Lemmas on Demand for Lambdas

Mathias Preiner, Aina Niemetz, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

***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-LIB v2 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.**

I. Introduction

The theory of arrays as axiomatized by McCarthy [14] 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. [4] introduced an approach to overcome these limitations by using restricted λ-terms to model array expressions (such as *memset* and *memcpy*), ordered data structures and partially interpreted functions within the SMT solver UCLID [17]. The SMT solver UCLID employs an eager SMT solving approach and therefore eliminates all λ-terms through β-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 λ-terms such as CVC4 [1] or Yices [8] also treat them eagerly, similarly to UCLID, and eliminate all occurrences of λ-terms by substituting them with their instantiated function body (cf. C-style macros). Eagerly eliminating λ-terms via β-reduction, however, may result in an exponential blow-up in the size of the formula [17]. Recently, an extension of the theory of arrays was proposed [10], which uses λ-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 λ-terms provided by an SMT solver. Instead, it reduces instances in the theory of arrays with λ-terms to a theory combination supported by solvers such as Boolector [3] (without native support for λ-terms), CVC4, STP [12], and Z3 [6].

In this paper, we generalize the decision procedure for the theory of arrays with bit vectors as introduced in [3] to lazily handle non-recursive and non-extensional λ-terms. We show how arrays, array operations and SMT-LIB v2 macros are represented in Boolector as λ-terms and introduce a lemmas on demand procedure for lazily handling λ-terms in Boolector in detail. We summarize an experimental evaluation and compare our results to solvers with SMT-LIB v2 macro support (CVC4, MathSAT [5], SONOLAR [13] and Z3) and finally, give an outlook on future work.

II. 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 $ite(\top,t,e) =_n t$ and $ite(\bot,t,e) =_n e$. As a notational convention, the subscript might be omitted in the following. We identify *read* and *write* as basic operations on array elements, where $read(a,i)$ denotes the value of array a at index i, and $write(a,i,e)$

This work was funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

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 [14]:

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

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

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

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:

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

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 λ-calculus, assuming the usual notation and terminology, including the notion of *function application*, *currying* and β*-reduction*. In general, we denote a λ-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 λ-term is restricted to exactly one. Functions with more than one parameter are transformed into a chain of nested λ-terms by means of *currying* (e.g. $f(x,y) := x+y$ is rewritten as $\lambda x \,.\, \lambda y \,.\, x+y$). As a notational convention, we will use $\lambda_{\bar{x}}$ as a shorthand for $\lambda x_0 \ldots \lambda x_k \,.\, t(x_0,\ldots,x_k)$ for $k \geq 0$. We identify the *function application* as an explicit operation on λ-terms and interpret it as instantiating a bound variable (all bound variables) of a λ-term (a curried λ-chain). We interpret β*-reduction* as a form of function application, where all formal parameter variables (bound variables) are substituted with their actual parameter terms. We will use $\lambda_{\bar{x}}[x_0 \backslash a_0, \ldots, x_n \backslash a_n]$ to indicate β-reduction of a λ-term $\lambda_{\bar{x}}$, where the formal parameters $x_0,\ldots,x_n$ are substituted with the actual argument terms $a_0,\ldots,a_n$.

III. λ-TERMS IN BOOLECTOR

In contrast to λ-term handling in other SMT solvers such as e.g. UCLID or CVC4, where λ-terms are eagerly eliminated, in Boolector we provide a lazy λ-term handling with *lemmas on demand*. We generalized the lemmas on demand decision procedure for the extensional theory of arrays introduced in [3] to handle lemmas on demand for λ-terms as follows.

In order to provide a uniform handling of arrays and λ-terms within Boolector, we generalized all arrays (and array operations) to λ-terms (and operations on λ-terms) by representing *array variables* as uninterpreted functions (UF), *read* operations as function applications, and *write* and *if-then-else* operations on arrays as λ-terms. We further interpret macros (as provided by the command *define-fun* in the SMT-LIB v2 format) as (curried) λ-terms. Note that in contrast to [3], our implementation currently does not support extensionality (equality) over arrays (λ-terms).

