Generalized, Efficient Array Decision Procedures

Leonardo de Moura, Nikolaj Bjørner

Abstract—**The theory of arrays is ubiquitous in the context of software and hardware verification and symbolic analysis. The basic array theory was introduced by McCarthy and allows to symbolically representing array updates. In this paper we present *combinatory array logic*, CAL, using a small, but powerful core of combinators, and reduce it to the theory of uninterpreted functions. CAL allows expressing properties that go well beyond the basic array theory. We provide a new efficient decision procedure for the base theory as well as CAL. The efficient procedure serves a critical role in the performance of the state-of-the-art SMT solver Z3 on array formulas from applications.**

I. Introduction

As part of formulating a programme of a mathematical theory of computation McCarthy [1] proposed a basic theory of arrays. The basic theory characterizes functions *store* and the binary selector $\_[\_]$ using two axioms: $\forall a, i, v \ . \ store(a, i, v)[i] \simeq v$ and $\forall a, i, j, v \ . \ i \simeq j \lor store(a, i, v)[j] \simeq a[j]$.

In this paper we develop an efficient saturation procedure for the basic (extensional) array theory as well as a powerful extension that we call *combinatory array logic* (CAL). Besides the *store* combinator the extension uses two new main combinators K (inspired by combinatory logic) and map_f (that maps f on arrays[1]). They have the characteristics:

$$K(v)[i] = v$$
$$map_f(a_1, \ldots, a_n)[i] = f(a_1[i], \ldots, a_n[i])$$

Ground satisfiability in the resulting theory is shown to be NP-complete. Our procedures are presented as inference rules. A useful contribution of this paper is strong filters for restricting the application of these rules while retaining completeness. The results are developed in the context of strongly disjoint theories, where finite domains are easy to handle. We show how default values of arrays can be reflected back into the array theory, but this construction is very sensitive to domain sizes. A technical report, available from the authors, describes CAL with the identity operator I.

The ideas described in this paper were already implemented in the version of the SMT solver Z3 submitted to the SMT 2008 (http://www.smtcomp.org). Z3 won the QF_AUFLIA division (arrays, uninterpreted functions and linear integer arithmetic), and was 25 times faster than the second place (Barcelogic). In the QF_A division, Z3 finished in second place, but this division consisted only of trivial artificial problems. The winner (Barcelogic) total runtime was 13.5 secs and Z3 was 17.3 secs. In Section VI, we also compare the performance of Z3 with and without using some of the proposed filters.

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA. {leonardo, nbjorner}@microsoft.com

[1] It is similar to Schönfinkel/Curry's B combinator, but not the well-known S combinator.

A. Related Work

Decision procedures for non-extensional theories of arrays with Presburger arithmetic constraints appeared in the early 80's [2], [3]. The theory remains important in the context of formal verification of hardware [4], [5].

A decision procedure for the theory of extensional arrays is given in [6]. It uses constrained equations between arrays to capture when arrays are equal except possibly on a finite set of indices. Rewriting approaches in the context of super-position are presented in [7] and [8]. An approach that also uses constrained equations is developed in [9]. It produces clauses with constrained equations, but a distinguishing feature of this system is that it uses the current congruence closure model to guide the search, thereby avoiding potentially redundant cases.

The theory of equality and uninterpreted functions is in a sense a base theory for the theory of arrays. Array access $a[i]$ can be treated as a binary uninterpreted function, and array updates can be compiled away using a finite set of instances. This was recognized for the theory of arrays as well as a number of other theories in [10]. The reduction approach is the basis of several implementations of the theory of arrays, including Yices [11], Z3 [12], and analyzed in DPT [13]. STP [14] also uses a reduction approach, and furthermore observes that it can be important to delay rewriting array read/write terms into conditional statements. As an alternative to eliminating array writes, [15] considers eliminating reads in favor of writes. The resulting procedure handles especially well cases where arrays obtained by multiple non-interfering overwrites are compared. The *map* and *array property fragments* [16] are classes of first-order formulas that can express array properties involving some arithmetic. Extensions are studied in [17], including a unary map operator. An entirely different approach to arrays represents models of arrays as regular automata [18]. They can decide formulas that use offset arithmetic on array indices.

In comparison our paper offers a general setting for optimized array decision procedures based on inference rules.

II. Preliminaries

We consider a many-sorted language. A *signature* Σ is a triple $(\Sigma^S, \Sigma^F, \Sigma^P)$ where Σ^S is a set of sorts, Σ^F is a set of function symbols, and Σ^P is a set of predicate symbols (each endowed with the corresponding arity and sort). We assume that, for each sort σ, the equality $\simeq_\sigma$ is a symbol that does not occur in Σ and that is always interpreted as the identity relation over (the interpretation of) σ. As a notational convention, we will always omit the subscript. We call 0-arity function symbols *constant* symbols, and 0-

arity predicate symbols *propositions*. Σ-*atoms*, Σ-*literals*, Σ-*clauses*, and Σ-*formulas* are defined in the usual way. A set of Σ-literals is called a Σ-constraint. Terms, literals, clauses and formulas are called *ground* when no variable appears in them. A *sentence* is a formulas in which free variables do not occur. A *CNF formula* is a conjunction $C_1 \wedge \ldots \wedge C_n$ of clauses. We will write CNF formulas as set of clauses. We use a, b, i, j, v and w for constants, where a and b are used for array constants, i and j for array indices, and v and w for array values. We use f for function symbols, p and q for predicate symbols, σ and τ for sorts, C for clauses, and φ for formulas. We use $v{:}\,\sigma$ to denote that constant symbol v has sort σ, and $f{:}\,(\sigma_1,\ldots,\sigma_n) \to \tau$ to denote that function symbol f has arity n, argument sorts $\sigma_1,\ldots,\sigma_n$, and result sort τ. Given two signatures Σ_1 and Σ_2, $\Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \subseteq \Sigma_2$ are defined as usual, we say Σ_1 and Σ_2 are *disjoint* if $\Sigma_1^F \cap \Sigma_2^F = \emptyset$ and $\Sigma_1^P \cap \Sigma_2^P = \emptyset$, and *strongly disjoint* if they are disjoint and $\Sigma_1^S \cap \Sigma_2^S = \emptyset$.

We use the standard notion of a Σ-structure M. It consists of a non-empty pairwise disjoint domains $|M|_\sigma$ for every sort σ, and a sort and arity-matching interpretation of the function and predicate symbols in Σ. We use ι and ν for elements of a domain $|M|_\sigma$. We use $M(f)$ (resp. $M(p)$) to denote the interpretation of the function (resp. predicate) symbol f (resp. p). The interpretation of an arbitrary term t is denoted by $M[\![t]\!]$, and is defined in the standard way. The truth of a Σ-formulas φ, denoted by $M[\![\varphi]\!]$, is also defined in the standard way. If $\Sigma_0 \subseteq \Sigma$ and M is a Σ-structure, the Σ_0-*reduct* of M is the Σ_0-structure $M{\downarrow}_{\Sigma_0}$ obtained from M by forgetting the interpretation of the symbols from $\Sigma \setminus \Sigma_0$.

