JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Dipl.-Ing.
Mathias Preiner, Bsc

Angefertigt am
Institut für Formale Modelle und Verifikation

Erstbeurteiler
Univ.-Prof. Dr.
Armin Biere

Zweitbeurteiler
Univ.-Prof. Ph.D.
Roderick Bloem

Februar 2017

Lambdas, Arrays and Quantifiers

Dissertation

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

im Doktoratsstudium der

Technischen Wissenschaften

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Satisfiability Modulo Theories (SMT) refers to the problem of deciding the satisfiability of a first-order formula with respect to one or more first-order theories. In many applications of hardware and software verification, SMT solvers are employed as back-end engine to solve complex verification tasks that usually combine multiple theories, such as the theory of fixed-size bit-vectors and the theory of arrays. This thesis presents several advances in the design and implementation of theory solvers for the theory of arrays, uninterpreted functions and quantified bit-vectors.

We introduce a decision procedure for non-recursive non-extensional first-order lambda terms, which is implemented in our SMT solver Boolector to handle the theory of arrays and uninterpreted functions. We discuss various implementation aspects and algorithms as well as the advantage of using lambda terms for array reasoning. We provide an extension of the lemmas on demand for lambdas approach to handle extensional arrays and discuss an optimization that improves the overall performance of Boolector.

We further investigate common array patterns in existing SMT benchmarks that can be represented by means of more compact and succinct lambda terms. We show that exploiting lambda terms for certain array patterns is beneficial for lemma generation since it allows us to produce stronger and more succinct lemmas, which improve the overall performance, particularly on instances from symbolic execution. Our results suggest that a more expressive theory of arrays might be desirable for users of SMT solvers in order to allow more succinct encodings of common array operations.

We further propose a new approach for solving quantified SMT problems, with a particular focus on the theory of fixed-size bit-vectors. Our approach combines counterexample-guided quantifier instantiation with a syntax-guided synthesis approach to synthesize interpretations for Skolem functions and terms for quantifier instantiations. We discuss a simple yet effective approach that does not rely on heuristic quantifier instantiation techniques, which are commonly employed by current state-of-the-art SMT procedures for handling quantified formulas. We show that our techniques are competitive compared to the state-of-the-art in solving quantified bit-vectors and discuss extensions and optimizations that improve the overall performance of our approach.

Zusammenfassung

Das "Satisfiability Modulo Theories (SMT)" Problem beschäftigt sich mit der Erfüllbarkeit von Formeln in Prädikatenlogik erster Stufe unter Berücksichtigung einer oder mehrerer Theorien. In vielen Anwendungen im Bereich der Hardware und Software Verifikation werden SMT Prozeduren (auch SMT Solver genannt) als Backend verwendet um komplexe Verifikationsprobleme zu lösen, die üblicherweise mehrere Theorien kombinieren, wie z.B. die Theorie der Bitvektoren mit der Theorie der Arrays. Diese Dissertation beschäftigt sich mit Entscheidungsprozeduren für die Theorie der Arrays kombiniert mit nicht-interpretierten Funktionen und für Bitvektoren in Kombination mit Quantoren.

Wir stellen eine Entscheidungsprozedur für nicht-rekursive, nicht-extensionale Lambdaterme erster Stufe vor, welche in unserem SMT Solver Boolector für die Theorie der Arrays kombiniert mit nicht-interpretierten Funktionen zum Einsatz kommt. Wir beschreiben unterschiedliche Implementierungsaspekte und Algorithmen und diskutieren die Vorteile von Lambdatermen in Bezug auf die Darstellung von Arrays. Zusätzlich stellen wir eine Erweiterung für unsere Entscheidungsprozedur vor, die uns ermöglicht mit Gleichheit von Arrays, die als Lambdaterme dargestellt werden, umzugehen und beschreiben eine Optimierung, die im Allgemeinen die Laufzeit von Boolector verbessert.

Weiters beschäftigen wir uns mit unterschiedlichen Array Strukturen, die in existierenden SMT Benchmarks zum Einsatz kommen und die mit Lambdatermen kompakter dargestellt werden können. Wir zeigen, dass die alternative Darstellungsform mit Lambdatermen die Erzeugung von besseren und stärkeren Lemmas in unserer Entscheidungsprozedur ermöglicht. Dies hat zur Folge, dass die Performanz besonders auf Benchmarks, die durch symbolische Programmausführung erzeugt wurden, wesentlich gesteigert werden kann.

Im letzten Teil dieser Dissertation beschäftigten wir uns mit einem neuen Ansatz zum Lösen von quantifizierten SMT Problemen mit Fokus auf quantifizierten Bitvektoren. Unser Ansatz kombiniert eine Technik namens "counterexample-guided quantifier instantiation" mit einer Technik aus dem Bereich der Synthese um Interpretationen für Skolemfunktionen und Terme für die Instantiierung von Quantoren zu synthetisieren. Unsere Ergebnisse zeigen, dass unser Ansatz zum Lösen von quantifizierten Bitvektoren im Vergleich zum aktuellen Stand der Technik kompetitiv ist. Weiters beschreiben wir eine Erweiterung und einige Optimierung für unseren neuen Ansatz, der im Allgemeinen die Performanz wesentlich verbessert.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Acknowledgements

I would like to thank my advisor Armin Biere for his guidance and continuous support throughout my PhD. I am truly grateful for his time, experience and ideas he shared with me. Working together with him was always a pleasure and an invaluable experience. Thank you Armin! I would like to thank my friends and family for their patience and continuous support, especially in the last couple of years. I would also like to thank my colleagues and co-authors – it was a pleasure to work with you. The biggest thanks I can think of goes to the most important person in my life. Thank you for your unconditional support, patience and being part of my life.

Contents

Part I

Prologue

Chapter 1

Introduction

In many applications in the field of formal methods for hardware and software development it is required to reason about specific domains such as integers, reals, bit-vectors or data structures. For example, one might ask if the statement $x + y < 42$ with $x \geq 42$ and $y \geq 42$ is true. An likely immediate response would be "no, this can not be true". However, this entirely depends on the domain of x, y and 42. If they are integers the statement is false. However, if they are bit-vectors, the statement is true since bit-vector arithmetic has overflow semantics and the result of arithmetic operations on bit-vectors of size n is always modulo 2^n. For example, if a bit-vector addition of two bit-vectors exceeds the maximum value that can be represented with the given size of the bit-vector, the value "wraps around" and starts with value 0 again. Consequently, the above statement is true in the domain of bit-vectors if $x = 42$ and $y = 255$ and both are of size 8, which yields value 41. As this example shows, in order to allow correct and meaningful reasoning within a domain it is important to define precise semantics, which can be formalized as first-order theories.

The Satisfiability Modulo Theories (SMT) problem is to decide if a first-order formula is satisfiable with respect to one or more of these first-order theories. Combinations of theories very often include the theory of arrays since it provides a natural way to model memory and actual array data structures. It defines two operations to access and update the contents of an array at a given position and allows reasoning on single array indices and elements. However, it lacks support to succinctly model common array operations such as *memset* and *memcpy* as defined in the standard C library, which modify multiple indices simultaneously.

In this thesis, we explore alternative ways to reason about arrays in SMT, in particular by utilizing non-recursive first-order lambda terms. This allows us to efficiently model array initialization and array update operations without the need to introduce universal quantifiers. For example, if we want to initialize n consecutive elements of array a with value c starting from index i, there are two obvious ways of modelling this. If size n is fixed, e.g., $n = 4$, we can assert value c for each element from index i up to $i + 3$:

$$a[i] = c \wedge a[i + 1] = c \wedge a[i + 2] = c \wedge a[i + 3] = c$$

However, there are two issues with this representation. First, if n is a large number, e.g., if the whole array should be initialized, it produces a large number

of the constraints above. Second, if n is not fixed it is not possible to specify the value for each index separately without using universal quantifiers. With quantifiers, however, the update operation can be specified by means of a universally quantified formula as follows.

$$\forall x \,.\, (i \leq x < i + n \rightarrow a[x] = c)$$

Lambda terms, on the other hand, enable us to use a functional representation for arrays, which allows us to model the state of array a after the update operation as follows.

$$\lambda x \,.\, \text{ite}(i \leq x < i + n,\, c,\, a[x])$$

The above lambda term yields value c if it is accessed within index range i and $i + n$, and an unmodified value $a[x]$ otherwise. The advantage of using lambda terms is their compact and succinct representations for various common array operations without the need for quantifiers. However, to natively handle such lambda terms, a specialized SMT procedure is required.

In this thesis, we present a new decision procedure based on lemmas on demand as presented in [11], which is a lazy SMT approach that iteratively refines an abstraction of the input formula with lemmas until convergence. Our procedure lazily handles non-recursive non-extensional first-order lambda terms. We discuss various implementation aspects and algorithms and show how lambda terms can be used for representing arrays and provide an extension to handle extensional arrays represented as lambda terms and discuss optimizations that improve the overall performance of the procedure. We further investigate various array patterns that occur in existing SMT benchmarks and extract and represent them as lambda terms. Extracting such patterns not only yields more compact representations, but it enables us to generate stronger and more succinct lemmas. As a consequence, this considerably improves the performance of our procedure. We describe several patterns that we found in existing benchmarks and provide algorithms to detect these patterns. Our results suggest that extending the theory of arrays with certain common array operations might be desirable for users of SMT solvers since they provide more succinct encodings of common array operations.

While non-recursive first-order lambda terms allow us to model various array operations in a compact way, it is not possible to specify relations between array elements since lambda terms can only be used to model the state of an array after some operation. As a consequence, we can not model arrays with properties such as sortedness that can be expressed by means of quantifiers. In this thesis, we present a new approach for solving quantified SMT problems that does not rely on current state-of-the-art techniques such as heuristic quantifier instantiation. We combine counterexample-guided quantifier instantiation (CE-QGI) with a syntax-guided synthesis approach to synthesize models and terms for quantifier instantiation. As a result, our approach is able to answer both

satisfiable and unsatisfiable in the presence of universally quantified formulas. In our experiments, we show that our approach is competitive with the current state-of-the-art in solving quantified bit-vectors. We further discuss an extension of our approach that allows us to generalize concrete counterexamples of CEGQI and synthesize terms that can be used as candidates for quantifier instantiation.

This thesis consists of two parts and is organized as follows. In Chapter 2 we briefly introduce the background of this thesis. We introduce SMT, some theories of interest and give an overview on the current state-of-the-art in solving quantified SMT problems. In Chapters 3-5, we revisit and discuss several new contributions to the peer-reviewed papers that are included as Chapters 7-9 in the second part of this thesis.

1.1 Contributions

The second part of this thesis consists of three peer-reviewed papers [51, 52, 53] whose main author is the author of this thesis. The included papers contain small modifications compared to the original publications such as fixed typos, layout changes and some minor notation adjustments. However, none of these modifications affect the content of the publications.

Paper A. [51] *Lemmas on Demand for Lambdas* with Aina Niemetz and Armin Biere. In Proceedings of the 2nd International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS 2013), affiliated to the 13th International Conference on Formal Methods in Computer Aided Design (FMCAD 2013), Portland, OR, USA, 2013.

Chapter 7 includes Paper A, where we decribe a lemmas on demand decision procedure for non-extensional non-recursive first-order lambda terms. The incentive for lambda terms in Boolector was given by A. Biere. M. Preiner developed and implemented the lemmas on demand procedure with contributions from A. Niemetz. The procedure in Paper A was described by M. Preiner with contributions from A. Niemetz. The experimental analysis was performed by M. Preiner and A. Niemetz. The co-authors further contributed with discussions and proof reading Paper A.

Paper B. [52] *Better Lemmas with Lambda Extraction* with Aina Niemetz and Armin Biere. In Proceedings of the 15th International Conference on Formal Methods in Computer Aided Design (FMCAD 2015), pages 128–135, Austin, TX, USA, 2015.

Chapter 8 includes Paper B, where we focus on finding and extracting common array patterns that occur in existing SMT benchmarks to represent them as lambda terms. The idea to extract array patterns as lambda terms was by M. Preiner. The pattern extraction was developed and implemented in Boolector by M. Preiner. The techniques in Paper B were described by M. Preiner

with contributions from A. Niemetz. The co-authors further contributed with discussions and proof reading Paper B.

Paper C. [53] *Counterexample-Guided Model Synthesis* with Aina Niemetz and Armin Biere. In Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017), 17 pages, to appear, Uppsala, Sweden, 2017.

Chapter 9 includes Paper C, where we present a new technique for solving quantified SMT formulas with a particular focus on quantified bit-vectors. Our technique combines counterexample-guided quantifier instantiation with a syntax-guided synthesis approach to synthesize models. The main idea of Paper C was by M. Preiner, who also developed and implemented the described techniques in Boolector. The techniques in Paper C were described by M. Preiner with contributions from A. Niemetz. The co-authors further contributed with discussions and proof reading Paper C.

Chapter 2

Background

The Satisfiability Modulo Theories (SMT) problem is to decide the satisfiability of first-order formulas with respect to one or more first-order theories. Examples for first-order theories include the theory of equality, of reals, of integers, of floating points, of fixed-sized bit-vectors, and of arrays. First-order theories serve two main purposes. First of all they allow reasoning about particular domains such as integers and bit-vectors. Secondly, while the satisfiability problem of first-order logic (FOL) is in general undecidable, the satisfiability problem for many first-order theories or fragments of theories is decidable, which allows to develop specialized and efficient decision procedures. In this thesis we adopt the notions and terminology of FOL and first-order theories in [4,9] with a particular interest in the theory of fixed-size bit-vectors and the theory of arrays.

A first-order theory T consists of a signature Σ and a set of axioms A. A signature Σ is a (possibly infinite) set of constant, function, and predicate symbols, whereas the set of axioms A is a set of closed FOL formulas, which define the meaning of the symbols in Σ. Each symbol in Σ is associated with an arity ≥ 0 and a sort. We refer to function symbols with arity 0 as constant symbols and call symbols occurring in Σ *interpreted* and all other symbols *uninterpreted*. A Σ-formula is constructed from the symbols in Σ using logical connectives ($\neg$, $\wedge$, $\vee$, ...), first-order variables and quantifiers. A Σ-formula φ is satisfiable modulo a theory T (T-satisfiable) if there exists a T-model (or model), i.e., a mapping from constant, predicate and function symbols in Σ to domain values, that satisfies φ under the interpretation of theory T.

Procedures for solving SMT (also referred to as SMT solvers) can be divided into *eager* approaches and *lazy* [57] approaches. Eager SMT approaches translate a given Σ-formula into an equisatisfiable propositional formula while applying various simplification techniques in order to reduce the size of the resulting formula. For example, *bit-blasting* is an eager SMT approach, which translates a bit-vector formula into an equisatisfiable propositional formula. This formula is then converted into conjunctive normal form (CNF) using Tseitin transformation [61] and given to a SAT solver, i.e., a decision procedure for the satisfiability problem of propositional logic. Lazy SMT approaches, on the other hand, are based on the integration of a SAT solver with one or more theory solvers. The SAT solver enumerates truth assignments of the Boolean abstraction of the input formula while the theory solvers check the consistency of the theory-specific parts

of the formula w.r.t. the current truth assignment. The majority of the current state-of-the-art SMT solvers employ lazy SMT approaches based on either the DPLL(T) framework [47] or lemmas on demand [5, 18].

In this thesis we focus on lemmas on demand, which we briefly introduce in Section 2.3. In Chapter 7 we discuss a new lemmas on demand decision procedure for non-recursive first-order lambda terms.

2.1 Theory of Fixed-Size Bit-Vectors

The theory of fixed-size bit-vectors provides a natural way of encoding bit-precise semantics to reason about circuits and programs in hardware and software verification. A fixed-size bit-vector is a fixed-length sequence of binary values (bits) and can be interpreted as signed or unsigned value, i.e., as a negative or positive number. For example, 0011 is a bit-vector of size 4 and represents the natural number 3. The signed representation of a bit-vector is expressed via two's complement, e.g., 1101 is the 4-bit representation of -3 in two's complement. The size of a bit-vector is a strictly positive natural number and different sizes correspond to different bit-vector sorts. As a consequence, the signature of the theory of fixed-size bit-vectors is infinite.

Table 2.1 depicts the set of bit-vector predicate and function symbols for the theory of fixed-size bit-vectors as defined by the SMT-LIBv2 standard [4]. Note

Operator	**Signature**	**Name**
bvult	$S_{[n]} \times S_{[n]} \to \mathit{Bool}$	unsigned less than
bvneg	$S_{[n]} \to S_{[n]}$	two's complement
bvnot	$S_{[n]} \to S_{[n]}$	bit-wise negation
bvand	$S_{[n]} \times S_{[n]} \to S_{[n]}$	bit-wise and
bvadd	$S_{[n]} \times S_{[n]} \to S_{[n]}$	addition
bvmul	$S_{[n]} \times S_{[n]} \to S_{[n]}$	multiplication
bvudiv	$S_{[n]} \times S_{[n]} \to S_{[n]}$	unsigned division
bvurem	$S_{[n]} \times S_{[n]} \to S_{[n]}$	unsigned remainder
bvshl	$S_{[n]} \times S_{[n]} \to S_{[n]}$	logical shift left
bvlshr	$S_{[n]} \times S_{[n]} \to S_{[n]}$	logical shift right
concat	$S_{[n]} \times S_{[m]} \to S_{[n+m]}$	concatenation
extract[i:j]	$S_{[n]} \to S_{[i-j+1]}$	extraction $(0 \leq j \leq i < n)$

Table 2.1: Set of bit-vector operations as defined for the theory of fixed-size bit-vectors in the SMT-LIBv2 standard. $S_{[n]}$ corresponds to a bit-vector sort of size n.

that the bit-vector predicate and function symbols defined in Table 2.1 can be combined to express other bit-vector operations not listed in the table. A set of extensions to the theory of fixed-size bit-vectors is defined in the QF_BV logic of SMT-LIBv2, which includes additional arithmetic and bit-wise operations. Bit-vector arithmetic has overflow semantics, which means that given an arithmetic operation on bit-vectors with size n, the result of the operation is always modulo 2^n. For example, the addition of bit-vector values 0011 (3) and 1110 (14) yields value 0001 (1) since $3 + 14 = 17 \bmod 2^4 = 1$.

The current state-of-the-art for solving quantifier-free bit-vector problems relies on bit-blasting. In [39] it was shown that the satisfiability problem for quantifier-free fixed-size bit-vectors is in general NExpTime-complete if a logarithmic encoding for the size of the bit-vectors is used (which is the case for bit-vectors in the SMT-LIBv2 standard). Until these complexity results it was often assumed that the satisfiability problem for quantifier-free bit-vectors is NP-complete, which only holds if the size of the bit-vectors is unary encoded.

2.2 Theory of Arrays

The theory of arrays provides a natural way to reason about memory in hardware and actual array data structures in software. It defines two operations *read* and *write* to access and update the contents of an array at a given position. The syntax read(a, i) represents the element of array a at index i, whereas write(a, i, e) represents an array that is identical to array a except that it stores element e at index i. In [43], McCarthy originally proposed the main read-over-write axioms, which axiomatize the read and write operations as follows.

$$\forall a, i, j, e \,.\, (i = j \rightarrow \text{read}(\text{write}(a,\, i,\, e),\, j) = e)$$

$$\forall a, i, j, e \,.\, (i \neq j \rightarrow \text{read}(\text{write}(a,\, i,\, e),\, j) = \text{read}(a,\, j))$$

The first axiom states that accessing a modified array at the updated index i yields the written element e. The second axiom asserts that the unmodified element of the original array a at index j is returned if the modified index i is not accessed.

The theory of arrays can be extended to support equality over arrays also denoted as the extensional theory of arrays [60]. In addition to the above read-over-write axioms it defines an axiom of extensionality, which is defined as follows.

$$\forall a, b \,.\, (a = b \leftrightarrow \forall i \,.\, (\text{read}(a,\, i) = \text{read}(b,\, i)))$$

The extensionality axiom states that if two arrays a and b are equal, then a and b must store the same element at each index i. Consequently, if each index i of arrays a and b store the same element, then a and b are equal.

Various algorithms have been developed to determine the satisfiability of quantifier-free formulas (ground formulas) over the (extensional) theory of arrays [10, 11, 31, 60]. One of the current state-of-the-art algorithms employs the so-called lemmas on demand approach and is implemented in our SMT solver Boolector. The original lemmas on demand approach as described in [11] and implemented in Boolector until version 1.5 won the QF_AUFBV divisions (bit-vectors combined with arrays and uninterpreted functions) at the SMT competitions in 2008, 2009, 2011 and 2012. The current lemmas on demand approach as introduced in Chapter 7 and implemented in Boolector since version 1.6, is a generalization of [11] to handle non-recursive first-order lambda terms, which won the QF_ABV divisions (bit-vectors combined with arrays) at the SMT competitions in 2014, 2015 and 2016. In the next section we will give an overview of the general idea of lemmas on demand.

2.3 Lemmas on Demand

Lemmas on demand as in [5,18] is a variant of lazy SMT, which iteratively refines a Boolean abstraction of the input formula with propositional lemmas until convergence. This abstraction refinement process is similar to the counterexample-guided abstraction refinement (CEGAR) approach [15], where an abstraction of a formula is refined based on the analysis of spurious counterexamples. The lemmas on demand approach as described in [11] and the approach currently implemented in Boolector supports the quantifier-free theory of fixed-size bit-vectors combined with the extensional theory of arrays. However, in contrast to the Boolean abstraction used in [5, 18], the approaches described in [11, 31, 32] and the approach introduced in Chapter 7 uses a bit-vector abstraction (also called bit-vector skeleton) of the input formula, which is refined with theory (bit-vector) lemmas.

Figure 2.1 depicts a high level view of the lemmas on demand decision procedure $\mathrm{DP_A}$ as described in [11], which we generalize to non-recursive first-order lambda terms in Chapter 7. In the first step of $\mathrm{DP_A}$, preprocessing is applied to input formula ϕ, which adds additional constraints to the formula that make it easier to keep track of array inequalities. Given the preprocessed formula π, $\mathrm{DP_A}$ produces a bit-vector skeleton $\alpha_\lambda(\pi)$ of formula π by introducing fresh bit-vector variables for each read operation in the formula. In case of extensionality, formula abstraction further introduces a fresh Boolean variable for each equality between arrays. In each step of the refinement loop, a decision procedure $\mathrm{DP_B}$ for bit-vectors is used to determine the satisfiability of the refined bit-vector skeleton $\alpha_\lambda(\pi) \wedge \xi$. If $\mathrm{DP_B}$ returns unsatisfiable, $\mathrm{DP_A}$ can immediately conclude with *unsat* since $\alpha_\lambda(\pi)$ is an over-approximation of formula ϕ. However, if $\mathrm{DP_B}$ returns satisfiable, it returns a candidate model $\sigma(\alpha_\lambda(\pi) \wedge \xi)$, which is checked for consistency w.r.t. formula π. In the consistency checking phase, $\mathrm{DP_A}$ checks if each previously introduced bit-vector and Boolean variable corresponding to

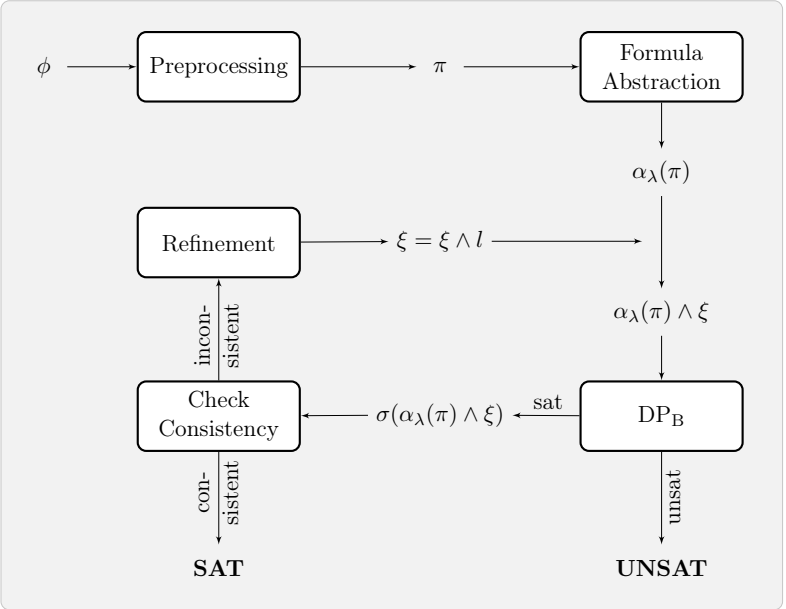


Figure 2.1: General workflow of the lemmas on demand approach as implemented in Boolector.

a read operation or an array equality is consistent with the underlying array terms. If $\sigma(\alpha_\lambda(\pi) \wedge \xi)$ is consistent $\mathrm{DP_A}$ concludes with *sat*. Otherwise, the candidate model is spurious and a lemma l is added to the set of formula refinements ξ. This process is repeated until either the refined bit-vector skeleton is unsatisfiable or the candidate model is consistent w.r.t the theory of arrays.

2.4 Quantifiers

In many applications of hardware and software verification, quantifiers are required to specify various properties of circuits and programs. For example, quantifiers can be useful for specifying universal safety properties, capturing frame axioms in software, defining loop invariants, or defining theory axioms for a theory of interest that is not natively supported by an SMT solver. While the majority of SMT solvers are able to efficiently handle quantifier-free formulas, only a few of them support reasoning with quantifiers. Several decidable fragments of first-order theories have been investigated in the past [1, 10, 33]. However, the main problem is that quantifiers in SMT are in general undecidable and consequently, no general decision procedure exists. Current state-of-the-art SMT solvers that support quantifiers usually employ *quantifier instantiation* and *model-based* techniques. Quantifier instantiation techniques are useful for proving the unsatisfiability of a formula, where universally quantified variables are

instantiated with ground terms to find conflicts at the ground level. However, they do not guarantee to find a ground conflict since the challenging part is to find the right instantiations. For example, E-matching [21] uses a heuristic that tries to find ground terms that have the same structural characteristics as in the quantified part of the formula. Quantifier instantiation techniques usually lack the ability to determine if a formula is satisfiable.

In [33] a technique called *complete instantiation* was introduced that guarantees completeness for certain restricted fragments of SMT for which it suffices to show that a quantified formula is satisfiable w.r.t. a relevant domain. This technique is usually combined with *model-based quantifier instantiation* (MBQI) [33], which allows to determine if a formula is satisfiable. The main idea of MBQI is as follows. Given formula $\varphi \wedge \forall x_1, \ldots, x_n \,.\, \psi[x_1, \ldots, x_n]$ with a ground part of the formula φ and a formula ψ containing universal variables $x_1, \ldots, x_n$. MBQI first checks if the quantifier-free part φ is satisfiable. If it unsatisfiable, MBQI concludes with unsatisfiable. Otherwise, a model M is constructed that contains interpretations for all uninterpreted symbols in φ. A second check is used to determine if model M is also a model for $\forall x_1, \ldots, x_n \,.\, \psi[x_1, \ldots, x_n]$. For this purpose, a formula ψ' is constructed by replacing all uninterpreted symbols in ψ with their interpretation in M. Then, universal variables $x_1, \ldots, x_n$ in ψ' are instantiated with fresh constants $a_1, \ldots, a_n$ and formula $\neg\psi[x_1/a_1, \ldots, x_n/a_n]'$ is checked if it is unsatisfiable. If this is the case, model M is valid and MBQI concludes with satisfiable and returns the model. Otherwise, a counterexample is generated for constants $a_1, \ldots, a_n$, which serves as a candidate for quantifier instantiation to create a new ground instance of $\forall x_1, \ldots, x_n \,.\, \psi[x_1, \ldots, x_n]$. With this approach it is possible to find ground instantiations that are not possible to find with E-matching.

Another model-based technique that can be used to determine if a formula is satisfiable in the presence of universal quantifiers is *finite model finding* [55] The main idea of this approach is to generate finite candidate models that treat uninterpreted sorts as finite domains and exhaustively instantiate universal formulas with the domain elements in order to determine if the formula is satisfiable. This technique is only applicable if universal quantifiers range over uninterpreted sorts or finite sorts such as bit-vectors and is particularly effective for problems that have small models. However, in practice, finite model finding is efficient for a wide range of problems that are of interest to applications in formal methods.

Certain classes of problems can be solved by a technique called *quantifier elimination* [7, 44]. The main idea of this technique is to construct a set of quantifier-free formulas that are equivalent to the original quantified problem and solve it by means of a theory solver for the quantifier-free fragment of the theory.

In Chapter 9 we discuss a model-based technique for solving quantified bit-vectors that is similar to MBQI, but uses a syntax-guided synthesis [2] approach to synthesize candidate models.

Chapter 3

Paper A. Lemmas on Demand for Lambdas

The theory of arrays as axiomatized in Chapter 2 provides a natural way to model memory (components) and array data structures in hardware and software verification. However, it lacks support for compact representations of array operations that update more than one index at the same time. Further, array update operations with a symbolic index range can not be modelled by means of single array read and write operations without using universal quantification. In order to overcome these limitations, in [12] Seshia et al. suggested to model array expressions, ordered data structures and partially interpreted functions as non-recursive first-order lambda terms. This approach was implemented in the SMT solver UCLID [58], and since UCLID implements an eager SMT approach, it eliminates all lambda terms in the input formula as a preprocessing step. This may in the worst-case result in an exponential blow-up in the size of the formula [58].

In Paper A, we describe a new decision procedure based on lemmas on demand, which lazily handles non-recursive non-extensional first-order lambda terms. Lemmas on demand is a CEGAR-based lazy SMT approach that iteratively refines an over-approximation of the input formula with lemmas until either the over-approximation becomes unsatisfiable or its model can be extended to satisfy the original input formula. We implemented our approach in our SMT solver Boolector, which supports the theory of fixed-size bit-vectors in combination with arrays and uninterpreted functions. The new procedure allows us to represent arrays and all array operations as lambda terms and uninterpreted functions and is the current default approach for solving the theory of arrays in Boolector. However, the original approach described in Paper A does not support extensional lambda terms and consequently, can not handle extensional arrays. At the SMT competition 2014, in the division for quantifier-free bit-vectors with arrays (QF_ABV), for benchmarks that were still extensional after rewriting, we had to rely on an older (internal) version of Boolector close to version 1.5.118, which implemented the old lemmas on demand approach described in [11]. In Section 3.1, we describe an extension to our original lemmas on demand for lambdas approach to handle equalities over lambda terms that

represent arrays and equalities over uninterpreted functions. This extension is implemented since Boolector version 2.1 and is used since the SMT competition in 2015. Our lemmas on demand procedure generates one lemma per refinement iteration, where each iteration produces some overhead. As a consequence, generating a large number of lemmas can have a negative impact on the performance of our procedure since every lemma triggers a new refinement iteration. In Section 3.2, we discuss an optimization of our lemmas on demand procedure, which reduces the overall number of required refinement iterations by using a conflict restart strategy.

3.1 Extensionality on Array Lambda Terms

The non-extensional theory of arrays enables us to reason about array elements, whereas the extensional theory of arrays also provides means to compare arrays and consequently enables us to reason about arrays as a whole. This can be particularly useful in verification applications like equivalence checking of memory, which verifies that two algorithms yield the same memory state after execution.

In the following, we discuss the modifications required to support extensional arrays in our lemmas on demand decision procedure DP_λ as introduced in Paper A. Note that we use the functional terminology and notation for arrays as introduced in Paper A, where read operations are represented as function applications, write operations as lambda terms and array variables as uninterpreted functions.

3.1.1 Adding Extensionality Support

In order to support extensional arrays in our lemmas on demand decision procedure DP_λ introduced in Paper A, minor modifications to the preprocessing and formula abstraction steps are required. Further, we need to extend the consistency checking and refinement steps with an additional phase to handle the axiom of extensionality as defined in Chapter 2. In the following, we will discuss the required modifications and additions to DP_λ in more detail.

Preprocessing As in [11], for every array equality $f_a = f_b$ in the input formula, we introduce two fresh function applications $f_a(k)$ and $f_b(k)$ with a fresh index k and add the following array inequality constraint to the top-level.

$$f_a \neq f_b \rightarrow \exists k \,.\, f_a(k) \neq f_b(k)$$

This constraint ensures that if f_a and f_b are not equal they differ at least in one position k. If this is the case, function applications $f_a(k)$ and $f_b(k)$ act as witnesses for the inequality. As a further preprocessing step, and as in [11], we add for each lambda term $g \coloneqq \lambda x.\mathrm{ite}(x = i,\, e,\, f_a(x))$, which represents a write

operation write(a, i, e), the following top-level constraint.

$$g(i) = e$$

This constraint enforces the consistency on write values for lambda terms that represent array write operations. Note that in our implementation this constraint is not added explicitly, but is handled implicitly during the initialization phase of our algorithm. However, for the sake of simplicity, assume that this constraint is added to the top-level.

Formula Abstraction In the formula abstraction step, a bit-vector skeleton $\alpha_\lambda(\pi)$ of the preprocessed formula π is constructed. This is done by traversing from the top-level constraints towards the inputs of the formula while introducing a fresh bit-vector variable for each function application and a fresh Boolean variable for each encountered array equality in formula π. The abstraction of array equalities as Boolean variables is done as in [11].

Consistency Checking The consistency checking step is extended with an additional phase that checks whether all the array equalities assigned to true in the bit-vector skeleton are consistent. Consistency checking now consists of two phases: phase (1) for checking if all abstracted function applications are consistent (as described in Section 7.7 for DP_λ), and phase (2) for checking the consistency of array equalities.

Refinement The refinement step is extended to generate instances of the extensionality axiom in case that array equality conflicts are detected. For each array equality $e \coloneqq (f_a = f_b)$ a set of conflicting indices $C(e)$ is generated, which is used to add the following lemma for each index $i \in C(e)$ as a refinement.

$$f_a = f_b \rightarrow f_a(i) = f_b(i)$$

This lemma enforces that if array f_a and f_b are equal then they also have to store the same element at the conflicting index i.