We represent an *array* as exactly one λ-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 λ-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* λ-term (e.g. $\lambda j \,.\, 0$). In contrast to SMT-LIB v2 macros, it is not required to represent arrays with curried λ-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 λ-terms with read index i as argument and represent them as $read(a,i) \equiv f_a(i)$ and $read(\lambda j \,.\, t(j), i) \equiv (\lambda j \,.\, t(j))(i)$, respectively.

We interpret *write* operations as λ-terms modeling the result of the *write* operation on array a at index i with value e, and represent them as $write(a,i,e) \equiv \lambda j \,.\, ite(i = j, e, f_a(j))$. A function application on a λ-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 β-reduction to a λ-term λ_w yields the same behaviour 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), β-reduction yields the written value e, whereas if $k \neq i$ (A3), β-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 [17].

We interpret *if-then-else* operations on arrays a and b as λ-terms, and represent them as $ite(c,a,b) \equiv \lambda\, j\, .\, ite(c, f_a(j), f_b(j))$. 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, λ-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 λ-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 λ-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 [10]. See also [17] for more examples. For instance, the *memset* with signature *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 $\lambda\, j\, .\, ite(i \leq j \wedge j < i+n, e, f_a(j))$. Note, n can be symbolic, and does not have to be a constant. In the same way, *memcpy* with signature *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 $\lambda\, j\, .\, ite(k \leq j \wedge j < k+n, f_a(i+j-k), f_b(j))$. 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 λ-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 λ-term handling in Boolector as simple as possible, whereas the second restriction limits λ-term handling in Boolector to non-higher order functions. Relaxing these restrictions will turn the considered λ-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.

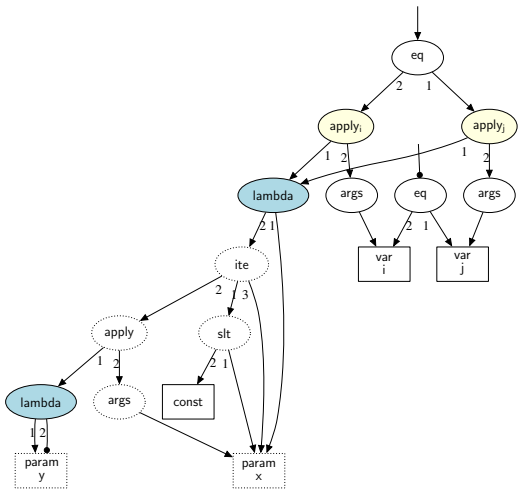


Fig. 1: DAG representation of formula ψ_1.

In contrast to treating SMT-LIB v2 macros as C-style macros, i.e., substituting every function application with the instantiated function body, in Boolector, we directly translate SMT-LIB v2 macros into λ-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 λ-terms and operations on λ-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 λ-term
- a *lambda* expression binds exactly one *param* expression, which may occur in a bit vector expression that represents the body of the λ-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

Example 1: Consider $\psi_1 \equiv f(i) = f(j) \wedge i \neq j$ with functions $f(x) := ite(x < 0, g(x), x)$, $g(y) := -y$ as depicted in Fig. 1. Both functions are represented as λ-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 β-reduction is applied.

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

IV. β-REDUCTION

In this section we discuss how concepts from the λ-calculus have been adapted and implemented in

our SMT solver Boolector. We focus on reduction algorithms for the non-recursive λ-calculus, which is rather atypical for the (vast) literature on λ-calculus. In the context of Boolector, we distinguish between *full* and *partial* β-reduction. They mainly differ in their application and the depth up to which λ-terms are expanded. In essence, given a function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$ *partial* β-reduction reduces only the top-most λ-term $\lambda_{\bar{x}}$, whereas *full* β-reduction reduces $\lambda_{\bar{x}}$ and every λ-term in the scope of $\lambda_{\bar{x}}$.