A collection of Σ-sentences is a Σ-theory. The free theory $T_\emptyset$ over a signature Σ is the first-order theory with an empty set of Σ-sentences. Let T_1 be a Σ_1-theory and T_2 be a Σ_2-theory. Then, T_1 and T_2 are *disjoint* (resp. *strongly disjoint*) if Σ_1 and Σ_2 are disjoint (resp. strongly disjoint). The combined theory $T_1 \oplus T_2$ is a $(\Sigma_1 \cup \Sigma_2)$-theory composed by the union of the Σ_1 and Σ_2-sentences. The *constraint satisfiability problem* for a theory T, also called the T-satisfiability problem, is the problem of deciding whether a Σ-constraint is satisfiable in a model of Σ-theory T. The constraint may contain variables, since these variables may be replaced by fresh constants, we can define the constraint satisfiability problem as the problem of deciding whether a finite conjunction of ground literals, in an expanded signature $\Sigma_\star$, is true in a $\Sigma_\star$-structure whose Σ-reduct is a model of T. The *satisfiability problem* can be similarly defined for ground CNF formulas.

III. A Simple Core Solver

The array theory decision procedures, proposed in this paper, are defined on top of a core solver as a set of inference rules. The basic architecture of the core solver is the usual one used in state-of-the-art SMT solvers, where a SAT solver is integrated with a decision procedure for the constraint satisfiability problem for a Σ-theory T [19]. In our core solver, the *core theory* T_{core} is the combined theory $T_\emptyset \oplus T_1 \oplus \ldots \oplus T_k$, where for each $i, j \in \{1,\ldots,k\}$, T_i and T_j are strongly disjoint, and $T_\emptyset$ and T_i are disjoint. This restriction admits a very simple combination method where the theories T_i's can be non-stably infinite and non-convex. Our combination method uses the *model-based theory combination* [20]. In the rest of the paper, we assume that one of the theories T_i's is the Σ_b-theory T_b of Boolean terms, where $\Sigma_b^S = \{\mathsf{bool}\}$, $\Sigma_b^F = \{\top{:}\,\mathsf{bool}, \bot{:}\,\mathsf{bool}\}$, $\Sigma_b^P = \emptyset$, and it contains the following two axioms:

$$\top \not\simeq \bot, \quad \forall x{:}\,\mathsf{bool}\,.\, x \simeq \top \vee x \simeq \bot$$

In our actual solver, the other theories in T_{core} are: *arithmetic*, *bit-vectors* and *scalar values*. For each $i \in \{1,\ldots,k\}$, let Σ_i be the signature of theory T_i, and let $\Sigma_\emptyset$ be the signature of $T_\emptyset$. Recall that the signature of $T_\emptyset$ is not fixed a priori, and w.l.o.g. we assume $\Sigma_1^S \cup \ldots \cup \Sigma_k^S \subseteq \Sigma_\emptyset^S$. We say a function (resp. predicate) symbol f (resp. q) is *interpreted* if $f \in \Sigma_1^F \cup \ldots \cup \Sigma_k^F$ (resp. $q \in \Sigma_1^P \cup \ldots \cup \Sigma_k^P$). Otherwise, we say the symbol is *uninterpreted*. We also assume uninterpreted predicates $q(v_1,\ldots,v_n)$ are represented as $f_q(v_1,\ldots,v_n) \simeq \top$. In the core solver, CNF formulas are represented in flattened form. A *CNF flattened formula* comprises of a sequence of definitions of the form:

$$v \equiv f(v_1,\ldots,v_n), \quad p \equiv q(v_1,\ldots,v_n), \quad p \equiv v \simeq w$$

and clauses of the form $l_1 \vee \ldots \vee l_n$, where f is a function symbol, p is an uninterpreted proposition, q is a predicate symbol, $v, w, v_1, \ldots, v_n$ are uninterpreted constants, each v is never defined *after* it is used, and each l_i is of the form p or $\neg p$. The constant v and proposition p, above, should be viewed as *names* for terms and atoms respectively. In an actual implementation, they are essentially pointers to these terms and atoms.

Example 1: The CNF formula $v \simeq w \wedge (v \geq w \vee f(v - w) \simeq 0)$ can be represented in flattened form as:

$$\begin{array}{lll} p_1 \equiv v \simeq w, & p_2 \equiv v \geq w, & v_1 \equiv v - w, \\ v_2 \equiv f(v_1), & v_3 \equiv 0, & p_3 \equiv v_2 \simeq v_3, \end{array} \; ; \; p_1, \;\; p_2 \vee p_3$$

The SAT solver, in our core solver, uses a DPLL based algorithm to build an assignment for all propositions. For each $i \in \{1,\ldots,k\}$, let $\mathfrak{S}_i$ be a decision procedure for theory T_i, and let $\mathfrak{S}_\emptyset$ be a decision procedure for the free theory $T_\emptyset$. In our implementation, $\mathfrak{S}_\emptyset$ is based on the congruence closure algorithm described in [21]. The state Γ of the core solver is composed by a propositional assignment, a set of definitions and clauses $F(\Gamma)$, and the states of procedures $\mathfrak{S}_i$'s and $\mathfrak{S}_\emptyset$. We use $\Gamma(p)$ to denote the assignment of proposition p in state Γ. When the SAT solver assigns a proposition $p \equiv q(v_1,\ldots,v_n)$ and $q \in \Sigma_i^P$, then procedure $\mathfrak{S}_i$ is notified. If $p \equiv v \simeq w$, then $\mathfrak{S}_\emptyset$ is notified, and if v has sort $\sigma \in \Sigma_i^S$, then procedure $\mathfrak{S}_i$ is also notified. The procedure $\mathfrak{S}_\emptyset$ maintains an equivalence relation $\sim_\Gamma$, which is the smallest equivalence relation that contains $\{(v, w) \mid p \equiv v \simeq w \in \Gamma, \text{ and } \Gamma(p) = \mathsf{true}\}$. As usual, the relation $\sim_\Gamma$ can be implemented using a union-find data-structure. The procedure $\mathfrak{S}_\emptyset$ also maintains the relation $\not\sim_\Gamma$ defined as $\{(v, w) \mid p \equiv v' \simeq w' \in \Gamma,\ \Gamma(p) = \mathsf{false},\ v \sim_\Gamma v',\ w \sim_\Gamma w'\}$. As a notational convention we will always omit the subscript in $\sim_\Gamma$ and $\not\sim_\Gamma$. Inference rules are written as:

