0/0	<1	**<1**	<1	**11**
hash_iter	4	<1	**<1**	4	22	<1	1	2	**10**	0/0	<1	1	3	22
hash_table	19	<1	**<1**	8	**162**	1	19	14	284	0/0	<1	14	4	176
linked_list	18	<1	**<1**	6	122	<1	1	3	69	0/0	<1	**<1**	9	**20**
others	2	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
priority_queue	6	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
ring_buffer	2	<1	**<1**	<1	**<1**	<1	**<1**	<1	**<1**	0/0	<1	**<1**	<1	**<1**
string	15	<1	**<1**	<1	25	<1	1	1	**17**	0/2	<1	<1	<1	11
total	157		**<1**		**451**		22		523	0/2		15		422

Table 8.2: Overall BMC performance of **SeaM**, **Arrays** and **Lambdas** on `aws-c-common` benchmark with **Yices2** solver. **cnt** , **fld** and **to** measures number of **total tasks**, **failed tasks** and **timeouts** per category respectively. Timeout threshold is set to 1,200 seconds.

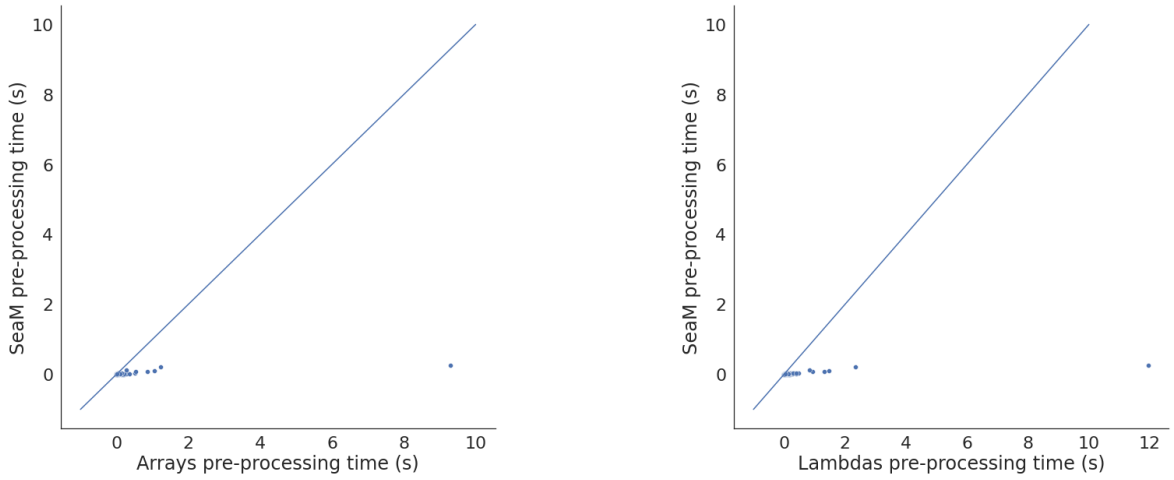


(a) SeaM vs. Lambdas (b) SeaM vs. Arrays

Figure 8.2: Per task solving time using Yices2

In terms of *per-processing time*, **SeaM** consistently out-performs the other two memory representations. The per-task comparisons are presented in Figure 8.3. Note that in general, *pre-processing time* is not significant in `aws-c-common` tasks. The BMC unroll

bound are general set below 100, resulting in short ROW sequences. Therefore, to study the impact of STORE-MAP, a technique aiming at reducing *pre-processing* overhead, we use crafted C programs from SV-COMP as benchmarks and apply larger BMC bounds in Sec. 8.3.



(a) SEAM vs. ARRAYS (b) SEAM vs. LAMBDAS

Figure 8.3: Per task pre-processing comparison

Conclusion Overall, SEABMC configured with **SEAM** produces VCs with **comparable, sometimes superior solving time** than **LAMBDAS** with both Z3 and Yices2. With Z3, **SEAM** produces VC with both **faster solving time** and **better soundness** than **ARRAYS**. With Yices2, **SEAM** produces VC with **better soundness** than **ARRAYS** at the cost of **slightly longer solving time**. Regardless of solver choice, **SEAM** achieves **negligible pre-processing time** overhead compared to the other memory representations. In conclusion, **SEAM** proves to be sound and highly performant in terms of solving time and pre-processing time for BMC on read-world industrial code base.