The extended decision procedure $\mathrm{DP}_{\lambda e}$ with support for extensional arrays is depicted in Figure 3.1 in pseudo-code. The main difference to the original approach DP_λ is the additional consistency checking and refinement phase for array equalities (lines 9-13), which is highlighted in bold line numbers. Function $\mathrm{consistent}_e$ checks the extensionality axiom for each array equality $e \in \pi$ that is assigned to true in the bit-vector skeleton and collects a set of conflicting indices C (line 9). If no conflict was found, i.e., if C is empty, the model of the bit-vector skeleton is consistent and $\mathrm{DP}_{\lambda e}$ concludes with *satisfiable*. However, if C is not empty, procedure lemmas_e generates lemmas for all conflicting indices in C, which are added to the set of refinements ξ (line 13). In the following, we discuss the notion of *conflicting index* and how the set of conflicting indices C is determined in more detail.

```
1  procedure DP_λe (φ)
2    π := preprocess_e(φ)
3    ξ := ⊤
4    loop
5      Γ := α_λe(π) ∧ ξ
6      r, σ := DP_B(Γ)
7      if r = unsatisfiable return unsatisfiable
8      if consistent_λ(π, σ)
9        C := consistent_e(π, σ)
10       if C = ∅
11         return satisfiable
12       else
13         ξ := ξ ∧ α_λe(lemmas_e(C))
14     else
15       ξ := ξ ∧ α_λe(lemma_λ(π, σ))
```

Figure 3.1: Extension of the lemmas on demand for lambdas procedure DP_λ with support for extensional arrays. Bold line numbers indicate the required additions compared to the original procedure.

3.1.2 Consistency Checking of Array Equalities

The main idea of checking the consistency of array equalities is as follows. Given an array equality $e \coloneqq (f_a = f_b)$, we generate candidate models for arrays f_a and f_b and compare them on each index. If the extensionality axiom is violated the corresponding index is added to the set of conflicting indices $C(e)$. In order to generate candidate models for arrays f_a and f_b, the consistency checking phase for array equalities is executed after all abstracted function applications are consistent, i.e., when procedure $\mathrm{consistent}_\lambda$ finishes without finding any conflicts. As described in Section 7.7, we maintain a hash table ρ, which maps each lambda term and UF symbol to a set of function applications. The hash table is initialized via initialization rule I (as defined in Section 7.7), which adds each function application $f(\ldots)$ to $\rho(f)$. During the first consistency checking phase, ρ is continuously extended via propagation rule P (as defined in Section 7.7). As a result, the set of function applications $\rho(f)$ for a function f consists of all function applications that (directly or indirectly) access function f under the current candidate model of the bit-vector skeleton. After procedure $\mathrm{consistent}_\lambda$ finishes without finding any conflicts, the function applications in ρ are consistent and can be used to generate candidate models for arrays in procedure $\mathrm{consistent}_e$. Note that function applications $f(i) \in \rho(f)$ are hashed by the current assign-

```
1  procedure consistent_e (π, σ)
2    C := ∅
3    for e := (f_a = f_b) ∈ π
4      if σ(e) = ⊥ continue
5      M_a := gen_model(f_a)
6      M_b := gen_model(f_b)
7      for i ∈ M_a
8        if i ∉ M_b or σ(M_a[i]) ≠ σ(M_b[i])
9          C := C ∪ {(e, i)}
10     for i ∈ M_b
11       if i ∉ M_a
12         C := C ∪ {(e, i)}
13   return C
```

Figure 3.2: Consistency checking algorithm for array equalities.

ment of the resp. indices $\sigma(i)$, i.e., indices i and j yield the same hash value if $\sigma(i) = \sigma(j)$.

Figure 3.2 depicts procedure consistent$_e$, which checks the consistency of array equalities in formula π and generates the set of conflicting indices C. Note that in procedure consistent$_e$ we only need to consider array equalities that are assigned to true in the bit-vector skeleton. All array equalities assigned to false do not have to be considered since it is sufficient to provide witnesses for the inequalities. These witnesses were added via the inequality constraints during the preprocessing step and are consistent since procedure consistent$_\lambda$ did not find any conflicts. As a consequence, the corresponding arrays are not equal. For every array equality $f_a = f_b \in \pi$ assigned to true, procedure consistent$_e$ checks if the extensionality axiom is violated. For this reason, we need to check if the computed models for f_a and f_b yield the same values on every index. Since f_a and f_b may be arbitrary array terms, procedure gen_model recursively collects all consistent function applications for f_a and f_b and their subterms in ρ (lines 5-6). The result M_a represents the current model of array f_a w.r.t. the current model of the bit-vector skeleton and maps indices to values.

The set of conflicting indices is determined in lines 7-12 by comparing models M_a and M_b on every index i. An index i is identified to be conflicting if M_a and M_b do not yield the same value on i, or if i occurs in M_a but not in M_b and vice-versa. The first case is checked in line 8 with condition $\sigma(M_a[i]) \neq \sigma(M_b[i])$, where $M_a[i]$ and $M_b[i]$ yield different values at the same index and consequently, violate the extensionality axiom. The second case is checked with $i \notin M_b$ (resp. $i \notin M_a$) in line 8 (resp. line 11). Index i is conflicting since the

value of the element at index i is undefined in M_b (resp. M_a), but is required to be the same as in M_a (resp. M_b).

After procedure consistent$_e$ determined all conflicting indices for the current candidate model of the bit-vector skeleton, the following lemmas are added as a refinement step.

$$\bigwedge_{(f_a = f_b,\, i) \in C} f_a = f_b \rightarrow f_a(i) = f_b(i)$$

In the next refinement iteration all function applications including the ones added via the extensionality lemmas are checked for consistency. This process is repeated until either the bit-vector skeleton becomes unsatisfiable or none of the consistency checking procedures detect any more conflicts.

Example 3.1. Consider formula ϕ, with indices i_0, i_1, i_2, values v_0, v_1, v_2, and write operations w_0, w_1, w_2 represented as lambda terms as follows.

$$\phi \coloneqq \underbrace{\lambda x.\mathrm{ite}(x = i_0,\, v_0,\, f_a(x))}_{w_0} = \underbrace{\lambda x.\mathrm{ite}(x = i_2,\, v_2,\, (\overbrace{\lambda y.\mathrm{ite}(y = i_1,\, v_1,\, f_b(y))}^{w_1})(x))}_{w_2}$$

In the first step, preprocessing generates an inequality constraint for array equality $w_0 = w_2$, and write value consistency constraints for w_0, w_1, and w_2.

$$\begin{aligned}\pi \coloneqq\ & w_0 = w_2 \\ & \wedge (w_0 \neq w_2 \rightarrow w_0(j) \neq w_2(j)) \\ & \wedge w_0(i_0) = v_0 \wedge w_1(i_1) = v_1 \wedge w_2(i_2) = v_2\end{aligned}$$

Since array equality $w_0 = w_2$ is asserted at the top-level, the left-hand side of the implication of the inequality constraint is always false and consequently, the impliciation simplifies to true and can therefore be omitted, which yields the following formula.

$$\pi \coloneqq w_0 = w_2 \wedge w_0(i_0) = v_0 \wedge w_1(i_1) = v_1 \wedge w_2(i_2) = v_2$$

In the next step, formula abstraction introduces a fresh Boolean variable e for array equality $w_0 = w_2$, and fresh bit-vector variables $u_{w_0}^{i_0}$, $u_{w_1}^{i_1}$, and $u_{w_2}^{i_2}$, for function applications $w_0(i_0)$, $w_1(i_1)$ and $w_2(i_2)$, which results in the following bit-vector skeleton.

$$\alpha_{\lambda e}(\pi) \coloneqq e \wedge u_{w_0}^{i_0} = v_0 \wedge u_{w_1}^{i_1} = v_1 \wedge u_{w_2}^{i_2} = v_2$$

Assume that DP$_\mathrm{B}$ produces a model $\sigma(\alpha_{\lambda e}(\pi))$ for formula $\alpha_{\lambda e}(\pi)$ such that

$$\begin{array}{llll}\sigma(e) = \top & \sigma(u_{w_0}^{i_0}) = \sigma(v_0) & \sigma(i_0) = \sigma(i_2) & \sigma(v_0) \neq \sigma(v_2) \\ & \sigma(u_{w_1}^{i_1}) = \sigma(v_1) & \sigma(i_1) \neq \sigma(i_2) & \sigma(v_0) = \sigma(v_1) \\ & \sigma(u_{w_2}^{i_2}) = \sigma(v_2) & & \end{array}$$

Consistency checking of function applications $w_0(i_0)$, $w_1(i_1)$, $w_2(i_2)$ w.r.t. $\sigma(\alpha_{\lambda e}(\pi))$ does not find any conflicts since all function applications are consistent due to the write value consistency constraints added in the preprocessing step. Procedure consistent$_\lambda$ produces the following final state of ρ.

$$\begin{aligned} \rho(w_0) &\coloneqq \{w_0(i_0)\} & \rho(f_a) &\coloneqq \{\} \\ \rho(w_1) &\coloneqq \{w_1(i_1)\} & \rho(f_b) &\coloneqq \{\} \\ \rho(w_2) &\coloneqq \{w_2(i_2)\} \end{aligned}$$

Since no conflicts were found, we continue checking array equality e and generate the models M_{w_0} and M_{w_2} for w_0 and w_2.

$$\begin{aligned} M_{w_0} &\coloneqq \rho(w_0) \cup \rho(f_a) = \{w_0(i_0)\} \\ M_{w_2} &\coloneqq \rho(w_2) \cup \rho(w_1) \cup \rho(f_b) = \{w_2(i_2), w_1(i_1)\} \end{aligned}$$

We identify index i_0 to be conflicting since $\sigma(i_0) = \sigma(i_2)$, but $\sigma(u_{w_0}^{i_0}) \neq \sigma(u_{w_2}^{i_2})$, and index i_1 to be conflicting since $i_1 \notin M_{w_0}$. As a consequence, we generate the following two lemmas and add them to the set of refinements ξ.

$$\xi \coloneqq (w_0 = w_2 \rightarrow w_0(i_0) = w_2(i_0)) \wedge (w_0 = w_2 \rightarrow w_0(i_1) = w_2(i_1))$$

Note that formula abstraction is applied to refinements ξ, which introduces new bit-vector variables $u_{w_2}^{i_0}$, $u_{w_0}^{i_1}$ and $u_{w_2}^{i_1}$ for function applications $w_2(i_0)$, $w_0(i_1)$ and $w_2(i_1)$. In the next round, assume that DP$_\mathrm{B}$ produces a model $\sigma(\alpha_{\lambda e}(\pi \wedge \xi))$ for formula $\alpha_{\lambda e}(\pi \wedge \xi)$ such that

$$\begin{array}{llllll} \sigma(e) = \top & \sigma(u_{w_0}^{i_0}) = \sigma(v_0) & \sigma(i_0) \neq \sigma(i_2) & \sigma(v_0) = \sigma(v_2) & \sigma(u_{w_0}^{i_0}) = \sigma(u_{w_2}^{i_0}) \\ & \sigma(u_{w_1}^{i_1}) = \sigma(v_1) & \sigma(i_1) \neq \sigma(i_2) & \sigma(v_0) = \sigma(v_1) & \sigma(u_{w_0}^{i_1}) = \sigma(u_{w_2}^{i_1}) \\ & \sigma(u_{w_2}^{i_2}) = \sigma(v_2) & \sigma(i_0) \neq \sigma(i_1) & & \sigma(u_{w_2}^{i_1}) = \sigma(v_1) \end{array}$$

Consistency checking of all function applications does not find any conflicts and yields the following state of ρ.

$$\begin{aligned} \rho(w_0) &\coloneqq \{w_0(i_0), w_0(i_1)\} & \rho(f_a) &\coloneqq \{w_0(i_1)\} \\ \rho(w_1) &\coloneqq \{w_1(i_1), w_2(i_0)\} & \rho(f_b) &\coloneqq \{w_2(i_0)\} \\ \rho(w_2) &\coloneqq \{w_2(i_2), w_2(i_1), w_2(i_0)\} \end{aligned}$$

This time, generating models M_{w_0} and M_{w_2} yields $\rho(w_0)$ and $\rho(w_2)$, respectively. Note that $w_1(i_1)$ does not occur in M_{w_2} since $w_2(i_1) \in \rho(w_2)$ has the same index and takes precedence over $w_1(i_1)$ while generating M_{w_2}.

$$\begin{aligned} M_{w_0} &\coloneqq \rho(w_0) \cup \rho(f_a) = \{w_0(i_0), w_0(i_1)\} \\ M_{w_2} &\coloneqq \rho(w_2) \cup \rho(w_1) \cup \rho(f_b) = \{w_2(i_2), w_2(i_1), w_2(i_0)\} \end{aligned}$$

We identify index i_2 as conflicting since $i_2 \notin M_{w_0}$, and add the following lemma as a refinement step.

$$\xi \coloneqq \xi \wedge (w_0 = w_2 \rightarrow w_0(i_2) = w_2(i_2))$$

In the final round, both consistency checking phases do not find any conflicts and our decision procedure $\mathrm{DP}_{\lambda e}$ concludes with satisfiable.

In contrast to the original algorithm proposed in [11], our approach for extensional arrays does not rely on upwards-propagation of read or write nodes. This is due to the fact that upwards-propagation in the presence of lambda terms is not as straightforward as in [11] since keeping track of the propagation paths for lemma generation would involve much more implementation overhead. Instead, we construct the current models of the corresponding arrays for each array equality, compare them and in case of a conflict add an instantiation of the array extensionality axiom as a lemma. As a consequence, the consistency checking and lemma generation for array equalities is much simpler, requires less implementation effort, and is still competitive to the original approach in [11], as shown in our experiments.

Note that generating the corresponding models after the first consistency checking phase is straightforward in the array case since this only requires to recursively collect all relevant function applications in ρ. However, for the general case, i.e., equality over arbitrary lambda terms, our approach does not work. One possible solution is to introduce universal quantifiers and add an additional constraint for each lambda term equality $f = g$ in the formula as follows.

$$f = g \rightarrow \forall \bar{x}.f(\bar{x}) = g(\bar{x})$$

However, this requires the solver to support universal quantifiers in combination with lambda terms, which is left to future work.

Note that in Paper A we used lambda terms to represent if-then-else on arrays and functions. In Boolector this turned out to be suboptimal in the extensional array case since in certain cases not all relevant function applications were collected via procedure gen_model due to some simplifications applied within these if-then-else lambda terms. Adding support for these special cases would have been too error-prone. As a consequence, we do not introduce lambda terms for if-then-else terms on arrays and functions.

3.1.3 Experiments

We extended the lemmas on demand for lambdas approach implemented in Boolector to support extensional arrays as discussed above. Further, for comparison purposes, we also implemented our approach in Boolector version 1.5.118, which implements the array decision procedure described in [11]. Each implementation in the two versions required about 300 lines of code. We evaluated our approach on all QF_ABV benchmarks of SMT-LIB [4] that contained array equalities after rewriting. The compiled benchmark set contains 1772 benchmarks in total, which are part of the brummayerbiere and dwp_formulas benchmark families. For this evaluation, we compared the following three configurations of Boolector.

(1) **Btor+e** Current version of Boolector, which implements $\mathrm{DP}_{\lambda e}$.

(2) **Btor15** An internal version of Boolector close to version 1.5.118, which was used at the SMT competition 2014 for extensional benchmarks.

(3) **Btor15+e** An extended version of Btor15, which implements procedures $\mathrm{consistent}_e$ and lemmas_e for extensional arrays as described in the previous section.

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.5 LTS. We set the limits for each solver/benchmark pair to 7GB of memory and 1200 seconds of CPU time. In case of a timeout, memory out, or an error, a penalty of 1200 seconds was added to the total CPU time.

Table 3.1 summarizes the results of all configurations grouped by the benchmark families brummayerbiere and dwp_formulas. Configuration Btor+e considerably outperforms the other two configurations. However, this is no surprise since configuration Btor+e is the current version of Boolector that won recent SMT competitions and its code base considerably changed since version 1.5.118. Therefore, we also implemented our approach in the old version of Boolector in order to provide a fair comparison. Configurations Btor15 and Btor15+e solve almost the same number of benchmarks. Overall, consid-

	Btor+e		**Btor15**		**Btor15+e**	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
bbiere (195)	**189**	**17751**	179	31654	179	29922
dwp (1577)	1577	1528	**1577**	**1268**	1576	3779
Total (1772)	**1766**	**19279**	1756	32922	1755	33701

Table 3.1: Results for all configurations on the extensional QF_ABV benchmarks grouped by benchmark families with a CPU time limit of 1200 seconds.

ering commonly solved instances only, configuration Btor15 generates 83538 lemmas (bbiere: 57189, dwp: 26349), whereas configuration Btor15+e generates in total 177482 lemmas (bbiere: 53938, dwp: 123544) of which 60061 lemmas (bbiere: 1532, dwp: 58529) are instantiations of the extensionality axiom. On the bbiere benchmarks Btor15+e requires about 3000 lemmas less compared to Btor+e and consequently, solves the 179 benchmarks slightly faster. Note that 88% (bbiere: 8%, dwp: 90%) of the extensionality lemmas are generated since an index occurs in only one of the array models. A reason for this might be that our approach introduces two fresh function applications for each extensionality lemma, which potentially increases the overall number of function applications to be checked for consistency. Introducing an additional propagation strategy for these indices instead of immediately generating a lemma might reduce the number of conflicting indices.

3.1.4 Conclusion

We presented a simple extension of our decision procedure for lambdas that enables us to handle extensional arrays represented as lambda terms. The same extension can also be employed for the lemmas on demand decision procedure originally implemented in Boolector for the theory of arrays, which was implemented in Boolector until version 1.5.118. Compared to the original algorithm, the implementation of our approach is rather simple (300 lines of code for each version) since it involves no upwards-propagations, but is competitive as shown in our experimental results. In our approach, lemma generation is not yet optimized and in some cases produces a lot of instantiations of the array extensionality axiom that could be avoided by an additional propagation strategy. We leave this enhancement to future work.

3.2 Eager Lemma Generation

Our lemmas on demand approach for lambdas DP_λ employs a consistency checking restart strategy, which restarts as soon as a conflict is detected. As a result, our approach generates one lemma per refinement step until either all function applications are consistent (no conflict can be found), or DP_B reports unsatisfiable. Each refinement step produces some overhead in terms of DP_B queries and function application checks in the consistency checking phase. The overhead caused by a single step is usually small, however, it can have a negative impact on the overall runtime with an increasing number of conflicts.

Therefore, we extended DP_λ with a new restart strategy, which enables us to generate multiple lemmas in one refinement step in order to reduce the overall number of refinement iterations. As a result, this reduces the overall number of DP_B queries and function application checks, however, at the cost of generating more lemmas. Since each lemma increases the size of the bit-vector skeleton handed to the underlying SAT procedure DP_B, generating a large number of

lemmas in each refinement step can have a negative impact on its runtime. Hence, it is important to find a good balance between the number of generated lemmas per refinement step and the overall number of refinement iterations. In the following, we discuss two strategies that generate multiple lemmas per refinement iteration, but with different levels of eagerness.

Our initial (eager) strategy was to generate lemmas for all conflicts in the current candidate model of the bit-vector skeleton prior to restarting. This was the default strategy of Boolector since version 2.2 and was enabled at the SMT competitions in 2015 and 2016. This approach significantly reduces the overall number of refinement iterations, however, in some cases, the number of generated lemmas had a negative impact on the overall runtime since too many lemmas were generated, which considerably increased the size of the bit-vector skeleton.

Our new (lazy) strategy implemented in Boolector since version 2.4 tries to address this issue by generating lemmas as long as the conflicts do not directly influence each other. That is, if a conflicting function application is detected and the value of one of its arguments already depends on a conflicting function application, we add all lemmas generated in the current round to the formula and restart consistency checking. Given a conflicting function application $f(a_1, \ldots, a_n)$, checking the restart criteria is realized as a depth-first-search (DFS) traversal of arguments $a_1, \ldots, a_n$. If during the traversal, a function application is encountered that produced a conflict in the current refinement iteration, consistency checking is restarted. The intuition for this criteria is that if a conflicting function application is found during the traversal, at least one value of arguments $a_1, \ldots, a_n$ depends on a conflict and consequently, is inconsistent. As a consequence, all lemmas generated in the current refinement iteration are added to the formula and $\mathrm{DP_B}$ is queried for a new candidate model. In order to keep the overhead of the traversal as small as possible, we do not traverse the complete subgraphs of $a_1, \ldots, a_n$, but stop the traversal at function applications. This turned out to be the best strategy since it provides a good balance between restarts and lemmas generated per refinement step.

3.2.1 Experiments

We implemented our new restart strategy in our SMT solver Boolector and evaluated it on all QF_ABV benchmarks (15091 in total) of SMT-LIB [4]. We compared the following three configurations of Boolector.

(1) **Btor** Boolector with the original restart strategy that generates one lemma per refinement iteration.

(2) **Btor+el** Boolector with the eager restart strategy that generates lemmas for all conflicts in the current candidate model.

(3) **Btor+ll** Boolector with the lazy restart strategy that generates lemmas as long as the conflicts do not depend each other.

	Btor		Btor+ll		Btor+el	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
bench_ab (119)	119	0.6	119	0.6	119	0.6
bmc (39)	39	313	39	293	39	290
bbiere (293)	**266**	**46746**	264	46595	264	46629
bbiere2 (22)	20	3903	20	3217	20	3215
bbiere3 (10)	10	0.5	10	0.5	10	0.6
btfnt (1)	1	49	1	42	1	35
calc2 (36)	36	1650	36	1652	36	1650
dwp (5765)	5763	5314	5763	4493	5763	4490
ecc (55)	54	1262	54	1266	54	1266
egt (7719)	7719	107	7719	107	7719	107
jager (2)	0	2400	0	2400	**2**	**1555**
klee (622)	622	124	622	115	622	115
pipe (1)	1	4.5	1	4.5	1	4.5
platania (275)	263	18197	**268**	**12135**	266	18398
sharing (40)	40	932	40	931	40	932
stp (40)	39	898	39	892	39	891
stp_samp (52)	52	2.0	52	2.1	52	2.0
Total (15091)	15044	81903	**15047**	**74144**	15047	79579

Table 3.2: Results for all configurations grouped by benchmark families.

The experiments were performed with the same hardware setup and resource limits (1200 seconds CPU time, 7GB memory) as in Section 3.1. Note that configuration Btor+el corresponds to the default strategy used in Boolector since version 2.2 and was enabled for the SMT competitions in 2015 and 2016. The new restart strategy enabled in configuration Btor+ll is the default strategy since Boolector version 2.4.

Table 3.2 summarizes the results of all configurations grouped by benchmark families. Overall, generating multiple lemmas per refinement step is an advantage for configurations Btor+ll and Btor+el and are able to solve more instances in less time compared to Btor. However, configuration Btor+el requires considerably more time than Btor+ll due to the fact that Btor+el generates lemmas for all conflicts in the current candidate model and consequently, produces more lemmas than Btor+ll. Since every lemma increases the size of the bit-vector skeleton, the number of lemmas also affects the time required by DP_B to solve it. This effect is especially pronounced on the platania benchmark family, where

	Btor	**Btor+ll**	**Btor+el**
Time [s]	23973	**20274**	24056
DP_B Time [s]	20730	**18401**	21967
DP_B Queries	205898	71649	**60628**
Lemmas	**252996**	349280	414080
Checks	117057731	**34016349**	63980400

Table 3.3: Lemmas on demand results on commonly solved instances.

on the commonly solved instances Btor+el produces almost twice as many lemmas (133k) than Btor+ll (71k) in total. As a consequence, the size of the bit-blasted bit-vector skeleton contained twice as many CNF variables and CNF clauses, which of course affected the runtime of the underlying SAT solver DP_B. For configuration Btor+el DP_B required 5026 seconds, whereas for configuration Btor+ll only 1453 seconds were spent in DP_B, which is an improvement by a factor of 3.5.

Table 3.3 summarizes the overall runtime, the runtime of DP_B, the number of DP_B queries (which corresponds to the number of refinement iterations), the number of generated lemmas, and the number of function applications checks on the 15040 benchmarks commonly solved by all configurations. As expected, configuration Btor generates the smallest number of lemmas and the highest number of DP_B queries. This is due to the fact that Btor restarts after each conflict and consequently, fixes conflicts consecutively, which produces less unnecessary lemmas, however, at the cost of increasing the overall number of DP_B queries. Note that for configuration Btor the difference between the number of DP_B queries and the number of generated lemmas is due the fact that Btor still generates multiple lemmas per refinement step for extensionality conflicts. Otherwise, the numbers would not differ. On the contrary, configuration Btor+el, which generates lemmas for all conflicts in the current candidate model, produces the highest number of lemmas and the smallest number of DP_B queries. The additional overhead in terms of lemmas and as a result the increase in formula size has a negative effect on the solving time of DP_B. Configuration Btor+ll significantly outperforms the other two configurations and requires 15% less runtime to solve all 15040 common benchmarks. The significant difference in function application checks compared to Btor+el is due to the no_init_multi_delete benchmarks in the platania benchmark family, which contain many function applications. Configuration Btor+ll solves these instances 10 times faster and requires only 25% of the refinement iterations of Btor+el.

3.3 Discussion

Our lemmas on demand approach for non-recursive first-order lambda terms allows us to represent arrays and array operations by means of lambda terms and uninterpreted functions. As shown in Paper B, this can be particularly beneficial if multiple array operations can be represented by more compact lambda terms. However, in the general case, where array operations can not be represented more succinctly, lambda terms produce some overhead in terms of memory consumption and runtime. For example, in Boolector, during construction of the formula each array write operation write(a, i, e) is translated to a lambda term $\lambda x.\text{ite}(x = i, e, a[x])$ on-the-fly, which introduces four additional terms. Further, consistency checking lambda terms requires to apply beta reduction, which is more expensive in terms of runtime compared to checking write operations. The best but also more complex approach would be to support both, where lambda terms are only used to combine multiple array operations. This requires that consistency checking and lemma generation support handling of both array operations and lambda terms, which is more involved compared to using only one kind of representation. However, we believe that this would be the optimal solution, which combines the best of both approaches.

Chapter 4

Paper B. Better Lemmas with Lambda Extraction

In Paper A we explore an alternative representation of arrays and array operations and employ non-recursive first-order lambda terms, which allows us to model common array operations not natively supported by the theory of arrays. As a simple example, consider the initialization of an entire array a with a constant c. If the domain of the index sort is finite, e.g., a bit-vector of size n, there are two obvious ways of representing this without quantifiers. First, we can use a sequence of 2^n write operations, where the top-most write operation represents array a.

$$\text{write}(\ldots \text{write}(\text{write}(b,\, 0,\, c),\, 1,\, c) \ldots,\, 2^n - 1,\, c)$$

Second, we can specify a conjunction of 2^n equalities over read operations to assert that the value at each index of array a is c.

$$\text{read}(a,\, 0) = c \wedge \text{read}(a,\, 1) = c \wedge \ldots \wedge \text{read}(a,\, 2^n - 1) = c$$

However, both approaches do not scale well for large domains of the index sort, since they produce too many read and write operations. Further, if the domain of the index sort is infinite, it is not possible to represent array initialization with the approaches mentioned above. As an alternative, we can use quantifiers, which allows a much more succinct representation.

$$\forall x \,.\, \text{read}(a,\, x) = c$$

However, this approach requires support for universal quantifiers and does not even scale for a simple array initialization pattern, as we will show in Section 4.1. Our lambda approach, on the other hand, handles finite and infinite index sort domains, where the initialized array above can be represented as follows.

$$\lambda x.c$$

In Paper B we focus on finding and extracting array patterns from existing SMT benchmarks to represent them as more compact lambda terms. We further describe a complementary technique denoted as lambda merging, which combines

multiple array operations into one lambda term. In combination with our lemmas on demand for lambdas approach, both techniques allow us to generate stronger and more succinct lemmas, which consequently prunes the search. We describe several array patterns and provide algorithms to detect and extract these patterns. We show that both techniques considerably improve the solver performance. Our results suggest that for certain array patterns (such as array initialization operations) it might be desirable to extend the theory of arrays in order to provide more succinct encodings and allow specialized SMT procedures that efficiently handle these operations.

4.1 Discussion

Using more compact and succinct representations for array operations does not only reduce the size of the input formula, but more importantly, considerably improves lemma generation of our lemmas on demand procedure. It allows us to generate lemmas that cover a range of indices instead of single indices, which significantly improves the overall performance. This is particularly useful on benchmarks from symbolic execution such as the klee benchmark family of the QF_ABV benchmark set. These benchmarks heavily rely on patterns that initialize large parts of an array with concrete values. As a result, on this benchmark set with lambda extraction we achieve an overall speed-up by a factor of 77.

Merging multiple array operations into one lambda term usually does not yield as compact lambda terms as lambda extraction, but it enables us to apply further simplifications. This is, e.g., useful for benchmarks that use sequences of write operations to initialize an array at symbolic indices. Consider, e.g., a sequence of write operations write(write(write(a, i, e), j, e), k, e), which corresponds to $\lambda x.\text{ite}(x = k, e, \text{read}(\lambda y.\text{ite}(y = j, e, \text{read}(\lambda z.\text{ite}(z = i, e, \text{read}(a, z)), y)), x))$. Applying lambda merging yields

$$\lambda x.\text{ite}(x = k, e, \text{ite}(x = j, e, \text{ite}(x = i, e, \text{read}(a, x)))),$$

which can then be simplified to

$$\lambda x.\text{ite}(x = k \vee x = j \vee x = i, e, \text{read}(a, x)).$$

In Paper B we investigated quantifier-free benchmarks and tried to represent multiple read and write operations by means of more compact lambda terms. However, we did not investigate patterns represented with quantifiers. Consider, e.g., the following patterns, which can be represented by means of quantifiers and lambda terms.

- **Initializations**

  Initialize entire arrays with either parallel updates or loops, e.g.,

  $\forall x\,.\,(\text{read}(a, x) = c)$, $\forall x\,.\,(\text{read}(a, x) = x)$, $\forall x\,.\,(\text{read}(a, x) = x + 1)$

  $\lambda x\,.\,c$, $\lambda x\,.\,x$, $\lambda x\,.\,x + 1$

- **Parallel updates**

  Update n elements of array a with value c starting from index i, which yields a new array b, e.g.,
  $b = memset(a, i, n, c)$
  $\forall x \,.\, (\text{read}(b,\, x) = \text{ite}(i \leq x < i + n,\, c,\, \text{read}(a,\, x)))$
  $\lambda x \,.\, \text{ite}(i \leq x < i + n,\, c,\, \text{read}(a,\, x))$

- **Copy operations**

  Copy n elements of array a starting from index i to array b at index j, which yields a new array b', e.g.,
  $b' = memcpy(a, b, i, j, n)$
  $\forall x \,.\, (\text{read}(b',\, x) = \text{ite}(j \leq x < j + n,\, \text{read}(a,\, i + x - j),\, \text{read}(b,\, x)))$
  $\lambda x \,.\, \text{ite}(j \leq x < j + n,\, \text{read}(a,\, i + x - j),\, \text{read}(b,\, x))$

The most intuitive approach for specifying the array operations above is using quantifiers, since this is directly supported by the SMT-LIBv2 standard. However, current state-of-the-art SMT solvers that support quantifiers lack the ability to efficiently handle these patterns. This can be illustrated with a simple array initialization pattern $\forall x \,.\, (\text{read}(a,\, x) = 0)$, which initializes an entire array a with the constant value 0. For this purpose, we compiled a set of benchmarks ABV-init (15091 in total), where we initialized the first array in every benchmark of the QF_ABV benchmark set of SMT-LIB with the pattern above. Note that this modification may change the status of some benchmarks from satisfiable to unsatisfiable. However, this is of no consequence for our experiment since we are mainly interested in identifying the overall effects of adding array initialization patterns. Note that due to the initialization pattern, more than one third of the benchmarks in QF_ABV changed the status from satisfiable to unsatisfiable, As a consequence, the majority of benchmarks in ABV-init is unsatisfiable.

For our experiment, we extended the current version 2.4 of our SMT solver Boolector and implemented a rewriting rule that transforms the initialization pattern for array a into the lambda term $\lambda x.0$ and adds a top-level equality $a = \lambda x.0$. We evaluated Boolector, CVC4[1] and Z3[2] on benchmark sets QF_ABV and ABV-init. Note that CVC4 and Z3 natively support quantifiers and do not extract lambda terms for the initialization pattern. A comparison of Boolector with quantifier support as introduced in Paper C is not included since it does not yet support the combination of quantified bit-vectors with lambda terms.

We set the resource limits for each solver/benchmark pair to 1200 seconds CPU time and 7GB of memory. In case of a timeout, memory out, error or an unknown result, a penalty of 1200 seconds was added to the total CPU time.

[1] commit 0dd2aa21f35b221ea96d277e9ea7cbc816ffe83c
[2] commit 40177f7bac4ab9615a32728154f6fd1fa6c8fcf9

	QF_ABV (15091)				**ABV-init** (15091)			
	Solved	Sat	Unsat	Time [s]	Solved	Sat	Unsat	Time [s]
Boolector	15047	10403	4644	75109	15075	4693	10382	31895
CVC4	14634	10067	4567	642777	10239	0	10239	5839596
Z3	14937	10340	4597	234606	13781	3714	10067	1734356

Table 4.1: Results of all solvers for benchmarks without (QF_ABV) and with array initialization (ABV-init) including penalties.

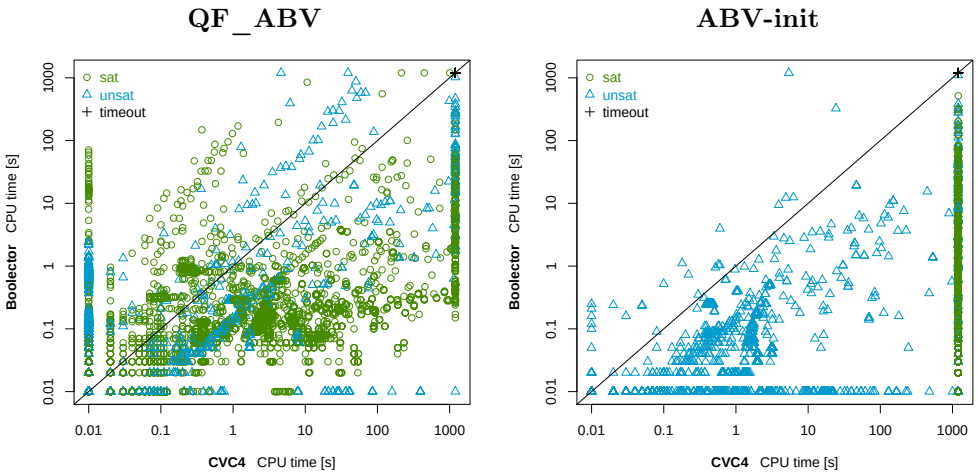


Figure 4.1: Runtime comparison of Boolector and CVC4 on benchmark sets QF_ABV and ABV-init.