$$\frac{\alpha_1,\ldots,\alpha_n}{C_1,\ldots,C_m}$$

where $\alpha_1, \ldots, \alpha_n$ are the *antecedents*, and should be viewed as queries to the current state Γ of the core solver. We use antecedents of the form: $a \equiv f(v_1, \ldots, v_n)$ (meaning: the definition is in Γ), $v \sim w$ (meaning: v and w are equivalent in Γ), $v{:}\sigma$ (meaning: Γ contains a constant v of sort σ), and $\Gamma(p) = \mathsf{false}$. The *consequents* $C_1, \ldots, C_n$ are clauses, not necessarily in flattened form, that should be added to the next state. For example, if the consequent is of the form $a[i] \simeq v$, it should be interpreted as $v' \equiv a[i],\ p \equiv v' \simeq v;\ p$. Note that new definitions are not created if they already exist in current state Γ. We use $\Gamma_1 \vdash_\gamma \Gamma_2$ to denote that inference rule γ was applied to state Γ_1 producing a new state Γ_2. We say an inference rule γ is *sound* with respect to a theory T if for all states Γ_1 and Γ_2 such that $\Gamma_1 \vdash_\gamma \Gamma_2$, we have $F(\Gamma_1)$ is equisatisfiable to $F(\Gamma_2)$ modulo theory T. A inference rule γ is *saturated* at state Γ if Γ already contains any consequent of γ, or if the consequents are already satisfied by the (partial) propositional assignment in Γ. In our implementation, the procedure $\mathfrak{S}_\emptyset$ uses the congruence inference rule:

$$\frac{w_1 \equiv f(v_1, \ldots, v_n),\ w_2 \equiv f(v'_1, \ldots, v'_n),\ v_1 \sim v'_1, \ldots, v_n \sim v'_n}{w_1 \simeq w_2}$$

Each procedure $\mathfrak{S}_i$ builds an interpretation (*candidate model*) for each constant $v{:}\sigma$ s.t. $\sigma \in \Sigma_i^S$. The combination method requires that the procedures agree on equalities between (uninterpreted) constants. For this purpose, the model-based theory combination, uses the models M_i that are built by each procedure $\mathfrak{S}_i$. Given two constants v and w, such that $M_i(v) = M_i(w)$, the procedure creates a definition $p_{v,w} \equiv v \simeq w$ (if it does not exist already), and $p_{v,w}$ is assigned to true in the SAT solver. This assignment is essentially a *guess* (i.e., decision), if this assignment triggers an inconsistency, then backtracking is used to fix the model. Many optimizations are possible [20] in the architecture described above, but they are beyond the scope of this paper. We say Γ is a *satisfiable final state* if all inference rules are saturated, all propositions are assigned, all clauses are satisfied, all constants $v{:}\sigma$ have an interpretation when $\sigma \in \Sigma_i^S$ for some $i \in \{1, \ldots, k\}$, and none of the procedures $\mathfrak{S}_i$'s and $\mathfrak{S}_\emptyset$ detected an inconsistency.

Example 2: Consider the following CNF formula, where $f{:}\,\mathsf{bool} \to \sigma$, $v{:}\,\mathsf{bool}$ and $w{:}\sigma$.

$$f(\top) \simeq w \ \wedge\ f(\bot) \simeq w \ \wedge\ f(v) \not\simeq w$$

This formula is unsatisfiable, and the core solver will detect it. The procedure $\mathfrak{S}_b$ will try to assign an interpretation for v because it has sort bool, but an inconsistency is detected (using the congruence rule) when it tries to assign v to $\top$ or $\bot$. Note that none of the procedures $\mathfrak{S}_i$ had to exchange cardinality constraints.

IV. Array Theory

The array theory T_A with signature Σ_A is parametric in the context of many-sorted logic. That is, given a non-empty set of sorts $\mathcal{S}$, Σ_A^S is the least set such that:

1. $\mathcal{S} \subset \Sigma_A^S$
2. $\sigma \in \Sigma_A^S$ and $\tau \in \Sigma_A^S$ implies $(\sigma \Rightarrow \tau) \in \Sigma_A^S$.

$$\mathsf{idx}\ \frac{a \equiv \mathit{store}(b,i,v)}{a[i] \simeq v}$$

$$\Downarrow\ \frac{a \equiv \mathit{store}(b,i,v), \quad w \equiv a'[j], \quad a \sim a'}{i \simeq j \vee a[j] \simeq b[j]}$$

$$\Uparrow\ \frac{a \equiv \mathit{store}(b,i,v), \quad w \equiv b'[j], \quad b \sim b'}{i \simeq j \vee a[j] \simeq b[j]}$$

$$\mathsf{ext}\ \frac{a{:}\,(\sigma \Rightarrow \tau), \quad b{:}\,(\sigma \Rightarrow \tau)}{a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}]}$$

Fig. 1. Array theory basic inference rules.

We say sorts of the form $(\sigma \Rightarrow \tau)$ are *array sorts* with *index sort* σ, and *value sort* τ. For each array sort $(\sigma \Rightarrow \tau)$, Σ_A^F contains the function symbols $\_[\_]{:}\,((\sigma \Rightarrow \tau), \sigma) \to \tau$, and $\mathit{store}{:}\,((\sigma \Rightarrow \tau), \sigma, \tau) \to (\sigma \Rightarrow \tau)$. There are no predicates, so $\Sigma_A^P = \emptyset$. We say $\_[\_]$ is the *array read* operation, and *store* the *array update* operation. The following scheme axiomatizes these two operators:

$$\forall a{:}\,(\sigma \Rightarrow \tau),\ i{:}\sigma,\ v{:}\tau\,.\ \mathit{store}(a,i,v)[i] \simeq v$$
$$\forall a{:}\,(\sigma \Rightarrow \tau),\ i{:}\sigma,\ j{:}\sigma,\ v{:}\tau\,.\ i \simeq j \vee \mathit{store}(a,i,v)[j] \simeq a[j]$$

We say that the function symbol *store* is an array combinator, that is, operations that build new arrays. Later, we define a richer set of array combinators. The following scheme is called the extensionality axiom scheme. Informally, it states that if two arrays store the same value at index i, for each index i, then they are equal.

$$\forall a{:}\,(\sigma \Rightarrow \tau),\ b{:}\,(\sigma \Rightarrow \tau)\,.\ \exists i{:}\,\sigma\,.\ a[i] \not\simeq b[i] \vee a \simeq b$$

Fig. 1 contains a basic set of inference rules for the array theory. Let us explain the first rules informally. Rule idx adds the assertion $a[i] \simeq v$ for every occurrence of a definition $a \equiv \mathit{store}(b,i,v)$. Rule $\Downarrow$ propagates read over a *store*. It fires if a is defined as a *store* and in the current state a is equivalent to a', where a' occurs in a read. It adds a clause forcing either the index j to be equal to the update index i, or the contents of a to agree with b on j. The clause is a tautology in the theory of arrays, it does not depend on $a \sim a'$. The other rules should be interpreted in a similar way. Later, we propose many refinements.

Theorem 3 (Soundness) idx, $\Downarrow$, $\Uparrow$, ext are sound.

Proof: The rules idx, $\Downarrow$, $\Uparrow$ are just instantiating the array axioms. The rule ext is instantiating the extensionality axiom by using a fresh skolem constant $k_{a,b}$. ■

Theorem 4 (Termination) idx, $\Downarrow$, $\Uparrow$, ext are terminating.