8.2 Simplification with ARM

In this section, we evaluate the impact of over-approximating pointer comparison with Address Range Map (ARM) on SEAM performance. In addition to *soundness*, *solving time* and *pre-processing time*, we also compare the metrics *syntactic size*: number of unique nodes in final VC DAG, and *ROW skips*: number of ROW simplifications.

Note on new metrics We consider *syntactic size* a reasonable metric only between configurations of the *same* memory representation, since different memory representations may use syntactically different expressions for equivalent semantics. For example, the two expressions in Figure 8.4 both represent a *read*-over-*write*. The expression `e1` in Figure 8.4a uses $\mathcal{T}_{\mathcal{A}}$ signatures resulting in a smaller syntactic size than the expression `e2` in Figure 8.4b ($9 < 13$). However, the comparison in syntactic size between `e1` and `e2` is not meaningful since they are semantically equivalent.

ROW skips contain ROW simplifications based on both *syntactic* pointer comparisons based on axiom 1 of Figure 4.2 and pointer comparisons resolved by ARM Theorem 4.

```
e1 = (select
         (store
              (store a p v)
              q
              w
         )
         i
     )
```

(a) Expression with $\mathcal{T}_{\mathcal{A}}$.

```
e2 = (ite
         (= i q)
         w
         (ite
              (= i p)
              v
              (select a i)
         )
     )
```

(b) Expression with `ite` only

Figure 8.4: Semantically equivalent expressions

We run SeaBmc configured with SeaM to verify 157 BMC tasks from `aws-c-common`. The experiments are repeated with ARM turned on (**SeaM**) and off (**SeaM-No-Arm**). Final VCs from both configurations are solved with Z3 and Yices2. The results are presented in Tab. 8.3.

Compared to configuration without ARM, SeaM with ARM resolves 42 more ROW skips on average accross all `aws-c-common` tasks. There are no *failures* in either configurations, indicating the **soundness** of ROW skips by ARM. Figure 8.5 plots the extra ROW skips resolved by ARM against syntactic size reduction. The plot indicates a positive linear correlation between number of ROW skips introduced by ARM and reduction in syntactic size. The extra ROW skips resulted in syntactic size reduction of 1.2× on average accross all `aws-c-common` tasks. The per-task comparison on syntactic size is shown in Figure 8.6.

	solving (s)				pre-proc. (s)		syntac. size		total skips	
	Z3		Yices2							
config	avg	sum	avg	sum	avg	sum	avg	sum	avg	sum
SEAM	**2**	**346**	**2**	**456**	<1	3	**745**	**116983**	**42**	**6639**
SEAM-NO-ARM	3	502	6	1051	<1	3	920	144473	<1	27
avg diff	×1.5		×3.0		×1.0		×1.2		+42	

Table 8.3: BMC performance of **SEAM** with and without ARM on `aws-c-common` benchmark with **Z3** and **Yices2**. Timeout threshold is 1,200 seconds.