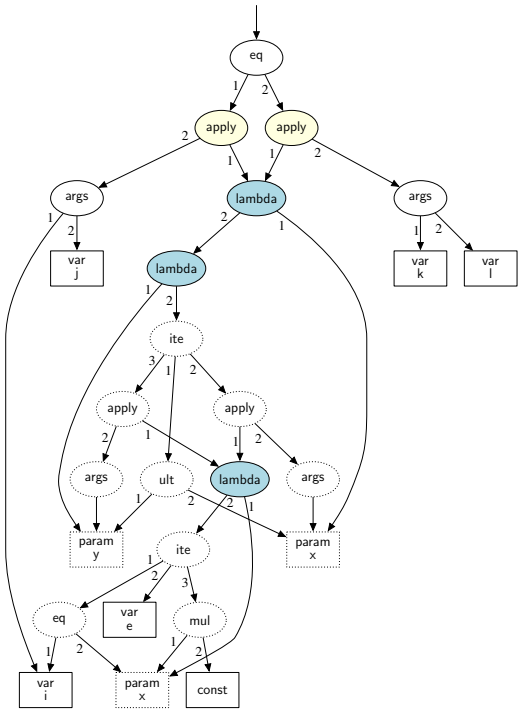
Full β-reduction of a function application on λ-term $\lambda_{\bar{x}}$ consists of a series of β-reductions, where λ-term $\lambda_{\bar{x}}$ and every λ-term $\lambda_{\bar{y}}$ within the scope of $\lambda_{\bar{x}}$ are instantiated, substituting all formal parameters with actual parameter terms. Since we do not allow partial function applications, full β-reduction guarantees to yield a term which is free of λ-terms. Given a formula with λ-terms, we usually employ full β-reduction in order to eliminate all λ-terms by substituting every function application with the term obtained by applying full β-reduction on that function application. In the worst case, full β-reduction results in an exponential blow-up. However, in practice, it is often beneficial to employ full β-reduction, since it usually leads to significant simplifications through rewriting. In Boolector, we incorporate this method as an optional rewriting step. We will use $\lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{f}}$ as a shorthand for applying full β-reduction to $\lambda_{\bar{x}}$ with arguments $a_0,\ldots,a_n$.

Partial β-reduction of a λ-term $\lambda_{\bar{x}}$, on the other hand, essentially works in the same way as what is referred to as β-reduction in the λ-calculus. Given a function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$, partial β-reduction substitutes formal parameters $x_0,\ldots,x_n$ with the actual argument terms $a_0,\ldots,a_n$ without applying β-reduction to other λ-terms within the scope of $\lambda_{\bar{x}}$. This has the effect that λ-terms are expanded function-wise, which we require for consistency checking. In the following, we use $\lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$ to denote the application of partial β-reduction to $\lambda_{\bar{x}}$ with arguments $a_0,\ldots,a_n$.

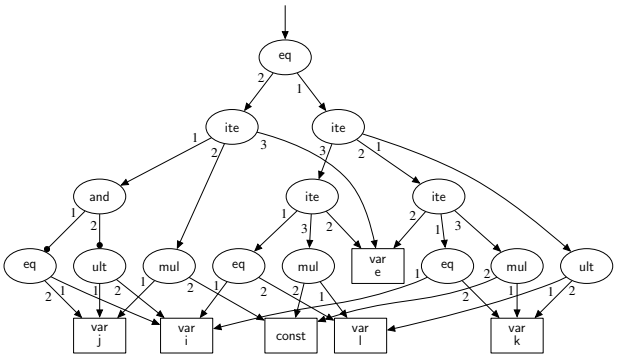
A. Full β-reduction

Given a function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$ and a DAG representation of $\lambda_{\bar{x}}$. Full β-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 $a_0,\ldots,a_n$ to the formal parameters $x_0,\ldots,x_n$.
2) While traversing down, for any λ-term $\lambda_{\bar{y}}$ in the scope of $\lambda_{\bar{x}}$, we need special handling for each function application $\lambda_{\bar{y}}(b_0,\ldots,b_m)$ as follows.



(a) Original formula ψ_2.



(b) Formula ψ_2' after full β-reduction of ψ_2.