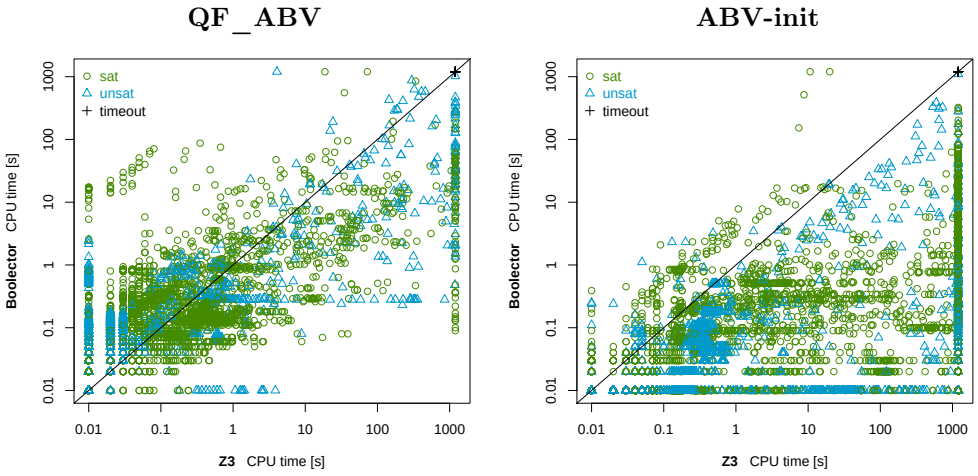


Figure 4.2: Runtime comparison of Boolector and Z3 on benchmark sets QF_ABV and ABV-init.

Table 4.1 summarizes the results for all solvers on benchmark sets QF_ABV and ABV-init. Adding the array initialization for one array already has a significant negative impact on the performance of CVC4 and Z3. Interestingly, CVC4 is not able to solve a single satisfiable instance, however, it outperforms Z3 on the number of unsatisfiable instances. Overall, CVC4 times out on 482 benchmarks, hits the memory limit on 12 benchmarks and reports unknown on 4358 benchmarks, and Z3 times out on 1126 benchmarks, hits the memory limit on 23 benchmarks and reports unknown in 161 cases. Boolector, on the other hand, is able to solve almost all of the benchmarks from the ABV-init benchmark set (11 timeout, 5 memory out). In fact, representing the initialized array as a lambda term enables Boolector to apply additional rewriting, which makes many benchmarks easier to solve. For CVC4 and Z3, depending on the internal solver architecture, a tighter integration between the array engine and the quantifier engine might yield similar results for this kind of patterns. Since the array engine is able to provide all relevant indices required for a complete instantiation of the quantifier in the initialization pattern, it should be possible to report satisfiable on these instances. Our propagation-based lemmas on demand approach as implemented in Boolector propagates read operations until fixpoint. As a result, all relevant read operations that access the initialized arrays are checked for consistency and fixed via lemmas in case of a conflict. The same idea could be used in combination with quantifier instantiation, where the array engine would instantiate quantifiers with indices on demand. This is similar to the approach for deciding the array property fragment [10], however, with the difference that quantifiers are instantiated lazily and lemmas are generated on demand.

As an interesting observation, Z3 was not able to solve 216 unsatisfiable benchmarks in the ABV-init set that were already determined to be unsatisfiable by Z3 without the array initialization (QF_ABV). Note that 183 of these 216 benchmarks in set QF_ABV were solved by Z3 in less than 100 seconds each. For CVC4 this behavior occurred only on 9 unsatisfiable instances of set ABV-init.

Figure 4.1 compares the runtime of Boolector and CVC4 on the QF_ABV and ABV-init benchmark set and shows a significant performance drop of CVC4 as soon as quantifiers are used for initializing arrays. On benchmark set ABV-init, CVC4 is not able to solve a single satisfiable instance but reports unknown, which is probably due to missing support for ABV in the model finding procedure. Considering unsatisfiable instances only, Boolector significantly outperforms CVC4. On the commonly solved instances (10238 in total) Boolector is over 23 times faster than CVC4 (733 vs. 17191 seconds). As a comparison, for the 4565 commonly solved unsatisfiable benchmarks of the QF_ABV benchmark set, Boolector requires 7219 seconds and CVC4 17604 seconds.

Figure 4.2 compares the runtime of Boolector and Z3 on both benchmark sets. Similar to CVC4, as soon as quantifiers for initializing arrays are involved, the performance of Z3 significantly drops. However, in contrast to CVC4, which is not able to solve a single satisfiable instance, Z3 solves 3714 satisfiable benchmarks. On the 13779 commonly solved instances, Boolector outperforms Z3 by

a factor of 33 and requires 4482 seconds in total, whereas Z3 requires 162325 seconds. Considering commonly solved unsatisfiable (satisfiable) instances only, Boolector requires 3091 (1792) seconds for solving 10067 (3712) benchmarks, whereas Z3 requires 44789 (117535) seconds in total.

Our experiment shows how a seemingly simple array initialization pattern involving quantifiers can have a considerable impact on the performance of state-of-the-art SMT solvers. With our specialized lemmas on demand for lambdas approach many of these patterns can be handled efficiently without affecting the performance of Boolector. However, we believe that similar results can be achieved with a tighter integration of the array and the quantifier engine, which is an interesting direction for future work. One possible approach to realize such a tighter integration would be a specialized procedure for the array property fragment [10] that combines our lemmas on demand approach with a lazy quantifier instantiation technique. This would allow to efficiently handle the array patterns discussed above, and further enables us to prove array properties such as sortedness, which can not be formulated with our lambda term approach.

4.2 Correction

The script that computed the results for commonly solved instances had a bug, which had the effect that it also included instances that hit a resource limit. As a result, for the commonly solved instances of configurations Btor and Btor+xm the numbers for the generated lemmas, reduction of CNF size and the time spent in the underlying SAT solver were incorrect. On the 13242 commonly solved QF_ABV benchmarks, configuration Btor generated 699027 (instead of 872913) and configuration Btor+xm 88762 lemmas (instead of 158175), which is a reduction by a factor of 7.9 (instead of 5.5). The size of the CNF is reduced by 24% on average (instead of 25%). Further, the time spent in the underlying SAT solver is reduced from 18175 to 13653 seconds (instead of 59638 to 40101 seconds), which is an improvement of 25% (instead of 33%).

Chapter 5

Paper C. Counterexample-Guided Model Synthesis

As discussed in Chapter 4, with non-recursive first-order lambda terms we can not express array properties that state, e.g., that an array is sorted or that all elements are within certain bounds. Further, our lemmas on demand for lambdas approach as introduced in Chapter 3 only handles extensionality of lambda terms that represent arrays and does not support equality over general lambda terms. When introducing quantifiers we are able to overcome these limitations. Consider, e.g., an array a that is sorted from index i to index j with $i < j$. We can express this sortedness of array a as $\forall x, y \,.\, (i \leq x \leq y < j \rightarrow a[x] \leq a[y])$. Equality of two arbitrary lambda terms f and g, on the other hand, can be asserted with $\forall x_1, \ldots, x_n \,.\, f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$.

In Paper C, we present a new approach called counterexample-guided model synthesis (CEGMS) for solving quantified SMT formulas with a particular interest in quantified bit-vectors. Our approach does not rely on current state-of-the-art techniques such as heuristic quantifier instantiation. It combines counterexample-guided quantifier instantiation with a syntax-guided synthesis approach called enumerative learning in order to synthesize interpretations for Skolem functions. Based on a set of ground instances, our approach tries to synthesize a candidate model for all Skolem functions. If the candidate model is valid our approach reports satisfiable and returns the model. Otherwise, a counterexample is generated, which is used for creating a new ground instance of the formula. These steps are repeated until either a ground conflict is found or a valid candidate model is synthesized. In Paper C we further introduce a dual CEGMS approach, which applies CEGMS to the negation of the formula in a parallel setting in order to synthesize quantifier instantiations that prove the unsatisfiability of the original formula. In our experiments, we compare CEGMS and its dual version to the state-of-the-art in solving quantified bit-vectors and show that it is competitive even though it does not employ any quantifier specific simplification techniques.

In Section 5.1 we discuss an extension of our CEGMS technique to generalize concrete counterexamples by means of synthesis. In Section 5.2 we evaluate the performance impact on CEGMS if quantifier specific simplifications are applied.

5.1 Synthesis of Quantifier Instantiations

As discussed in Paper C, our dual CEGMS approach is used to synthesize quantifier instantiations. However, we only utilize the final result of applying CEGMS to the dual formula and consequently, do not exchange intermediate results between the original and the dual formula. If dual CEGMS concludes with unsatisfiable on the dual formula (i.e., the original formula is satisfiable), our approach currently does not provide a model for the original formula. Further, if the input formula contains uninterpreted functions, dual CEGMS can not exploit the duality of the formula (as described in Paper C) and is therefore not applicable.

Our CEGMS technique is a model finding procedure that iteratively refines a set of ground instances of the input formula until either a valid model is synthesized or the set of ground instances becomes unsatisfiable. In each refinement step, based on the set of ground instances a candidate model is synthesized and checked for validity. If the candidate model is valid, the CEGMS procedure returns satisfiable. Otherwise, a counterexample is generated, which is used to create a new ground instance of the formula. A counterexample corresponds to a concrete assignment to universal variables for which the current candidate model does not hold. In the following, we extend our CEGMS technique to generalize these counterexamples by means of synthesis in order to find more general candidates for quantifier instantiation. We add an additional synthesis step to the refinement loop of our CEGMS procedure, which allows us to directly utilize the counterexamples generated in each refinement iteration to synthesize quantifier instantiations. This extension is a first step to combine the strengths of CEGMS and dual CEGMS into one procedure in order to overcome the limitations of dual CEGMS mentioned above.

Figure 5.1 shows the extended version of our CEGMS algorithm. Bold line numbers indicate the modified parts of the algorithm in comparison to the CEGMS procedure introduced in Paper C. Since quantifier instantiations are synthesized in each refinement step w.r.t. the counterexamples generated so far, we maintain a global set of counterexamples C. In each refinement step, if the current candidate model is not valid, the generated counterexample M_C (line 10) is added to set C (line 13). Based on the current set of counterexamples C, a set of quantifier instantiation candidates M_{QI} is synthesized, which maps universal variables to synthesized terms (line 14). Set M_{QI} is then used to create a new ground instance of formula φ_G by substituting the universal variables $\mathbf{u}$ with the corresponding quantifier instantiation candidates in M_{QI} (line 16). Note that if procedure synthesize is not able to synthesize any terms, it returns an empty set for M_{QI}. In this case, only the instance created via CEGQI is added to the set

```
1  function CEGMS_QI (φ)
2    G := ⊤, x := var_∀(φ), C := {}
3    φ_sk := skolemize(preprocess(φ))          // apply Skolemization
4    f := sym_sk(φ_sk)                         // Skolem symbols
5    φ_G := φ_sk[u/x]                          // ground φ_sk with fresh u
6    while true
7      r, M_G := sat(G)                        // check set of ground instances
8      if r = unsat return unsat               // found ground conflict
9      M_S := synthesize(f, G, M_G, φ_G)       // synthesize candidate model
10     r, M_C := sat(¬φ_G[M_S(f)/f])           // check candidate model
11     if r = unsat return sat                 // candidate model is valid
12     G := G ∧ φ_G[M_C(u)/u]                  // new ground inst. via CEGQI
13     C := C ∪ {M_C}                          // save counterexample
14     M_QI := synthesize(u, ¬φ_G, C, φ_G)     // synthesize quantifier inst.
15     if M_QI ≠ ∅
16       G := G ∧ φ_G[M_QI(u)/u]               // new ground inst. via M_QI
```

Figure 5.1: Extended version of our CEGMS algorithm 9.2 with quantifier instantiation synthesis. The bold line numbers indicate the modified parts of the original algorithm.

of ground instances G.

The main idea of maintaining a set of counterexamples C is to simulate model M_G for ground instances G of the dual CEGMS procedure. For example, given a formula $\forall \mathbf{x} \exists \mathbf{y}.\varphi[\mathbf{x}, \mathbf{y}]$ and its dual version $\exists \mathbf{x} \forall \mathbf{y}.\neg\varphi[\mathbf{x}, \mathbf{y}]$, in the dual case, set G consists of ground instances $g_1 \wedge g_2 \wedge \ldots \wedge g_n$ with g_i being of the form $\neg\varphi[\mathbf{x}, M_C(\mathbf{y})/\mathbf{y}]$. The model M_G is used to synthesize terms for variables $\mathbf{x}$, which correspond to quantifier instantiations in the original formula. In our new approach, we utilize the counterexamples generated in each refinement step (which satisfy $\neg\varphi_G[M_S(\mathbf{f})/\mathbf{f}]$) to synthesize quantifier instantiation candidates. This has the advantage that we do not have to maintain an additional set of negated ground instances to generate a model in each refinement step for the synthesis of quantifier instantiations.

For this purpose, we need to modify procedure synthesize (line 14) to also support the synthesis of terms for universal variables. First, the input selection of procedure synthesize (Section 9.4) needs to be extended to also consider the universal case, where all *existential* variables on which a universal variable depends are selected as inputs. Further, as described in Section 9.5, for the existential case, the signature computation of the enumerative learning algorithm

(procedure enumlearn 9.3) uses the set of ground instances G and the current model M_G to generate a signature for a term t synthesized for f. A signature of t is a tuple of Boolean values, where the i-th value corresponds to the evaluation $M[\![g_i]\!]$ of the i-th ground instance $g_i \in G[t/f]$ under current model M_G. For a universal variable u and the corresponding synthesized term t, the signature is computed by evaluating $\neg\varphi_G[t/u]$ for every counterexample $M_C \in C$, which yields tuple $(M_C[\![\neg\varphi_G[t/u]]\!] \mid M_C \in C)$.

Note that our counterexample generalization differs from our dual CEGMS approach in several ways. First of all, it does not exploit the concept of duality as dual CEGMS. Thus, it is possible to employ our approach in the presence of uninterpreted functions. However, due to the duality of the input formula, dual CEGMS is able to report satisfiable if the dual formula is unsatisfiable (but can not provide a model for the original formula). With counterexample generalization, we can not detect this case. Further, counterexample generalization generates quantifier instantiations in each refinement step, whereas dual CEGMS provides one "final" quantifier instantiation that immediately produces a ground conflict. However, in some cases it may be easier to find ground conflicts if multiple terms are used as candidates for quantifier instantiation.

5.1.1 Experiments

We implemented the extended version of our CEGMS approach in our SMT solver Boolector and evaluated it on benchmark sets BV and BV_{LNIRA} with the same hardware setup and resource limits as in Paper C (1200 seconds CPU time, 7GB memory). For the evaluation we added the following two new configurations of Boolector and compared them to the configurations used in Paper C.

(1) **Btor+sg** Boolector with CEGMS (configuration Btor+s in Paper C) and counterexample generalization enabled.

(2) **Btor+dsg** Boolector with dual CEGMS (configuration Btor+ds in Paper C) and counterexample generalization enabled.

Table 5.1 summarizes the results of configurations Btor+sg and Btor+dsg compared to the Boolector configurations evaluated in Paper C. On the BV benchmark set, counterexample generalization barely improves the overall performance of configurations Btor+sg and Btor+dsg. However, on benchmark set $\text{BV}_{\text{BVLNIRA}}$, both Btor+sg and Btor+dsg solve considerably more benchmarks compared to Btor+s and Btor+ds, where the majority is unsatisfiable. This is expected since counterexample generalization helps to find unsatisfiable instances due to the additional synthesis of quantifier instantiations. Configuration Btor+sg solves 190 additional instances of which 188 are unsatisfiable and now even outperforms Btor+ds by 12 instances. Further, with a CPU time limit of 1200 seconds, configuration Btor+sg solves the most unsatisfiable instances of all configurations. The difference of solved unsatisfiable instances between

	BV (191)					**BV_{LNIRA}** (4838)				
	Slvd	Sat	Unsat	Time [s]	Uniq	Slvd	Sat	Unsat	Time [s]	Uniq
Btor	142	51	91	59529	0	4527	465	4062	389123	3
Btor+s	164	72	92	32996	0	4526	467	4059	390661	1
Btor+d	162	67	95	35877	0	4572	518	4054	342412	4
Btor+ds	172	77	95	24163	0	4704	517	4187	187411	2
Btor+sg	163	71	92	34162	0	4716	469	**4247**	164274	0
Btor+dsg	**173**	**78**	95	**23789**	6	**4761**	**519**	4242	**122174**	**3**

Table 5.1: Results for all configurations on the BV and BV_{LNIRA} benchmarks.

Btor+sg and Btor+dsg is due to the fact that for configuration Btor+dsg the effective CPU time limit per thread is 600 seconds, but these benchmarks require more than 600 seconds to be solved by Btor+sg. With a CPU time limit of 1200 seconds per thread, Btor+dsg is also able to solve these benchmarks. An interesting observation is that Btor+dsg solves one more instance on the BV benchmark set compared to Btor+ds, but solves 6 instances that can not be solved by other configurations. This is due to the fact that counterexample generalization introduces additional overhead in each refinement iteration while synthesizing quantifier instantiations. As a consequence, while Btor+dsg is able to solve 6 more instances, it loses 5 other instances due to the overhead. Further, on benchmark set BV_{LNIRA}, configuration Btor+sg solves the most number of unsatisfiable instances. However, compared to configuration Btor+dsg it solves 50 satisfiable instances less. This is due to the fact that in these cases, dual CEGMS determines that the dual formula is unsatisfiable and consequently, concludes with satisfiable. On the BV benchmark set, this is the case for 7 out of the additional 10 satisfiable instances solved by Btor+dsg.

Figure 5.2 illustrates the distribution of refinement iterations required by configuration Btor+sg for the solved instances of benchmark sets BV and BV_{LNIRA}. On the BV benchmark set, at least one refinement iteration was required to solve an instance. For the majority of solved benchmarks (161 out of 163) Btor+sg required at most two refinement iterations. The remaining two instances were solved within 4 and 5 iterations respectively. On the BV_{LNIRA} benchmark set half of the solved instances were either solved by rewriting (2328 instances) or the simplified formula did not contain any universal quantifiers (104 instances). The majority of the remaining instances (41%) were solved within one refinement iteration. For the other solved instances at least two refinement iterations were required. The maximum number of refinement iterations required was 98 (one instance).

Figures 5.3 and 5.4 illustrate the effect of counterexample generalization on the runtime of configurations Btor+sg and Btor+dsg, respectively. For configu-

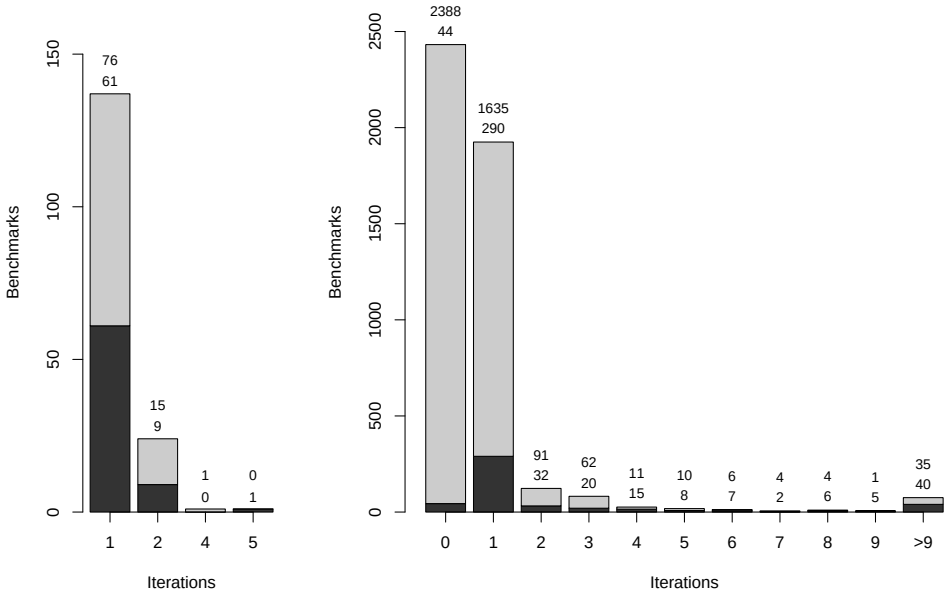


Figure 5.2: Distribution of refinement iterations required on benchmark sets BV (left) and BV_{LNIRA} (right) with configuration Btor+sg. The dark gray color corresponds to satisfiable instances, the light gray color to unsatisfiable instances.

ration Btor+sg, on the solved instances of the BV benchmark set the overhead of counterexample generalization is negligible, whereas on the BV_{LNIRA} benchmark set it amounts to 8% of the overall runtime. The three (satisfiable) outliers on the BV benchmark set are due to counterexample generalization producing additional ground instances, which are passed down to the SAT solver. These benchmarks are part of the ranking benchmark family and were very unstable in our experiments if the set of ground instances changed. Investigating this behavior is left to future work. The overhead for configuration Btor+dsg on the BV_{LNIRA} is similar to Btor+sg with 8%. On the BV benchmark set, however, the overhead amounts to 25% of the total runtime.

Table 5.2 summarizes the results of configurations Btor+sg and Btor+dsg compared to CVC4, Q3B, and Z3, which we also evaluated in Paper C. On the BV benchmark set Q3B still solves the highest number of benchmarks. However, on the BV_{LNIRA} benchmark set configuration Btor+dsg now solves more instances than Z3 and the highest number of satisfiable instances. Still, Z3 solves the highest number of unsatisfiable instances (9 more than Btor+sg), which we assume is due to its heuristic quantifier instantiation techniques.

Figure 5.5 depicts a cactus plot over the runtime of configurations Btor+sg and Btor+dsg compared to CVC4, Q3B, and Z3 on benchmark BV_{LNIRA}. Even

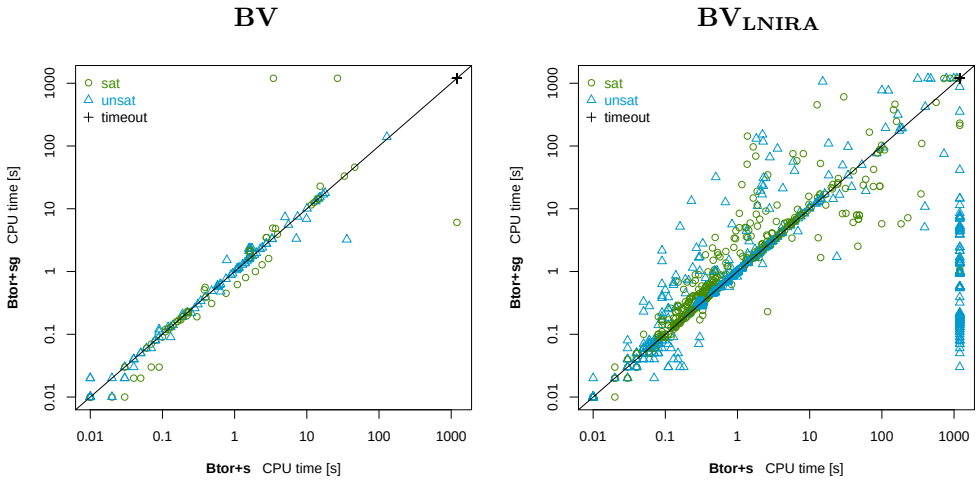


Figure 5.3: Comparison of CEGMS with counterexample generalization enabled (Btor+sg) and disabled (Btor+s) on the BV and BV_{LNIRA} benchmarks.

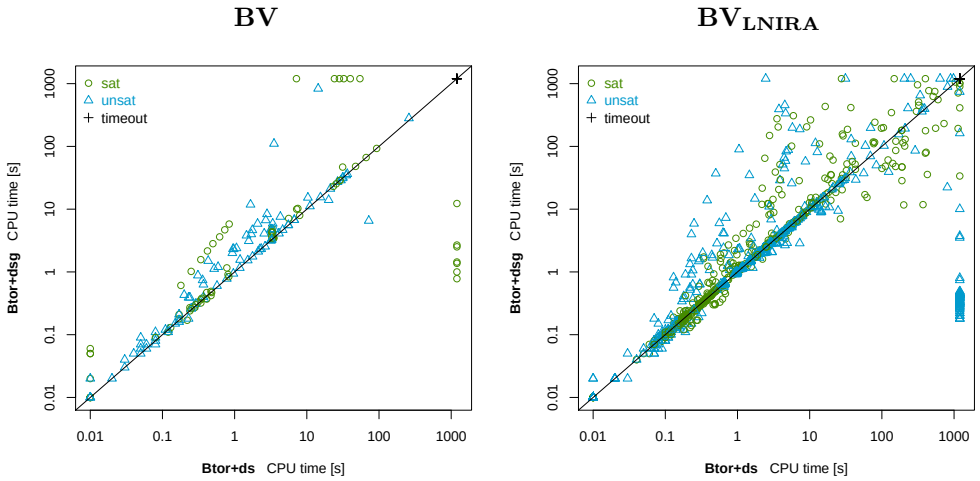


Figure 5.4: Comparison of dual CEGMS with counterexample generalization enabled (Btor+dsg) and disabled (Btor+ds) on the BV and BV_{LNIRA} benchmarks.

	BV (191)					**BV_{LNIRA}** (4838)				
	Slvd	Sat	Unsat	Time [s]	Uniq	Slvd	Sat	Unsat	Time [s]	Uniq
Btor+ds	172	77	95	24163	0	4704	517	4187	187411	4
CVC4	145	64	81	57652	0	4362	339	4023	580402	2
Q3B	**187**	**93**	94	**9086**	9	4367	327	4040	581252	4
Z3	161	69	92	36593	0	4732	476	**4256**	130405	**10**
Btor+sg	163	71	92	34162	0	4716	469	4247	164274	1
Btor+dsg	173	78	95	23789	0	**4761**	**519**	4242	**122174**	1

Table 5.2: Results for all solvers on the BV and BV_{LNIRA} benchmarks with a CPU time limit of 1200 seconds.

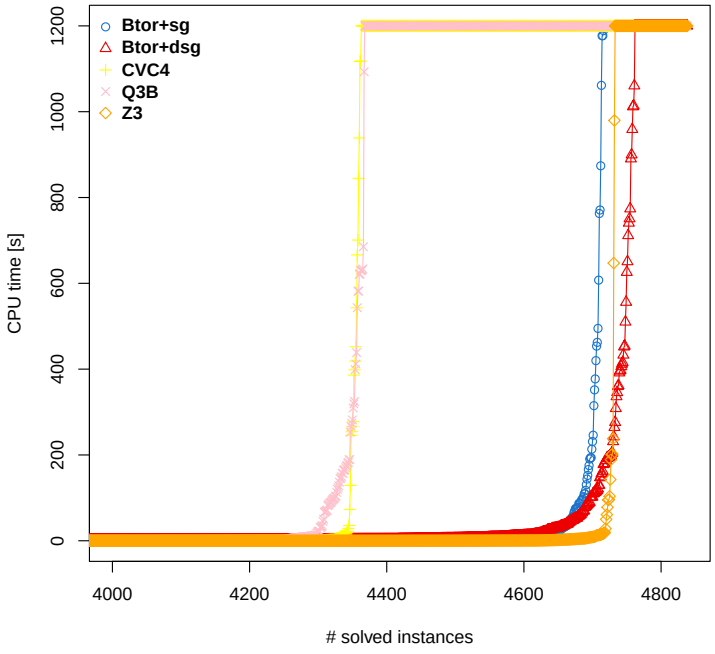


Figure 5.5: Cactus plot of runtime of all solvers on benchmark set BV_{LNIRA} with a CPU time limit of 1200 seconds.

though Btor+dsg solves the highest number of instances, Z3 solves many benchmarks in less time. This is on the one hand due to the synthesis overhead produced in every refinement step and on the other hand due to the lack of quantifier specific simplifications as discussed in Paper C. Employing simplifications that potentially eliminate existential and universal variables also reduces the synthesis work since we have to synthesize terms for less variables. Further, reducing the scopes of the variables may also have an impact on synthesis since it reduces the number of inputs for each variable and consequently reduces the number of enumerated expressions.

In general, counterexample generalization adds additional overhead in terms of runtime, however, it considerably increases the overall number of solved instances for both configurations Btor+sg and Btor+dsg with the majority being unsatisfiable. In our experiments, we identified the signature computation as the main cause for the synthesis overhead, which affects counterexample generalization as well as CEGMS (and its dual version). The signature computation can be expensive since it has to be done for every enumerated expression in order to check if the current expression satisfies all ground instances and if it has the same signature as an already enumerated expression. Improving the synthesis procedure is one of our top-priority directions for future work as it will improve CEGMS, dual CEGMS and the counterexample generalization approach at the same time.

5.2 Quantifier Specific Simplifications

In this section we investigate quantifier specific simplification techniques and evaluate their effectiveness in combination with our extended CEGMS approach as introduced in Section 5.1. For our evaluation we implemented the following three simplification techniques in Boolector.

Miniscoping [56] is a technique to minimize the scope of a quantifier by applying the following rules.

$$\forall x.(\varphi[x] \wedge \psi[x]) \rightsquigarrow (\forall x.\varphi[x]) \wedge (\forall x.\psi[x]) \tag{MS1}$$

$$\exists x.(\varphi[x] \vee \psi[x]) \rightsquigarrow (\exists x.\varphi[x]) \vee (\exists x.\psi[x]) \tag{MS2}$$

$$Qx.(\varphi[x] \diamond \psi) \rightsquigarrow (Qx.\varphi[x]) \diamond \psi \quad \text{with } Q \in \{\forall, \exists\},\ \diamond \in \{\wedge, \vee\} \tag{MS3}$$

In our case, we employ a lightweight version of miniscoping, which only applies rule MS3 since it does not introduce additional quantifiers. Introducing additional universal and existential quantifiers may result in more overhead when synthesizing candidate models and quantifier instantiations. Implementing rules MS1 and MS2, and evaluating the additional synthesis overhead against the benefit of having smaller scopes is left to future work. In general, minimizing

the scope of quantifiers is beneficial for our CEGMS approach since it reduces the set of inputs used for synthesizing candidate models and quantifier instantiations. As discussed in Paper C, the set of inputs used to synthesize a term for a variable x is selected w.r.t. the variables on which x depends. Consequently, reducing the number of inputs also reduces the number of expressions to be enumerated.

The second technique eliminates universal quantifiers from a formula by applying the following equality substitution rule (e.g., [48])

$$\forall x, y.(x = t \rightarrow \varphi[x, y]) \rightsquigarrow (\forall y.\varphi[x/t, y]), \tag{DER}$$

where x does not occur in t. This technique is sometimes also referred to as *destructive equality resolution* (DER). Eliminating universal variables helps our CEGMS approach in two ways. First of all, it reduces the number of variables for which we need to synthesize quantifier instantiations. Second, since we apply Skolemization as the last step in our simplifications, DER also reduces the number of inputs used for synthesis.

The third technique eliminates existential quantifiers from a formula by applying the following equality substitution rule (e.g., [48])

$$\exists x, y.(x = t \wedge \varphi[x, y]) \rightsquigarrow (\exists y.\varphi[x/t, y]), \tag{CER}$$

where x does not occur in t. This technique is sometimes referred to as *constructive equality resolution* (CER) [38]. CER is the dual case of DER and eliminates existential variables, which helps to improve the synthesis of candidate models as well as quantifier instantiations. In Boolector, we employ a technique called *variable substitution*, which is similar to CER and works on ground formulas. Variable substitution finds top-level equalities of the form $v = t$ with v being a constant and t being an arbitrary term. If v does not occur in t, it gets substituted with term t in the formula.

We implemented above simplification techniques in Boolector and repeated the experiments of Section 5.1 with configurations Btor+sg and Btor+dsg on benchmark sets BV and BV_{LNIRA}. We used the same hardware setup and resource limits (1200 seconds CPU time, 7GB memory, 1200 seconds penalty) as in Section 5.1.