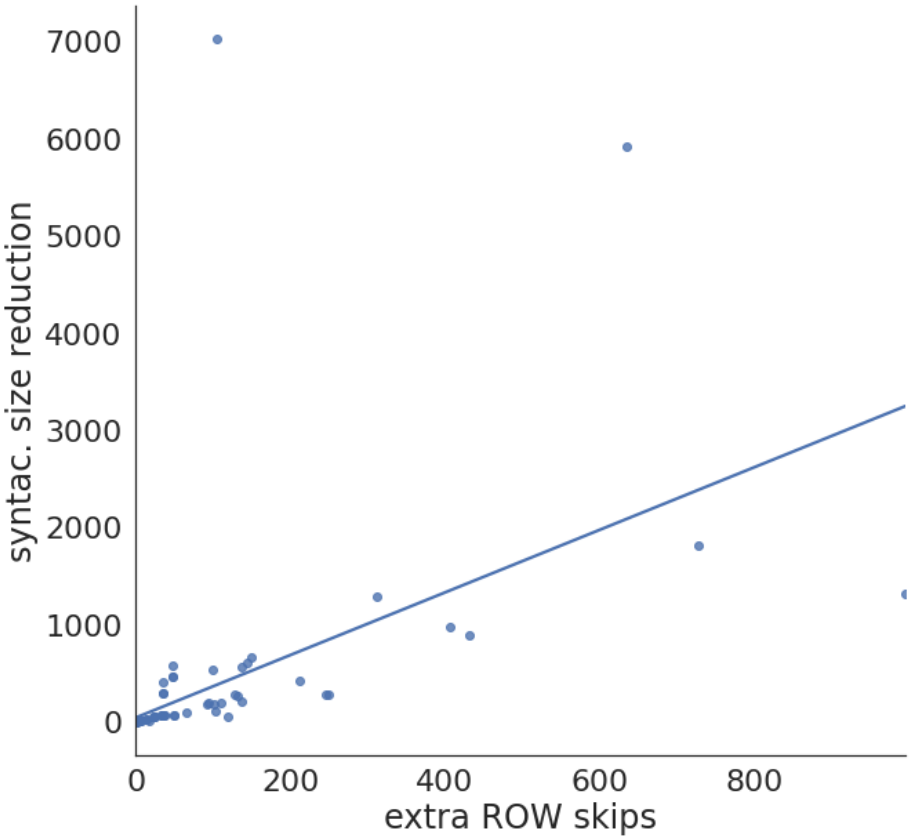


Figure 8.5: # of extra ROW skips vs. reduction in syntactic size.

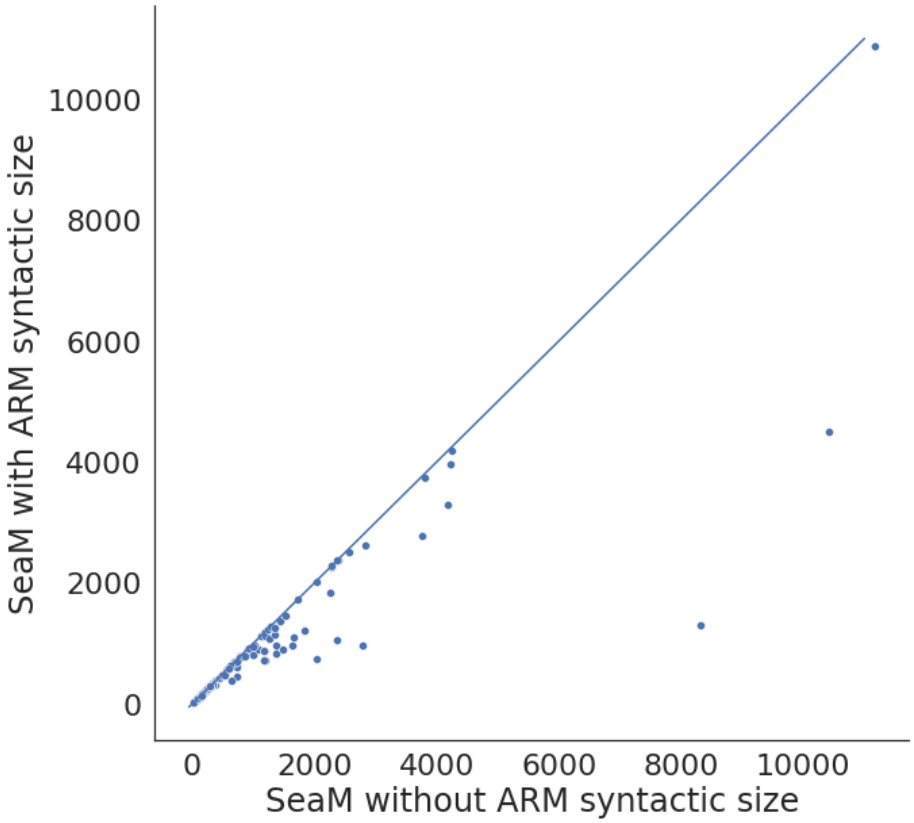
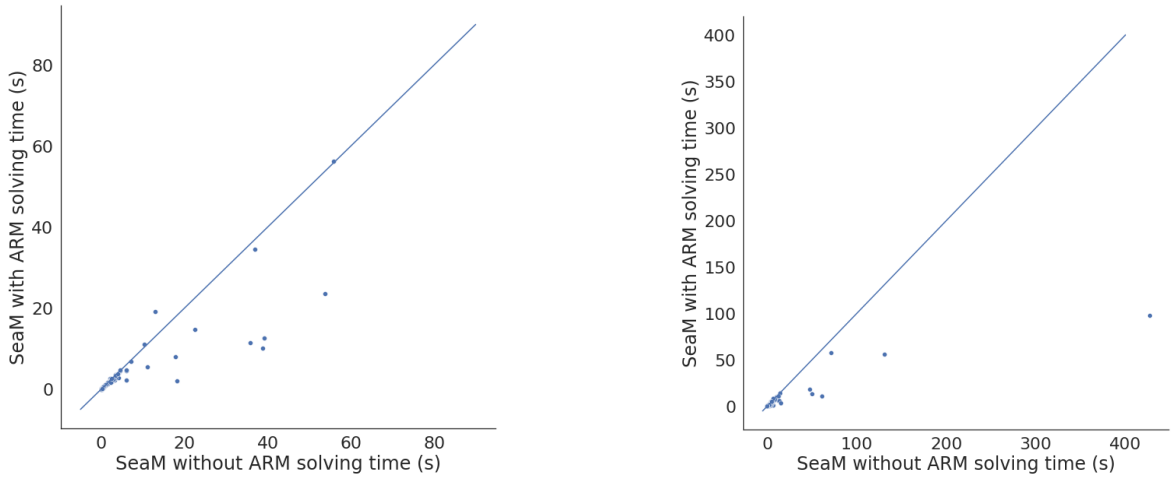