Fig. 2: Full β-reduction of formula ψ_2.

   a) Visit arguments $b_0,\ldots,b_m$ first, and obtain rebuilt arguments $b_0',\ldots,b_m'$.
   b) Assign rebuilt arguments $b_0',\ldots,b_m'$ to $\lambda_{\bar{y}}$ and apply β-reduction to $\lambda_{\bar{y}}(b_0',\ldots,b_m')$.

   This ensures a bottom-up construction of the β-reduced DAG (see step 3.), since all arguments $b_0',\ldots,b_m'$ passed to a λ-term $\lambda_{\bar{y}}$ are β-reduced and rebuilt prior to applying β-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 $\lambda_{\bar{y}}(b_0,\ldots,b_m)$ with $\lambda_{\bar{y}}[y_0 \backslash b_0',\ldots,y_m \backslash b_m']_{\mathbf{f}}$

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

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

Example 2: Given a formula $\psi_2 \equiv f(i,j) = f(k,l)$ and two functions $g(x) := ite(x = i, e, 2 * x)$ and $f(x,y) := ite(y < x, g(x), g(y))$ as depicted in Fig. 2a. Applying full β-reduction to formula ψ_2 yields formula ψ_2' as illustrated in Fig. 2b. Function application $f(i,j)$ has been reduced to $ite(j \geq i \wedge i \neq j, 2 * j, e)$ and $f(k,l)$ to $ite(l < k, ite(k = i, e, 2 * k), ite(l = i, e, 2 * l))$.

B. Partial β-reduction

Given a function application $\lambda_{\bar{x}}(a_0, \ldots, a_n)$ and a DAG representation of $\lambda_{\bar{x}}$. The scope of a partial β-reduction $\beta_p(\lambda_{\bar{x}})$ is defined as the sub-DAG obtained by cutting off all λ-terms in the scope of $\lambda_{\bar{x}}$. Assume that we have an assignment for arguments $a_0, \ldots, a_n$, and for all non-parameterized expressions in the scope of $\beta_p(\lambda_{\bar{x}})$. The partial β-reduction algorithm substitutes *param* expressions $x_0, \ldots, x_n$ with $a_0, \ldots, a_n$ and rebuilds $\lambda_{\bar{x}}$. Similar to full β-reduction, we perform a DFS post-order traversal of the DAG as follows.

1) Initially, we instantiate $\lambda_{\bar{x}}$ by assigning arguments $a_0, \ldots, a_n$ to the formal parameters $x_0, \ldots, x_n$.
2) While traversing down the DAG, we require special handling for the following expressions:
   - function applications $\lambda_{\bar{y}}(b_0, \ldots, b_m)$
     a) Visit arguments $b_0, \ldots, b_m$, obtain rebuilt arguments $b_0', \ldots, b_m'$.
     b) Do not assign rebuilt arguments $b_0', \ldots, b_m'$ 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'

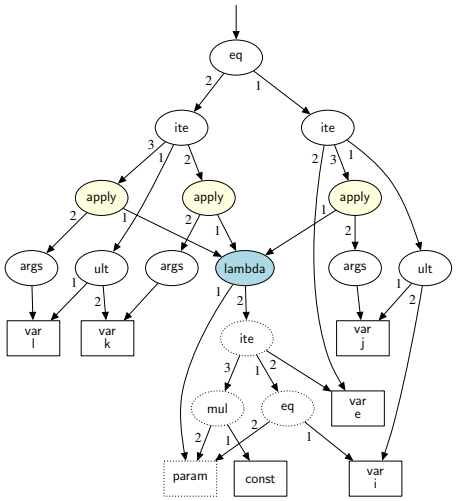


Fig. 3: Partial β-reduction of formula ψ_2.

   - *if-then-else*: substitute expression $ite(c, t_1, t_2)$ with t_1 if $c = \top$, and t_2 otherwise

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

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

Example 3: Consider ψ_2 from Ex. 2. Applying partial β-reduction to ψ_2 yields the formula depicted in Fig. 3, where function application $f(i,j)$ has been reduced to $ite(j < i, e, g(j))$ and $f(k,l)$ to $ite(l < k, g(k), g(l))$.