Table 5.3 summarizes the results for configurations Btor+sg and Btor+dsg on benchmark set BV, where we compared every combination of the simplification techniques. Enabling miniscoping (ms) improves the performance of configurations Btor+sg and Btor+dsg, where the number of inputs for an existential variable is reduced by 80% on average for each benchmark. This is particularly useful since reducing the number of inputs also reduces the number of expressions that need to be enumerated during synthesis. The number of inputs for the universal variables is unaffected since the quantifier prefix for each benchmark in the fixpoint family is $\forall\exists$. On the benchmarks of the ranking family miniscoping was not able to reduce any quantifier scopes. However, all benchmarks have

	Btor+sg				Btor+dsg			
	Slvd	Sat	Unsat	Time [s]	Slvd	Sat	Unsat	Time [s]
default	163	71	92	34166	173	78	95	23538
ms	168	74	94	28452	175	80	95	20651
cer	162	72	90	35380	171	78	93	25260
der	165	73	92	32388	169	75	94	28060
cer+der	170	80	90	26157	**177**	**82**	**95**	**18913**
ms+cer	**177**	**82**	**95**	**17984**	177	82	95	19154
ms+der	167	73	94	29416	171	77	94	26074
ms+cer+der	175	81	94	20300	176	81	95	19849

Table 5.3: Results for configurations Btor+sg and Btor+dsg on benchmark set BV (191 benchmarks in total) with different simplification techniques enabled

an $\exists\forall$ quantifier prefix and consequently, the inputs for existential variables can not be further reduced. Enabling only DER (der) or CER (cer) is less effective and does not achieve any improvements compared to the default version except for Btor+sg with DER. In most cases configurations with DER or CER enabled solve even less instances, which is due to the fact that the theory solvers used for finding and checking a candidate model receive a different formula. This may produce different models and counterexamples, which in some cases may direct the search into different directions. With DER the number of universal variables is reduced by 63% on average, whereas with CER no reduction was achieved. However, by combining CER and DER (cer+der) configurations Btor+sg and Btor+dsg solve 7 and 4 more instances. For configuration Btor+dsg the combination of CER and DER achieves the best results on the BV benchmark set. This combination reduces the number of existential variables for only 10 benchmarks (small-equiv-fixpoint-* benchmarks). However, on these benchmarks the reduction of existential variables is on average 55%, which is the reason for the increase in the number of solved instances. The small-equiv-fixpoint-* benchmarks can be considerably simplified with CER when combined with either DER or miniscoping. The combination of miniscoping with CER (ms+cer) solves the most instances for both Btor+sg and Btor+dsg, whereas ms+der seems less effective. Combination ms+cer eliminates on average 39% of the existential variables. The reduction of universal variables for combination ms+der is with 28% on average less compared to DER only. This is due to the fact that in some cases miniscoping pushes universal quantifiers all the way down to inequalities ($\forall x.x \neq t$), for which the DER rule is not applicable anymore. However, these cases would be eliminated with additional rewriting since $\forall x.x \neq t$ simplifies to false if the domain of x contains more than one value, which is always the case for bit-vectors (smallest bit-vector size is 1 with domain values $\{0, 1\}$). Interest-

	Btor+sg				Btor+dsg			
	Slvd	Sat	Unsat	Time [s]	Slvd	Sat	Unsat	Time [s]
default	4715	469	4246	165096	**4760**	**520**	**4240**	**123698**
ms	4713	469	4244	168876	4754	515	4239	131026
cer	4708	464	4244	173230	4757	517	4240	127100
der	4709	465	4244	171189	4757	517	4240	126765
cer+der	**4715**	**469**	**4246**	**164842**	4758	518	4240	124500
ms+cer	4708	466	4242	172446	4752	513	4239	127812
ms+der	4707	465	4242	172888	4753	514	4239	128810
ms+cer+der	4712	468	4244	168607	4753	514	4239	131086

Table 5.4: Results for configurations Btor+sg and Btor+dsg on benchmark set $\mathrm{BV}_{\mathrm{LNIRA}}$ (4838 benchmarks in total) with different simplification techniques enabled

ingly, the combination of miniscoping and CER on configuration Btor+sg even outperforms the best configuration of Btor+dsg in terms of runtime. Enabling all simplification techniques (ms+cer+der) does not yield the best performance on the BV benchmark set, however, it is still close to the best combinations. This combination reduces the number of existential variables by 41% and the number of universal variables by 28% on average.

Table 5.4 summarizes the results for configurations Btor+sg and Btor+dsg on benchmark set $\mathrm{BV}_{\mathrm{LNIRA}}$ with all combinations of simplification techniques. Interestingly, on this benchmark set none of the simplification techniques achieved any improvement.

Figures 5.6 and 5.7 compare different combinations of simplification techniques for configuration Btor+sg on the BV and $\mathrm{BV}_{\mathrm{BVLNIRA}}$ benchmark sets. The plots for configuration Btor+dsg are not included since they exhibit a behavior similar to configuration Btor+sg.

By enabling simplifications Btor+sg improves considerably compared the configuration without simplifications and is able to solve 14 more instances in almost half of the runtime. For configuration Btor+dsg the improvement is at most 4 instances, however, the default version of Btor+dsg already solves 10 more instances compared to the default version of Btor+sg, which leaves less room for improvement. The majority of the additionally solved instances were satisfiable. These results suggest that sufficiently reducing the number of existential variables is always beneficial for our CEGMS approach.

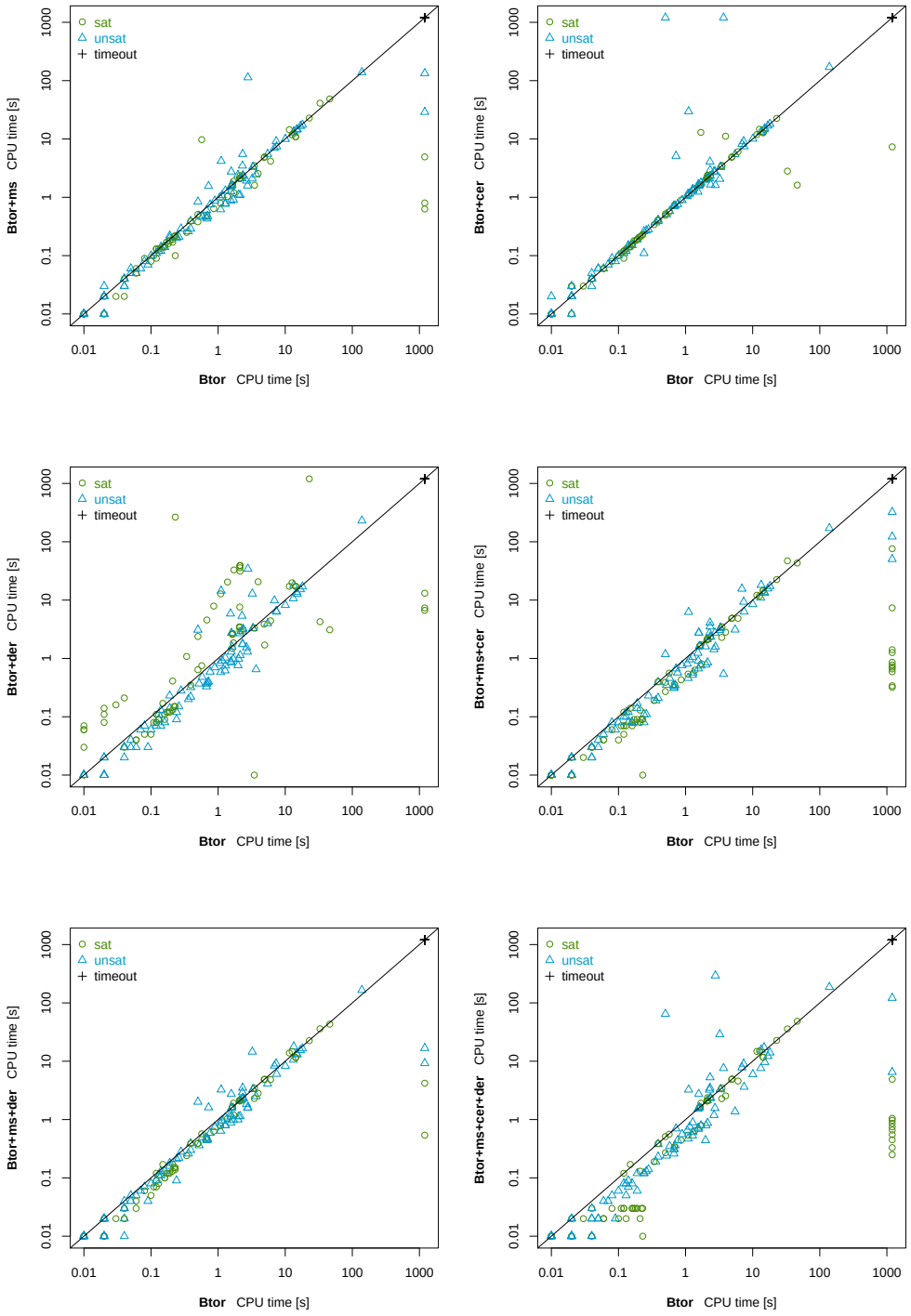


Figure 5.6: Comparison of configuration Btor+sg with different simplification technique on the $\mathrm{BV}_{\mathrm{LNIRA}}$ benchmark set.

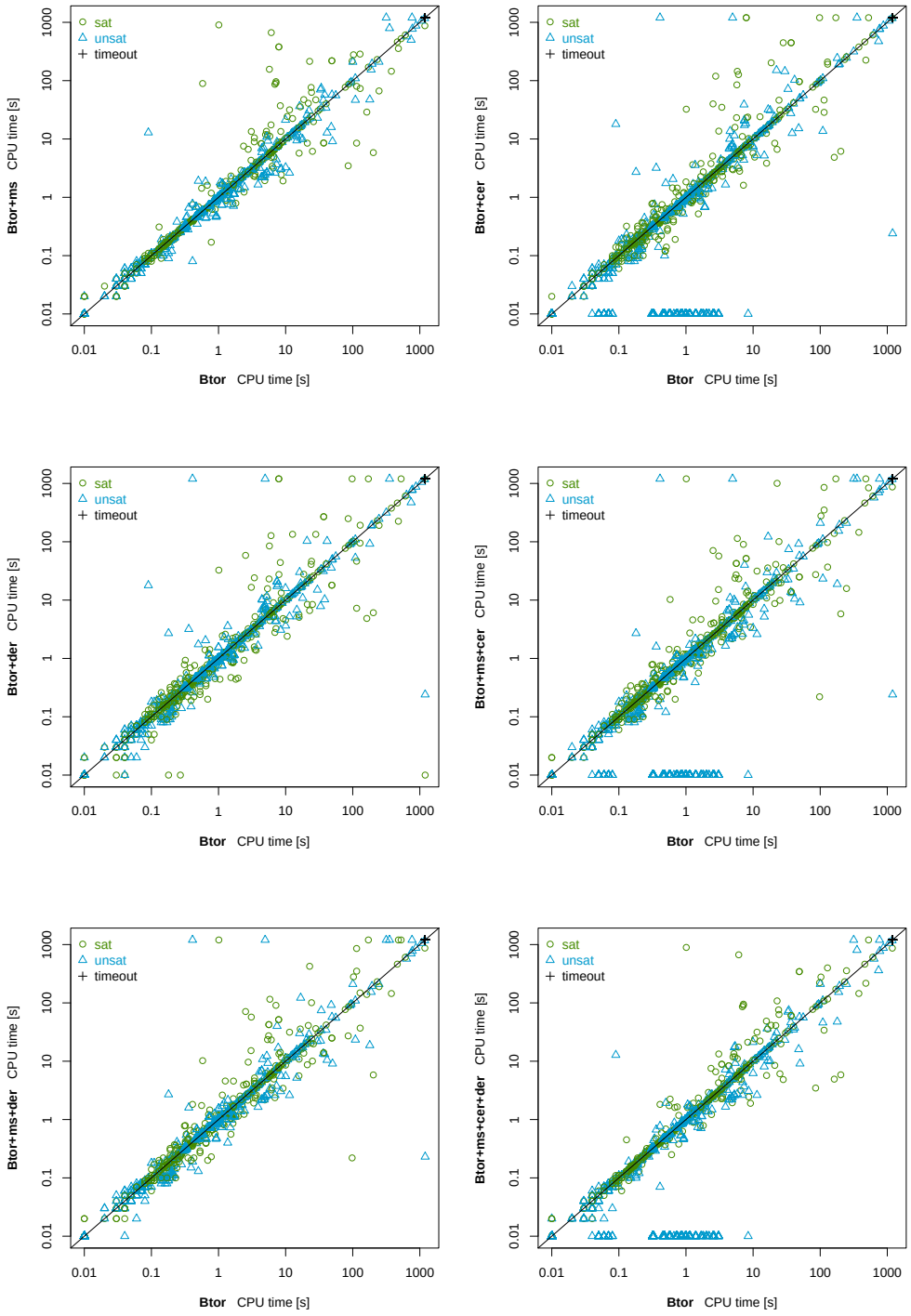


Figure 5.7: Comparison of configuration Btor+sg with different simplification technique on the BV_{LNIRA} benchmark set.

5.3 Discussion

In Section 5.1, we discussed an extension to our original CEGMS technique, which allows us to synthesize quantifier instantiations in each refinement iteration based on a set of concrete counterexamples. As a result, this considerably improved the number of solved unsatisfiable instances for both techniques CEGMS and dual CEGMS, where dual CEGMS was even able to outperform all other solvers on the BV_{LNIRA} benchmark set. The gap between CEGMS and dual CEGMS became smaller, however, employing the dual CEGMS approach is still beneficial. In Section 5.2, we investigated the impact of some quantifier specific simplification techniques on our CEGMS technique. The results of our experiments are mixed. Depending on the benchmark set the employed simplification techniques can either improve or worsen the performance of CEGMS. However, the fact that CEGMS and dual CEGMS are competitive with the state-of-the-art without any simplification techniques is very promising since for other approaches [38, 63] simplification techniques are crucial.

Our dual CEGMS approach is currently not applicable in the presence of uninterpreted function symbols. In this case we can not exploit the duality concept as described in Paper C. Finding a solution for this problem is left to future work. A more interesting question is, however, if we can improve CEGMS combined with counterexample generalization in order to make dual CEGMS obsolete.

We further plan to extend CEGMS in Boolector to also support the combination with uninterpreted functions, arrays and lambda terms. This would allow us to handle various array properties such as sortedness (as discussed in Section 4.1) as well as extensional lambda terms. Applying our CEGMS approach to other theories such as linear integer arithmetic (LIA) or linear real arithmetic (LRA) is another interesting direction for future work.

Chapter 6

Conclusion

In this thesis, we explored an alternative approach to reason about arrays in SMT by means of non-recursive first-order lambda terms and presented a new decision procedure to lazily handle these lambda terms. We discussed various scenarios where lambda terms are beneficial when used for representing array operations. Further, we presented a technique called counterexample-guided model synthesis (CEGMS) for solving quantified SMT problems, with a particular focus on quantified bit-vectors. This approach does not rely on techniques commonly used in current state-of-the-art SMT solvers such as heuristic quantifier instantiation or finite model finding. Instead, it employs a combination of counterexample-guided quantifier instantiation with a syntax-guided synthesis approach to synthesize models and quantifier instantiations. Our experimental results showed that our technique is competitive with the state-of-the-art in solving quantifier bit-vectors even though we did not employ any quantifier specific simplification techniques. The initial work on these topics were part of the peer-reviewed papers A, B and C, which were included as Chapters 7-9 in the second part of this thesis. In the first part, we revisited the topics of Papers A-C and discussed several new contributions that extend and improve our presented techniques.

In Chapter 3, we presented a simple extension to the lemmas on demand procedure introduced in Paper A to handle extensional lambda terms that represent arrays. In our experiments, we showed that our approach for handling extensionality is competitive with the lemmas on demand procedure originally implemented in Boolector for the theory of arrays [11]. We further discussed an optimization of our lemmas on demand approach that allows us to generate lemmas more eagerly, which improves the overall performance of the procedure. We revisited the approach on how lambdas are treated in our SMT solver Boolector and proposed a more refined approach, which only employs lambda terms if multiple array operations can be combined. We leave this enhancement and the other discussed optimizations to future work.

In Chapter 4, we revisited the work presented in Paper B and discussed various array patterns, which can be represented by lambda terms and quantifiers. We evaluated both approaches based on a simple array initialization pattern. Our experiment showed that using quantifiers for this kind of patterns can have a considerable negative impact on the performance of state-of-the-art SMT solvers.

However, this is mainly due to the lack of a specialized procedure that is able to efficiently handle such array patterns in combination with quantifiers. We briefly discussed an idea for a specialized theory solvers that combines a lemmas on demand approach for the theory of arrays with a lazy quantifier instantiation technique, which may achieve results similar to our lambda approach. We leave the development of such a specialized theory solver to future work.

In Chapter 5, we described an extension for our CEGMS approach introduced in Paper C to generalize concrete counterexamples by means of synthesis. The proposed technique considerably improves the performance of our CEGMS approach (and its dual version), particularly on unsatisfiable benchmarks. Improving our new technique in order to render dual CEGMS obsolete is left to future work. In Paper C, we did not employ any quantifier specific simplifications. Hence, in Chapter 5, we investigated the effect of the simplification techniques miniscoping, destructive and constructive equality resolution on our CEGMS approach. In some cases, we were able to solve more instances due to simplifications, but most of the time there was no or little performance gain. However, (dual) CEGMS was already competitive without employing quantifier specific simplification techniques, which usually are crucial for other state-of-the-art approaches for solving quantified SMT problems. Improving the overall CEGMS procedure and applying it to more theories (e.g, LIA and LRA) is an another interesting direction for future work.

The work presented in Papers A and B and parts of Chapter 3 contributed to Boolector winning the QF_ABV division at the SMT competitions in 2014, 2015 and 2016, and the QF_UFBV division in 2015 and 2016. The work in Paper C and its extension introduced in Chapter 5 are a promising alternative approach for solving quantified SMT problems and we believe that there is still a lot of room for improving these techniques.

Part II

Publications

A

Chapter 7

Paper A. Lemmas on Demand for Lambdas

Published In Proceedings of the 2nd International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS 2013), affiliated to the 13th International Conference on Formal Methods in Computer Aided Design (FMCAD 2013), Portland, OR, USA, 2013.

Authors Mathias Preiner, Aina Niemetz and Armin Biere.

Modifications Sequences $x_1, \ldots, x_n$ now start with 1 instead of 0. Instead of λ term (β-reduction) we now write lambda term (beta reduction). Solver configurations in experiments use uniform naming scheme.

Abstract We generalize the lemmas on demand decision procedure for array logic as implemented in Boolector to handle non-recursive and non-extensional lambda terms. We focus on the implementation aspects of our new approach and discuss the involved algorithms and optimizations in more detail. Further, we show how arrays, array operations and SMT-LIBv2 macros are represented as lambda terms and lazily handled with lemmas on demand. We provide experimental results that demonstrate the effect of native lambda support within an SMT solver and give an outlook on future work.

7.1 Introduction

The theory of arrays as axiomatized by McCarthy [43] enables us to reason about memory (components) in software and hardware verification, and is particularly important in the context of deciding satisfiability of first-order formulas w.r.t. first-order theories, also known as *Satisfiability Modulo Theories (SMT)*. However, it is restricted to array operations on single array indices and lacks support for efficiently modeling operations such as memory initialization and parallel updates (*memset* and *memcpy* in the standard C library).

In 2002, Seshia et al. [12] introduced an approach to overcome these limitations by using restricted lambda terms to model array expressions (such as *memset* and *memcpy*), ordered data structures and partially interpreted functions within the SMT solver UCLID [58]. The SMT solver UCLID employs an eager SMT solving approach and therefore eliminates all lambda terms through beta reduction, which replaces each argument variable with the corresponding argument term as a preliminary rewriting step. Other SMT solvers that employ a lazy SMT solving approach and natively support lambda terms such as CVC4 [3] or Yices [25] also treat them eagerly, similarly to UCLID, and eliminate all occurrences of lambda terms by substituting them with their instantiated function body (cf. C-style macros). Eagerly eliminating lambda terms via beta reduction, however, may result in an exponential blow-up in the size of the formula [58]. Recently, an extension of the theory of arrays was proposed [27], which uses lambda terms similarly to UCLID. This extension provides support for modeling *memset*, *memcpy* and loop summarizations. However, it does not make use of native support of lambda terms provided by an SMT solver. Instead, it reduces instances in the theory of arrays with lambda terms to a theory combination supported by solvers such as Boolector [11] (without native support for lambda terms), CVC4, STP [32], and Z3 [17].

In this paper, we generalize the decision procedure for the theory of arrays with bit-vectors as introduced in [11] to lazily handle non-recursive and non-extensional lambda terms. We show how arrays, array operations and SMT-LIBv2 macros are represented in Boolector as lambda terms and introduce a lemmas on demand procedure for lazily handling lambda terms in Boolector in detail. We summarize an experimental evaluation and compare our results to solvers with SMT-LIBv2 macro support (CVC4, MathSAT [14], SONOLAR [42] and Z3) and finally, give an outlook on future work.

7.2 Preliminaries

We assume the usual notions and terminology of first-order logic and are mainly interested in many-sorted languages, where bit-vectors of different bit width correspond to different sorts and array sorts correspond to a mapping $(\tau_i \Rightarrow \tau_e)$ from index sort τ_i to element sort τ_e. Our approach is focused primarily on the

quantifier-free first-order theories of *fixed size bit-vectors*, *arrays* and *equality with uninterpreted functions*, but not restricted to the above.

We call 0-arity function symbols *constant* symbols and a, b, i, j, and e denote constants, where a and b are used for array constants, i and j for array indices, and e for an array value. For each bit-vector of size n, the equality $=_n$ is interpreted as the identity relation over bit-vectors of size n. We further interpret the *if-then-else* bit-vector term ite_n as $\text{ite}(\top, t, e) =_n t$ and $\text{ite}(\bot, t, e) =_n e$. As a notational convention, the subscript might be omitted in the following. We identify *read* and *write* as basic operations on array elements, where $\text{read}(a, i)$ denotes the value of array a at index i, and $\text{write}(a, i, e)$ denotes the modified array a overwritten at position i with value e. The theory of arrays (without extensionality) is axiomatized by the following axioms, originally introduced by McCarthy in [43]:

$$i = j \rightarrow \text{read}(a, i) = \text{read}(a, j) \tag{A1}$$

$$i = j \rightarrow \text{read}(\text{write}(a, i, e), j) = e \tag{A2}$$

$$i \neq j \rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j) \tag{A3}$$

The *array congruence* axiom A1 asserts that accessing array a at two equal indices i and j produces the same element. The *read-over-write* axioms A2 and A3 ensure a basic characteristic of arrays: A2 asserts that accessing a modification to an array a at the index it has most recently been updated (i), produces the value it has been updated with (e). A3 captures the case when a modification to an array a is accessed at an index other than the one it has most recently been updated at (j), which produces the unchanged value of the original array a at position j. Note that we assume that all variables a, i, j and e in axioms A1, A2 and A3 are universally quantified.

From the theory of equality with uninterpreted functions we primarily focus on the following axiom:

$$\forall \bar{x}, \bar{y}. \bigwedge_{i=1}^{n} x_i = y_i \rightarrow f(\bar{x}) = f(\bar{y}) \tag{EUF}$$

The *function congruence* axiom (EUF) asserts that a function evaluates to the same value for the same argument values.

We only consider a non-recursive lambda calculus, assuming the usual notation and terminology, including the notion of *function application*, *currying* and *beta reduction*. In general, we denote a lambda term λ_x as $\lambda x.t(x)$, where x is a variable *bound* by λ_x and $t(x)$ is a term in which x may or might not occur. We interpret $t(x)$ as defining the *scope* of bound variable x. Without loss of generality, the number of bound variables per lambda term is restricted to exactly one. Functions with more than one parameter are transformed into a chain of nested lambda terms by means of *currying* (e.g. $f(x, y) \coloneqq x + y$ is rewritten

as $\lambda x.\lambda y.x + y$). As a notational convention, we will use $\lambda_{\bar{x}}$ as a shorthand for $\lambda x_1 \ldots \lambda x_k.t(x_1, \ldots, x_k)$ for $k \geq 1$. We identify the *function application* as an explicit operation on lambda terms and interpret it as instantiating a bound variable (all bound variables) of a lambda term (a curried lambda chain). We interpret *beta reduction* as a form of function application, where all formal parameter variables (bound variables) are substituted with their actual parameter terms. We will use $\lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]$ to indicate beta reduction of a lambda term $\lambda_{\bar{x}}$, where the formal parameters $x_1, \ldots, x_n$ are substituted with the actual argument terms $a_1, \ldots, a_n$.

7.3 Lambda terms in Boolector

In contrast to lambda term handling in other SMT solvers such as e.g. UCLID or CVC4, where lambda terms are eagerly eliminated, in Boolector we provide a lazy lambda term handling with *lemmas on demand*. We generalized the lemmas on demand decision procedure for the extensional theory of arrays introduced in [11] to handle lemmas on demand for lambda terms as follows.

In order to provide a uniform handling of arrays and lambda terms within Boolector, we generalized all arrays (and array operations) to lambda terms (and operations on lambda terms) by representing *array variables* as uninterpreted functions (UF), *read* operations as function applications, and *write* and *if-then-else* operations on arrays as lambda terms. We further interpret macros (as provided by the command *define-fun* in the SMT-LIBv2 format) as (curried) lambda terms. Note that in contrast to [11], our implementation currently does not support extensionality (equality) over arrays (lambda terms).

We represent an *array* as exactly one lambda term with exactly one bound variable (parameter) and define its representation as $\lambda j.t(j)$. Given an array of sort $(\tau_i \Rightarrow \tau_e)$ and its lambda term representation $\lambda j.t(j)$, we require that bound variable j is of sort index τ_i and term $t(j)$ is of sort element τ_e. Term $t(j)$ is not required to contain j and if it does not contain j, it represents a *constant* lambda term (e.g. $\lambda j.0$). In contrast to SMT-LIBv2 macros, it is not required to represent arrays with curried lambda chains, as arrays are accessed at one single index at a time (cf. *read* and *write* operations on arrays).

We treat *array variables* as UF with exactly one argument and represent them as f_a for array variable a.

We interpret *read* operations as function applications on either UF or lambda terms with read index i as argument and represent them as $\text{read}(a, i) \equiv f_a(i)$ and $\text{read}(\lambda j.t(j), i) \equiv (\lambda j.t(j))(i)$, respectively.

We interpret *write* operations as lambda terms modeling the result of the *write* operation on array a at index i with value e, and represent them as $\text{write}(a, i, e) \equiv \lambda j.\text{ite}(i = j, e, f_a(j))$. A function application on a lambda term λ_w representing a *write* operation yields value e if j is equal to the modified index i, and the unmodified value $f_a(j)$, otherwise. Note that applying beta reduction

to a lambda term λ_w yields the same behavior described by array axioms A2 and A3. Consider a function application on $\lambda_w(k)$, where k represents the position to be read from. If $k = i$ (A2), beta reduction yields the written value e, whereas if $k \neq i$ (A3), beta reduction returns the unmodified value of array a at position k represented by $f_a(k)$. Hence, these axioms do not need to be explicitly checked during consistency checking. This is in essence the approach to handle arrays taken by UCLID [58].

We interpret *if-then-else* operations on arrays a and b as lambda terms, and represent them as $\text{ite}(c,\ a,\ b) \equiv \lambda j.\text{ite}(c,\ f_a(j),\ f_b(j))$. Condition c yields either function application $f_a(j)$ or $f_b(j)$, which represent the values of arrays a and b at index j, respectively.

In addition to the base array operations introduced above, lambda terms enable us to succinctly model array operations like e.g. *memcpy* and *memset* from the standard C library, which we previously were not able to efficiently express by means of *read*, *write* and *ite* operations on arrays. In particular, both *memcpy* and *memset* could only be represented by a fixed sequence of *read* and *write* operations within a constant index range, such as copying exactly 5 words etc. It was not possible to express a variable range, e.g. copying n words, where n is a symbolic (bit-vector) variable.

With lambda terms however, we do not require a sequence of array operations as it usually suffices to model a parallel array operation by means of exactly one lambda term. Further, the index range does not have to be fixed and can therefore be within a variable range. This type of high level modeling turned out to be useful for applications in software model checking [27]. See also [58] for more examples. For instance, the *memset* with signature $\text{memset}(a,\ i,\ n,\ e)$, which sets each element of array a within the range $[i, i+n[$ to value e, can be represented as $\lambda j.\text{ite}(i \leq j \wedge j < i + n,\ e,\ f_a(j))$. Note, n can be symbolic, and does not have to be a constant. In the same way, *memcpy* with signature $\mathit{memcpy}\ (a, b, i, k, n)$, which copies all elements of array a within the range $[i, i+n[$ to array b, starting from index k, is represented as $\lambda j.\text{ite}(k \leq j \wedge j < k + n,\ f_a(i + j - k),\ f_b(j))$. As a special case of *memset*, we represent *array initialization* operations, where all elements of an array are initialized with some (constant or symbolic) value e, as $\lambda j.e$.

Introducing lambda terms does not only enable us to model arrays and array operations, but further provides support for arbitrary functions (macros) by means of currying, with the following restrictions: (1) functions may not be recursive and (2) arguments to functions may not be functions. The first restriction enables keeping the implementation of lambda term handling in Boolector as simple as possible, whereas the second restriction limits lambda term handling in Boolector to non-higher order functions. Relaxing these restrictions will turn the considered lambda calculus to be Turing-complete and in general render the decision problem to be undecidable. As future work it might be interesting to consider some relaxations.

In contrast to treating SMT-LIBv2 macros as C-style macros, i.e, substituting

every function application with the instantiated function body, in Boolector, we directly translate SMT-LIBv2 macros into lambda terms, which are then handled lazily via lemmas on demand. Formulas are represented as directed acyclic graphs (DAG) of bit-vector and array expressions. Further, in this paper, we propose to treat arrays and array operations as lambda terms and operations on lambda terms, which results in an expression graph with no expressions of sort array ($\tau_i \Rightarrow \tau_e$). Instead, we introduce the following four additional expression types of sort bit-vector:

- a *param* expression serves as a placeholder variable for a variable bound by a lambda term
- a *lambda* expression binds exactly one *param* expression, which may occur in a bit-vector expression that represents the body of the lambda term
- an *args* expression is a list of function arguments
- an *apply* expression represents a function application that applies arguments *args* to a *lambda* expression

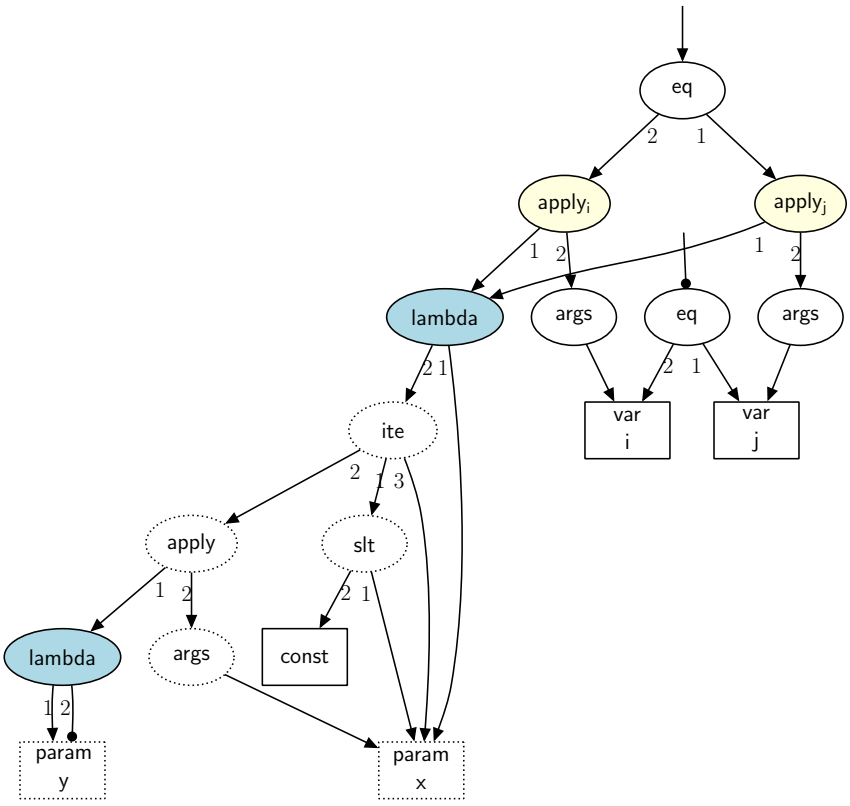


Figure 7.1: DAG representation of formula ψ_1.

Example 7.1. Consider $\psi_1 \equiv f(i) = f(j) \wedge i \neq j$ with functions $f(x) \coloneqq \text{ite}(x < 0,\, g(x),\, x)$, $g(y) \coloneqq -y$ as depicted in Figure 7.1. Both functions are represented as lambda terms, where function $g(y)$ returns the negation of y and is used in function $f(x)$, which computes the absolute value of x. Dotted nodes indicate parameterized expressions, i.e., expressions that depend on *param* expressions, and serve as templates that are instantiated as soon as beta reduction is applied.

In order to lazily evaluate lambda terms in Boolector we implemented two beta reduction approaches, which we will discuss in the next section in more detail.

7.4 Beta reduction

In this section we discuss how concepts from the lambda calculus have been adapted and implemented in our SMT solver Boolector. We focus on reduction algorithms for the non-recursive lambda calculus, which is rather atypical for the (vast) literature on lambda calculus. In the context of Boolector, we distinguish between *full* and *partial* beta reduction. They mainly differ in their application and the depth up to which lambda terms are expanded. In essence, given a function application $\lambda_{\bar{x}}(a_1, \ldots, a_n)$ *partial* beta reduction reduces only the top-most lambda term $\lambda_{\bar{x}}$, whereas *full* beta reduction reduces $\lambda_{\bar{x}}$ and every lambda term in the scope of $\lambda_{\bar{x}}$.

Full beta reduction of a function application on lambda term $\lambda_{\bar{x}}$ consists of a series of beta reductions, where lambda term $\lambda_{\bar{x}}$ and every lambda term $\lambda_{\bar{y}}$ within the scope of $\lambda_{\bar{x}}$ are instantiated, substituting all formal parameters with actual parameter terms. Since we do not allow partial function applications, full beta reduction guarantees to yield a term which is free of lambda terms. Given a formula with lambda terms, we usually employ full beta reduction in order to eliminate all lambda terms by substituting every function application with the term obtained by applying full beta reduction on that function application. In the worst case, full beta reduction results in an exponential blow-up. However, in practice, it is often beneficial to employ full beta reduction, since it usually leads to significant simplifications through rewriting. In Boolector, we incorporate this method as an optional rewriting step. We will use $\lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_f$ as a shorthand for applying full beta reduction to $\lambda_{\bar{x}}$ with arguments $a_1, \ldots, a_n$.

Partial beta reduction of a lambda term $\lambda_{\bar{x}}$, on the other hand, essentially works in the same way as what is referred to as beta reduction in the lambda calculus. Given a function application $\lambda_{\bar{x}}(a_1, \ldots, a_n)$, partial beta reduction substitutes formal parameters $x_1, \ldots, x_n$ with the actual argument terms $a_1, \ldots, a_n$ without applying beta reduction to other lambda terms within the scope of $\lambda_{\bar{x}}$. This has the effect that lambda terms are expanded function-wise, which we require for consistency checking. In the following, we use $\lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$ to denote the application of partial beta reduction to $\lambda_{\bar{x}}$ with arguments $a_1, \ldots, a_n$.

7.4.1 Full beta reduction

Given a function application $\lambda_{\bar{x}}(a_1, \ldots, a_n)$ and a DAG representation of $\lambda_{\bar{x}}$. Full beta reduction of $\lambda_{\bar{x}}$ consecutively substitutes formal parameters with actual argument terms while traversing and rebuilding the DAG in depth-first-search (DFS) post-order as follows.

1. Initially, we instantiate $\lambda_{\bar{x}}$ by assigning arguments $a_1, \ldots, a_n$ to the formal parameters $x_1, \ldots, x_n$.