Figure 8.6: SEAM with and without ARM syntactic size per task.

On average accross all `aws-c-common` tasks, SEAM with ARM resulted in VCs with 1.5× faster solving time for Z3 and 3.0× for Yices2 compared to configuration without ARM. The per-task comparison between the two configurations are shown in Figure 8.7. The plot shows that VCs simplified with ARM is faster to solve for both solvers accross all `aws-c-common` tasks with very few exceptions. In terms of *pre-processing time*, the experiments show that ARM does not result in any additional overhead, which is explained by its linear running time by design as described in Chapter 5.



(a) Z3 (b) Yices2

Figure 8.7: SeaM with and without ARM solving time per task.

Conclusion ARM has a positive impact on the performance of SeaBmc with SeaM while maintaining **soundness**. On VCs generated from read-world industrial code base, ARM resolves significantly **more ROW skips** than the approach with syntactic pointer comparison only. Simplification with ARM resulted in **syntactically smaller** VCs and **faster solving time** for Z3 and Yices2, with **negligible pre-processing time** overhead.

8.3 Overhead reduction with Store-Map

In this section, we evaluate the impact of Store-Map on *pre-processing time*. In Sec. 8.1 and Sec. 8.2, SeaM spends around 3 seconds on *pre-processing* for all 157 `aws-c-common` tasks. On average, the pre-processing time per task is less than 1 second. Without changing settings such as BMC unrolling bound, impact of Store-Map is insignificant on `aws-c-common` tasks. Alternatively, we adopted 10 C programs from the **array-crafted** section of Sv-Comp benchmarks collection. These programs all perform a large number of repeated memory access and modification operations on a single integer array. We performed BMC on the 10 programs with SeaBmc using three memory representations: 1. **Lambdas**; 2. **SeaM**: SeaM with Store-Map; 3. **SeaM-linear**: SeaM with linear nested memory *write*s. For all tasks we use only Z3 as SMT solver since this experiment aims to evaluate *pre-processing* performance. Also, as the results reveal, SeaBmc with

Config.	pre-proc. time (s) sum	diff	solving time (s)	syntac. size	Store-Map simp.
SeaM	**3**	–	<1	47425	23192
SeaM-linear	102	×34	<1	47425	–
Lambdas	233	×77	<1	8919	–

Table 8.4: BMC performance on 10 Sv-Comp programs.

all three memory representations does the heavy-lifting during pre-processing instead of during SMT solving.