Proof: None of the rules create definitions of the form $a \equiv \mathit{store}(b,i,v)$. Thus, rule idx can only be applied once for each occurrence of *store* in the input. Assume the input formula has n array constants $(a{:}\,(\sigma \Rightarrow \tau))$, and m definitions of the form $v \equiv a[j]$. Then, rule ext can be applied at most n^2 times, and at most n^2 skolem constant $k_{a,b}$ are created. The rules $\Downarrow$ and $\Uparrow$ can be applied at most $(n^2 + m)n$ times. ■

Definition 5 (Map) Given sets S_σ and S_τ, a *map* from S_σ to S_τ is a finite set of pairs (ι, ν) where $\iota \in S_\sigma$ and $\nu \in S_\tau$. We say the map G is *functional* iff for all (ι_1, ν_1) and (ι_2, ν_2) in G, $\iota_1 = \iota_2$ implies that $\nu_1 = \nu_2$.

Theorem 6 (Completeness) idx, $\Downarrow$, $\Uparrow$, ext are complete.

Proof: Assume all rules are saturated in the satisfiable final state Γ, and let M be the model produced by the core solver for this final state. Note that symbols *store* and $\_[\_]$ are considered to be uninterpreted in the core solver. The core solver guarantees that for any pair of constants v_1 and v_2, $v_1 \sim v_2$ iff $M(v_1) = M(v_2)$. Our goal is to build a model M^λ that satisfies Γ and the array axioms. For every non array sort sort σ, $|M^\lambda|_\sigma = |M|_\sigma$. The domain for the array sorts is defined inductively. Let σ' be an array sort of the form $(\sigma \Rightarrow \tau)$, then $|M^\lambda|_{\sigma'}$ is the set of functions from $|M^\lambda|_\sigma$ to $|M^\lambda|_\tau$. The interpretation for each $\_[\_]\colon((\sigma \Rightarrow \tau), \sigma) \rightarrow \tau$ is just the function application. More formally, given $\rho \in |M^\lambda|_{(\sigma \Rightarrow \tau)}$ and $\iota \in |M^\lambda|_\sigma$, $M^\lambda(\_[\_])(\rho, \iota) = \rho(\iota)$. The interpretation for each $\mathit{store}\colon((\sigma \Rightarrow \tau), \sigma, \tau) \rightarrow (\sigma \Rightarrow \tau)$ is $M^\lambda(\mathit{store})(\rho, \iota, \nu) = \rho'$, where ρ' is the function:

$$\rho'(x) = \begin{cases} \nu & \text{if } x = \iota, \\ \rho(x) & \text{otherwise.} \end{cases}$$

It is easy to check that the interpretations for $\_[\_]$ and *store* satisfy the array axioms. Our next goal is to assign an interpretation to all constants in Γ such that:

1. For any pair of constants v_1 and v_2 in Γ, $M(v_1) = M(v_2)$ iff $M^\lambda(v_1) = M^\lambda(v_2)$. We say this is the *equivalence property.*
2. The interpretation of constants satisfies all definitions of the form $a \equiv \mathit{store}(b, i, v)$ and $v \equiv a[i]$ in Γ.

Let $\sqsubset$ be a total order on sorts such that for all array sorts $(\sigma \Rightarrow \tau)$, $\sigma \sqsubset (\sigma \Rightarrow \tau)$ and $\tau \sqsubset (\sigma \Rightarrow \tau)$. We define the interpretation for the constants using $\sqsubset$. That is, if $\sigma_1 \sqsubset \sigma_2$, then we define the interpretation for all constants $a_1\colon\sigma_1$ before defining the interpretation for any constant $a_2\colon\sigma_2$. Moreover, when we construct the interpretation for $a_2\colon\sigma_2$ we assume that the equivalence property holds for all constants $a_1\colon\sigma_1$ where $\sigma_1 \sqsubset \sigma_2$. The "base case" is easy, for each constant $v\colon\sigma$ in Γ, such that σ is not an array sort, $M^\lambda(v) = M(v)$. We also define $M^\lambda(f) = M(f)$ for every interpreted function symbol f. For each sort σ, Let δ_σ be an arbitrary element of $|M^\lambda|_\sigma$. Now, we define an interpretation for an array constant $a\colon(\sigma \Rightarrow \tau)$ assuming that the interpretation for all constants $i\colon\sigma$ and $v\colon\tau$ was already defined. First, we define a map $\mathsf{graph}(a)$ as the set $\{(M^\lambda(i), M^\lambda(v)) \mid v \equiv a'[i] \in \Gamma,\ a \sim a'\}$. The map $\mathsf{graph}(a)$ is functional, because the equivalence property holds for all constants $i\colon\sigma$ and $v\colon\tau$; and for any two entries $v_1 \equiv a_1[i_1]$ and $v_2 \equiv a_2[i_2]$ the core solver guarantees that $M(a_1) = M(a_2)$ and $M(i_1) = M(i_2)$ implies that $M(v_1) = M(v_2)$. Then, we define $M^\lambda(a)$ as:

$$M^\lambda(a)(\iota) = \begin{cases} \nu & \text{if } (\iota, \nu) \in \mathsf{graph}(a), \\ \delta_\sigma & \text{otherwise.} \end{cases}$$

Now, we show that for any array constants $a\colon(\sigma \Rightarrow \tau)$ and $b\colon(\sigma \Rightarrow \tau)$, the equivalence property holds:

1. If $a \sim b$, then by construction $M^\lambda(a) = M^\lambda(b)$.
2. If $a \not\sim b$, then rule ext guarantees that $\mathsf{graph}(a)$ contains $(M^\lambda(k_{a,b}), \nu_1)$ and $\mathsf{graph}(b)$ contains $(M^\lambda(k_{a,b}), \nu_2)$ such that $M^\lambda(\nu_1) \neq M^\lambda(\nu_2)$. Therefore, $M^\lambda(a) \neq M^\lambda(b)$.

It is easy to check that the definitions of the form $v \equiv a[i]$ are satisfied by M^λ. Now, we show that M^λ also satisfies all definitions of the form $a \equiv \mathit{store}(b, i, v)$. Recall that all rules are saturated in the final state. First, rule idx guarantees that $M^\lambda(a)(M^\lambda(i)) = M^\lambda(v)$. Now, let $\mathsf{index}(a)$ be the set $\{\iota \mid (\iota, \nu) \in \mathsf{graph}(a)\}$. The rules $\Uparrow$ and $\Downarrow$ guarantee that $\mathsf{index}(a) = \mathsf{index}(b) \cup \{M^\lambda(i)\}$, and $M^\lambda(a)(\iota) = M^\lambda(b)(\iota)$ for every $\iota \in \mathsf{index}(a) \setminus \{M^\lambda(i)\}$. Finally, we have $M^\lambda(a)(\iota) = M^\lambda(b)(\iota) = \delta_\sigma$ for every $\iota \notin \mathsf{index}(a)$. Therefore, every definition of the form $a \equiv \mathit{store}(b, i, v)$ is satisfied. The construction of the interpretation $M^\lambda(f)$, for each uninterpreted function symbol f in Γ, is similar to the one used for array constants. The only difference is that $\mathsf{graph}(f)$ is a tuple of size $\text{arity}(f) + 1$ instead of being a pair. It guarantees that all definitions of the form $v \equiv f(w_1, \ldots, w_n)$ are satisfied by M^λ. Note that the equivalence property guarantees that for every definition of the form $p \equiv v \simeq w$ in Γ, $M[\![v \simeq w]\!]$ iff $M^\lambda[\![v \simeq w]\!]$, and consequently for every clause C in Γ, $M[\![C]\!]$ iff $M^\lambda[\![C]\!]$. Thus, M^λ satisfies all array axioms, definitions and clauses in Γ. ■