2. While traversing down, for any lambda term $\lambda_{\bar{y}}$ in the scope of $\lambda_{\bar{x}}$, we need special handling for each function application $\lambda_{\bar{y}}(b_1, \ldots, b_m)$ as follows.

   a) Visit arguments $b_1, \ldots, b_m$ first, and obtain rebuilt arguments $b'_1, \ldots, b'_m$.

   b) Assign rebuilt arguments $b'_1, \ldots, b'_m$ to $\lambda_{\bar{y}}$ and apply beta reduction to $\lambda_{\bar{y}}(b'_1, \ldots, b'_m)$.

   This ensures a bottom-up construction of the beta reduced DAG (see step 3.), since all arguments $b'_1, \ldots, b'_m$ passed to a lambda term $\lambda_{\bar{y}}$ are beta reduced and rebuilt prior to applying beta reduction to $\lambda_{\bar{y}}$.

3. During up-traversal of the DAG we rebuild all visited expressions bottom-up and require special handling for the following expressions:

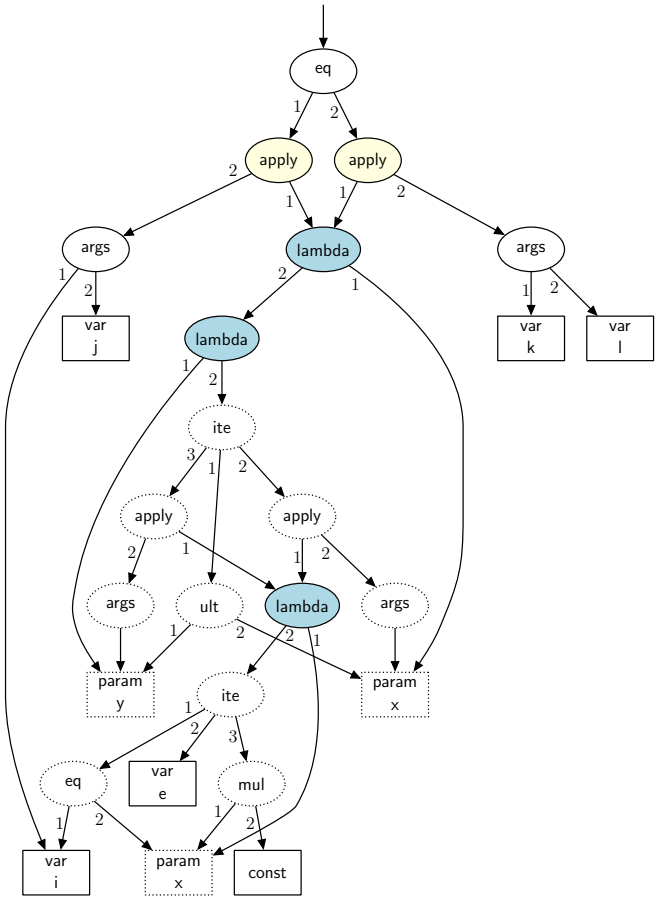
   - *param*: substitute *param* expression y_i with current instantiation b'_i
   - *apply*: substitute expression $\lambda_{\bar{y}}(b_1, \ldots, b_m)$ with $\lambda_{\bar{y}}[y_1 \backslash b'_1, \ldots, y_m \backslash b'_m]_f$

We further employ following optimizations to improve the performance of the full beta reduction algorithm.

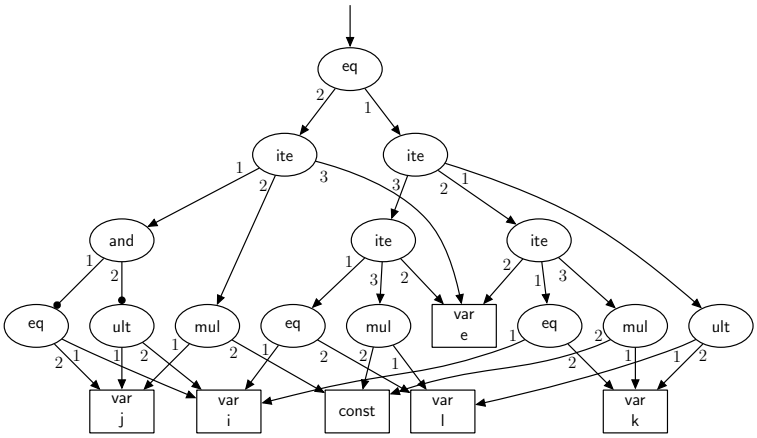
- *Skip expressions that do not need rebuilding*
  Given an expression e within the scope of a lambda term $\lambda_{\bar{x}}$. If e is not parameterized and does not contain any lambda term, e is not dependent on arguments passed to $\lambda_{\bar{x}}$ and may therefore be skipped.

- *Lambda scope caching*
  We cache rebuilt expressions in a lambda scope to prevent rebuilding parameterized expressions several times.

Example 7.2. Given a formula $\psi_2 \equiv f(i,j) = f(k,l)$ and two functions $g(x) \coloneqq \text{ite}(x = i,\ e,\ 2{*}x)$ and $f(x,y) \coloneqq \text{ite}(y < x,\ g(x),\ g(y))$ as depicted in Figure 7.2a. Applying full beta reduction to formula ψ_2 yields formula ψ'_2 as illustrated in Figure 7.2b. Function application $f(i,j)$ has been reduced to $\text{ite}(j \geq i \wedge i \neq j,\ 2 * j,\ e)$ and $f(k,l)$ to $\text{ite}(l < k,\ \text{ite}(k = i,\ e,\ 2 * k),\ \text{ite}(l = i,\ e,\ 2 * l))$.



(a) Original formula ϕ_2.



(b) Formula ψ_2' after full beta reduction of ψ_2.

Figure 7.2: Full beta reduction of formula ψ_2.

7.4.2 Partial beta reduction

Given a function application $\lambda_{\bar{x}}(a_1, \ldots, a_n)$ and a DAG representation of $\lambda_{\bar{x}}$. The scope of a partial beta reduction $\beta_p(\lambda_{\bar{x}})$ is defined as the sub-DAG obtained by cutting off all lambda terms in the scope of $\lambda_{\bar{x}}$. Assume that we have an assignment for arguments $a_1, \ldots, a_n$, and for all non-parameterized expressions in the scope of $\beta_p(\lambda_{\bar{x}})$. The partial beta reduction algorithm substitutes *param* expressions $x_1, \ldots, x_n$ with $a_1, \ldots, a_n$ and rebuilds $\lambda_{\bar{x}}$. Similar to full beta reduction, we perform a DFS post-order traversal of the DAG as follows.

1. Initially, we instantiate $\lambda_{\bar{x}}$ by assigning arguments $a_1, \ldots, a_n$ to the formal parameters $x_1, \ldots, x_n$.

2. While traversing down the DAG, we require special handling for the following expressions:
   - function applications $\lambda_{\bar{y}}(b_1, \ldots, b_m)$
     a) Visit arguments $b_1, \ldots, b_m$, obtain rebuilt arguments $b'_1, \ldots, b'_m$.
     b) Do not assign rebuilt arguments $b'_1, \ldots, b'_m$ to $\lambda_{\bar{y}}$ and stop down-traversal at $\lambda_{\bar{y}}$.
   - ite(c, t_1, t_2)
     Since we have an assignment for all non-parameterized expressions within the scope of $\beta_p(\lambda_{\bar{x}})$, we are able to evaluate condition c. Based on that we either select t_1 or t_2 to further traverse down (the other branch is omitted).

3. During up-traversal of the DAG we rebuild all visited expressions bottom-up and require special handling for the following expressions:
   - *param*: substitute *param* expression y_i with current instantiation b'_i
   - *if-then-else*: substitute expression ite(c, t_1, t_2) with t_1 if $c = \top$, and t_2 otherwise

For partial beta reduction, we have to modify the first of the two optimizations introduced for full beta reduction.

- *Skip expressions that do not need rebuilding*
  Given an expression e in the scope of partial beta reduction $\beta_p(\lambda_{\bar{x}})$. If e is not parameterized, in the context of partial beta reduction, e is not dependent on arguments passed to $\lambda_{\bar{x}}$ and may be skipped.

Example 7.3. Consider ψ_2 from Ex. 7.2. Applying partial beta reduction to ψ_2 yields the formula depicted in Figure 7.3, where function application $f(i, j)$ has been reduced to ite$(j < i, e, g(j))$ and $f(k, l)$ to ite$(l < k, g(k), g(l))$.

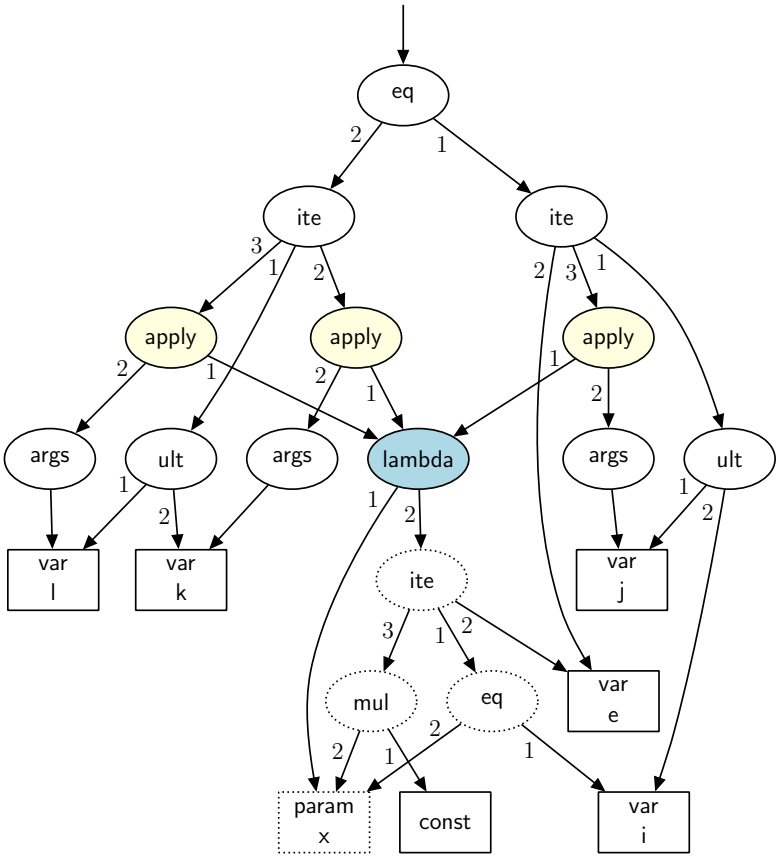


Figure 7.3: Partial beta reduction of formula ψ_2.

7.5 Decision Procedure

The idea of *lemmas on demand* goes back to [20] and actually represents one extreme variant of the lazy SMT approach [57]. Around the same time, a related technique was developed in the context of bounded model checking [26], which lazily encodes all-different constraints over bit-vectors (see also [6]). In constraint programming the related technique of lazy clause generation [49] is effective too.

In this section, we introduce lemmas on demand for non-recursive lambda terms based on the algorithm introduced in [11]. A top-level view of our lemmas on demand decision procedure for lambda terms (DP_λ) is illustrated in Figure 7.4 and proceeds as follows. Given a formula ϕ, DP_λ uses a bit-vector skeleton of the preprocessed formula π as formula abstraction $\alpha_\lambda(\pi)$. In each iteration, an underlying decision procedure DP_B determines the satisfiability of the formula abstraction refined by formula refinement ξ, i.e., in DP_B, we eagerly encode the refined formula abstraction Γ to SAT and determine its satisfiability by means of a SAT solver. As Γ is an over-approximation of ϕ, we immediately conclude

```
procedure DP_λ (φ)
  π := preprocess(φ)
  ξ := ⊤
  loop
    Γ := α_λ(π) ∧ ξ
    r, σ := DP_B(Γ)
    if r = unsatisfiable return unsatisfiable
    if consistent_λ(π, σ) return satisfiable
    ξ := ξ ∧ α_λ(lemma_λ(π, σ))
```

Figure 7.4: Lemmas on demand for lambda terms DP_λ.

with *unsatisfiable* if Γ is unsatisfiable. If Γ is satisfiable, we have to check if the current satisfying assignment σ (as provided by procedure DP_B) is consistent w.r.t. preprocessed formula π. If σ is consistent, i.e., if it can be extended to a valid satisfying assignment for the preprocessed formula π, we immediately conclude with *satisfiable*. Otherwise, assignment σ is spurious, $\mathrm{consistent}_\lambda(\pi, \sigma)$ identifies a violation of the function congruence axiom EUF, and we generate a symbolic lemma $\mathrm{lemma}_\lambda(\pi, \sigma)$, which is added to formula refinement ξ in its abstracted form $\alpha_\lambda(\mathrm{lemma}_\lambda(\pi, \sigma))$.

Note that in ϕ, in contrast to the decision procedure introduced in [11], all array variables and array operations in the original input have been abstracted away and replaced by corresponding lambda terms and operations on lambda terms. Hence, various integral components of the original procedure (α_λ, $\mathrm{consistent}_\lambda$, lemma_λ) have been adapted to handle lambda terms as follows.

7.6 Formula Abstraction

In this section, we introduce a partial formula abstraction function α_λ as a generalization of the abstraction approach presented in [11]. Analogous to [11], we replace function applications by fresh bit-vector variables and generate a bit-vector skeleton as formula abstraction. Given π as the preprocessed input formula ϕ, our abstraction function α_λ traverses down the DAG structure starting from the roots, and generates an over-approximation of π as follows.

1. Each bit-vector variable and symbolic constant is mapped to itself.
2. Each function application $\lambda_{\bar{x}}(a_1, \ldots, a_n)$ is mapped to a fresh bit-vector variable.
3. Each bit-vector term $t(y_1, \ldots, y_m)$ is mapped to $t(\alpha_\lambda(y_1), \ldots, \alpha_\lambda(y_m))$.

Note that by introducing fresh variables for function applications, we essentially cut off lambda terms and UF and therefore yield a pure bit-vector skeleton, which is subsequently eagerly encoded to SAT.

Example 7.4. Consider formula ψ_1 from Ex. 7.1, which has two roots. The abstraction function α_λ performs a consecutive down-traversal of the DAG from both roots. The resulting abstraction is a mapping of all bit-vector terms encountered during traversal, according to the rules 1-3 above. For function applications (e.g. $\mathsf{apply_i}$) fresh bit-vector variables (e.g. $\alpha_\lambda(\mathsf{apply_i})$) are introduced, where the remaining sub-DAGs are therefore cut off. The resulting abstraction $\alpha_\lambda(\psi_1)$ is given in Figure 7.5.

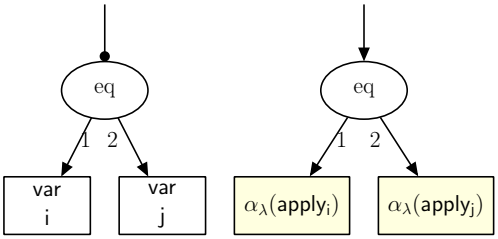


Figure 7.5: Formula abstraction $\alpha_\lambda(\psi_1)$.

7.7 Consistency Checking

In this section, we introduce a consistency checking algorithm consistent$_\lambda$ as a generalization of the consistency checking approach presented in [11]. However, in contrast to [11], we do not propagate so-called access nodes but function applications and further check axiom EUF (while applying partial beta reduction to evaluate function applications under a current assignment) instead of checking array axioms A1 and A2. Given a satisfiable over-approximated and refined formula Γ, procedure consistent$_\lambda$ determines whether a current satisfying assignment σ (as provided by the underlying decision procedure $\mathrm{DP_B}$) is spurious, or if it can be extended to a valid satisfying assignment for the preprocessed input formula π. Therefore, for each function application in π, we have to check both if the assignment of the corresponding abstraction variable is consistent with the value obtained by applying partial beta reduction, and if axiom EUF is violated. If consistent$_\lambda$ does not find any conflict, we immediately conclude that formula π is satisfiable. However, if current assignment σ is spurious w.r.t. preprocessed formula π, either axiom EUF is violated or partial beta reduction yields a conflicting value for some function application in π. In both cases, we generate a lemma as formula refinement. In the following we will equally use function symbols f, g, and h for UF symbols and lambda terms.

In order to check axiom EUF, for each lambda term and UF symbol we maintain a hash table ρ, which maps lambda terms and UF symbols to function applications. We check consistency w.r.t. π by applying the following rules.

I: For each $f(\bar{a})$, if $\bar{a}$ is not parameterized, add $f(\bar{a})$ to $\rho(f)$

C: For any pair $s \coloneqq g(\bar{a}),\ t \coloneqq h(\bar{b}) \in \rho(f)$ check
$\bigwedge_{i=1}^{n} \sigma(\alpha_\lambda(a_i)) = \sigma(\alpha_\lambda(b_i)) \rightarrow \sigma(\alpha_\lambda(s)) = \sigma(\alpha_\lambda(t))$

B: For any $s \coloneqq \lambda_{\bar{y}}(a_1, \ldots, a_n) \in \rho(\lambda_{\bar{x}})$ with $t \coloneqq \lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$, check rule P, if P fails, check $eval(t) = \sigma(\alpha_\lambda(s))$

P: For any $s \coloneqq \lambda_{\bar{y}}(a_1, \ldots, a_n) \in \rho(\lambda_{\bar{x}})$ with
$t \coloneqq g(b_1, \ldots, b_m) = \lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$,
if $n = m \ \wedge \bigwedge_{i=1}^{n} a_i = b_i$, propagate s to $\rho(g)$

Given a lambda term (UF symbol) f and a corresponding hash table $\rho(f)$. Rule I, the *initialization* rule, initializes $\rho(f)$ with all non-parameterized function applications on f. Rule C corresponds to the function congruence axiom and is applied whenever we add a function application $g(a_1, \ldots, a_n)$ to $\rho(f)$. Rule B is a consistency check w.r.t. the current assignment σ, i.e., for every function application s in $\rho(f)$, we check if the assignment of $\sigma(\alpha_\lambda(s))$ corresponds to the assignment evaluated by the partially beta reduced term $\lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$. Finally, rule P represents a crucial optimization of consistent$_\lambda$, as it avoids unnecessary conflicts while checking rule B. If rule P applies, both function applications s and t have the same arguments. As function application $s \in \rho(\lambda_{\bar{x}})$, rule C implies that $s = \lambda_{\bar{x}}(a_1, \ldots, a_n)$. Therefore, function applications s and t must produce the same function value as $t \coloneqq \lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p = \lambda_{\bar{y}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$, i.e., function application t must be equal to the result of applying partial beta reduction to function application s. Assume we encode t and add it to the formula. If $\mathrm{DP_B}$ guesses an assignment s.t. $\sigma(\alpha_\lambda(t)) \neq \sigma(\alpha_\lambda(s))$ holds, we have a conflict and need to add a lemma. However, this conflict is unnecessary, as we know from the start that both function applications must map to the same function value in order to be consistent. We avoid this conflict by propagating s to $\rho(g)$.

Figure 7.6 illustrates our consistency checking algorithm consistent$_\lambda$, which takes the preprocessed input formula π and a current assignment σ as arguments, and proceeds as follows. First, we initialize stack S with all non-parameterized function applications in formula π (cf. nonparam_apps(π)) and order them top-down, according to their appearance in the DAG representation of π. The top-most function application then represents the top of stack S, which consists of tuples $(g, f(a_1, \ldots, a_n))$, where f and g are initially equal and $f(a_1, \ldots, a_n)$

```
procedure consistent_λ (π, σ)
  S ≔ nonparam_apps(π)
  while S ≠ ∅
    g, f(a_1, ..., a_n) ≔ pop(S)
    encode(f(a_1, ..., a_n))
    if not congruent(g, f(a_1, ..., a_n))      // check rule C
      return ⊥
    add(f(a_1, ..., a_n), ρ(g))
    if is_UF(g) continue
    encode(g)
    t ≔ g[x_1\a_1, ..., x_n\a_n]_p
    if assigned(t)
      if σ(t) ≠ σ(α_λ(f(a_1, ..., a_n)))       // check rule B
        return ⊥
    elif t = h(a_1, ..., a_n)                  // check rule P
      push(S, (h, f(a_1, ..., a_n)))
      continue
    else
      apps ≔ fresh_apps(t)
      for a ∈ apps
        encode(a)
      if eval(t) ≠ σ(α_λ(f(a_1, ..., a_n)))    // check rule B
        return ⊥
      for h(b_1, ..., b_m) ∈ apps
        push(S, (h, h(b_1, ..., b_m)))
  return ⊤
```

Figure 7.6: Procedure consistent$_\lambda$ in pseudo-code.

denotes the function application propagated to function g. In the main consistency checking loop, we check rules C and B for each tuple as follows. First we check if $f(a_1, \ldots, a_n)$ violates the function congruence axiom EUF w.r.t. function g and return $\bot$ if this is the case. Note that for checking rule C, we require an assignment for arguments $a_1, \ldots, a_n$, hence we encode them on-the-fly. If rule C is not violated and function f is an uninterpreted function, we continue to check the next tuple on stack S. However, if f is a lambda term we still need to check rule B, i.e., we need to check if the assignment $\sigma(\alpha_\lambda(f(a_1, \ldots, a_n)))$ is consistent with the value produced by $g[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$. Therefore, we first encode all non-parameterized expressions in the scope of partial beta reduction $\beta_p(g)$ (cf. encode(g)) before applying partial beta reduction with arguments $a_1, \ldots, a_n$, which yields term t. If term t has an assignment, we can immediately check if it differs from assignment $\sigma(\alpha_\lambda(f(a_1, \ldots, a_n)))$ and return $\bot$ if this is the case. However, if term t does not have an assignment, which is the case when t has been instantiated from a parameterized expression, we have to compute the value for term t. Note that we could also encode term t to get an assignment $\sigma(t)$, but this might add a considerable amount of superfluous clauses to the SAT solver. Before computing a value for t we check if rule P applies and propagate $f(a_1, \ldots, a_n)$ to h if applicable. Otherwise, we need to compute a value for t and check if t contains any function applications that were instantiated and not yet encoded (cf. fresh_apps(t)) and encode them if necessary. Finally, we compute the value for t (cf. eval(t)) and compare it to the assignment of $\alpha_\lambda(f(a_1, \ldots, a_n))$. If the values differ, we found an inconsistency and return $\bot$. Otherwise, we continue consistency checking the newly encoded function applications apps. We conclude with $\top$, if all function applications have been checked successfully and no inconsistencies have been found.

7.7.1 Lemma generation

Following [11], we introduce a lemma generation procedure lemma$_\lambda$, which generates a symbolic lemma whenever our consistency checker detects an inconsistency. Depending on whether rule C or B was violated, we generate a symbolic lemma as follows. Assume that rule C was violated by function applications $s \coloneqq g(a_1, \ldots, a_n),\ t \coloneqq h(b_1, \ldots, b_n) \in \rho(f)$. We first collect all conditions that lead to the conflict as follows.

1. Find the shortest possible propagation path p^s (p^t) from function application s (t) to function f.

2. Collect all *ite* conditions $c_1^s, \ldots, c_j^s$ ($c_1^t, \ldots, c_l^t$) on path p^s (p^t) that were $\top$ under given assignment σ.

3. Collect all *ite* conditions $c_1^s, \ldots, c_k^s$ ($c_1^t, \ldots, c_m^t$) on path p^s (p^t) that were $\bot$ under 1iven assignment σ.

We generate the following (in general symbolic) lemma:

$$\bigwedge_{i=1}^{j} c_i^s \ \wedge \bigwedge_{i=1}^{k} \neg c_i^s \ \wedge \bigwedge_{i=1}^{l} c_i^t \ \wedge \bigwedge_{i=1}^{m} \neg c_i^t \ \wedge \bigwedge_{i=1}^{n} a_i = b_i \rightarrow s = t$$

Assume that rule B was violated by a function application $s \coloneqq \lambda_{\bar{y}}(a_1, \ldots, a_n) \in \rho(\lambda_{\bar{x}})$. We obtained $t \coloneqq \lambda_{\bar{x}}[x_1 \backslash a_1, \ldots, x_n \backslash a_n]_p$ and collect all conditions that lead to the conflict as follows.

1. Collect *ite* conditions $c_1^s, \ldots, c_j^s$ and $c_1^s, \ldots, c_k^s$ for s as in steps 1-3 above.

2. Collect all *ite* conditions $c_1^t, \ldots, c_l^t$ that evaluated to $\top$ under current assignment σ when partially beta reducing $\lambda_{\bar{x}}$ to obtain t.

3. Collect all *ite* conditions $c_1^t, \ldots, c_m^t$ that evaluated to $\bot$ under current assignment σ when partially beta reducing $\lambda_{\bar{x}}$ to obtain t.

We generate the following (in general symbolic) lemma:

$$\bigwedge_{i=1}^{j} c_i^s \ \wedge \bigwedge_{i=1}^{k} \neg c_i^s \ \wedge \bigwedge_{i=1}^{l} c_i^t \ \wedge \bigwedge_{i=1}^{m} \neg c_i^t \rightarrow s = t$$

Example 7.5. Consider formula ψ_1 and its preprocessed formula abstraction $\alpha_\lambda(\psi_1)$ from Ex. 7.1. For the sake of better readability, we will use λ_x and λ_y to denote functions f and g, and further use a_i and a_j as a shorthand for $\alpha_\lambda(\mathsf{apply_i})$ and $\alpha_\lambda(\mathsf{apply_j})$. Assume we run $\mathrm{DP_B}$ on $\alpha_\lambda(\psi_1)$ and it returns a satisfying assignment σ such that $\sigma(i) \neq \sigma(j)$, $\sigma(a_i) = \sigma(a_j)$, $\sigma(i) < 0$ and $\sigma(a_i) \neq \sigma(-i)$. First, we check consistency for $\lambda_x(i)$ and check rule C, which is not violated as $\sigma(i) \neq \sigma(j)$, and continue with checking rule B. We apply partial beta reduction and obtain term $t \coloneqq \lambda_x[x/i]_{\mathbf{p}} = \lambda_y(i)$ (since $\sigma(i) < 0$) for which rule P is applicable. We propagate $\lambda_x(i)$ to λ_y, check if $\lambda_x(i)$ is consistent w.r.t. λ_y, apply partial beta reduction, obtain $t \coloneqq \lambda_y[y/i]_{\mathbf{p}} = -i$ and find an inconsistency according to rule B: $\sigma(a_i) \neq \sigma(-i)$ but we obtained $\sigma(a_i) = \sigma(-i)$. We generate lemma $i < 0 \rightarrow a_i = -i$. Assume that in the next iteration DB_P returns a new satisfying assignment σ such that $\sigma(i) \neq \sigma(j)$, $\sigma(a_i) = \sigma(a_j)$, $\sigma(i) < 0$, $\sigma(a_i) = \sigma(-i)$ and $\sigma(j) > \sigma(-i)$. We first check consistency for $\lambda_x(i)$, which is consistent due to the lemma we previously generated. Next, we check rule C for $\lambda_x(j)$, which is not violated since $\sigma(i) \neq \sigma(j)$, and continue with checking rule B. We apply partial beta reduction and obtain term $t \coloneqq \lambda_x[x/j]_{\mathbf{p}} = j$ (since $\sigma(j) > \sigma(-i)$ and $\sigma(i) < 0$) and find an inconsistency as $\sigma(a_i) = \sigma(-i)$, $\sigma(a_i) = \sigma(a_j)$ and $\sigma(j) > \sigma(-i)$, but $\sigma(a_j) = \sigma(j)$. We then generate lemma $j > 0 \rightarrow a_j = j$.

7.8 Experiments

We applied our lemmas on demand approach for lambda terms on three different benchmark categories: (1) *crafted*, (2) *SMT'12*, and (3) *application*. For the *crafted* category, we generated benchmarks using SMT-LIBv2 macros, where the instances of the first benchmark set (*macro blow-up*) tend to blow up in formula size if SMT-LIBv2 macros are treated as C-style macros. The benchmark sets *fisher-yates SAT* and *fisher-yates UNSAT* encode an incorrect and correct but naive implementation of the Fisher-Yates shuffle algorithm [30], where the instances of the *fisher-yates SAT* also tend to blow up in the size of the formula if SMT-LIBv2 macros are treated as C-style macros. The *SMT'12* category consists of all non-extensional QF_AUFBV benchmarks used in the SMT competition 2012. For the *application* category, we considered the *instantiation* benchmarks[1] generated with `LLBMC` as presented in [27]. The authors also kindly provided the same benchmark family using lambda terms as arrays, which is denoted as *lambda*.

We performed all experiments on 2.83GHz Intel Core 2 Quad machines with 8GB of memory running Ubuntu 12.04.2 setting a memory limit of 7GB and a time limit for the *crafted* and the *SMT'12* benchmarks of 1200 seconds. For the *application* benchmarks, as in [27] we used a time limit of 60 seconds. We evaluated four different versions of Boolector: (1) our lemmas on demand for lambda terms approach DP_λ (Btor), (2) DP_λ without optimization rule P (Btor-p), (3) DP_λ with full beta reduction (Btor+b), and (4) the version submitted to the SMT competition 2012 ($Btor_{sc12}$). For comparison we used the following SMT solvers: CVC4 1.2, MathSAT 5.2.6, SONOLAR 2013-05-15, STP 1673 (svn revision), and Z3 4.3.1. Note that we limited the set of solvers to those which currently support SMT-LIBv2 macros and the theory of fixed-size bit-vectors. As a consequence, we did not compare our approach to UCLID (no bit-vector support) and Yices, which both have native lambda term support, but lack support for the SMT-LIBv2 standard.

As indicated in Tables 7.1, 7.2 and 7.3, we measured the number of solved instances (Solved), timeouts (TO), memory outs (MO), total CPU time (Time), and total memory consumption (Space) required by each solver for solving an instance. If a solver ran into a timeout, 1200 seconds (60 seconds for category *application*) were added to the total time as a penalty. In case of a memory out, 1200 seconds (60 seconds for *application*) and 7GB were added to the total CPU time and total memory consumption, respectively.

Table 7.1 summarizes the results of the *crafted* benchmark category. On the *macro blow-up* benchmarks, Btor and Btor-p benefit from lazy lambda term handling and thus, outperform all those solvers which try to eagerly eliminate SMT-LIBv2 macros with a very high memory consumption as a result. The only solver not having memory problems on this benchmark set is SONOLAR.

[1] `http://llbmc.org/files/downloads/vstte-2013.tgz`

	macro blow-up (100)				
	Solved	TO	MO	Time [10^3s]	Space [GB]
Btor	100	0	0	24.2	9.4
Btor-p	**100**	**0**	**0**	**18.2**	**8.4**
Btor+b	28	49	23	91.5	160.0
CVC4	21	0	79	95.7	551.6
MathSAT	51	2	47	64.6	395.0
SONOLAR	26	74	0	90.2	1.7
Z3	21	0	79	95.0	552.2

	fisher-yates SAT (18)				
	Solved	TO	MO	Time [10^3s]	Space [GB]
Btor	**7**	**10**	**1**	**14.0**	**7.5**
Btor-p	4	13	1	17.3	7.0
Btor+b	6	1	11	15.0	76.4
CVC4	5	1	12	15.7	83.6
MathSAT	6	10	2	14.7	17.3
SONOLAR	3	14	1	18.1	6.9
Z3	6	12	0	14.8	0.2

	fisher-yates UNSAT (19)				
	Solved	TO	MO	Time [10^3s]	Space [GB]
Btor	5	13	1	17.4	7.1
Btor-p	4	14	1	18.2	6.9
Btor+b	9	0	10	12.1	72.0
CVC4	3	4	12	19.2	82.1
MathSAT	6	11	2	15.9	14.7
SONOLAR	3	15	1	19.2	6.8
Z3	**10**	**9**	**0**	**11.2**	**2.2**

Table 7.1: Results *crafted* benchmark.

	SMT'12 (149)				
	Solved	TO	MO	Time [10^3s]	Space [GB]
Btor	139	10	0	19.9	14.8
Btor-p	134	15	0	26.3	14.5
Btor+b	137	11	1	21.5	22.7
Btor$_{\mathbf{sc12}}$	**140**	**9**	**0**	**15.9**	**10.3**

Table 7.2: Results *SMT'12* benchmark.

However, it is not clear how SONOLAR handles SMT-LIBv2 macros. Surprisingly, on these benchmarks Btor-p performs better than Btor with optimization rule P, which needs further investigation. On the *fisher-yates SAT* benchmarks Btor not only solves the most instances, but requires 107 seconds for the first 6 instances, for which Btor+b, MathSAT and Z3 need more than 300 seconds each. Btor-p does not perform as well as Btor due to the fact that on these benchmarks optimization rule P is heavily applied. In fact, on these benchmarks, rule P applies to approx. 90% of all propagated function applications on average. On the *fisher-yates UNSAT* benchmarks Z3 and Btor+b solve the most instances, whereas Btor and Btor-p do not perform so well. This is mostly due to the fact that these benchmarks can be simplified significantly when macros are eagerly eliminated, whereas partial beta reduction does not yield as much simplifications. We measured overhead of beta reduction in Btor on these benchmarks and it turned out that for the *macro blow-up* and *fisher-yates UNSAT* instances the overhead is negligible (max. 3% of total runtime), whereas for the *fisher-yates SAT* instances beta reduction requires over 50% of total runtime.

Table 7.2 summarizes the results of running all four Btor versions on the *SMT'12* benchmark set. We compared our three approaches Btor, Btor-p, and Btor+b to Btor$_{\text{sc12}}$, which won the QF_AUFBV track in the SMT competition 2012. In comparison to Btor+b, Btor solves 5 unique instances, whereas Btor+b solves 3 unique instances. In comparison to Btor$_{\text{sc12}}$, both solvers combined solve 2 unique instances. Overall, on the *SMT'12* benchmarks Btor$_{\text{sc12}}$ still outperforms the other approaches. However, our results still look promising since none of the approaches Btor, Btor-p and Btor+b are heavily optimized yet. On these benchmarks, the overhead of beta reduction in Btor is around 7% of the total runtime.