The BMC results of SeaBmc with three memory representations are presented in Tab. 8.4. We also present the total number of simplifications introduced by Store-Map, including R-HIT and R-SKIP in Figure 6.5, in the **Store-Map simplifications** column. Overall, Store-Map results in over 23,000 R-HIT and R-SKIP rewrites over 10 tasks. Total pre-processing time is reduced by 34× compared to **SeaM-linear**, and 77× compared to **Lambdas**, while the solving time of result VCs are identically negligible across all memory representations. Pre-processing time per task is always the shortest for **SeaM** as shown in Figure 8.8. Note that between the two configurations of **SeaM**, SMT solving time and syntactic size are identical. This is consistent with Store-Map's designed purpose of reducing ROW simplification time during pre-processing, it does not resolve additional pointer equalities.

Case study bAnd To further explore the correlation between pre-processing time and size of ROW sequences in VC, we create a case study focusing on a single program from Sv-Comp **array-crafted**: bAnd, as shown in Figure 8.9. The program maintains an integer array `x` of *fixed* size `N` and initializes it with non-deterministic values on lines 19-21. On line 23, `x` is passed into a function named `bAnd`, which *reads* existing elements of `x` and calculates the accumulated bit-wise and (&) product of all elements in the array and stores it in `ret`. In the loop on lines 25-29, each iteration swaps elements from two shifting indices of `x`; next the bit-wise & product of the shuffled array calculated and stored in `ret2`. Each iteration then verifies the equality between `ret` and `ret2`. By adjusting the variable `SELECT_SIZE`, we can adjust the size of ROW sequences in the translated VC.

In this case study, we perform BMC on bAnd program using SeaBmc configured with **Lambdas**, **SeaM** and **SeaM-linear**. The experiment is repeated with increasing values of `SELECT_SIZE`, ranging from 10 to 100. The pre-processing time of BMC results on bAnd with different memory representations vs. size of `SELECT_SIZE` is plotted in Figure 8.10. As size of ROW sequences in VC increases with `SELECT_SIZE`, SeaBmc pre-processing

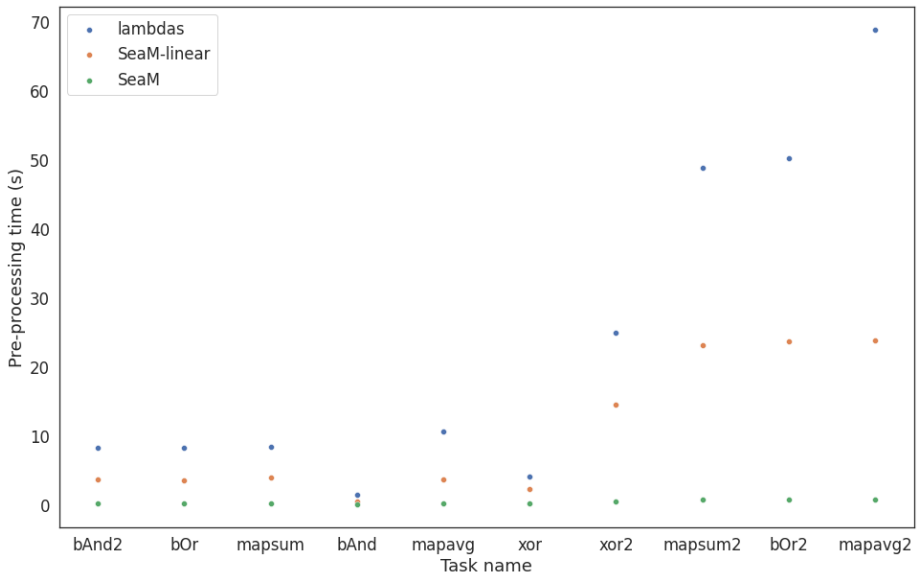


Figure 8.8: Pre-processing time on Sv-Comp tasks.