V. Decision Procedure

The idea of *lemmas on demand* goes back to [7] and actually represents one extreme variant of the lazy SMT approach [16]. Around the same time, a related technique was developed in the context of bounded model checking [9], which lazily encodes all-different constraints over bit vectors (see also [2]). In constraint programming the related technique of lazy clause generation [15] is effective too.

In this section, we introduce lemmas on demand for non-recursive λ-terms based on the algorithm introduced in [3]. A top-level view of our lemmas on demand decision procedure for λ-terms (DP_λ) is illustrated in Fig. 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

```
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))
```

Fig. 4: Lemmas on demand for λ-terms DP_λ.

its satisfiability by means of a SAT solver. As Γ is an over-approximation of ϕ, we immediately conclude 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, $consistent_\lambda(\pi,\sigma)$ identifies a violation of the function congruence axiom EUF, and we generate a symbolic lemma $lemma_\lambda(\pi,\sigma)$ which is added to formula refinement ξ in its abstracted form $\alpha_\lambda(lemma_\lambda(\pi,\sigma))$.

Note that in ϕ, in contrast to the decision procedure introduced in [3], all array variables and array operations in the original input have been abstracted away and replaced by corresponding λ-terms and operations on λ-terms. Hence, various integral components of the original procedure (α_λ, $consistent_\lambda$, $lemma_\lambda$) have been adapted to handle λ-terms as follows.

VI. Formula Abstraction

In this section, we introduce a partial formula abstraction function α_λ as a generalization of the abstraction approach presented in [3]. Analogous to [3], 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 $\lambda_{\bar{x}}(a_0,\ldots,a_n)$ is mapped to a fresh bit vector variable.
3) Each bit vector term $t(y_0,\ldots,y_m)$ is mapped to $t(\alpha_\lambda(y_0),\ldots,\alpha_\lambda(y_m))$.

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

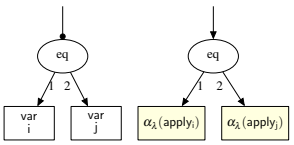


Fig. 5: Formula abstraction $\alpha_\lambda(\psi_1)$.

Example 4: Consider formula ψ_1 from Ex. 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 Fig. 5.

VII. Consistency Checking

In this section, we introduce a consistency checking algorithm $consistent_\lambda$ as a generalization of the consistency checking approach presented in [3]. However, in contrast to [3], we do not propagate so-called access nodes but function applications and further check axiom EUF (while applying partial β-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 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 β-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 β-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 λ-terms.

In order to check axiom EUF, for each λ-term and UF symbol we maintain a hash table ρ, which maps λ-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 $s := g(\bar{a})$, $t := h(\bar{b}) \in \rho(f)$ check
$\bigwedge_{i=0}^{n} \sigma(\alpha_\lambda(a_i)) = \sigma(\alpha_\lambda(b_i)) \rightarrow \sigma(\alpha_\lambda(s)) = \sigma(\alpha_\lambda(t))$

B: For any $s := \lambda_{\bar{y}}(a_0,\ldots,a_n) \in \rho(\lambda_{\bar{x}})$ with
$t := \lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$,
check rule **P**, if **P** fails, check $eval(t) = \sigma(\alpha_\lambda(s))$

P: For any $s := \lambda_{\bar{y}}(a_0,\ldots,a_n) \in \rho(\lambda_{\bar{x}})$ with
$t := g(b_0,\ldots,b_m) = \lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$,
if $n = m \wedge \bigwedge_{i=0}^{n} a_i = b_i$, propagate s to $\rho(g)$