Finally, Table 7.3 summarizes the results of the *application* category. We used the benchmarks obtained from the instantiation-based reduction approach presented in [27] (*instantiation* benchmarks) and compared our new approaches to STP, the same version of the solver that outperformed all other solvers on these benchmarks in the experimental evaluation of [27]. On the *instantiation* benchmarks Btor+b and STP solve the same number of instances in roughly the same time. However, Btor+b requires less memory for solving those instances.

	instantiation (45)				
	Solved	TO	MO	Time [s]	Space [MB]
Btor	37	8	0	576	235
Btor-p	35	10	0	673	196
Btor+b	**44**	**1**	**0**	**138**	**961**
Btor$_{\mathbf{sc12}}$	39	6	0	535	308
STP	44	1	0	141	3814

	lambda (45)				
	Solved	TO	MO	Time [s]	Space [MB]
Btor	37	8	0	594	236
Btor-p	35	10	0	709	166
Btor+b	**45**	**0**	**0**	**52**	**676**
Btor$_{\mathbf{sc12}}$	-	-	-	-	-
STP	-	-	-	-	-

Table 7.3: Results *application* benchmarks.

Btor, Btor-p and Btor$_{sc12}$ did not perform so well on these benchmarks because in contrast to Btor+b and STP, they do not eagerly eliminate read operations, which is beneficial on these benchmarks. The *lambda* benchmarks consist of the same problems as *instantiation*, using lambda terms for representing arrays. On these benchmarks, Btor+b clearly outperforms Btor and Btor-p and solves all 45 instances within a fraction of time. Btor$_{sc12}$ and STP do not support lambda terms as arrays and therefore were not able to participate on this benchmark set. By exploiting the native lambda term support for arrays in Btor+b, in comparison to the *instantiation* benchmarks we achieve even better results. Note that on the *lambda* (*instantiation*) benchmarks, the overhead in Btor+b for applying full beta reduction was around 15% (less than 2%) of the total runtime.

Benchmarks, binaries of Boolector and all log files of our experiments can be found at: `http://fmv.jku.at/difts-rev-13/lloddifts13.tar.gz`.

7.9 Conclusion

In this paper, we introduced a new decision procedure for handling non-recursive and non-extensional lambda terms as a generalization of the array decision procedure presented in [11]. We showed how arrays, array operations and SMT-LIBv2 macros are represented in Boolector and evaluated our new approach with 3 dif-

ferent benchmark categories: *crafted*, *SMT'12* and *application*. The *crafted* category showed the benefit of lazily handling SMT-LIBv2 macros where eager macro elimination tends to blow-up the formula in size. We further compared our new implementation to the version of Boolector that won the QF_AUFBV track in the SMT competition 2012. With the *application* benchmarks, we demonstrated the potential of native lambda term support within an SMT solver. Our experiments look promising even though we employ a rather naive implementation of beta reduction in Boolector and also do not incorporate any lambda term specific rewriting rules except full beta reduction.

In future work we will address the performance bottleneck of the beta reduction implementation and will further add lambda term specific rewriting rules. We will analyze the impact of various beta reduction strategies on our lemmas on demand procedure and will further add support for extensionality over lambda terms. Finally, with the recent and ongoing discussion within the SMT-LIB community to add support for recursive functions, we consider extending our approach to recursive lambda terms.

7.10 Acknowledgements

We would like to thank Stephan Falke, Florian Merz and Carsten Sinz for sharing benchmarks and Bruno Duterte for explaining the implementation and limits of lambdas in SMT solvers, and more specifically in Yices.

B

Chapter 8

Paper B. Better Lemmas with Lambda Extraction

Published In Proceedings of the 15th International Conference on Formal Methods in Computer Aided Design (FMCAD 2015), pages 128–135, Austin, TX, USA, 2015.

Authors Mathias Preiner, Aina Niemetz and Armin Biere.

Abstract In Satisfiability Modulo Theories (SMT), the theory of arrays provides operations to access and modify an array at a given index, e.g., *read* and *write*. However, common operations to modify multiple indices at once, e.g., *memset* or *memcpy* of the standard C library, are not supported. We describe algorithms to identify and extract array patterns representing such operations, including *memset* and *memcpy*. We represent these patterns in our SMT solver Boolector by means of compact and succinct lambda terms, which yields better lemmas and increases overall performance. We describe how extraction and merging of lambda terms affects lemma generation, and provide an extensive experimental evaluation of the presented techniques. It shows a considerable improvement in terms of solver performance, particularly on instances from symbolic execution.

8.1 Introduction

The theory of arrays, which for instance has been axiomatized by McCarthy [43], enables reasoning about "memory" in both software and hardware verification. It provides two operations *read* and *write* for accessing and modifying arrays on *single* array indices. While these two operations can be used to capture many aspects of modeling memory, they are not sufficient to succinctly encode array operations over *multiple* indices or a range of indices, e.g., *memset* or *memcpy* from the standard C library. Such array operations can therefore only be represented verbosely by means of a constant number of read and write operations. It is further impossible to reason about a variable number of indices e.g., a memset operation of variable size (without introducing quantifiers).

To overcome these limitations, Seshia et. al. [12] introduced an approach to model arrays by means of restricted lambda terms. This also enabled their SMT solver UCLID [58] to reason about ordered data structures and partially interpreted functions. However, UCLID employs the eager SMT approach and thus eliminates all lambda terms as a rewriting step prior to bit-blasting the formula to SAT, which might result in an exponential blow-up in the size of the formula [58].

An extension to the theory of arrays by Sinz et.al. [27] uses lambda terms similarly to UCLID in order to model memset and memcpy operations as well as loop summarizations, which in essence are initialization loops for arrays. As UCLID, this approach suffers from the problem of exponential explosion through eager lambda elimination.

To avoid exponential lambda elimination, in [51] we introduced a new decision procedure, which lazily handles non-recursive and non-extensional lambda terms. That decision procedure enabled us to succinctly represent array operations such as memset and memcpy as well as other array initialization patterns by means of lambda terms within our SMT solver Boolector. Lambda terms also allow to reason about variable ranges of indices without the need for quantifiers.

In this paper, we continue this thread of research and describe various patterns of operations on arrays occurring in benchmarks from SMT-LIB [4]. We provide algorithms to identify these patterns, and to extract succinct lambda terms from them. Extraction leads to stronger, as well as fewer lemmas. This improves performance by orders of magnitude on certain benchmarks, particularly on instances from symbolic execution [13]. We further describe a technique called *lambda merging*. Our extensive experimental evaluation shows that both techniques considerably improve the performance of Boolector, the winner of the QF_ABV track of the SMT competition 2014.

8.2 Preliminaries

We assume the usual notions and terminology of first-order logic and are mainly interested in many-sorted languages, where bit vectors of different bit width correspond to different sorts, and array sorts correspond to a mapping $(\tau_i \Rightarrow \tau_e)$ from index sort τ_i to element sort τ_e. We primarily focus on the *quantifier-free* theories of *fixed size bit vectors* and *arrays*. However, our approach is not restricted to the above.

In general, we refer to 0-arity function symbols as *constant* symbols. Symbols a, b, i, j, and e denote constants, where a and b are used for array constants, i and j for array indices, and e for an array element. We denote an *if-then-else* over bit vector terms with condition c, then branch t_1, and else branch t_2 as $\text{ite}(c, t_1, t_2)$, which is interpreted as $\text{ite}(\top, t_1, t_2) = t_1$ and $\text{ite}(\bot, t_1, t_2) = t_2$. We identify operations *read* and *write* as basic array operations (cf. *select* and *store* in SMT-LIBv2 notation) for accessing and modifying arrays. A *read* operation $\text{read}(a, i)$ denotes the element of array a at index i, whereas a *write* operation $\text{write}(a, i, e)$ represents the modified array a with element e written to index i. The non-extensional theory of arrays is axiomatized by the following axioms originally introduced by McCarthy in [43]:

$$i = j \rightarrow \text{read}(a, i) = \text{read}(a, j) \tag{A1}$$

$$i = j \rightarrow \text{read}(\text{write}(a, i, e), j) = e \tag{A2}$$

$$i \neq j \rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j) \tag{A3}$$

Axiom A1 asserts that accessing array a at two indices that are equal always yields the same element. Axiom A2 asserts that accessing a modified array on the updated index i yields the written element e, whereas axiom A3 ensures that the unmodified element of the original array a at index j is returned if the modified index i is not accessed.

A *write sequence* of n (consecutive) write operations of the form $a_1 = \text{write}(a_0, i_1, e_1), \ldots, a_n = \text{write}(a_{n-1}, i_n, e_n)$ is denoted as $(a_k \coloneqq \text{write}(a_{k-1}, i_k, e_k))_{k=1}^{n}$ with array a_0 as the *base array* of the write sequence. In the following we use $a_n = \text{write}(a, \bar{i}, \bar{e})$ as shorthand for write sequences.

In [51] we use *uninterpreted functions* (UF) and *lambda* terms to represent array variables and array operations, respectively. Consequently, a read on an array of sort $\tau_i \Rightarrow \tau_e$ is represented as a function application $f(i)$ on either an UF f or a lambda term $f \coloneqq \lambda j.t$, where function f maps terms of sort τ_i to terms of sort τ_e. Furthermore, write operations $\text{write}(a, i, e)$ are represented as lambda terms $\lambda j.\text{ite}(i = j, e, a(j))$, where given an array a, a function application yields element e if j is equal to the modified index i and the unchanged element $a(j)$, otherwise. Lambda terms allow us to succinctly model array operations such as *memset* and *memcpy* from the standard C library, or arrays initialized with a

constant value. For example, memset with signature memset(a, i, n, e), which sets each element of array a to e within the range $[i, i+n[$, can be represented as $\lambda j.\text{ite}(i \leq j < i+n,\ e,\ a(j))$. In this paper, we use read operations and function applications interchangeably.

8.3 Extracting Lambdas

Currently, the SMT-LIBv2 standard only supports write operations for modifying the contents of an array at one index at a time. Hence, quasi-parallel array operations like memset or memcpy usually have to be represented as a *fixed* sequence of consecutive write operations, where copying or setting n indices always requires n write operations. Further, modeling such array operations with a *variable* range is not possible (without quantifiers), since it would require a variable number of write operations. Lambda terms, however, provide means to succinctly represent parallel array operations, and further allow to model these operations with *variable* ranges. For example, modeling memset(a, i, n, e) with a sequence of writes for some fixed n produces n nested write operations write(write(...(write(a, i, e), $i+1$, e)...), $i+n-1$, e) which could be represented in a more compact way by means of a *single* lambda term $\lambda j.\text{ite}(i \leq j < i+n,\ e,\ a(j))$.

In the following, we describe several array operation patterns we identified by analyzing QF_ABV benchmarks in the SMT-LIB benchmark library. These patterns can not be captured compactly by means of write and read operations alone, but they can be succinctly represented using lambda terms. For each pattern identified in a formula, lambda terms are extracted and used instead of the original array operations, which are defined as follows.

8.3.1 Memset Pattern

The probably most common pattern is the *memset pattern* modeling the memset(a, i, n, e) operation, which updates n elements of array a within range $[i, i+n[$ to a value e starting from address i. This is the pattern already described above, and it is represented by the lambda term

$$\lambda_{mset} \coloneqq \lambda j.\text{ite}(i \leq j < i+n,\ e,\ a(j)).$$

Lambda term λ_{mset} yields value e if index j is within the range $[i, i+n[$, and the unmodified value from array a at position j otherwise. Note that in actual benchmarks, e.g., those from SMT-LIB, the upper bound n is constant, while indices, as well as values are usually symbolic.

8.3.2 Memcpy Pattern

The *memcpy pattern* models the memcpy(a, b, i, k, n) operation, which copies n elements from source array a starting at address i to destination array b at

address k. If arrays a and b are syntactically distinct, or if the source and destination addresses do not overlap, i.e., $(i + n < k)$ or $(k + n < i)$, *memcpy* can be represented as

$$\lambda_{mcpy} \coloneqq \lambda j.\text{ite}(k \leq j < k + n,\, a(i + j - k),\, b(j)).$$

Lambda term λ_{mcpy} returns the value copied from source array a if it is accessed within the copied range $[k, k + n[$, and the value from destination array b at position j otherwise.

Assume arrays a and b are syntactically equal, then aliasing occurs. Writing to array b at overlapping memory regions modifies elements in a to be copied to the destination address. This is not captured by lambda term λ_{mcpy}, since λ_{mcpy} behaves like a *memmove* operation. It ensures that elements of a at the overlapping memory region are copied before being overwritten. The following lambda term λ_{mcpyo} can be used to model *memcpy* applied to potentially overlapping memory regions.

$$\begin{aligned}\lambda_{mcpyo} \coloneqq \lambda j.\text{ite}(&k \leq j < k + n,\\ &\text{ite}(i \leq k < i + n,\\ &\quad a(i + ((j - k) \bmod (k - i))),\\ &\quad a(i + j - k)),\\ &b(j)).\end{aligned}$$

If condition $i \leq k < i + n$ holds, source and destination memory regions overlap and consequently, the elements of the overlapping memory region always contain the repeated sequence of the elements of array a in range $[i, k[$. This corresponds to the value $a(i + (j - k) \bmod (k - i))$, where $k - i$ represents the size of the non-overlapping memory region and thus, the number of elements that occur repeatedly. If the memory regions do not overlap, the behavior of lambda terms λ_{mcpyo} and λ_{mcpy} is equivalent. For the rest of this paper, we focus on *memcpy* with non-overlapping memory regions.

8.3.3 Loop Initialization Pattern

The *loop initialization pattern* models array initialization operations that can be expressed with the following loop

$$\textbf{for}\ (j = i;\, j < i + n;\, j = j + inc)\ \{a[j] = e;\},$$

where, starting from index i, the loop counter is incremented by a constant inc greater than one. Consequently, every inc-th element of an array a is modified within the range $[i, i + n[$. The above loop pattern corresponds to the lambda term

$$\lambda_{i \to e} \coloneqq \lambda j.\text{ite}(i \leq j \wedge j < i + n\ \wedge (inc \mid (j - i)),\, e,\, a(j)).$$

The memset pattern is actually a special case of this pattern with $inc = 1$. Further, the divisibility condition $inc \mid (j - i)$ makes sure that there exists a c such that index $j = i + c \cdot inc$ or equivalently $((j - i) \bmod inc = 0)$.

It is also possible that the value written on an index i depends on i itself. We found two such patterns in benchmarks. They can be expressed with the following loops

$$\textbf{for } (j = i;\ j < i + n;\ j = j + inc)\ \{a[j] = j\},$$

$$\textbf{for } (j = i;\ j < i + n;\ j = j + inc)\ \{a[j] = j + 1\}$$

or equivalently with the following lambda terms

$$\lambda_{i \to i} \coloneqq \lambda j.\mathrm{ite}(i \leq j \wedge j < i + n\ \wedge (inc \mid (j - i)),\ j,\ \ a(j))$$

$$\lambda_{i \to i+1} \coloneqq \lambda j.\mathrm{ite}(i \leq j \wedge j < i + n\ \wedge (inc \mid (j - i)),\ j + 1,\ \ a(j)).$$

Note that with $inc = 1$, the condition $inc \mid (j - i)$ is redundant and can be omitted. Further, this set of patterns is of course just a subset of all possible structures in benchmarks for which lambdas can be extracted. The ones discussed in this paper are those that we observed in actual benchmarks, and which turn out to be useful in our experiments.

8.3.4 Lemma Generation

Extracting lambda terms from write sequences does not only yield more compact array representations but improves the lemmas generated during search. As an example, consider a memset operation with range $[i, i + n[$ and value e, which is represented as a sequence of write operations

$$b \coloneqq \mathrm{write}(\mathrm{write}(\ldots(\mathrm{write}(a,\ i,\ e),\ i + 1,\ e)\ldots),\ i + n - 1,\ e).$$

A read operation on array b at index j may produce a conflict on index i, where $\mathrm{read}(b,\ j) \neq e$. As a consequence, the following lemma is generated.

$$(\bigwedge_{k=1}^{n-1} j \neq i + k) \wedge j = i \to \mathrm{read}(b,\ j) = e$$

In the worst case, this might be repeated for all the indices $i + k$ with $k \in [1, n[$, which also results in n lemmas of the above form. However, if we use a lambda term to represent memset, then a conflict produces a single lemma of the form

$$i \leq j \wedge j < i + n \to \mathrm{read}(b,\ j) = e,$$

which is more succinct and stronger as it covers an index range instead of single indices. This effect can be observed in our experiments in Sect. 8.4.2 as well. If applicable, the number of generated lemmas is reduced. This improves runtime and more instances are solved.

```
1  procedure extract_lambdas (φ)
2    for write sequence a_n := write(a, ī, ē) ∈ φ
3      and i_1, ..., i_n are distinct
4      ρ_{i→e} := index_value_map(a_n)
5      p_set := find_mset_patterns(ρ_{i→e})
6      p_cpy := find_mcopy_patterns(ρ_{i→e})
7      p_loop := find_lp_patterns(ρ_{i→e})
8      b := a_0
9      for p ∈ p_set
10       b := mk_memset(b, p.i, p.n, p.e)
11     for p ∈ p_cpy
12       b := mk_memcopy(p.a, b, p.i, p.k, p.n)
13     for p ∈ p_loop
14       b := mk_loop_init(b, p.i, p.n, p.inc)
15     for (i, e) ∈ ρ_{i→e}
16       b := mk_write(b, i, e)
17     φ := φ[a_n/b]
```

Figure 8.1: Main lambda extraction algorithm in pseudo-code.

8.3.5 Algorithms

Figure 8.1 depicts the main *lambda extraction* algorithm extract_lambdas. The purpose of this procedure is to initially identify and extract array patterns from each sequence of write operations in formula ϕ (lines 5-7). The identified patterns are then used to create lambda terms on top of each other resulting in a new lambda term b, which is equisatisfiable to the original write sequence a_n (lines 8-16), and is used to substitute a_n in ϕ. Figures 8.2, 8.3, and 8.4 depict the algorithms for identifying and extracting the actual array patterns. In essence, they all can be split into the following three steps. Given a sequence of write operations,

1. group *write indices* w.r.t. the corresponding pattern,
2. identify *index sequences* in these grouped write indices,
3. and create new pattern for identified *index sequence*.

In the following we describe the algorithms for identifying and extracting array patterns in more detail. A high level view of the main lambda extraction algorithm extract_lambdas is given in Figure 8.1. Given a formula ϕ,

```
1  procedure find_mset_patterns (ρ_{i→e})
2    patterns := [], ρ_{e→i} := {}
3    for (index, value) ∈ ρ_{i→e}
4      ρ_{e→i}[value].add(index)
5    for (value, indices) ∈ ρ_{e→i}
6      indices := sort(indices)
7    l := 0, u := 0
8    while u < len(indices)
9      while u + 1 < len(indices) and indices[u + 1] − indices[u] = 1
10       u += 1
11       if l ≠ u
12         Pattern p
13         p.i := indices[l]
14         p.n := indices[u] − indices[l] + 1
15         p.e := value
16         patterns.add(p)
17         ρ_{i→e} := ρ_{i→e} \ {indices[i] | ∀i ∈ [l, u]}
18         l := u + 1            // next sequence
19       u += 1
20   return patterns
```

Figure 8.2: Memset pattern extraction algorithm in pseudo-code.

for any write sequence $a_n = \text{write}(a, \bar{i}, \bar{e})$ with distinct indices $i_1, \ldots, i_n$, extract_lambdas initially generates a map $\rho_{i \to e}$, which maps indices $i_1, \ldots, i_n$ to values $e_1, \ldots, e_n$ (line 4), and is then used to extract memset (p_{set}), memcpy (p_{cpy}), and loop initialization patterns (p_{loop}) (lines 5-7). Note that procedures find_mset_patterns, find_mcopy_patterns, and find_lp_patterns remove all index/value pairs included in extracted patterns from $\rho_{i \to e}$. As a consequence, at line 8, map $\rho_{i \to e}$ contains all index/value pairs for which no pattern was extracted. The actual memset, memcpy, and loop initialization lambda terms are then created on top of each other with base array a_0 of write sequence a_n as the initial base array (lines 8-14). For the remaining index/value pairs in $\rho_{i \to e}$, lambda terms representing write operations are created on top of the previously generated lambda terms, and the resulting term b is then used to substitute the original write sequence a_n. Note that indices $i_1, \ldots, i_n$ are required to be distinct constants (line 3) as otherwise, reordering write sequence a_n does not result in an equisatisfiable sequence. As an example, assume in-

```
1  procedure find_mcopy_patterns (ρ_{i→e})
2    patterns := [], offset_groups := {}
3    for (index, value) ∈ ρ_{i→e} and index = dst + o and value = a(src + o)
4      offset_groups[dst, a, src].add(o)
5      for (dst, a, src) ∈ offset_groups
6        indices := sort(offset_groups[dst, a, src])
7        u := 0, l := 0
8        while u < len(indices)
9          while u + 1 < len(indices)
10           and indices[u + 1] − indices[u] = 1
11           u += 1
12           if l ≠ u
13             Pattern p
14             p.a := a
15             p.i := src + indices[l]
16             p.k := dst + indices[l]
17             p.n := indices[u] − indices[l] + 1
18             patterns.add(p)
19             ρ_{i→e} := ρ_{i→e} \ {indices[i] | ∀i ∈ [l, u]}
20             l := u + 1              // next sequence
21         u += 1
22   return patterns
```

Figure 8.3: Memcopy pattern extraction algorithm in pseudo-code.

dices i and j are equal and values e_i and e_j are distinct. Accessing sequence $a_{ij} \coloneqq \text{write}(\text{write}(a,\, i,\, e_i),\, j,\, e_j)$ at index j yields value e_j. However, accessing sequence $a_{ji} \coloneqq \text{write}(\text{write}(a,\, j,\, e_j),\, i,\, e_i)$ at index j yields e_i since $i = j$. Thus, a_{ji} is not equisatisfiable to a_{ij}.

Figures 8.2, 8.3, and 8.4 illustrate the algorithms for the actual pattern extraction, which we describe in more detail in the following. Procedure find_mset_patterns as in Figure 8.2 extracts *memset* patterns, i.e., in essence, it identifies index sequences that map to the same value. Given map $\rho_{i \to e}$, the procedure initially generates a reverse map $\rho_{e \to i}$, which maps values to indices and therefore groups indices that map to the same value (lines 3-4). For each index group indices in $\rho_{e \to i}$, find_mset_patterns sorts indices in ascending order (line 6) and identifies index sequences $s \coloneqq (i_k)_{k=l}^{u}$ with $i_k \coloneqq i_{k-1} + 1$ within

lower bound l ($i_l \coloneqq \text{indices}[l]$) and upper bound u ($i_u \coloneqq \text{indices}[u]$) (lines 7-18). If sequence s includes at least two indices (i.e., $u \neq l$), a new memset pattern p with start address $p.i$, size $p.n$ and value $p.e$ is created and added to list patterns (lines 12-16). All indices included in sequence s are removed from map $\rho_{i\to e}$ (line 17), since these indices are covered by a detected pattern. If all index groups have been processed, procedure find_mset_patterns returns the list of detected memset patterns.

Figure 8.3 illustrates procedure find_mcopy_patterns for extracting *memcpy* patterns. Assume that write operation write(b, $dst + o$, $a(src + o)$) represents a single memcpy operation memcpy(a, b, src, dst, n) with offset o and $src \leq o < src + n$, which copies one element from source address $src + o$ of array a to destination address $dst + o$ of array b. Consequently, $\rho_{i\to e}$ maps indices of the form $dst + o$ to values of the form $a(src + o)$. Initially, the procedure collects all offsets o from the indices in $\rho_{i\to e}$ and groups them by destination address dst, source array a, and source address src (lines 3-4). Note that a group of offsets corresponds to the memory regions copied from source address of array a to destination address of array b. For each offset group indices in offset_groups, find_mcopy_patterns identifies index sequences $s \coloneqq (i_k)_{k=l}^{u}$ similar to procedure find_mset_patterns (lines 9-11). If a sequence with at least two indices is found, a new memcpy pattern with source array $p.a$, source address $p.i$, destination address $p.k$, and size $p.n$ is created and added to the patterns list (lines 13-18). As for find_mset_patterns, indices included in a sequence s are removed from $\rho_{i\to e}$ (line 19). If all offset groups have been processed, procedure find_mcopy_patterns returns the list of detected memcpy patterns.

Figure 8.4 illustrates procedure find_lp_patterns for extracting *loop initialization* patterns. Initially, all indices in map $\rho_{i\to e}$ are categorized w.r.t. the three loop initialization patterns defined above, which correspond to the map $\rho_{e\to i}$, and the lists $\rho_{i\to i}$ and $\rho_{i\to i+1}$. Map $\rho_{e\to i}$ groups indices that map to the same value, list $\rho_{i\to i}$ contains indices that map to themselves, and list $\rho_{i\to i+1}$ contains all indices i that map to $i + 1$ (lines 4-9). For index groups $\rho_{i\to i+1}$ and $\rho_{i\to i+1}$, and for each index group in $\rho_{e\to i}$, procedure find_lpp_aux identifies sequences $s \coloneqq (i_k)_{k=l}^{u}$ with $i_k \coloneqq i_{k-1} + inc$ and $inc \geq 1$ within lower bound l ($i_l = \text{indices}[l]$) and upper bound u ($i_u \coloneqq \text{indices}[u]$) (lines 10-13). Identifying index sequences in find_lpp_aux is similar to find_mset_patterns, except that increment inc can be greater than one. For each sequence, inc is initially set to $\text{indices}[u+1] - \text{indices}[u]$ (lines 19-20), which defines the increment value between neighbouring indices, e.g., $(l, l + inc, l + 2 \cdot inc, l + 3 \cdot inc, \ldots, u)$. If a sequence with at least two indices is found, a new *loop initialization* pattern with lower bound $p.i$, size $p.n$, and increment $p.inc$ is created and added to the *patterns* list. Index sequences found in $\rho_{e\to i}$ correspond to $\lambda_{i\to e}$ patterns. These require a $p.e$ value, which is saved in addition (but remains unused for sequences $\rho_{i\to i}$ and $\rho_{i\to i+1}$). As before, indices included in a detected sequence s are removed from map $\rho_{i\to e}$ (line 30). If index group indices has been processed, procedure find_lpp_aux returns the list of detected loop initialization patterns.

```
1  procedure find_lp_patterns (ρ_{i→e})
2    patterns := [], ρ_{e→i} := {}
3    ρ_{i→i} := [], ρ_{i→i+1} := []
4    for (index, value) ∈ ρ_{i→e}
5      ρ_{e→i}[value].add(index)
6      if value = index
7        ρ_{i→i}.add(index)
8      elif value = index + 1
9        ρ_{i→i+1}.add(index)
10   for (value, indices) ∈ ρ_{e→i}
11     patterns.add(find_lpp_aux(ρ_{i→e}, ρ_{e→i}))
12   patterns.add(find_lpp_aux(ρ_{i→e}, ρ_{i→i}))
13   patterns.add(find_lpp_aux(ρ_{i→e}, ρ_{i→i+1}))
14   return patterns

15 procedure find_lpp_aux (ρ_{i→e}, indices)
16   patterns := [], indices := sort(indices)
17   l := 0, u := 0
18   while u < len(indices)
19     if u + 1 < len(indices)
20       inc := indices[u + 1] − indices[u]
21     while u + 1 < len(indices) and indices[u + 1] − indices[u] = inc
22       u += 1
23     if l ≠ u
24       Pattern p
25       p.i := indices[l]
26       p.n := indices[u] − indices[l] + 1
27       p.e := value
28       p.inc := inc
29       patterns.add(p)
30       ρ_{i→e} := ρ_{i→e} \ {indices[i] | ∀i ∈ [l, u]}
31       l := u + 1              // next sequence
32     u += 1
33   return patterns
```

Figure 8.4: Loop initialization pattern extraction algorithm in pseudo-code.

In case that write expressions in a write sequence are *shared*, i.e., they also appear in the formula outside of the sequence, we still extract patterns for the whole sequence. This may duplicate parts, which is not a problem since the extracted lambda terms are succinct and the "duplication" only affects the index range check of a lambda and is therefore negligible.

There are two common approaches for representing the initialization of an array variable a with n concrete values: with (1) *write* sequences of size n with array a as base array, or (2) n *read* operations on array a by asserting for each index $i \in i_1, \ldots, i_n$ that $\text{read}(a, i) = e$. In case (1), we are able to directly represent such array initializations by means of lambda terms. However, in case (2), we first have to translate the read operations into sequences of write operations. For example, given an array a that is initialized with some values e on indices $1 - 4$, we could either represent this as a sequence of write operations with a fresh array variable b as base array

$$a \coloneqq \text{write}(\text{write}(\text{write}(\text{write}(b,\, 1,\, e),\, 2,\, e),\, 3,\, e),\, 4,\, e),$$

or with the following four equalities asserted to be true

$$\text{read}(a,\, 1) = e,\ \text{read}(a,\, 2) = e,\ \text{read}(a,\, 3) = e,\ \text{read}(a,\, 4) = e.$$

However, the initialization with read/value equalities can also be represented as lambda term

$$a \coloneqq \lambda j.\text{ite}(1 \leq j \leq 4,\, e,\, b(j)),$$

where array b is a fresh array variable. In order to extract lambda terms from these equalities, we translate them into sequences of write operations and apply the lambda extraction algorithms to it. The only requirement is that, for the same reason as for the write sequence case, the read indices have to be distinct.

8.4 Merging Lambdas

Lambda terms extracted from a sequence of write operations often do not cover all indices in the sequence. Some might be left over. In order to preserve equisatisfiability, we use the uncovered write operations to create a new write sequence on top of the extracted lambda terms (cf. lines 15-16 in Figure 8.2). Note that as we represent write operations as lambda terms, we actually generate a sequence of lambda terms (representing write operations) on top of the extracted terms. Given a sequence of lambda terms of size n, however, we can apply a rewriting technique we refer to as *lambda merging*, which inlines the function bodies of lambda terms $\lambda_1, \ldots, \lambda_{n-1}$. The result is a single lambda term with a function body consisting of the function bodies of lambda terms $\lambda_1, \ldots, \lambda_n$. This technique may not yield representations as compact as lambda extraction, but merging function bodies of consecutive lambdas often enables additional simplifications. As an example consider write sequence $a_n \coloneqq \text{write}(a,\, \bar{i},\, \bar{e})$ of size n,

where $e_1, \ldots, e_n$ are equal. It corresponds to the following lambda sequence.

$$\begin{aligned} \lambda_n &\coloneqq \lambda j_n.\text{ite}(j_n = i_n,\, e,\, \lambda_{n-1}(j_n)), \\ &\vdots \\ \lambda_1 &\coloneqq \lambda j_1.\text{ite}(j_1 = i_1,\, e,\, a_0(j_1)) \end{aligned}$$

If we apply lambda merging to $\lambda_n, \ldots, \lambda_1$ and inline function bodies, we obtain the following lambda term

$$\begin{aligned} \lambda_{n'} \coloneqq \lambda j_n.\text{ite}(j_n = i_n,\, e,\, & \\ & \ddots \\ & \text{ite}(j_n = i_1,\, e,\, a_0(j_n)))\ldots) \end{aligned}$$

Note that $\lambda_{n'}$ can be further simplified by merging the if-then-else terms into one (since the if-branch of each if-then-else contains value e), which results in lambda term $\lambda_{n''}$.

$$\lambda_{n''} \coloneqq \lambda j_n.\text{ite}(j_n = i_n \vee \ldots \vee j_n = i_1,\, e,\, a_0(j_n))$$

8.4.1 Lemma Generation

Merged lambdas can be more compact than write sequences and may even be beneficial for lemma generation. For example, a read operation on $\lambda_{n''}$ at index j may produce a conflict on index i_1, where $\text{read}(\lambda_{n''},\, j) \neq e$. As a consequence, the following lemma is generated.

$$j = i_n \vee \ldots \vee j = i_1 \rightarrow \text{read}(\lambda_{n''},\, j) = e.$$

The resulting lemma covers all cases where $\text{read}(\lambda_{n''},\, j)$ could produce a conflict on indices $i_2, \ldots, i_n$. In the original write sequence version, however, it might need n lemmas.

8.4.2 Algorithm

Figure 8.5 illustrates procedures merge_lambdas and rec_merge for merging lambda sequences. Given formula ϕ, for every lambda sequence λ_n, procedure merge_lambdas recursively merges the lambda terms in λ_n into lambda term b, which is then used to substitute lambda sequence λ_n in formula ϕ (line 4). Procedure rec_merge recursively traverses the lambda sequence starting at the top most lambda term λ_n and substitutes every bound variable j_i by the variable j_n, which is bound by the top most lambda term λ_n. In the base case ($i = 0$), the procedure returns a fresh read operation on base array a_0 at index j_n (which substitutes variable j_1). Else, it performs a recursive call on λ_{i-1}, which yields term t_{i-1}. For every lambda term λ_i with $i < n$, rec_merge generates lambda term t_i by substituting all occurrences of variable j_i in the function body of λ_i by

```
1  procedure merge_lambdas (φ)
2    for write sequence λ_n := write(a, ī, ē) ∈ φ
3      b := rec_merge(n, j_n, λ_n)
4      φ := φ[λ_n/b]

5  procedure rec_merge (n, j_n, λ_i)
6    if i = 0 return a_0(j_n)                  // base array a_0
7    t_{i-1} := rec_merge(n, j_n, λ_{i-1})     // λ_i = λj_i.ite(j_i = i_i, e_i, λ_{i-1}(j_i))
8    if i < n
9      t_i := ite(j_i = i_i, e_i, λ_{i-1}(j_i))[j_i/j_n]
10     return t_i[λ_{i-1}(j_n)/t_{i-1}]
11   return λ_n[λ_{i-1}(j_n)/t_{i-1}]          // top-most lambda a_n
```

Figure 8.5: Merge lambdas algorithm in pseudo-code.

j_n, and returns the lambda term obtained by substituting all occurrences of read operation read$(\lambda_{i-1}, j_n,)$ in t_i with term t_{i-1} (line 10). For the top most lambda ($i = n$), procedure rec_merge returns the lambda term obtained by substituting all occurrences of read operation read$(\lambda_{i-1}, j_n,)$ in λ_n with term t_{i-1} (line 11).