Config.	pre-proc. time (s) sum	diff	solving time (s)
SeaM	**4**	–	<1
SeaM-linear	16	×4	<1
Lambdas	52	×13	<1

Table 8.5: BMC performance summary on bAnd.

time grows close to linearly under all three configurations. The rate of pre-processing time increase in relation to size of ROW is ranked as the following: **Lambdas** > **SeaM-linear** > **SeaM**. Solving time across the three configurations are identical and very insignificant, as shown in Tab. 8.5. Through this case study, we show that **SeaM** is more scalable than **Lambdas** as size of ROW sequences increases. With Store-Map data structure, ROW simplification can be performed with best-case running time logarithmic to number of *write*s. This is reflected by the even slower increase of pre-processing time as size of ROW sequences increases in VC when Store-Map is enabled in SeaM.

Conclusion On crafted Sv-Comp benchmarks, SeaBmc with SeaM achieved similar level of simplification during pre-processing with much smaller pre-processing time compared to Lambdas. When Store-Map data structure is enabled over *linear* nested *write*s,

```
#define N 101
#define SELECT_SIZE ?
int bAnd (int x[N]) {
  int i;
  long long res;
  res = x[0];
  for (i = 1; i < N; i++) {
    res = res & x[i];
  }
  return res;
}


int main(void) {
  int x[N];
  int temp;
  int ret;
  int ret2;
  for (int i = 0; i < N; i++) {
    x[i] = nd_int();
  }

  ret = bAnd(x);

  for (int i = 0, j = SELECT_SIZE; i < j; i++, j--) {
      temp = x[i]; x[i] = x[j]; x[j] = temp;
      ret2 = bAnd(x);
      sassert(ret == ret2);
  }
  return 0;
}
```

Figure 8.9: bAnd program source code.

the performance gain is even greater. The case study with bAnd affirms that as size of ROW sequences grows, SeaM can be more scalable than Lambdas. Once again, Store-Map introduces further improvement in scalability compared to linear data structure approach.

8.4 Conclusion

Overall, SeaM is a sound memory representation for SeaBmc with comparable or better performance in terms of solving time compared to existing best performing memory representations against real-world industrial benchmarks. Performance in terms of pre-processing time is superior to both **Lambdas** and **Arrays** under all configurations.

ARM proves to be effective for resolving pointer comparisons. Extra simplifications by ARM have positive impact on SMT solving time of BMC on real-world industrial benchmarks. The improvement inflicts negligible overhead in pre-processing time.

On benchmarks requiring simplification on longer sequences of ROW, SeaM proves to be more performant than Lambdas in terms of pre-processing time while achieving similar

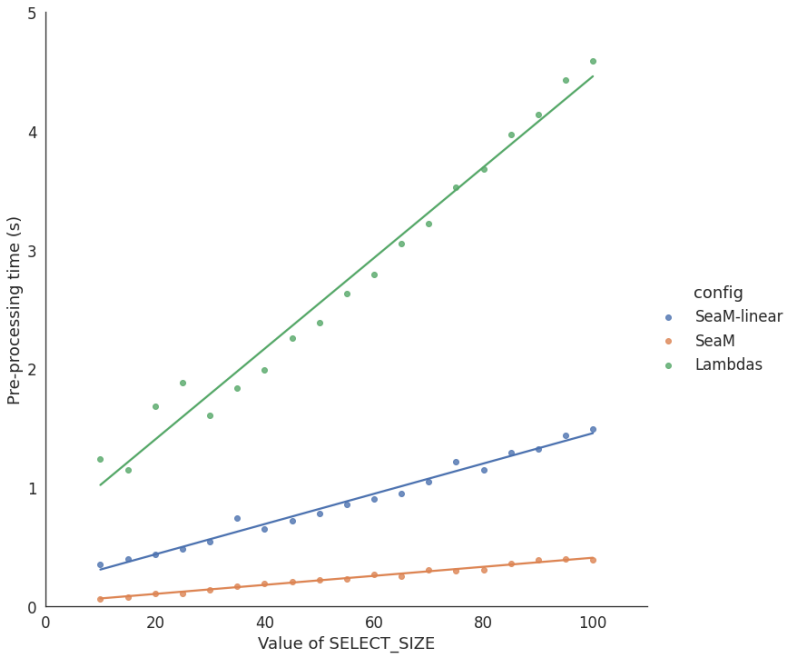


Figure 8.10: Pre-processing time of SeaM configurations with increasing `SELECT_SIZE`.

level of simplification measured by solving time. Store-Map proves to be effective in further reducing pre-processing time when enabled in SeaM. As size of ROW sequences increases, SeaM proves to be more scalable in a crafted scenario than Lambdas. Store-Map further improves scalability upon SeaM with linear data structure.