Given a λ-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 $g(a_0,\ldots,a_n)$ 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 β-reduced term $\lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$. Finally, rule **P** represents a crucial optimization of *consistent*$_\lambda$, as it avoids unnecessary conflicts while checking **B**. If **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 $s = \lambda_{\bar{x}}(a_0,\ldots,a_n)$. Therefore, function applications s and t must produce the same function value as $t := \lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}} = \lambda_{\bar{y}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$, i.e., function application t must be equal to the result of applying partial β-reduction to function application s. Assume we encode t and add it to the formula. If 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 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 $(g, f(a_0,\ldots,a_n))$, where f and g are initially equal and $f(a_0,\ldots,a_n)$ denotes the function application propagated to function g. In the main consistency checking

```
procedure consistent_λ(π, σ)
    S ← nonparam_apps(π)
    while S ≠ ∅
        (g, f(a_0,...,a_n)) ← pop(S)
        encode(f(a_0,...,a_n))
        /* check rule C */
        if not congruent(g, f(a_0,...,a_n))
            return ⊥
        add(f(a_0,...,a_n), ρ(g))
        if is_UF(g) continue
        encode(g)
        /* check rule B */
        t ← g[x_0\a_0,...,x_n\a_n]_p
        if assigned(t)
            if σ(t) ≠ σ(α_λ(f(a_0,...,a_n)))
                return ⊥
        elif t = h(a_0,...,a_n) /* check rule P */
            push(S, (h, f(a_0,...,a_n)))
            continue
        else
            apps ← fresh_apps(t)
            for a ∈ apps
                encode(a)
            if eval(t) ≠ σ(α_λ(f(a_0,...,a_n)))
                return ⊥
            for h(b_0,...,b_m) ∈ apps
                push(S, (h, h(b_0,...,b_m)))
    return ⊤
```

Fig. 6: Procedure *consistent*$_\lambda$ in pseudo-code.

loop, we check rules **C** and **B** for each tuple as follows. First we check if $f(a_0,\ldots,a_n)$ 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 $a_0,\ldots,a_n$, 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 λ-term we still need to check rule **B**, i.e., we need to check if the assignment $\sigma(\alpha_\lambda(f(a_0,\ldots,a_n)))$ is consistent with the value produced by $g[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$. Therefore, we first encode all non-parameterized expressions in the scope of partial β-reduction $\beta_p(g)$ (cf. `encode(`g`)`) before applying partial β-reduction with arguments $a_0,\ldots,a_n$, which yields term t. If term t has an assignment, we can immediately check if it differs from assignment $\sigma(\alpha_\lambda(f(a_0,\ldots,a_n)))$ 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 $f(a_0,\ldots,a_n)$ 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. $\texttt{eval}(t)$) and compare it to the assignment of $\alpha_\lambda(f(a_0,\ldots,a_n))$. 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.

A. Lemma generation

Following [3], 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 $s := g(a_0,\ldots,a_n)$, $t := h(b_0,\ldots,b_n) \in \rho(f)$. 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 $c_0^s,\ldots,c_j^s$ ($c_0^t,\ldots,c_l^t$) on path p^s (p^t) that were $\top$ under given assignment σ.
3) Collect all *ite* conditions $c_0^s,\ldots,c_k^s$ ($c_0^t,\ldots,c_m^t$) on path p^s (p^t) that were $\bot$ under given assignment σ.

We generate the following (in general symbolic) lemma:

$$\bigwedge_{i=0}^{j} c_i^s \wedge \bigwedge_{i=0}^{k} \neg c_i^s \wedge \bigwedge_{i=0}^{l} c_i^t \wedge \bigwedge_{i=0}^{m} \neg c_i^t \wedge \bigwedge_{i=0}^{n} a_i = b_i \rightarrow s = t$$

Assume that rule **B** was violated by a function application $s := \lambda_{\bar{y}}(a_0,\ldots,a_n) \in \rho(\lambda_{\bar{x}})$. We obtained $t := \lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]_{\mathbf{p}}$ and collect all conditions that lead to the conflict as follows.

1) Collect *ite* conditions $c_0^s,\ldots,c_j^s$ and $c_0^s,\ldots,c_k^s$ for s as in steps 1-3 above.
2) Collect all *ite* conditions $c_0^t,\ldots,c_l^t$ that evaluated to $\top$ under current assignment σ when partially β-reducing $\lambda_{\bar{x}}$ to obtain t.
3) Collect all *ite* conditions $c_0^t,\ldots,c_m^t$ that evaluated to $\bot$ under current assignment σ when partially β-reducing $\lambda_{\bar{x}}$ to obtain t.