A. Redundant Axioms

The rules $\Downarrow$, $\Uparrow$, ext produce clauses of the form:

$$i \simeq j \vee a[j] \simeq b[j] \tag{1}$$

$$a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}] \tag{2}$$

The proof of Theorem 6 makes it clear that it is unnecessary to add the clauses of the form (1) to Γ, when Γ already contains a clause $i' \simeq j' \vee a'[j'] \simeq b'[j']$ such that $a \sim a'$, $b \sim b'$, $i \sim i'$, and $j \sim j'$. Similarly, it is unnecessary to add (2) to Γ, when Γ already contains a clause $a' \simeq b' \vee a'[k_{a',b'}] \not\simeq b'[k_{a',b'}]$ such that $a \sim a'$ and $b \sim b'$. Thus, in our implementation, we use a data-structure for storing a set of tuples (a, b, i, j) for (1), and a set of tuples (a, b) for (2). Given a tuple t, this data-structure provides a constant time function for checking whether the data-structure contains a tuple congruent to t or not. Before including (1) and (2) into Γ, we check whether the data-structure already contains a congruent tuple. If it does, we discard the new clause. Otherwise, we include it into Γ and update the data-structure. This data-structure is similar to the hashtable used to implement congruence closure procedures [21].

B. Restricted Extensionality

In the proof of Theorem 6, rule ext is used to guarantee that for all array constants a_1 and a_2:

$$M(a_1) \neq M(a_2) \text{ implies } M^\lambda(a_1) \neq M^\lambda(a_2) \tag{3}$$

when proving the *equivalence property*: for any pair of constants v_1 and v_2 in Γ, $M(v_1) = M(v_2)$ iff $M^\lambda(v_1) =$

$$\mathsf{ext}_{\not\simeq}\ \frac{p \equiv a \simeq b, \quad \Gamma(p) = \mathsf{false}}{a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}]}$$

$$\mathsf{ext}_r\ \frac{a\colon(\sigma \Rightarrow \tau), \quad b\colon(\sigma \Rightarrow \tau), \quad \{a,b\} \subseteq \mathsf{foreign}}{a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}]}$$

Fig. 2. Restricted extensionality inference rules.

$M^\lambda(v_2)$. We say an array constant a is *foreign* iff there is a b such that $a \sim b$, and b occurs as the argument of an uninterpreted function symbol f, or as the index of an array read $v \equiv a'[b]$. Observe that (3) is only needed for showing that:

1. $\mathsf{graph}(a')$ and $\mathsf{graph}(f)$ are functional when $\mathsf{index}(a)$ and $\mathsf{index}(f)$ contain $M^\lambda(a_1)$ and $M^\lambda(a_2)$. That is, a_1 and a_2 are foreign.
2. $M[\![a \simeq b]\!] = \mathsf{false}$ implies $M^\lambda[\![a \simeq b]\!] = \mathsf{false}$.

So, this observation suggests a simple optimization where ext is only applied to pairs of array constants a and b when: a and b are foreign, or $a \simeq b$ is assigned to false by the core solver.

Definition 7 (Foreign) Given a state Γ, the set $\mathsf{foreign}$ of *foreign constants* is the least set s.t.:

1. $v \equiv f(\ldots,a,\ldots)$ and $a\colon(\sigma \Rightarrow \tau)$ implies $a \in \mathsf{foreign}$,
2. $v \equiv a[b]$ and $b\colon(\sigma \Rightarrow \tau)$ implies $b \in \mathsf{foreign}$,
3. $a \sim b$ and $a \in \mathsf{foreign}$ implies $b \in \mathsf{foreign}$.

Fig. 2 contains the set of rules for implementing this refinement.

Theorem 8: The rules idx, $\Downarrow$, $\Uparrow$, $\mathsf{ext}_{\not\simeq}$ and ext_r are sound, terminating and complete.

Another optimization is based on the observation that it is unnecessary to add (2) to Γ, if a and b already store different values at some index. More formally, we have:

Definition 9 (Already Disequal) Given a state Γ, $(a,b) \in \mathsf{already\text{-}diseq}$ iff there are two definitions $v_1 \equiv a_1[i_1]$ and $v_2 \equiv a_2[i_2]$ in Γ such that $v_1 \not\sim v_2$, $a \sim a_1$, $b \sim b_1$, and $i_1 \sim i_2$.

C. Restricted $\Uparrow_r$

Definition 10 (Linearity) Given a state Γ, the set $\mathsf{non\text{-}linear}$ of *non-linear constants* is the least set such that:

1. $a_1 \equiv store(b_1,i_1,v_1)$, $a_2 \equiv store(b_2,i_2,v_2)$, a_1 is not a_2 and $a_1 \sim a_2$ implies $\{a_1,a_2\} \subseteq \mathsf{non\text{-}linear}$,
2. $a \equiv store(b,i,v)$ and $a \in \mathsf{non\text{-}linear}$ implies $b \in \mathsf{non\text{-}linear}$,
3. $a \in \mathsf{non\text{-}linear}$ and $a \sim b$ implies $b \in \mathsf{non\text{-}linear}$.

We say a is *linear* if $a \notin \mathsf{non\text{-}linear}$.

In many software verification applications, we observed that the set $\mathsf{non\text{-}linear}$ is very small. This observation suggests a simple optimization, where rule $\Uparrow$ is only applied to array constants in the set $\mathsf{non\text{-}linear}$. Given a map m and a constant j, we use $m \setminus \{j\}$ to denote the set $\{(\iota,\nu) \mid (\iota,\nu) \in m,\ \iota \neq M^\lambda(j)\}$. The basic idea is to use $\mathsf{graph}(b) \setminus \{i\}$ to complete the map $\mathsf{graph}(a)$ whenever Γ contains a definition of the form $a \equiv store(b,i,v)$ and b is linear. Fig. 3 contains the restricted version of $\Uparrow$.

$$\Uparrow_r\ \frac{a \equiv store(b,i,v), \quad w \equiv b'[j], \quad b \sim b', \quad b \in \mathsf{non\text{-}linear}}{i \simeq j \vee a[j] \simeq b[j]}$$

Fig. 3. Restricted $\Uparrow_r$ inference rule.

Theorem 11: The rules idx, $\Downarrow$, $\Uparrow_r$, $\mathsf{ext}_{\not\simeq}$ and ext_r are sound, terminating and complete.