8.5 Experimental Evaluation

We implemented lambda extraction and merging in our SMT solver Boolector and evaluated our techniques on all non-extensional benchmarks from the QF_ABV category of the SMT-LIBv2 benchmark library. Six configurations are considered: (1) Btor, (2) Btor+e, (3) Btor+m, (4) Btor+x, (5) Btor+xme, and (4) Btor+xm. The base line Btor is an improved version of Boolector that won the QF_ABV track of the SMT competition in 2014. For the other configurations, **x** indicates that lambda extraction is enabled, **m** indicates that lambda merging is enabled, and **e** indicates an eager solving approach by reducing the formula to QF_BV. It eliminates lambda terms with beta reduction, and the remaining read operations, i.e., applications of uninterpreted functions (UF), by Ackermann reduction. The Btor+e and Btor+xme configurations essentially simulate an eager approach similar to that of UCLID [58].

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.2 LTS. The memory and time limit for each solver/benchmark pair was set to 7 GB and 1200 seconds CPU time, respectively. In case of a timeout or memory out, a penalty of 1200 seconds was added to the total CPU time. Note that the time and memory limits and the hardware used for our experiments differ from the

	QF_ABV (13317)			
	Solved	TO	MO	Time [s]
Btor	13242	68	7	122645
Btor+e	13242	49	26	120659
Btor+m	13259	50	8	105647
Btor+x	13256	54	7	99834
Btor+xme	13246	47	24	111114
Btor+xm	**13263**	46	8	**84760**

Table 8.1: Overall results on QF_ABV benchmarks (13317 in total).

setup used at the SMT competition 2014.

Table 8.1 depicts the overall results consisting of the number of solved benchmarks (Solved), number of timeouts (TO), number of memory outs (MO), and the CPU time (Time) of all four configurations on the QF_ABV benchmarks. Enabling either lambda extraction (Btor+x) or lambda merging (Btor+m) improves the number of solved benchmarks by up to 17 instances and the runtime by up to 19% compared to Btor. Combining both techniques (Btor+xm) solves 21 more benchmarks and requires 30% less runtime compared to Btor. This suggests, that lambda extraction and merging have orthogonal effects. They complement each other and in combination improve solver performance further (most of the time). However, if the eager solving approach is employed, both configurations Btor+e and Btor+xme do not show a notable improvement in terms of solved instances (less timeouts, but more memory outs). This is due to the high memory consumption caused by eager elimination of lambda terms and UFs, where Btor+e in total consumes 2.6 times (397 GB), and Btor+xme 2.3 times (347 GB) more memory than Btor. The other four configurations require roughly the same amount of memory. Table 8.2 depicts the overall results and the number of extracted patterns grouped by QF_ABV benchmark families in more detail. On benchmark families *bmc*, *bbiere2*, *klee*, *platania*, and *stp* Btor+xm considerably improves in terms of runtime and number of solved instances compared to Btor. On the *bbiere2* and *platania* benchmark families, the combined use of lambda extraction and lambda merging yields significantly better results than both Btor+x and Btor+m alone. The most notable improvement in terms of runtime is achieved on the *klee* benchmark family, where all three configurations with *lambda extraction* enabled improve by orders of magnitude compared to Btor. The *klee* benchmark family consists of symbolic execution benchmarks obtained from KLEE [13], a symbolic virtual machine built on top of the LLVM compiler infrastructure. Previous versions of Boolector were shown to have rather poor performance on these benchmarks [50], which is confirmed by our experiments. This is due to the extreme version of lazy SMT in Boolec-

	Btor		Btor+e		Btor+m		Btor+x		Btor+xme		Btor+xm		Extracted Patterns				
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	λ_{mset}	λ_{mcpy}	$\lambda_{i\to e}$	$\lambda_{i\to i}$	$\lambda_{i\to i+1}$
bench (119)	119	2	119	3	119	2	119	0.3	119	0.3	119	0.3	208	0	34	0	0
bmc (38)	38	1361	39	769	39	921	39	197	**39**	**88**	39	182	256	3	56	0	0
bbiere (98)	75	29455	75	30301	75	28944	75	29359	75	29167	**75**	**28854**	0	10	0	0	0
bbiere2 (22)	17	7299	21	2617	18	6927	18	7842	**22**	**2034**	20	3241	1392	0	8	0	0
bbiere3 (8)	0	9600	1	8464	1	8435	0	9600	1	8463	1	8435	0	0	0	0	0
btfnt (1)	1	134	1	144	1	134	1	134	1	146	1	134	0	0	0	0	0
calc2 (36)	36	862	36	1528	36	864	36	863	36	1527	36	863	0	0	0	0	0
dwp (4188)	4187	2668	**4188**	**2216**	4187	2090	4187	2666	4187	2235	4187	2089	42	0	0	0	0
ecc (55)	**54**	**1792**	54	1745	54	1792	54	1845	54	1808	54	1845	125	0	0	0	0
egt (7719)	7719	222	7719	544	7719	221	7719	225	7719	275	**7719**	**212**	3893	0	0	0	0
jager (2)	0	2400	0	2400	0	2400	0	2400	0	2400	0	2400	14028	0	239	0	0
klee (622)	622	12942	620	4408	622	12688	622	169	**622**	**126**	622	154	9373	0	10049	0	0
pipe (1)	1	10	1	14	1	10	1	10	1	14	1	10	0	0	0	0	0
platania (275)	247	42690	238	58807	256	35005	255	34993	240	56172	**258**	**31189**	0	0	0	58	120
sharing (40)	40	2460	40	2459	40	2459	40	2460	40	2458	40	2458	0	0	0	0	0
stp (40)	34	8749	38	4238	39	2755	38	7072	38	4200	**39**	**2695**	60	0	297	0	0
stp_sa (52)	52	0.7	52	0.5	52	0.6	52	0.6	**52**	**0.5**	52	0.7	0	0	0	0	0
totals (13317)	13242	122645	13242	120659	13259	105647	13256	99834	13246	111114	**13263**	**84760**	29377	13	10683	58	120

Table 8.2: Overall results and number of extracted patterns on all QF_ABV benchmarks grouped by benchmark family.

tor, using lemmas on demand. In our experiments, Btor requires almost 13000 seconds to solve the 622 *klee* benchmarks, while lambda extraction improved runtime by up to a factor of 500 compared to Btor. This effect is illustrated by the scatter plot in Figure 8.6, which shows that the runtime on most of the benchmarks is improved by a factor of 10 to 100. The *klee* benchmarks contain many instances of the λ_{mset} and $\lambda_{i \to e}$ patterns, where Btor+xm was able to extract 9373 and 10049 lambda terms with an average size of 108 and 11, respectively. On most of the benchmarks where Btor+xm was able to extract lambda terms, the runtime improved. The only exceptions are the two benchmarks in the *jager* benchmark family, on which Btor+xm still timed out even though 14028 λ_{mset} and 239 $\lambda_{i \to e}$ patterns were extracted. In total, Btor+xm was able to extract 29377 λ_{mset}, 13 λ_{mcpy}, 10683 $\lambda_{i \to e}$, 58 $\lambda_{i \to i}$, and 120 $\lambda_{i \to i+1}$ patterns with an average size of 40, 7, 12, 39, and 38, respectively. The overall time required by Btor+xm for extracting and merging the lambda terms amounts to 41 and 24 seconds, which is less than 0.01% of the total runtime and therefore negligible.

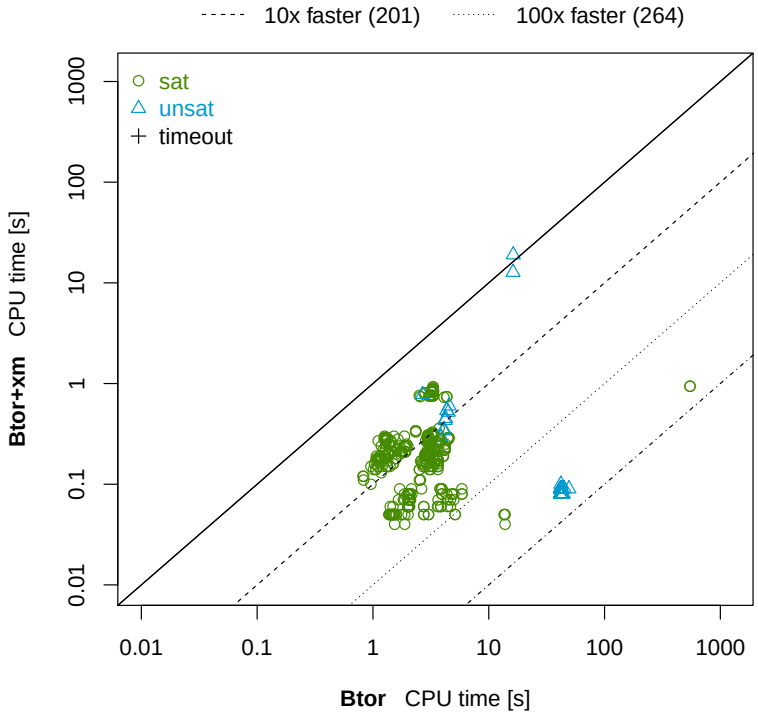


Figure 8.6: Btor vs. Btor+xm on *klee* benchmark family.

	k=1	2	3	4	5	6	7	8	9	10	11
Btor	0.1	0.4	8	42	296	T	T	T	T	T	T
SONOLAR	0.1	0.2	2	15	201	T	T	T	T	T	T
MathSAT	0.1	0.3	2	9	70	709	T	M	T	T	T
Yices	**0.0**	**0.0**	0.1	0.6	2	8	23	93	371	T	T
Btor+xm	0.1	0.1	0.1	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.2**	**0.2**

	12	13	14	15	16	17	18	19	20	21
Btor	T	T	T	T	T	T	T	T	T	M
SONOLAR	T	T	T	T	T	T	M	M	M	M
MathSAT	T	T	T	T	T	T	T	M	M	M
Yices	T	T	T	T	T	T	M	M	M	T
Btor+xm	**0.3**	**0.5**	**1**	**2**	**6**	**14**	**44**	**140**	**463**	M

Table 8.3: Runtime in seconds on *memcpy* benchmarks of size 2^k copied elements. T denotes out of time, M denotes out of memory.

Benchmark family *bbiere* contains 11 benchmarks, which encode a *memcpy* operation on two non-overlapping memory regions and verify the correctness of the memcpy algorithm. The benchmarks are parameterized by the size of the copied memory region starting from size 2 up to size 12. We generated 21 additional benchmarks with size 2 to 2^{21} (i.e., 2^k with $1 \leq k \leq 21$) in order to evaluate how Btor+xm scales on these benchmarks. For comparison we additionally ran the top three solvers after Boolector at the SMT competition 2014, Yices [22] version 2.3.1, MathSAT [14] version 5.3.5, and SONOLAR [41] version 2014-12-04 on these benchmarks. Table 8.3 depicts the runtime of all solvers on the additional *memcpy* benchmarks of size 2 to 2^{21}, where T denotes out of time, and M denotes out of memory. Btor and SONOLAR are able to solve these benchmarks up to size 2^5, MathSAT up to size 2^6, Yices up to size 2^9, and Btor+xm up to size 2^{20}. For the largest instance parsing consumes most of the runtime (∼60%). For sizes greater than 2^{20}, Boolector is not able to fit the input formula into 7 GB of memory, which results in a memory out.

Finally, we measured the impact of lambda extraction and lambda merging w.r.t. the number of generated lemmas. Since every lemma generated in Boolector entails an additional call to the underlying SAT solver, the number of generated lemmas usually correlates with the runtime of the solver. On the QF_ABV benchmarks commonly solved by Btor and Btor+xm (13242 in total), Btor generates 872913 lemmas, whereas Btor+xm generates 158175 lemmas, which is a reduction by a factor of 5.5. Consequently, the size of the CNF is reduced by 25%

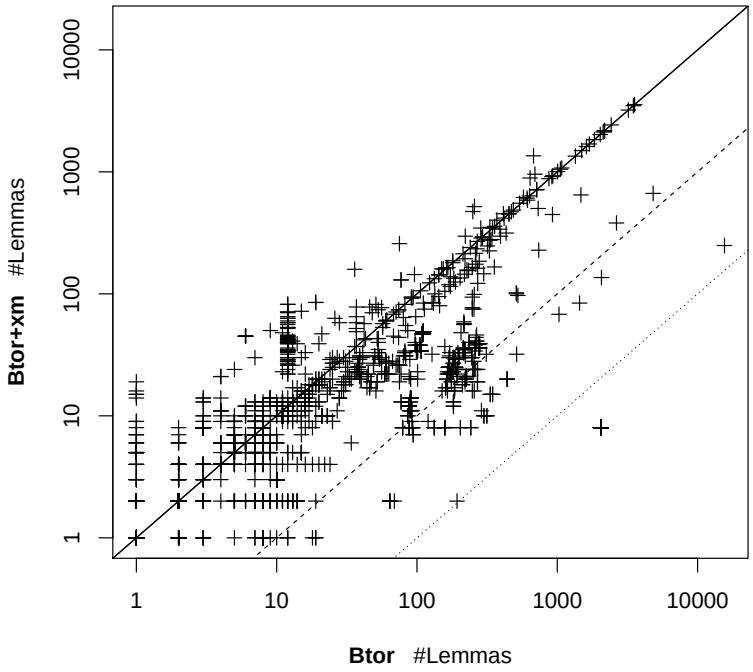


Figure 8.7: Number of generated lemmas Btor vs. Btor+xm on commonly solved QF_ABV benchmarks (13242 in total).

on average (no matter whether variables or clauses are counted). This is further illustrated in Figure 8.7. On these benchmarks the reduction of the time spent in the underlying SAT solver is reduced from 59638 to 40101, i.e., an improvement of 33%.

8.6 Conclusion

We discussed patterns of array operations occurring in actual benchmarks and presented a technique denoted as *lambda extraction*, which utilizes such patterns to extract compact and more succinct lambda terms. Another new complementary technique, called *lambda merging*, can still be exploited if lambda extraction is not applicable. These techniques allow to produce stronger and more succinct lemmas.

In the experimental analysis, based on our SMT solver Boolector, it was shown that these techniques reduce the number of generated lemmas by a factor of 5.5, and the overall size of the bit-blasted CNF by 25% on average. To summarize, we were able to considerably improve the overall performance of Boolector and

achieve speedups up to orders of magnitude, particularly on benchmarks from symbolic execution.

We believe, that there are additional patterns in software and hardware verification benchmarks, which can be extracted as lambdas and used to speed-up array reasoning further. Our results also suggest, that a more expressive theory of arrays might be desirable for users of SMT solvers, in order to allow more succinct encodings of common array operation patterns.

C

Chapter 9

Paper C. Counterexample-Guided Model Synthesis

Published In Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017), 17 pages, to appear, Uppsala, Sweden, 2017.

Authors Mathias Preiner, Aina Niemetz and Armin Biere.

Abstract In this paper we present a new approach for solving quantified formulas in Satisfiability Modulo Theories (SMT), with a particular focus on the theory of fixed-size bit-vectors. We combine counterexample-guided quantifier instantiation with a syntax-guided synthesis approach, which allows us to synthesize both Skolem functions and terms for quantifier instantiations. Our approach employs two ground theory solvers to reason about quantified formulas. It neither relies on quantifier specific simplifications nor heuristic quantifier instantiation techniques, which makes it a simple yet effective approach for solving quantified formulas. We implemented our approach in our SMT solver Boolector and show in our experiments that our techniques are competitive compared to the state-of-the-art in solving quantified bit-vectors.

9.1 Introduction

Many techniques in hardware and software verification rely on quantifiers for describing properties of programs and circuits, e.g., universal safety properties, inferring program invariants [35], finding ranking functions [16], and synthesizing hardware and software [36, 59]. Quantifiers further allow to define theory axioms to reason about a theory of interest not supported natively by an SMT solver.

The theory of fixed-size bit-vectors provides a natural way of encoding bit-precise semantics as found in hardware and software. In recent SMT competitions, the division for quantifier-free fixed-size bit-vectors was the most competitive with an increasing number of participants every year. Quantified bit-vector reasoning, however, even though a highly required feature, is still very challenging and did not get as much attention as the quantifier-free fragment. The complexity of deciding quantified bit-vector formulas is known to be NExpTime-hard and solvable in ExpSpace [40]. Its exact complexity, however, is still unknown.

While there exist several SMT solvers that efficiently reason about quantifier-free bit-vectors, only CVC4 [3], Z3 [19], and Yices [23] support the quantified bit-vector fragment. The SMT solver CVC4 employs counterexample-guided quantifier instantiation (CEGQI) [54], where a ground theory solver tries to find concrete values (counterexamples) for instantiating universal variables by generating models of the negated input formula. In Z3, an approach called model-based quantifier instantiation (MBQI) [33] is combined with a model finding procedure based on templates [63]. In contrast to only relying on concrete counterexamples as candidates for quantifier instantiation, MBQI additionally uses symbolic quantifier instantiation to generalize the counterexample by selecting ground terms to rule out more spurious models. The SMT solver Yices provides quantifier support limited to exists/forall problems [24] of the form $\exists \mathbf{x} \forall \mathbf{y}.P[\mathbf{x}, \mathbf{y}]$. It employs two ground solver instances, one for checking the satisfiability of a set of generalizations and generating candidate solutions for the existential variables $\mathbf{x}$, and the other for checking if the candidate solution is correct. If the candidate model is not correct, a model-based generalization procedure refines the candidate models.

Recently, a different approach based on binary decision diagrams (BDD) was proposed in [38]. Experimental results of its prototype implementation Q3B show that it is competitive with current state-of-the-art SMT solvers. However, employing BDDs for solving quantified bit-vectors heavily relies on formula simplifications, variable ordering, and approximation techniques to reduce the size of the BDDs. If these techniques fail to substantially reduce the size of the BDDs this approach does not scale. Further, in most applications it is necessary to provide models in case of satisfiable problems. However, it is unclear if a bit-level BDD-based model can be lifted to produce more succinct word-level models.

In this paper, we combine a variant of CEGQI with a syntax-guided synthesis [2] approach to create a model finding algorithm called *counterexample-guided model synthesis* (CEGMS), which iteratively refines a synthesized candi-

date model. Unlike Z3, our approach synthesizes Skolem functions based on a set of ground instances without the need for specifying function or circuit templates up-front. Further, we can apply CEGMS to the negation of the formula in a parallel dual setting to synthesize quantifier instantiations that prove the unsatisfiability of the original problem. Our approach is a simple yet efficient technique that does not rely on quantifier specific simplifications, which have previously been found to be particularly useful [63]. Our experimental evaluation shows that our approach is competitive with the state-of-the-art in solving quantified bit-vectors. However, even though we implemented it in Boolector, an SMT solver for the theory of bit-vectors with arrays and uninterpreted functions, our techniques are not restricted to the theory of quantified bit-vectors.

9.2 Preliminaries

We assume the usual notions and terminology of first-order logic and primarily focus on the theory of *quantified fixed-size bit-vectors*. We only consider many-sorted languages, where bit-vectors of different size are interpreted as bit-vectors of different sorts.

Let Σ be a signature consisting of a set of function symbols $f : n_1, \ldots, n_k \to n$ with arity $k \geq 0$ and a set of bit-vector sorts with size $n, n_1, \ldots, n_k$. For the sake of simplicity and w.l.o.g., we assume that sort *Bool* is interpreted as a bit-vector of size one with constants $\top$ (1) and $\bot$ (0), and represent all predicate symbols as function symbols with a bit-vector of size one as the sort of the co-domain. We refer to function symbols occurring in Σ as *interpreted*, and those symbols not included in Σ as *uninterpreted*. A *bit-vector term* is either a bit-vector variable or an application of a bit-vector function of the form $f(t_1, \ldots, t_k)$, where $f \in \Sigma$ or $f \notin \Sigma$, and $t_1, \ldots, t_k$ are bit-vector terms. We denote bit-vector term t of size n as $t_{[n]}$ and define its *domain* as $\mathcal{BV}_{[n]}$, which consists of all *bit-vector values* of size n. Analogously, we represent a bit-vector value as an integer with its size as a subscript, e.g., $1_{[4]}$ for 0001 or $-1_{[4]}$ for 1111.

We assume the usual interpreted symbols for the theory of bit-vectors, e.g., $=_{[n]}, +_{[n]}, *_{[n]}, \mathit{concat}_{[n+m]}, <_{[n]}$, etc., and will omit the subscript specifying their bit-vector size if the context allows. We further interpret an ite(c, t_0, t_1) as an *if-then-else* over bit-vector terms, where ite($\top$, t_0, t_1) $= t_0$ and ite($\bot$, t_0, t_1) $= t_1$.

In general, we refer to 0-arity function symbols as *constant* symbols, and denote them by a, b, and c. We use f and g for non-constant function symbols, P for predicates, x, y and z for variables, and t for arbitrary terms. We use symbols in bold font, e.g., $\mathbf{x}$, as a shorthand for tuple $(x_1, \ldots, x_k)$, and denote a formula (resp. term) that may contain variables $\mathbf{x}$ as $\varphi[\mathbf{x}]$ (resp. $t[\mathbf{x}]$). If a formula (resp. term) does not contain any variables we refer to it as *ground* formula (resp. term). We further use $\varphi[t/x]$ as a notation for *replacing* all occurrences of x in φ with a term t. Similarly, $\varphi[\mathbf{t}/\mathbf{x}]$ is used as a shorthand for $\varphi[t_1/x_1, \ldots, t_k/x_k]$.

Given a quantified formula $\varphi[\mathbf{x}, \mathbf{y}]$ with universal variables $\mathbf{x}$ and existential variables $\mathbf{y}$, *Skolemization* [56] eliminates all existential variables $\mathbf{y}$ by introducing *fresh* uninterpreted function symbols with arity ≥ 0 for the existential variables $\mathbf{y}$. For example, the *skolemized* form of formula $\exists y_1 \forall \mathbf{x} \exists y_2 . P(y_1, \mathbf{x}, y_2)$ is $\forall \mathbf{x} . P(f_{y_1}, \mathbf{x}, f_{y_2}(\mathbf{x}))$, where f_{y_1} and f_{y_2} are fresh uninterpreted symbols, which we refer to as *Skolem symbols*. The subscript denotes the existential variable that was eliminated by the corresponding Skolem symbol. We write $skolemize(\varphi)$ for the application of Skolemization to formula φ, $var_{\forall}(\varphi)$ for the set of universal variables in φ, and $sym_{sk}(\varphi)$ for the set of Skolem symbols in φ.

A Σ-structure M maps each bit-vector sort of size n to its domain $\mathcal{BV}_{[n]}$, each function symbol $f : n_1, \ldots, n_k \to n \in \Sigma$ with arity $k > 0$ to a total function $M(f) : \mathcal{BV}_{[n_1]}, \ldots, \mathcal{BV}_{[n_k]} \to \mathcal{BV}_{[n]}$, and each constant symbol with size n to an element in $\mathcal{BV}_{[n]}$. We use $M' \coloneqq M\{x \mapsto v\}$ to denote a Σ-structure M' that maps variable x to a value v of the same sort and is otherwise identical to M. The evaluation $M(x_{[n]})$ of a variable $x_{[n]}$ and $M(c_{[n]})$ of a constant c in M is an element in $\mathcal{BV}_{[n]}$. The evaluation of an arbitrary term t in M is denoted by $M[\![t]\!]$ and is recursively defined as follows. For a constant c (resp. variable x) $M[\![c]\!] = M(c)$ (resp. $M[\![x]\!] = M(x)$). A function symbol f is evaluated as $M[\![f(t_1, \ldots, t_k)]\!] = M(f)(M[\![t_1]\!], \ldots, M[\![t_k]\!])$. A Σ-structure M is a *model* of a formula φ if $M[\![\varphi]\!] = \top$. A formula is *satisfiable* if and only if it has a model.

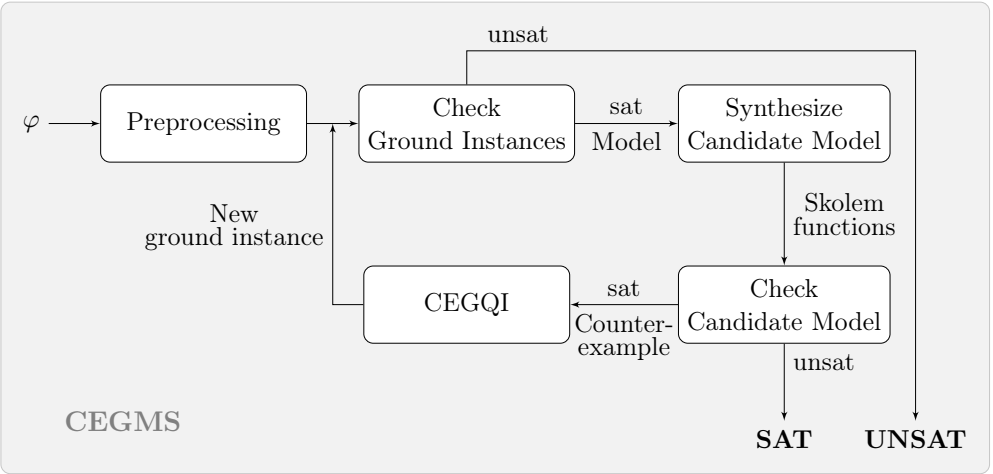


Figure 9.1: Basic workflow of our CEGMS approach.

9.3 Overview

In essence, our *counterexample-guided model synthesis* (CEGMS) approach for solving quantified bit-vector problems combines a variant of counterexample-guided quantifier instantiation (CEGQI) [54] with the syntax-guided synthesis approach in [2] in order to synthesize Skolem functions. The general workflow of our approach is depicted in Figure 9.1 and introduced as follows.

Given a quantified formula φ as input, CEGMS first applies Skolemization as a preprocessing step and initializes an empty set of ground instances. This empty set is, in the following, iteratively extended with ground instances of φ, generated via CEGQI. In each iteration, CEGMS first checks for a ground conflict by calling a ground theory solver instance on the set of ground instances. If the solver returns *unsatisfiable*, a ground conflict was found and the CEGMS procedure concludes with UNSAT. If the solver returns *satisfiable*, it produces a model for the Skolem symbols, which serves as a base for synthesizing a candidate model for all Skolem functions. If the candidate model is valid, the CEGMS procedure concludes with SAT. However, if the candidate model is invalid, the solver generates a counterexample, which is used to create a new ground instance of the formula via CEGQI. The CEGMS procedure terminates, when either a ground conflict occurs, or a valid candidate model is synthesized.

9.4 Counterexample-Guided Model Synthesis

The main refinement loop of our CEGMS approach is realized via CEGQI [54], a technique similar to the *generalization by substitution* approach described in [24], where a concrete counterexample to universal variables is used to create a ground instance of the formula, which then serves as a refinement for the candidate model. Similarly, every refinement step of our CEGMS approach produces a ground instance of the formula by instantiating its universal variables with a counterexample if the synthesized candidate model is not valid. The counterexample corresponds to a concrete assignment to the universal variables for which the candidate model does not satisfy the formula. Figure 9.2 introduces the main algorithm of our CEGMS approach as follows.

Given a quantified bit-vector formula φ, we represent φ as a directed acyclic graph (DAG), with the Boolean layer expressed by means of *AND* and *NOT*. As a consequence, it is not possible to transform φ into negative normal form (NNF) and we therefore apply quantifier normalization as a preprocessing step to ensure that a quantifier does not occur in both negated and non-negated form. For the same reason, an *ite*-term is eliminated in case that a quantifier occurs in its condition. Note that if φ is not in NNF, it is sufficient to keep track of the polarities of the quantifiers, i.e., to count the number of negations from the root of the formula to the resp. quantifier, and flip the quantifier if the number of negations is odd. If a quantifier occurs negative and positive, the scope of the

```
1  function CEGMS (φ)
2    G := ⊤, x := var_∀(φ)
3    φ_sk := skolemize(preprocess(φ))        // apply Skolemization
4    f := sym_sk(φ_sk)                       // Skolem symbols
5    φ_G := φ_sk[u/x]                        // ground φ_sk with fresh u
6    while true
7      r, M_G := sat(G)                      // check set of ground instances
8      if r = unsat return unsat             // found ground conflict
9      M_S := synthesize(f, G, M_G, φ_G)     // synthesize candidate model
10     r, M_C := sat(¬φ_G[M_S(f)/f])         // check candidate model
11     if r = unsat return sat               // candidate model is valid
12     G := G ∧ φ_G[M_C(u)/u]                // new ground instance via CEGQI
```

Figure 9.2: High level view of our CEGMS approach.

quantifier is duplicated, the quantification is flipped, and the negative occurrence is substituted with the new subgraph. Further note that preprocessing currently does not include any further simplification techniques such as miniscoping or destructive equality resolution (DER) [63].

After preprocessing, Skolemization is applied to the normalized formula, and all universal variables $\mathbf{x}$ in φ_{sk} are instantiated with fresh bit-vector constants $\mathbf{u}$ of the same sort. This yields ground formula φ_G. Initially, procedure CEGMS starts with an empty set of ground instances G, which is iteratively extended with new ground instances during the refinement loop.

In the first step of the loop, an SMT solver instance checks whether G contains a ground conflict (line 7). If this is the case, procedure CEGMS has found conflicting quantifier instantiations and concludes with *unsatisfiable*. Else, the SMT solver produces model M_G for all Skolem symbols in G, i.e., every Skolem constant is mapped to a bit-vector value, and every uninterpreted function corresponding to a Skolem function is mapped to a partial function mapping bit-vector values. Model M_G is used as a base for synthesizing a candidate model M_S that satisfies G. The synthesis of candidate models M_S will be introduced in more detail in the next section. In order to check if M_S is also a model that satisfies φ, we check with an additional SMT solver instance if there exists an assignment to constants $\mathbf{u}$ (corresponding to universal variables $\mathbf{x}$), such that candidate model M_S does not satisfy formula φ (line 10).

If the second SMT solver instance returns unsatisfiable, no such assignment to constants $\mathbf{u}$ exists and consequently, candidate model M_S is indeed a valid model for the Skolem functions and procedure CEGMS returns with *satisfiable*. Else, the SMT solver produces a concrete counterexample for constants $\mathbf{u}$, for which

candidate model M_S does not satisfy formula φ. This counterexample is used as a quantifier instantiation to create a new ground instance $g_i \coloneqq \varphi_G[M_C(\mathbf{u})/\mathbf{u}]$, which is added to $G \coloneqq G \wedge g_i$ as a refinement (line 12) and considered in the next iteration for synthesizing a candidate model. These steps are repeated until either a ground conflict is found or a valid candidate model was synthesized.

Our CEGMS procedure creates in the worst-case an unmanageable number of ground instances of the formula prior to finding either a ground conflict or a valid candidate model, infinitely many in case of infinite domains. In the bit-vector case, however, it produces in the worst-case exponentially many ground instances in the size of the domain. Since, given a bit-vector formula, there exist only finitely many such ground instances, procedure CEGMS will always terminate. Further, if CEGMS concludes with satisfiable, it returns with a model for the existential variables.

9.5 Synthesis of Candidate Models

In our CEGMS approach, based on a concrete model M_G we apply synthesis to find general models M_S to accelerate either finding a valid model or a ground conflict. Consider formula $\varphi \coloneqq \forall xy \exists z.z = x + y$, its skolemized form $\varphi_{sk} \coloneqq \forall xy.f_z(x, y) = x + y$, some ground instances $G \coloneqq f_z(0, 0) = 0 \wedge f_z(0, 1) = 1 \wedge f_z(1, 2) = 3$, and model $M_G \coloneqq \{f_z(0, 0) \mapsto 0, f_z(0, 1) \mapsto 1, f_z(1, 2) \mapsto 3\}$ that satisfies G. A simple approach for generating a Skolem function for f_z would be to represent model $M_G(f_z)$ as a lambda term $\lambda xy.\text{ite}(x = 0 \wedge y = 0,\ 0,\ \text{ite}(x = 0 \wedge y = 1,\ 1,\ \text{ite}(x = 1 \wedge y = 2,\ 3,\ 0)))$ with base case constant 0, and check if it is a valid Skolem function for f_z. If it is not valid, a counterexample is generated and a new ground instance is added via CEGQI to refine the set of ground instances G. However, this approach, in the worst-case, enumerates exponentially many ground instances until finding a valid candidate model. By introducing a modified version of a syntax-guided synthesis technique called *enumerative learning* [62], CEGMS is able to produce a more succinct and more general lambda term $\lambda xy.x + y$, which satisfies the ground instances G and formula φ_{sk}.

Enumerative learning as in [62] systematically enumerates expressions that can be built based on a given syntax and checks whether the generated expression conforms to a set of concrete test cases. These expressions are generated in increasing order of a specified complexity metric, such as, e.g., the size of the expression. The algorithm creates larger expressions by combining smaller ones of a given size, which is similar to the idea of dynamic programming. Each generated expression is evaluated on the concrete test cases, which yields a vector of values also denoted as *signature*. In order to reduce the number of enumerated expressions, the algorithm discards expressions with identical signatures, i.e., if two expressions produce the same signature the one generated first will be stored and the other one will be discarded. Figure 9.3 depicts a simplified version of the enumerative learning algorithm as employed in our CEGMS approach, while

```
1  function enumlearn (f, I, O, T, M)
2    S ≔ ∅, E[1] ≔ I, size = 0
3    while true
4      size ≔ size + 1                    // increase expression size to create
5      for t ∈ enumexps(size, O, E)       // enumerate all expr. of size size
6        s ≔ eval(M, T[t/f])              // compute signature of t
7        if s ∉ S                         // expression not yet created
8          S ≔ S ∪ {s}                    // cache signature
9          if checksig(s) return t        // t conforms to test cases T
10         E[size] ≔ E[size] ∪ {t}        // store expression t
```

Figure 9.3: Simplified version of enumerative learning [62] employed in CEGMS.

a more detailed description of the original algorithm can be found in [62].

Given a Skolem symbol f, a set of inputs I, a set of operators O, a set of test cases T, and a model M, algorithm enumlearn attempts to synthesize a term t, such that $T[t/f]$ evaluates to true under model M. This is done by enumerating all terms t that can be built with inputs I and bit-vector operators O. Enumerating all expressions of a certain size (function enumexps) follows the original enumerative learning approach [62]. Given an expression size *size* and a bit-vector operator o, the size is partitioned into partitions of size $k = arity(o)$, e.g., (1,3) (3,1) (2,2) for $size = 4$ and $k = 2$. Each partition $(s_1, \ldots, s_k)$ specifies the size s_i of expression e_i, and is used to create expressions of size *size* with operator o, i.e., $\{o(e_1, \ldots, e_k) \mid (e_1, \ldots, e_k) \in E[s_1] \times \ldots \times E[s_k]\}$, where $E[s_i]$ corresponds to the set of expressions of size s_i. Initially, for $size = 1$, function enumexps enumerates inputs only, i.e., $E[1] = I$.