We generate the following (in general symbolic) lemma:

$$\bigwedge_{i=0}^{j} c_i^s \wedge \bigwedge_{i=0}^{k} \neg c_i^s \wedge \bigwedge_{i=0}^{l} c_i^t \wedge \bigwedge_{i=0}^{m} \neg c_i^t \rightarrow s = t$$

Example 5: Consider formula ψ_1 and its preprocessed formula abstraction $\alpha_\lambda(\psi_1)$ from Ex. 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 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 β-reduction and obtain term $t := \lambda_x[x/i]_{\mathbf{p}} = \lambda_y(i)$ (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 β-reduction, obtain $t := \lambda_y[y/i]_{\mathbf{p}} = -i$ 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 β-reduction and obtain term $t := \lambda_x[x/j]_{\mathbf{p}} = j$ (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 $j > 0 \rightarrow a_j = j$.

VIII. Experiments

We applied our lemmas on demand approach for λ-terms on three different benchmark categories: (1) *crafted*, (2) *SMT'12*, and (3) *application*. For the *crafted* category, we generated benchmarks using SMT-LIB v2 macros, where the instances of the first benchmark set (*macro blow-up*) tend to blow up in formula size if SMT-LIB v2 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 [11], where the instances of the *fisher-yates SAT* also tend to blow up in the size of the formula if SMT-LIB v2 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 [10]. The authors also kindly provided the same benchmark family using λ-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 [10]

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

	Solver	Solved	TO	MO	Time [10^3s]	Space [GB]
macro blow-up	Boolector	100	0	0	24.2	9.4
	Boolector$_{nop}$	100	0	0	18.2	8.4
	Boolector$_{\beta}$	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	Boolector	7	10	1	14.0	7.5
	Boolector$_{nop}$	4	13	1	17.3	7.0
	Boolector$_{\beta}$	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	Boolector	5	13	1	17.4	7.1
	Boolector$_{nop}$	4	14	1	18.2	6.9
	Boolector$_{\beta}$	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 I: Results *crafted* benchmark.

we used a time limit of 60 seconds. We evaluated four different versions of Boolector: (1) our lemmas on demand for λ-terms approach DP_λ (Boolector), (2) DP_λ without optimization rule **P** (Boolector$_{nop}$), (3) DP_λ with full β-reduction (Boolector$_{\beta}$), and (4) the version submitted to the SMT competition 2012 (Boolector$_{sc12}$). 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-LIB v2 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 λ-term support, but lack support for the SMT-LIB v2 standard.

As indicated in Tables I, II and III, 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 I summarizes the results of the *crafted* benchmark category. On the *macro blow-up* benchmarks, Boolector and Boolector$_{nop}$ benefit from lazy λ-term handling and thus, outperform all those solvers which try to eagerly eliminate SMT-LIB v2 macros with a very high memory consumption as a result. The only solver not having memory problems on this benchmark set is SONOLAR. However, it is not clear how SONOLAR handles SMT-LIB v2 macros. Surprisingly, on these benchmarks Boolector$_{nop}$ performs better than Boolector with optimization rule **P**, which needs further investigation. On the *fisher-yates SAT* benchmarks Boolector not only solves the most instances, but requires 107 seconds for the first 6 instances, for which Boolector$_{\beta}$, MathSAT and Z3 need more than 300 seconds each. Boolector$_{nop}$ does not perform as well as Boolector 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 Boolector$_{\beta}$ solve the most instances, whereas Boolector and Boolector$_{nop}$ 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 β-reduction does not yield as much simplifications. We measured overhead of β-reduction in Boolector 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 run time), whereas for the *fisher-yates SAT* instances β-reduction requires over 50% of total run time.

	Solver	Solved	TO	MO	Time [10^3s]	Space [GB]
SMT'12	Boolector	139	10	0	19.9	14.8
	Boolector$_{nop}$	134	15	0	26.3	14.5
	Boolector$_{\beta}$	137	11	1	21.5	22.7
	Boolector$_{sc12}$	140	9	0	15.9	10.3

TABLE II: Results *SMT'12* benchmark.