Proof: The proof is similar to the proof of Theorem 6, but we use the completion procedure described above before defining $M^\lambda(a)$. ■

V. Combinatory Array Logic

In this section, we consider the extended array theory T_{CAL} with signature Σ_{CAL}. T_{CAL} contains two new families of combinators: the *constant-value* array combinators, and the *map* combinators. For each sort $(\sigma \Rightarrow \tau)$, Σ^F_{CAL} contains the function symbol $K\colon \tau \to (\sigma \Rightarrow \tau)$. For each function symbol $f\colon(\tau_1,\ldots,\tau_k) \to \tau$, interpreted or not, and sort σ, Σ^F_{CAL} contains the function symbol $map_f\colon((\sigma \Rightarrow \tau_1),\ldots,(\sigma \Rightarrow \tau_k)) \to (\sigma \Rightarrow \tau)$. We say map_f is the *pointwise array extension* of f. The following scheme axiomatizes the new combinators:

$$\forall v\colon\tau,\ i\colon\sigma\,.\,K(v)[i] \simeq v$$
$$\forall a_1\colon(\sigma \Rightarrow \tau_1),\ldots,a_k\colon(\sigma \Rightarrow \tau_k),\ i\colon\sigma\,.$$
$$map_f(a_1,\ldots,a_k)[i] \simeq f(a_1[i],\ldots,a_k[i])$$

Similarly, given a predicate symbol $p\colon(\tau_1,\ldots,\tau_k)$, we define the pointwise extension combinator map_p for predicates as:

$$\begin{aligned}&\forall b_1\colon(\sigma \Rightarrow \tau_1),\ldots,b_k\colon(\sigma \Rightarrow \tau_k),\ i\colon\sigma\,.\\ &\quad(\neg p(b_1[i],\ldots,b_k[i]) \vee map_p(b_1,\ldots,b_k)[i] \simeq \top)\ \wedge\\ &\quad(p(b_1[i],\ldots,b_k[i]) \vee map_p(b_1,\ldots,b_k)[i] \simeq \bot)\end{aligned}$$

Due to space limitations, we only discuss combinators of the form map_f. The extension to map_p is straight-forward.

From now on, for each sort τ, we assume the core theory contains the if-then-else operator $ite\colon(\mathsf{bool},\tau,\tau) \to \tau$. The following scheme axiomatizes ite:

$$\forall x_1,x_2\colon\tau.\ ite(\top,x_1,x_2) \simeq x_1\ \wedge ite(\bot,x_1,x_2) \simeq x_2$$

A. Versatility

The extended combinators allow to easily express some functions over sets and bags. The idea is to represent a set of elements of sort σ as an array of sort $(\sigma \Rightarrow \mathsf{bool})$. We list a few of these below.

$\emptyset$	$= K(\bot)$	$\overline{a}$	$= map_{ite}(a,K(\bot),K(\top))$
$\{v\}$	$= store(K(\bot),v,\top)$	$a \cup b$	$= map_{ite}(a,K(\top),b)$
$v \in a$	$= a[v]$	$a \cap b$	$= map_{ite}(a,b,K(\bot))$

Similarly, a bag (or multi-set) of elements of sort σ can be encoded as an array of sort $(\sigma \Rightarrow \mathsf{int})$. Then, the empty bag is encoded as $K(0)$, the set of elements in a bag a is $map_{>}(a,K(0))$, the multi-set extension of a set a is $map_{ite}(a,K(1),K(0))$, and the join operation $a \uplus b$ on bags is encoded as $map_{+}(a,b)$. On the other hand, the cardinality of a set/bag cannot be expressed.

Notice also that $store(a,i,v) = map_{ite}(\{i\},K(v),a)$, so we could use $store$ as a derived combinator if we instead assume ite and the singleton combinator.

$$\mathsf{K}{\Downarrow}\ \frac{a \equiv K(v), \quad w \equiv a'[j], \quad a \sim a'}{a[j] \simeq v}$$

$$\mathsf{map}{\Downarrow}\ \frac{a \equiv map_f(b_1, \ldots, b_n), \quad w \equiv a'[j], \quad a \sim a'}{a[j] \simeq f(b_1[j], \ldots, b_n[j])}$$

$$\mathsf{map}{\Uparrow}\ \frac{\begin{array}{c} a \equiv map_f(b_1, \ldots, b_n), \quad w \equiv b'_k[j], \\ b_k \sim b'_k, \text{ for some } \ k \in \{1, \ldots, n\} \end{array}}{a[j] \simeq f(b_1[j], \ldots, b_n[j])}$$

$$\epsilon_{\not\simeq}\ \frac{v \equiv a[i], \quad i{:}\sigma, \quad i \text{ is not } \epsilon_\sigma}{\epsilon_\sigma \not\simeq i} \qquad \epsilon\delta\ \frac{a{:}(\sigma \Rightarrow \tau)}{a[\epsilon_\sigma] \simeq \delta_a}$$

Fig. 4. Extended combinators inference rules.

B. Extended Inference Rules

Fig. 4 contains the inference rules for the new combinators. In the proof of Theorem 6, for every array constant a, we defined $M^\lambda(a)(\iota) = \delta_\sigma$ if $\iota \notin \mathsf{index}(a)$, where δ_σ is an arbitrary value of $|M^\lambda|_\sigma$. That is, δ_σ is the *default value* of every array constant in Γ. This simple construction is not possible when combinators K and map_f are used, because they constrain the default value of array constants. Given an array constant a, we use the fresh constant $\delta_a{:}\sigma$ to denote the *default value* for array a. The rule $\epsilon\delta$ exposes the default value δ_a (of an array constant a) by accessing a at an index ϵ_σ. We have a fresh constant ϵ_σ for each sort σ. The rule $\epsilon_{\not\simeq}$ enforces that ϵ_σ is different from any other index i of sort σ. Of course, in general, the rule $\epsilon_{\not\simeq}$ is not sound if the interpretation of sort σ is finite. The following example illustrates the problem.

Example 12: Let i be a constant of sort σ. Then, the formula $store(K(v_1), i_1, w_1) \simeq K(v_2)$, $v_1 \not\simeq v_2$ is satisfiable in a structure where the interpretation of σ has only one element. On the other hand, a procedure based on the inference rules in Fig. 1 and 4 will return unsatisfiable.

Theorem 13: Considering the simplifying assumption that the intended interpretation of every index sort σ is infinite, then the rules idx, $\Downarrow$, $\Uparrow$, $\mathsf{ext}_{\not\simeq}$, ext_r, $\mathsf{K}{\Downarrow}$, $\mathsf{map}{\Downarrow}$, $\mathsf{map}{\Uparrow}$, $\epsilon_{\not\simeq}$ and $\epsilon\delta$ are sound, terminating and complete.