Chapter 9

Related Work

Memory encoding in BMC is a mature yet still robust field of study. CBMC [9], one of the earliest BMC tool for C programs, uses a combination of theory of arrays and theory of uninterpreted functions to represent memory allocation and operations. SMACK [31] is another BMC tool for LLVM. It also leverages alias analysis to partition memory. SMACK also aims to preserve LLVM IR memory features by translating LLVM IR into the Boogie *intermediate verification language* (IVL) [13], which in spirit is similar to the purpose of $\mathcal{T}_{\mathcal{M}}$. The authors of LLBMC ([17], [15], [25], [33]) introduced extended theory of memory $\mathcal{T}_{\mathcal{ASC}}$ with sound and efficient encoding of `set` and `copy` operations; it also introduced simulation of array operations with λ functions. SeaM borrows the VCGen semantics of `memset` and `memcpy` from $\mathcal{T}_{\mathcal{ASC}}$, but opts to perform eager rewriting of ROW terms over using λ functions.

A different direction for the problem of encoding rich memory operations is to extend solvers to support theories modeling memory properties. [14] introduces a theory of heaps for CHC clauses. The theory of heaps also makes distinction between pointers (*Address sort*) and other scalar values. The work defines new SMT theory semantics in SMT and CHC solvers, unlike SeaM which rewrites VC to contain only *ite*s and array constants (uninterpreted functions). The theory of heaps also maintains properties lke validity and allocation safety, which is not handled by SeaM: SeaBmc handles *temporal* and *spatial* safety with shadow memory and fat pointers. A work in similar direction from Rakamarić et. al. [30] models the memory of a Heap Manipulating Program (HMP) with an acyclic singly-linked list; the work extends the MathSAT [4] solver reasons about unbound reachability of program states in an HMP. Although both capturing properties of heap manipulating programs, [30] and SeaM have very different abstraction models.

The theory of arrays is still one of the more prominent memory representations across different verification and program analysis techniques. Many domain specific memory encodings still contain array terms in final VC. Therefore, improvements on decision procedures for $\mathcal{T}_{\mathcal{A}}$ is a straighforward direction for improving verification performance. Literature on techniques that eliminate ROW terms with *eager* ([19], [22]) rewriting is abundant. One approach substitutes array terms with new variables and delays solving substitutes terms *lazily* [19]. Another "lazy" approach adds array lemmas *on demand* [6] only when inconsistencies occur between indices equality and read variables equality. Most of the improved decision procedures for $\mathcal{T}_{\mathcal{A}}$ are implemented in solvers. In theory, ARRAYS memory representation could be modified with $\mathcal{T}_{\mathcal{M}}$ ROW simplifications while keeping array terms in final VC to take advantage of solvers with more advanced array decision procedures.

The STORE-MAP technique takes inspirations from *map list* data structure in [18]. STORE-MAP also partially sorts memory write sequence in *temporal* order with *commute* rewrite rules in addition to grouping *spatially* pairwise-comparable indices like *map list*. In essence, the application spaces of STORE-MAP and *map list* are quite different. *Map list* aims at generic array simplifications without language-specific information, while STORE-MAP focuses on simplifications under the context of BMC for C programs. [18] also uses intervals abstract domain for over-approximate scalar values. However, [18] does not include abstract semantics of *ite*, which is understandable considering the work mostly uses formulas generated from Symbolic Execution as benchmarks. Note that our Abstract Interpretation approach ARM abstracts symbolic scalar values with $\top$. This is not as precise as [18], which propagates intervals of symbolic values from *assertions*. We believe in the future similar level of precision is feasible in ARM through taking advantage of `assume` statements in SEA-IR.

Chapter 10

Conclusion

In this thesis we explore the advantage of preserving language specific properties in memory encodings for BMC in terms of solving time of verification conditions and pre-processing overhead.