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

Finally, Table III summarizes the results of the *application* category. We used the benchmarks obtained from the instantiation-based reduction approach presented in [10] (*instantiation* benchmarks) and compared our

	Solver	Solved	TO	MO	Time [s]	Space [MB]
instantiation	Boolector	37	8	0	576	235
	Boolector$_{nop}$	35	10	0	673	196
	Boolector$_{\beta}$	44	1	0	138	961
	Boolector$_{sc12}$	39	6	0	535	308
	STP	44	1	0	141	3814
lambda	Boolector	37	8	0	594	236
	Boolector$_{nop}$	35	10	0	709	166
	Boolector$_{\beta}$	45	0	0	52	676
	Boolector$_{sc12}$	-	-	-	-	-
	STP	-	-	-	-	-

TABLE III: Results *application* benchmarks.

new approaches to STP, the same version of the solver that outperformed all other solvers on these benchmarks in the experimental evaluation of [10]. On the *instantiation* benchmarks Boolector$_{\beta}$ and STP solve the same number of instances in roughly the same time. However, Boolector$_{\beta}$ requires less memory for solving those instances. Boolector, Boolector$_{nop}$ and Boolector$_{sc12}$ did not perform so well on these benchmarks because in contrast to Boolector$_{\beta}$ 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 λ-terms for representing arrays. On these benchmarks, Boolector$_{\beta}$ clearly outperforms Boolector and Boolector$_{nop}$ and solves all 45 instances within a fraction of time. Boolector$_{sc12}$ and STP do not support λ-terms as arrays and therefore were not able to participate on this benchmark set. By exploiting the native λ-term support for arrays in Boolector$_{\beta}$, in comparison to the *instantiation* benchmarks we achieve even better results. Note that on the *lambda* (*instantiation*) benchmarks, the overhead in Boolector$_{\beta}$ for applying full β-reduction was around 15% (less than 2%) of the total run time.

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.

IX. Conclusion

In this paper, we introduced a new decision procedure for handling non-recursive and non-extensional λ-terms as a generalization of the array decision procedure presented in [3]. We showed how arrays, array operations and SMT-LIB v2 macros are represented in Boolector and evaluated our new approach with 3 different benchmark categories: *crafted*, *SMT'12* and *application*. The *crafted* category showed the benefit of lazily handling SMT-LIB v2 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 λ-term support within an SMT solver. Our experiments look promising even though we employ a rather naive implementation of β-reduction in Boolector and also do not incorporate any λ-term specific rewriting rules except full β-reduction.

In future work we will address the performance bottleneck of the β-reduction implementation and will further add λ-term specific rewriting rules. We will analyze the impact of various β-reduction strategies on our lemmas on demand procedure and will further add support for extensionality over λ-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 λ-terms.

X. 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.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
[2] A. Biere and R. Brummayer. Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver. In *FMCAD*, pages 1–4. IEEE, 2008.
[3] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. *JSAT*, 6(1-3):165–201, 2009.
[4] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *CAV*, volume 2404 of *LNCS*, pages 78–92. Springer, 2002.
[5] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
[6] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. ETAPS'08*, pages 337–340, 2008.
[7] L. M. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *CADE*, volume 2392 of *LNCS*. Springer, 2002.
[8] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, Aug. 2006.
[9] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *ENTCS*, 89(4):543–560, 2003.
[10] S. Falke, F. Merz, and C. Sinz. Extending the Theory of Arrays: memset, memcpy, and Beyond. In *Proc. VSTTE'13*.
[11] R. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, 1953.
[12] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. CAV'07*. Springer-Verlag, 2007.
[13] F. Lapschies, J. Peleska, E. Gorbachuk, and T. Mangels. SONOLAR SMT-Solver. System Desc. SMT-COMP'12. http://smtcomp.sourceforge.net/2012/reports/sonolar.pdf, 2012.
[14] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.
[15] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
[16] R. Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4):141–224, 2007.
[17] S. A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, CMU, 2005.