Proof: The restricted version of the rule $\Uparrow$ is not considered here. We consider this optimization, in the context of extended combinators, in Section D. The proof is similar to the proof of Theorem 6, but the construction of M^λ is slightly different. We define the map $\mathsf{graph}(a)$ as before, but we define $M^\lambda(a)$ as:

$$M^\lambda(a)(\iota) = \begin{cases} \nu & \text{if } (\iota, \nu) \in \mathsf{graph}(a), \\ M^\lambda(\delta_a) & \text{otherwise.} \end{cases}$$

Now, we show that M^λ satisfies all definitions of the form $a \equiv store(b, i, v)$, $a \equiv K(v)$ and $a \equiv map_f(b_1, \ldots, b_k)$. Let $\mathsf{index}(a)$ be the set $\{\iota \mid (\iota, \nu) \in \mathsf{graph}(a)\}$. First, let us consider definitions of the form $a \equiv store(b, i, v)$, The rule idx guarantees that $M^\lambda(a)(M^\lambda(i)) = M^\lambda(v)$. The rules $\Uparrow$ and $\Downarrow$ guarantee that $\mathsf{index}(a) = \mathsf{index}(b) \cup \{M^\lambda(i)\}$, and $M^\lambda(a)(\iota) = M^\lambda(b)(\iota)$ for every $\iota \in \mathsf{index}(a) \setminus \{M^\lambda(i)\}$.

$$\mathsf{blast}\ \frac{a{:}(\sigma \Rightarrow \tau), \quad \mathsf{size}(\sigma) = k}{a[\sigma_1] \simeq \delta_{a,1}, \ \ldots, \ a[\sigma_k] \simeq \delta_{a,k}}$$

Fig. 5. Blasting inference rule.

Now, we just need to show that for every $\kappa \notin \mathsf{index}(a)$, $M^\lambda(a)(\kappa) = M^\lambda(b)(\kappa)$. This equality is a consequence of the following observations:

$$\begin{array}{lll} & M^\lambda(a)(\kappa) & \\ = & M^\lambda(\delta_a) & \text{(by def. of } M^\lambda) \\ = & M^\lambda(a)(M^\lambda(\epsilon_\sigma)) & \text{(by rule } \epsilon\delta) \\ = & M^\lambda(b)(M^\lambda(\epsilon_\sigma)) & \text{(by rule } \epsilon_{\not\simeq},\ M^\lambda(\epsilon_\sigma) \neq M^\lambda(i)) \\ = & M^\lambda(\delta_b) & \text{(by rule } \epsilon\delta) \\ = & M^\lambda(b)(\kappa) & \text{(by def. of } M^\lambda) \end{array}$$

For definitions of the form $a \equiv K(v)$, by rules $\mathsf{K}{\Downarrow}$ and $\epsilon\delta$, it is easy to see that $M^\lambda(a)(\iota) = M^\lambda(v)$ for all $\iota \in |M^\lambda|_\sigma$. Finally, we consider definitions of the form $a \equiv map_f(b_1, \ldots, b_k)$. The rule $\mathsf{map}{\Downarrow}$ guarantees that for all $\iota \in \mathsf{index}(a)$ the map_f axiom holds. The rule $\mathsf{map}{\Uparrow}$ guarantees that $\mathsf{index}(b_1) \cup \ldots \cup \mathsf{index}(b_k) \subseteq \mathsf{index}(a)$. Hence, if $\kappa \notin \mathsf{index}(a)$, then $\kappa \notin \mathsf{index}(b_1) \cup \ldots \cup \mathsf{index}(b_k)$. Then, by rule $\epsilon\delta$ and the definition of M^λ, we have $M^\lambda(a)(\kappa) = M^\lambda(a)(M^\lambda(\epsilon_\sigma))$, and for each $i \in \{1, \ldots, k\}$, $M^\lambda(b_i)(\kappa) = M^\lambda(b_i)(M^\lambda(\epsilon_\sigma))$. Since $M^\lambda(\epsilon_\sigma) \in \mathsf{index}(a)$, the map_f axiom is also satisfied for all $\kappa \notin \mathsf{index}(a)$. ■

A procedure using rule $\epsilon_{\not\simeq}$ may track how many times this rule was used. Let $\mathsf{num}(\sigma)$ be the number of times rule $\epsilon_{\not\simeq}$ was applied to ϵ_σ for indices of sort σ. Now, assume the size $\mathsf{size}(\sigma)$ of the intended interpretation of a sort σ is known. Then, it is sound to apply $\epsilon_{\not\simeq}$ when $\mathsf{num}(\sigma) < \mathsf{size}(\sigma)$. In practice, if $\mathsf{size}(\sigma)$ is very big (e.g., σ is the sort of bit-vectors of size 32), then it is "sound" to apply rule $\epsilon_{\not\simeq}$. If $\mathsf{size}(\sigma)$ is small and $\mathsf{num}(\sigma) \geq \mathsf{size}(\sigma)$, then instead of using rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$ a procedure may use the rule blast described in Fig. 5. In rule blast, each $\delta_{a,i}$ is a fresh constant, and σ_i is an interpreted constant that is a name for the i-th value in the intended interpretation of σ. For example, if σ is the sort bool, then $\mathsf{size}(\sigma) = 2$, σ_1 is $\top$ and σ_2 is $\bot$.

Finally, we consider the case where $\mathsf{size}(\sigma)$ is not known (e.g., σ is an uninterpreted sort). Then, given a formula φ, if a procedure using rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$ returns unsatisfiable, then we know that φ is unsatisfiable in any structure where the size of the interpretation of each index sort σ is greater than $\mathsf{num}(\sigma)$. The value $\mathsf{num}(\sigma)$ gives us a bound on the size of any interpretation of σ. A complete and sound procedure can be implemented using these bounds and the rule blast. This is essentially the equivalent of a finite model finding procedure. In general, this procedure is quite expensive. For example, if F contains n uninterpreted sorts $\sigma_1, \ldots, \sigma_n$, then we have to consider $\mathsf{num}(\sigma_1) \times \ldots \times \mathsf{num}(\sigma_n)$ additional satisfiability subproblems. If all of them are unsatisfiable, then F is indeed unsatisfiable.

$$\mathsf{U}\delta\ \frac{a \equiv \mathit{store}(b,i,v)}{\delta(a) \simeq \delta(b)} \qquad \mathsf{K}\delta\ \frac{a \equiv K(v)}{\delta(a) \simeq v}$$

$$\mathsf{map}\delta\ \frac{a \equiv \mathit{map}_f(b_1,\ldots,b_n)}{\delta(a) \simeq f(\delta(b_1),\ldots,\delta(b_n))}$$

Fig. 6. Default value inference rules.

The constants δ_a enable another filter for the rule ext_r. The idea is to only apply ext_r when $\delta_a \sim \delta_b$. The basic observation is that $M^\lambda(a) \neq M^\lambda(b)$ if $M^\lambda(\delta_a) \neq M^\lambda(\delta_b)$, when the index sort σ has a sufficiently big interpretation. This observation is equivalent to the filter used in [13]. The filter is not sound if $\mathsf{size}(\sigma)$ is finite (and small) because we might have $\mathsf{index}(a) = \mathsf{index}(b) = |M^\lambda|_\sigma$ and there is no $\iota \in |M^\lambda|_\sigma$ s.t. $M^\lambda(a)(\iota) = M^\lambda(\delta_a)$ and $M^\lambda(b)(\iota) = M^\lambda(\delta_b)$.