We introduced an expressive First-Order theory called Theory of Memory $\mathcal{T}_{\mathcal{M}}$. $\mathcal{T}_{\mathcal{M}}$ preserves C-specific pointer arithmetic and comparison semantics with *pointer sort* σ_P. $\mathcal{T}_{\mathcal{M}}$ encodes C `memset` and `memcpy` functions succinctly and soundly. We also created eager rewrite rules that remove all $\mathcal{T}_{\mathcal{M}}$ memory operation signatures from final VC provided to solver backend, leaving only *ite*s and uninterpreted functions. Based on $\mathcal{T}_{\mathcal{M}}$, we introduce two optimization techniques.

First, we identified the problem of exponential overhead during direct comparisons between terms of σ_P sort involving nested *ite*s. We introduced ARM, an abstract value of pointer terms that can be constructed from pointer terms in *polynomial* time to syntactic size of pointer term ASTs. We showed that ARM $\sqcap$ operation over-approximates pointer (dis)-equality check in *linear* time. We complemented $\mathcal{T}_{\mathcal{M}}$ ROW simplifications with faster resolution of pointer (dis-equalities) from ARM at the cost of sound over-approximation.

Next, we observed potential scalability issue from *linear* ROW simplification rewrites. Taking advantage of $\mathcal{T}_{\mathcal{M}}$ pointer comparison semantics, we compress *write*s with pairwise-comparable pointers in special data structures named Store-Map with best-case *logarithmic* time-complexity for each ROW simplification.

We have consolidated the above contributions into a memory representation SeaM in SeaBmc. On real-world C program benchmarks, SeaM yields faster overall VC solving time than Arrays memory representation; SeaM is comparable in VC solving time performance with Lambdas. The pre-processing time SeaM is noticeably superior to both

ARRAYS and LAMBDAS. Against real-world C program benchmarks and crafted SV-COMP benchmarks respectively, both ARM and STORE-MAP show positive impact on the overall performance of SEAM.

References

[1] *Arrays*, pages 291–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[2] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[3] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. A write-based solver for SAT modulo the theory of arrays. In Alessandro Cimatti and Robert B. Jones, editors, *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8. IEEE, 2008.

[4] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. The mathsat 3 system. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 315–321, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[6] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *J. Satisf. Boolean Model. Comput.*, 6(1-3):165–201, 2009.

[7] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[8] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In Tibor Gyimóthy, editor, *Compiler Construction, 6th International Conference, CC'96,*

Linköping, Sweden, April 24-26, 1996, Proceedings, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1996.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[13] Robert DeLIne and Rustan Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, March 2005.

[14] Zafer Esen and Philipp Rümmer. A theory of heap for constrained horn clauses (extended technical report). *CoRR*, abs/2104.04224, 2021.

[15] Stephan Falke, Florian Merz, and Carsten Sinz. The bounded model checker LLBMC. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 706–709. IEEE, 2013.

[16] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2013.

[17] Stephan Falke, Carsten Sinz, and Florian Merz. A theory of arrays with set and copy operations. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series in Computing*, pages 98–108. EasyChair, 2012.

[18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 363–380. EasyChair, 2018.

[19] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[20] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, Mountain View, CA, USA, October 1995.

[21] Red Hat Inc. Cve-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, 2013. Accessed: 2022-08-09.

[22] Daniel Kroening and Ofer Strichman. *Arrays*, pages 171–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[23] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[24] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[25] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, volume 7152 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

[26] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007.

[27] Mathias Preiner, Aina Niemetz, and Armin Biere. Better lemmas with lambda extraction. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, pages 128–135. IEEE, 2015.

[28] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Verifying verified code. In *Automated Technology for Verification and Analysis*, pages 187–202, Cham, 2021. Springer International Publishing.

[29] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Bounded model checking for llvm. In *Formal Methods in Computer-Aided Design*. Springer International Publishing, 2022.

[30] Zvonimir Rakamarić, Roberto Bruttomesso, Alan J. Hu, and Alessandro Cimatti. Verifying heap-manipulating programs in an smt framework. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis*, pages 237–252, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[31] Zvonimir Rakamarić and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 106–113, Cham, 2014. Springer International Publishing.

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[33] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*. USENIX Association, 2010.

[34] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.