For each generated term t, a signature s is computed from a set of test cases T with function eval. In the original algorithm [62], set T contains concrete examples of the input/output relation of f, i.e., it defines a set of output values of f under some concrete input values. In our case, model $M(f)$ may be used as a test set T, since it contains a concrete input/output relation on some bit-vector values. However, we are not looking for a term t with that concrete input/output value behavior, but a term t that at least satisfies the set of current ground instances G. Hence, we use G as our test set and create a signature s by evaluating every ground instance $g_i \in G[t/f]$, resulting in a tuple of Boolean constants, where the Boolean constant at position i corresponds to the value $M[\![g_i]\!]$ of ground instance $g_i \in G[t/f]$ under current model M. Procedure checksig returns true if signature s contains only the Boolean constant $\top$, i.e., if every ground instance $g_i \in G$ is satisfied.

As a consequence of using G rather than $M(f)$ as a test set T, the expres-

```
1  function synthesize (f, G, M_G, φ_G)
2    M_S := M_G, O := ops(φ_G)            // choose operators O w.r.t. φ_G
3    for f ∈ f
4      I := inputs(f, φ_G)                // choose inputs for f
5      t := enumlearn(f, I, O, G, M_S)    // synthesize term t
6      if t ≠ null
7        M_S := M_S{f ↦ t}                // update model
8    return M_S
```

Figure 9.4: Synthesis of candidate models in CEGMS.

sion enumeration space is even more pruned since computing the signature of f w.r.t. G yields more identical expressions (and consequently, more expressions get discarded). Note that the evaluation via function eval does not require additional SMT solver calls, since the value of ground instance $g_i \in G[t/f]$ can be computed via evaluating $M[\![g_i]\!]$.

Algorithm synthesize produces Skolem function candidates for every Skolem symbol $f \in \mathbf{f}$, as depicted in Figure 9.4. Initially, a set of bit-vector operators O is selected, which consists of those operators appearing in formula φ_G. Note that we do not select all available bit-vector operators of the bit-vector theory in order to reduce the number of expressions to enumerate. The algorithm then selects a set of inputs I, consisting of the universal variables on which f depends and the constant values that occur in formula φ_G. Based on inputs I and operators O, a term t for Skolem symbol f is synthesized and stored in model M_S (lines 4-7). If algorithm enumlearn is not able to synthesize a term t, model $M_G(f)$ is used instead. This might happen if function enumlearn hits some predefined limit such as the maximum number of expressions enumerated.

In each iteration step of function synthesize, model M_S is updated if enumlearn succeeded in synthesizing a Skolem function. Thus, in the next iterations, previously synthesized Skolem functions are considered for evaluating candidate expressions in function enumlearn. This is crucial to guarantee that each synthesized Skolem function still satisfies the ground instances in G. Otherwise, M_S may not rule out every counterexample generated so far, and thus, validating the candidate model may result in a counterexample that was already produced in a previous refinement iteration. As a consequence, our CEGMS procedure would not terminate even for finite domains since it might get stuck in an infinite refinement loop while creating already existing ground instances.

The number of inputs and bit-vector operators used as base for algorithm enumlearn significantly affects the size of the enumeration space. Picking too many inputs and operators enumerates too many expressions and algorithm enumlearn will not find a candidate term in a reasonable time, whereas restrict-

ing the number of inputs and operators too much may not yield a candidate expression at all. In our implementation, we kept it simple and maintain a set of base operators $\{ite, \sim\}$, which gets extended with additional bit-vector operators occurring in the original formula. The set of inputs consists of the constant values occurring in the original formula and the universal variables on which a Skolem symbol depends. Finding more restrictions on the combination of inputs and bit-vector operators in order to reduce the size of the enumeration space is an important issue, but left to future work.

Example 9.1. Consider $\varphi \coloneqq \forall x \exists y.(x < 0 \rightarrow y = -x) \wedge (x \geq 0 \rightarrow y = x)$, and its skolemized form $\forall x.(x < 0 \rightarrow f_y(x) = -x) \wedge (x \geq 0 \rightarrow f_y(x) = x)$, where y and consequently $f_y(x)$ corresponds to the absolute value function *abs(x)*. For synthesizing a candidate model for f_y, we first pick the set of inputs $I \coloneqq \{x, 0\}$ and the set of operators $O \coloneqq \{-, \sim, <, ite\}$ based on formula φ. Note that we omitted operators $\geq$ and $\rightarrow$ since they can be expressed by means of the other operators. The ground formula and its negation are defined as follows.

$$\varphi_G \coloneqq (u < 0 \rightarrow f_y(u) = -u) \wedge (u \geq 0 \rightarrow f_y(u) = u)$$

$$\neg\varphi_G \coloneqq (u < 0 \ \wedge \ f_y(u) \neq -u) \vee (u \geq 0 \ \wedge \ f_y(u) \neq u)$$

For every refinement round i, the table below shows the set of ground instances G, the synthesized candidate model $M(f_y)$, formula $\neg\varphi_G[M_S(f_y)/f_y]$ for checking the candidate model, and a counterexample M_C for constant u if the candidate model was not correct.

$\mathbf{i}$	$\mathbf{G}$	$\mathbf{M_S(f_y)}$	$\neg\varphi_\mathbf{G}[\mathbf{M_S(f_y)}/\mathbf{f_y}]$	$\mathbf{M_C(u)}$
1	$\top$	$\lambda x.0$	$(u < 0 \wedge 0 \neq -u) \vee (u \geq 0 \wedge 0 \neq u)$	1
2	$f_y(1) = 1$	$\lambda x.x$	$(u < 0 \wedge u \neq -u) \vee (u \geq 0 \wedge u \neq u)$	-1
3	$f_y(-1) = 1$	$\lambda x.ite($ $x < 0, -x, x)$	$(u < 0 \wedge ite(u < 0, -u, u) \neq -u) \vee$ $(u \geq 0 \wedge ite(u < 0, -u, u) \neq u)$	-

In the first round, the algorithm starts with ground formula $G \coloneqq \top$. Since any model of f_y satisfies G, for the sake of simplicity, we pick $\lambda x.0$ as candidate, resulting in counterexample $u = 1$, and refinement $\varphi_G[1/u] \equiv f_y(1) = 1$ is added to G. In the second round, lambda term $\lambda x.x$ is synthesized as candidate model for f_y since it satisfies $G \coloneqq f_y(1) = 1$. However, this is still not a valid model for f_y and counterexample $u = -1$ is produced, which yields refinement $\varphi_G[-1/u] \equiv f_y(-1) = 1$. In the third and last round, $M_S(f_y) \coloneqq \lambda x.ite(x < 0, -x, x)$ is synthesized and found to be a valid model since $\neg\varphi_G[M_S(f_y)/f_y]$ is unsatisfiable, and CEGMS concludes with satisfiable.

9.6 Dual Counterexample-Guided Model Synthesis

Our CEGMS approach is a model finding procedure that enables us to synthesize Skolem functions for satisfiable problems. However, for the unsatisfiable case we rely on CEGQI to find quantifier instantiations based on concrete counterexamples that produce conflicting ground instances. In practice, CEGQI is often successful in finding ground conflicts. However, it may happen that way too many quantifier instantiations have to be enumerated (in the worst-case exponentially many for finite domains, infinitely many for infinite domains). In order to obtain better (symbolic) candidates for quantifier instantiation, we exploit the concept of duality of the input formula and simultaneously apply our CEGMS approach to the original input and its negation (the *dual* formula).

Given a quantified formula φ and its negation, the dual formula $\neg\varphi$, e.g., $\varphi \coloneqq \forall\mathbf{x}\exists\mathbf{y}.P[x,y]$ and $\neg\varphi \coloneqq \exists\mathbf{x}\forall\mathbf{y}.\neg P[x,y]$. If $\neg\varphi$ is satisfiable, then there exists a model $M(\mathbf{x})$ to its existential variables $\mathbf{x}$ such that $\varphi[M(\mathbf{x})/\mathbf{x},\mathbf{y}]$ is unsatisfiable. That is, a model in the dual formula $\neg\varphi$ can be used as a quantifier instantiation in the original formula φ to immediately produce a ground conflict. Similarly, if $\neg\varphi$ is unsatisfiable, then there exists no quantifier instantiation in φ such that φ is unsatisfiable. As a consequence, if we apply CEGMS to the dual formula and it is able to synthesize a valid candidate model, we obtain a quantifier instantiation that immediately produces a ground conflict in the original formula. Else, if our CEGMS procedure concludes with unsatisfiable on the dual formula, there exists no model to its existential variables and therefore, the original formula is satisfiable.

Dual CEGMS enables us to simultaneously search for models and quantifier instantiations, which is particularly useful in a parallel setting. Further, applying synthesis to produce quantifier instantiations via the dual formula allows us to create terms that are not necessarily ground instances of the original formula. This is particularly useful in cases where heuristic quantifier instantiation techniques based on E-matching [21] or model-based quantifier instantiation [33] struggle due to the fact that they typically select terms as candidates for quantifier instantiation that occur in some set of ground terms of the input formula, as illustrated by the following example.

Example 9.2. Consider the unsatisfiable formula $\varphi \coloneqq \forall x.a * c + b * c \neq x * c$, where $x = a + b$ produces a ground conflict. Unfortunately, $a + b$ is not a ground instance of φ and is consequently not selected as a candidate by current state-of-the-art heuristic quantifier instantiation techniques. However, if we apply CEGMS to the dual formula $\forall abc\exists x.a * c + b * c = x * c$, we obtain $\lambda xyz.x + y$ as a model for Skolem symbol $f_x(a,b,c)$, which corresponds to the term $a + b$ if instantiated with (a,b,c). Selecting $a + b$ as a term for instantiating variable x in the original formula results in a conflicting ground instance, which immediately allows us to determine unsatisfiability.

Note that if CEGMS concludes unsatisfiable on the dual formula, we currently do not produce a model for the original formula. Generating a model would require further reasoning, e.g., proof reasoning, on the conflicting ground instances of the dual formula and is left to future work.

Further, dual CEGMS currently only utilizes the final result of applying CEGMS to the dual formula. Exchanging intermediate results (synthesized candidate models) between the original and the dual formula in order to prune the search is an interesting direction for future work.

In the context of quantified Boolean formulas (QBF), the duality of the given input has been previously successfully exploited to prune and consequently speed up the search in circuit-based QBF solvers [34]. In the context of SMT, in previous work we applied the concept of duality to optimize lemmas on demand approach for the theory of arrays in Boolector [45].

9.7 Experiments

We implemented our CEGMS technique and its dual version in our SMT solver Boolector [46], which supports the theory of bit-vectors combined with arrays and uninterpreted functions. We evaluated our approach on two sets of benchmarks (5029 in total). Set BV (191) contains all BV benchmarks of SMT-LIB [4], whereas set BV_{LNIRA} (4838) consists of all LIA, LRA, NIA, NRA benchmarks of SMT-LIB [4] translated into bit-vector problems by substituting every integer or real with a bit-vector of size 32, and every arithmetic operator with its signed bit-vector equivalent.

We evaluated four configurations of Boolector[1]: (1) Btor, the CEGMS version without synthesis, (2) Btor+s, the CEGMS version with synthesis enabled, (3) Btor+d, the dual CEGMS version without synthesis, (4) Btor+ds, the dual CEGMS version with synthesis enabled. We compared our approach to the current development versions of the state-of-the-art SMT solvers CVC4[2] [3] and Z3[3] [19], and the BDD-based approach implemented as a prototype called Q3B[4] [38]. The tool Q3B runs two processes with different approximation strategies in a parallel portfolio setting, where one process applies over-approximation and the other under-approximation. The dual CEGMS approach implemented in Boolector is realized with two parallel threads within the solver, one for the original formula and the other for the dual formula. Both threads do not exchange any information and run in a parallel portfolio setting.

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.5 LTS. We set the limits for each solver/benchmark pair to 7GB of memory and 1200 seconds

[1] Boolector commit 4f7837876cf9c28f42649b368eaffaf03c7e1357
[2] CVC4 commit d19a95344fde1ea1ff7d784b2c4fc6d09f459899
[3] Z3 commit 186afe7d10d4f0e5acf40f9b1f16a1f1c2d1706c
[4] Q3B commit 68301686d36850ba782c4d0f9d58f8c4357e1461

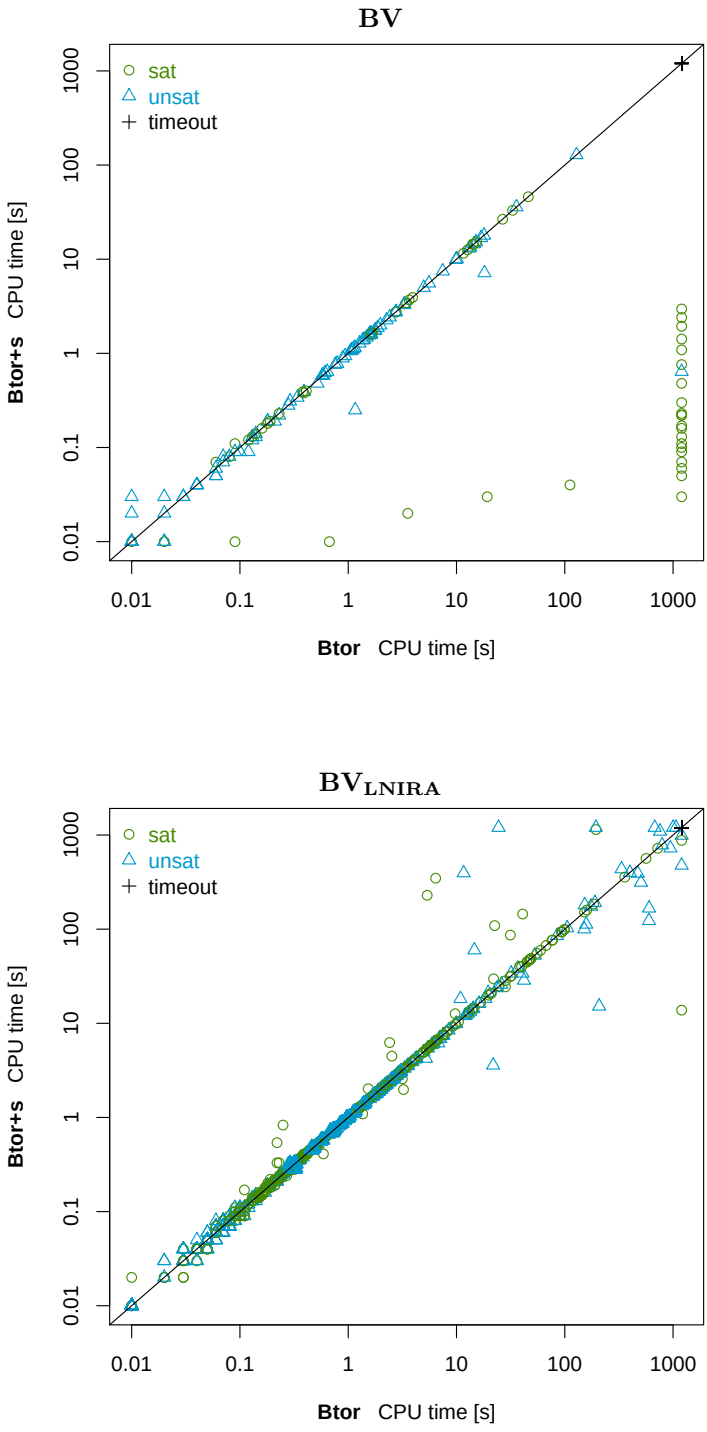


Figure 9.5: Comparison of Boolector with model synthesis enabled (Btor+s) and disabled (Btor) on the BV and BV$_{\text{LNIRA}}$ benchmarks.

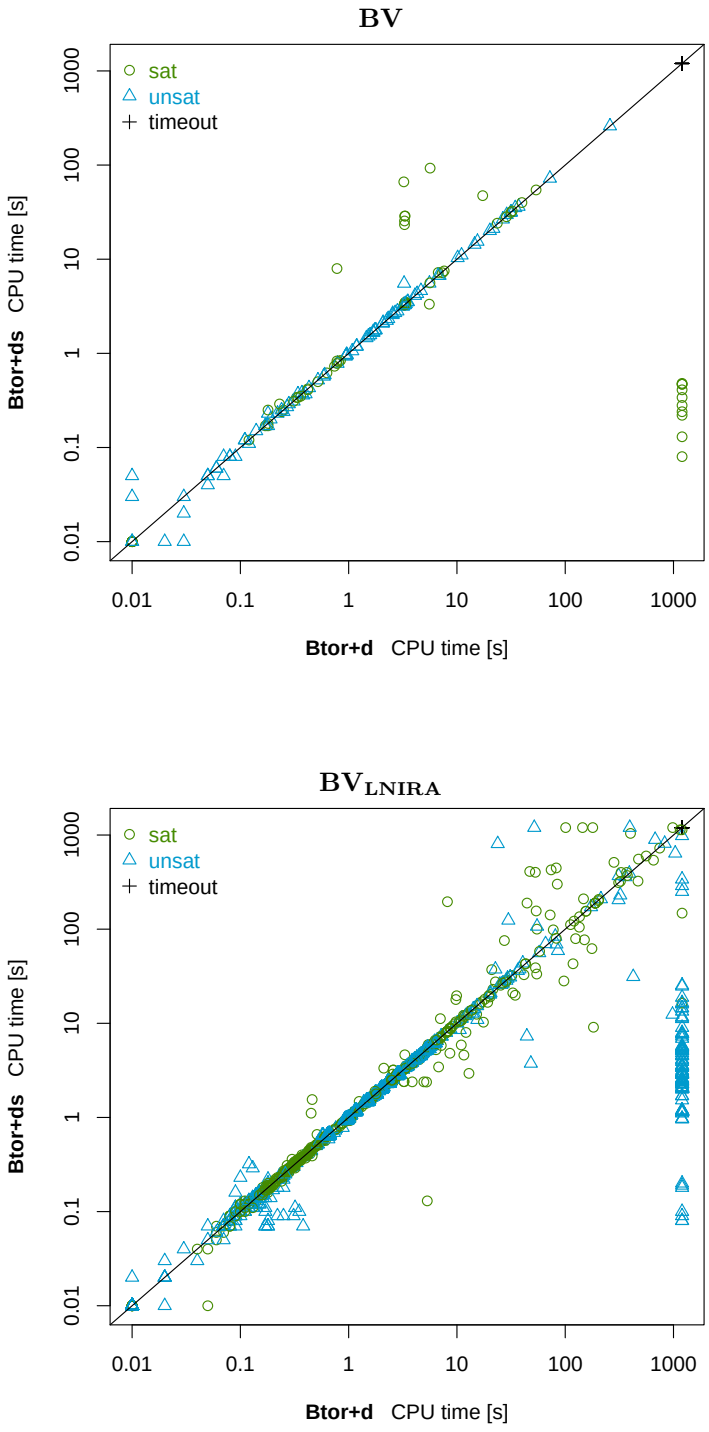


Figure 9.6: Comparison of dual CEGMS with model synthesis enabled (Btor+ds) and disabled (Btor+d) on the BV and BV_{LNIRA} benchmarks.

of CPU time (not wall clock time). In case of a timeout, memory out, or an error, a penalty of 1200 seconds was added to the total CPU time.

Figure 9.5 illustrates the effect of our model synthesis approach by comparing configurations Btor and Btor+s on the BV and BV_{LNIRA} benchmark sets. On the BV benchmark set, Btor+s solves 22 more instances (21 satisfiable, 1 unsatisfiable) compared to Btor. The gain in the number of satisfiable instances is due to the fact that CEGMS is primarily a model finding procedure, which allows to find symbolic models instead of enumerating a possibly large number of bit-vector values, which seems to be crucial on these instances.

On set BV_{LNIRA}, however, Btor+s does not improve the overall number of solved instances, even though it solves two satisfiable instances more than Btor. Note that benchmark set BV_{LNIRA} contains only a small number of satisfiable benchmarks (at most 12% = 575 benchmarks[5]), where configuration Btor already solves 465 instances without enabling model synthesis. For the remaining satisfiable instances, the enumeration space may still be too large to synthesize a model in reasonable time and may require more pruning by introducing more syntactical restrictions for algorithm *enumlearn* as discussed in Section 9.5.

Figure 9.6 shows the effect of model synthesis on the dual configurations Btor+d and Btor+ds on benchmark sets BV and BV_{LNIRA}. On the BV benchmark set, configuration Btor+ds is able to solve 10 more instances of which all are satisfiable. On the BV_{LNIRA} benchmark set, compared to Btor+d, configuration Btor+ds is able to solve 132 more instances of which all are unsatisfiable. The significant increase is due to the successful synthesis of quantifier instantiations (133 cases).

Table 9.1 summarizes the results of all four configurations on both benchmark sets. Configuration Btor+ds clearly outperforms all other configurations w.r.t. the number of solved instances and runtime on both benchmark sets. Out of all 77 (517) satisfiable instances in set BV (BV_{LNIRA}) solved by Btor+ds, 32 (321) were solved by finding a ground conflict in the dual CEGMS approach. In case of configuration Btor+d, out of 67 (518) solved satisfiable instances, 44 (306) were solved by finding a ground conflict in the dual formula. As an interesting observation, 16 (53) of these instances were not solved by Btor. Note, however, that Btor+d is not able to construct a model for these instances due to the current limitations of our dual CEGMS approach as described in Section 9.6.

On the BV benchmark set, model synthesis significantly reduces the number of refinement iterations. Out of 142 commonly solved instances, Btor+s required 165 refinement iterations, whereas Btor required 664 refinements. On the 4522 commonly solved instances of the BV_{LNIRA} benchmark set, Btor+s requires 5249 refinement iterations, whereas Btor requires 5174 refinements. The difference in the number of refinement iterations is due to the fact that enabling model synthesis may produce different counterexamples that requires the CEGMS pro-

[5]Boolector, CVC4, Q3B, and Z3 combined solved 4263 unsatisfiable and 533 satisfiable instances, leaving only 42 instances unsolved

	BV (191)					**BV_{LNIRA}** (4838)				
	Slvd	Sat	Unsat	Time [s]	Uniq	Slvd	Sat	Unsat	Time [s]	Uniq
Btor	142	51	91	59529	0	4527	465	4062	389123	3
Btor+s	164	72	92	32996	0	4526	467	4059	390661	1
Btor+d	162	67	95	35877	0	4572	**518**	4054	342412	4
Btor+ds	**172**	**77**	**95**	**24163**	0	**4704**	517	**4187**	**187411**	**135**

Table 9.1: Results for all configurations on the BV and BV_{LNIRA} benchmarks.

cedure to sometimes create more refinements. However, as noted earlier, enabling model synthesis on set BV_{LNIRA} does not improve the overall number of solved instances in the non-dual case.

We analyzed the terms produced by model synthesis for both Btor+s and Btor+ds on both benchmark sets. On the BV benchmark set, mainly terms of the form $\lambda \mathbf{x}.c$ and $\lambda \mathbf{x}.x_i$ with a bit-vector value c and $x_i \in \mathbf{x}$ have been synthesized. On the BV_{LNIRA} benchmarks, additional terms of the form $\lambda \mathbf{x}.(x_i \ op \ x_j)$, $\lambda \mathbf{x}.(c \ op \ x_i)$, $\lambda \mathbf{x}.\sim(c * x_i))$ and $\lambda \mathbf{x}.(x_i + (c + \sim x_j))$ with a bit-vector operator *op* were synthesized. On these benchmarks, more complex terms did not occur.

Figure 9.7 depicts two cactus plots over the runtime of our best configuration Btor+ds and the solvers CVC4, Q3B, and Z3 on the benchmark sets BV and BV_{LNIRA}. On both benchmark sets, configuration Btor+ds solves the second highest number of benchmarks after Q3B (BV) and Z3 (BV_{LNIRA}). On both benchmark sets, a majority of the benchmarks seem to be trivial since they were solved by all solvers within one second.

Table 9.2 summarizes the results of all solvers on both benchmark sets. On the BV benchmark set, Q3B solves with 187 instances the highest number of benchmarks, followed by Btor+ds with a total of 172 solved instances. Out of all 19 benchmarks unsolved by Btor+ds, 9 benchmarks are solved by Q3B and CVC4 through simplifications only. We expect Boolector to also benefit from introducing quantifier specific simplification techniques, which is left to future work. On the BV_{LNIRA} set, Z3 solves the most instances (4732) and Btor+ds again comes in second with 4704 solved instances. In terms of satisfiable instances, however, Btor+ds solves the highest number of instances (517). In terms of unsatisfiable instances, Z3 clearly has an advantage due to its heuristic quantifier instantiation techniques and solves 69 instances more than Btor+ds, out of which 66 were solved within 3 seconds. The BDD-based approach of Q3B does not scale as well on the BV_{LNIRA} set as on the BV set benchmark set and is even outperformed by Btor+s. Note that most of the benchmarks in BV_{LNIRA} involve more bit-vector arithmetic than the benchmarks in set BV.

Finally, considering Btor+ds, a wall clock time limit of 1200 seconds increases the number of solved instances of set BV_{LNIRA} by 11 (and by 6 for Q3B). On set BV, the number of solved instances does not increase.

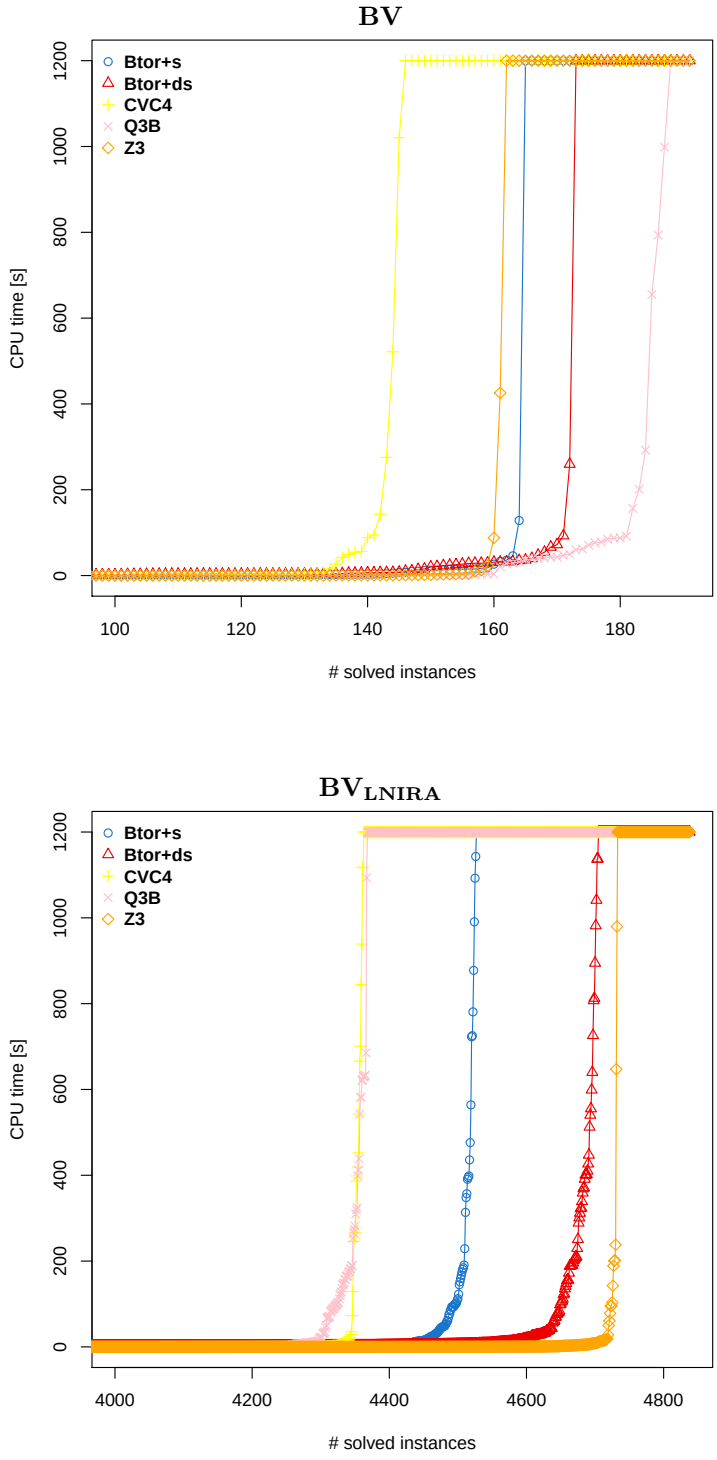


Figure 9.7: Cactus plot of the runtime of all solvers on benchmark sets BV and BV_{LNIRA}.

	BV (191)					**BV_{LNIRA}** (4838)				
	Slvd	Sat	Unsat	Time [s]	Uniq	Slvd	Sat	Unsat	Time [s]	Uniq
Btor+ds	172	77	**95**	24163	2	4704	**517**	4187	187411	**19**
CVC4	145	64	81	57652	0	4362	339	4023	580402	3
Q3B	**187**	**93**	94	**9086**	**9**	4367	327	4040	581252	5
Z3	161	69	92	36593	0	**4732**	476	**4256**	**130405**	11

Table 9.2: Results for all solvers on the BV and BV_{LNIRA} benchmarks with a CPU time limit of 1200 seconds (not wall clock time).

9.8 Conclusion

We presented CEGMS, a new approach for handling quantifiers in SMT, which combines CEGQI with syntax-guided synthesis to synthesize Skolem functions. Further, by exploiting the duality of the input formula dual CEGMS enables us to synthesize terms for quantifier instantiation. We implemented CEGMS in our SMT solver Boolector. Our experimental results show that our technique is competitive with the state-of-the-art in solving quantified bit-vectors even though Boolector does not yet employ any quantifier specific simplification techniques. Such techniques, e.g., miniscoping or DER were found particularly useful in Z3. CEGMS employs two ground theory solvers to reason about arbitrarily quantified formulas. It is a simple yet effective technique, and there is still a lot of room for improvement. Model reconstruction from unsatisfiable dual formulas, symbolic quantifier instantiation by generalizing concrete counterexamples, and the combination of quantified bit-vectors with arrays and uninterpreted functions are interesting directions for future work. It might also be interesting to compare our approach to the work presented in [8, 28, 29, 37].

Binary of Boolector, the set of translated benchmarks (BV_{LNIRA}) and all log files of our experimental evaluation can be found at `http://fmv.jku.at/tacas17`.

Bibliography

[1] Aharon Abadi, Alexander Moshe Rabinovich, and Mooly Sagiv. Decidable fragments of many-sorted logic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, pages 17–31, 2007.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[5] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 236–249, 2002.

[6] Armin Biere and Robert Brummayer. Consistency checking of all different constraints over bit-vectors within a SAT solver. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–4, 2008.

[7] Nikolaj Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 316–330, 2010.

[8] Nikolaj Bjørner and Mikolás Janota. Playing with quantified satisfaction. In *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015.*, pages 15–27, 2015.

[9] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006.

[11] Robert Brummayer and Armin Biere. Lemmas on Demand for the Extensional Theory of Arrays. *JSAT*, 6(1-3):165–201, 2009.

[12] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 78–92, 2002.

[13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, pages 93–107, 2013.

[15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[16] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 236–250, 2010.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. ETAPS'08*, pages 337–340, 2008.

[18] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT'02), Cincinnati, USA*, 2002.

[19] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pages 337–340, 2008.

[20] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, pages 438–455, 2002.

[21] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[22] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 737–744, 2014.

[23] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[24] Bruno Dutertre. Solving exists/forall problems in yices. *Workshop on Satisfiability Modulo Theories*, 2015.

[25] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[26] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[27] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, pages 108–128, 2013.

[28] Azadeh Farzan and Zachary Kincaid. Linear arithmetic satisfiability via strategy improvement. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 735–743, 2016.

[29] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Automated discovery of simulation between programs. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 606–621, 2015.

[30] R.A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, 1953.

[31] V. Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Computer Science Department, Stanford University, 2007.

[32] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 519–531, 2007.

[33] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.

[34] Alexandra Goultiaeva and Fahiem Bacchus. Exploiting QBF duality on a circuit representation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[35] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 120–135, 2009.

[36] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 117–124, 2006.

[37] Ajith K. John and Supratik Chakraborty. A layered algorithm for quantifier elimination from linear modular constraints. *Formal Methods in System Design*, 49(3):272–323, 2016.

[38] Martin Jonás and Jan Strejcek. Solving quantified bit-vector formulas using binary decision diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 267–283, 2016.

[39] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 44–56, 2012.

[40] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.

[41] Florian Lapschies, Jan Peleska, and Elena Gorbachuk. System Description: SONOLAR SMT-COMP 2014. `http://smtcomp.sourceforge.net/2014/systemDescriptions/sonolar-smtcomp2014.pdf`, 2014.

[42] Florian Lapschies, Jan Peleska, Elena Gorbachuk, and Tatiana Mangels. SONOLAR SMT-Solver. System Desc. SMT-COMP'12. `http://smtcomp.sourceforge.net/2012/reports/sonolar.pdf`, 2012.

[43] John McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.

[44] David Monniaux. Quantifier elimination by lazy model enumeration. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 585–599, 2010.

[45] Aina Niemetz, Mathias Preiner, and Armin Biere. Turbo-charging lemmas on demand with don't care reasoning. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 179–186, 2014.

[46] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[47] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[48] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [56], pages 335–367.

[49] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[50] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 53–68, 2013.

[51] Mathias Preiner, Aina Niemetz, and Armin Biere. Lemmas on demand for lambdas. In *Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems, Portland, OR, USA, October 19, 2013.*, 2013.

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

[53] Mathias Preiner, Aina Niemetz, and Armin Biere. Counterexample-guided model synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, to appear*, 2017.

[54] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 198–216, 2015.

[55] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstic. Finite model finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 640–655, 2013.

[56] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[57] Roberto Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4):141–224, 2007.

[58] Sanjit A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, CMU, 2005.

[59] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.

[60] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 29–37, 2001.

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

[62] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296, 2013.

[63] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 239–246, 2010.