C. Default Value Propagation

In this section, we use the simplifying assumption that every index sort is infinite. A corollary of Theorem 8 is that if a formula φ is satisfiable in the extended array theory, then it is satisfied in a structure M^λ where for every array constant a there is a value $M^\lambda(\delta_a)$ s.t. there is only a finite number of indices ι such that $M^\lambda(a)(\iota) \neq M^\lambda(\delta_a)$. Thus, we say that every array constant a has a *default value*. This observation suggests an alternative to rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$. The idea is to propagate constraints about the default value of each array constant a. We use distinguished function symbols δ, and encode the default value of a as the term $\delta(a)$. Fig. 6 contains the inference rules for propagating default values.

The distinguished functions may be used to encode properties of arrays. If we want to force a set b to be finite, we can assert $\delta(b) = \bot$ as part of the formula checked for satisfiability.

D. Restricted $\Uparrow_r$ and $\mathsf{map}\Uparrow_r$ for Extended Combinators

Now, we consider the problem of restricting the inference rules $\Uparrow$ and $\mathsf{map}\Uparrow$. The construction used in Theorem 11 does not work. For example, the extended array theory has combinators that take many array arguments. Given a definition of the form $a \equiv \mathsf{C}(\ldots,b,\ldots)$, where C is an arbitrary combinator, the basic idea was to use (a subset of) $\mathsf{graph}(b)$ to complete the map $\mathsf{graph}(a)$, when b is linear. However, if a combinator contains many array arguments $b_1,\ldots,b_k$, then we may have $\iota \in \mathsf{index}(b_i)$, but $\iota \notin \mathsf{index}(b_j)$ for some i and j in $\{1,\ldots,k\}$. It is incorrect to assume $M^\lambda(b_j)(\iota) = M^\lambda(b_j)(M^\lambda(\epsilon_\sigma))$, because $M^\lambda(b_j)$ may not have been defined yet when constructing $M^\lambda(a)$. The following example illustrates this problem.

Example 14: Let a, b and c be arrays $(\sigma \Rightarrow \mathsf{bool})$. Let us assume we are using a restricted version of $\mathsf{map}\Uparrow$ similar to $\Uparrow_r$. Now, consider the following formula:

$$a \simeq \mathit{map}_{ite}(a,b,c)\ \wedge\ b[j] \simeq \bot\ \wedge\ c[j] \simeq \top$$

The constant a is a linear parent [2], because its equivalence class contains only one combinator $\mathit{map}_{ite}(a,b,c)$. Therefore, the restricted version of $\mathsf{map}\Uparrow$ does not instantiate the *map* axiom, and the unsatisfiability is not detected. Note that if $a[j]$ is $\top$, then we have an inconsistency because $b[j] \simeq \bot$. Similarly, if $a[j]$ is $\bot$, then we also have an inconsistency because $c[j] \simeq \top$.

[2] defined in the technical report

The example above suggests a simple solution based on a total order $\prec$ on constants compatible with the order $\sqsubset$ on sorts. By compatible, we mean that if $v{:}\,\sigma$, $w{:}\,\tau$ and $\sigma \sqsubset \tau$ implies $v \prec w$. The order $\prec$ allows us to define a notion of stratification that complements the definition of linearity defined in Section IV.C. We use $a \preceq b$ to denote $a \prec b$ or $a = b$.

Definition 15 (Linear Stratification) Given a state Γ, the set $\mathsf{non\text{-}linear\text{-}stratified}$ of *non-linear-stratified constants* is the least set such that:

1. $a_1 \equiv \mathsf{C}(\ldots)$, $a_2 \equiv \mathsf{C}'(\ldots)$, a_1 is not a_2 and $a_1 \sim a_2$ implies $\{a_1,a_2\} \subseteq \mathsf{non\text{-}linear\text{-}stratified}$,
2. $a \equiv \mathsf{C}(\ldots,b,\ldots)$, $b{:}\,(\sigma \Rightarrow \tau)$, $b \sim b'$ and $a \preceq b'$ implies $a \in \mathsf{non\text{-}linear\text{-}stratified}$,
3. $a \equiv \mathsf{C}(\ldots,b,\ldots)$, $b{:}\,(\sigma \Rightarrow \tau)$ and $a \in \mathsf{non\text{-}linear\text{-}stratified}$ implies $b \in \mathsf{non\text{-}linear\text{-}stratified}$,
4. $a \in \mathsf{non\text{-}linear\text{-}stratified}$ and $a \sim b$ implies $b \in \mathsf{non\text{-}linear\text{-}stratified}$.

where, C and C' are arbitrary combinators. We say a is *linear-stratified* if $a \notin \mathsf{non\text{-}linear\text{-}stratified}$.
Now, we restrict the application of the rules $\Uparrow$ and $\mathsf{map}\Uparrow$ using the $\mathsf{non\text{-}linear\text{-}stratified}$ instead of $\mathsf{non\text{-}linear}$. Let $\Uparrow_r$ and $\mathsf{map}\Uparrow_r$ be the restricted version of these rules. If a is linear stratified and $a \equiv \mathsf{C}(\ldots,b,\ldots)$, then we can assume that $M^\lambda(b)$ was already defined when defining $M^\lambda(a)$.

Theorem 16: Considering the simplifying assumption that the intended interpretation of every index sort σ is infinite, then the rules idx, $\Downarrow$, $\Uparrow_r$, $\mathsf{ext}_{\not\simeq}$, ext_r, $\mathsf{K}\Downarrow$, $\mathsf{map}\Downarrow$, $\mathsf{map}\Uparrow_r$, $\epsilon_{\not\simeq}$ and $\epsilon\delta$ are sound, terminating and complete.

Proof: The proof is similar to the proof of Theorem 8. Let $\overline{a}$ be the greatest constant, with respect to the order $\prec$, in the equivalence class containing a. Now, we define $M^\lambda(\overline{a})$ only after all constants $\overline{b} \prec \overline{a}$ were already defined. Similarly, if $f{:}\,(\sigma_1,\ldots,\sigma_k) \to \tau$ is uninterpreted, then we define $M^\lambda(f)$ after all all constants of sorts $\sigma_1,\ldots,\sigma_k$ and τ were already defined. If $\overline{a}$ is linear-stratified and $a \equiv \mathit{map}_f(b_1,\ldots,b_k)$, then we define $M^\lambda(\overline{a})$ as:

$$M^\lambda(\overline{a})(\iota) = \begin{cases} \nu \text{ if } (\iota,\nu) \in \mathsf{graph}(\overline{a}), \\ M^\lambda(f)(M^\lambda(b_1)(\iota),\ldots,M^\lambda(b_k)(\iota)), \text{ otherwise.} \end{cases}$$

The construction for definitions of the form $a \equiv \mathit{store}(b,i,v)$ is similar. If $\overline{a}$ is not linear-stratified, then we use the same construction used in the proof of Theorem 8. After defining $M^\lambda(\overline{a})$, we make $M^\lambda(a) = M^\lambda(\overline{a})$ for every a in the equivalence class of $\overline{a}$. ∎

The proof of Theorem 4 established that the reduction from $T_A \oplus T_{\mathsf{core}}$ to T_{core} required at most a cubic number of new terms. The reduction of $T_{\mathsf{CAL}} \oplus T_{\mathsf{core}}$ to T_{core} can also be bounded by a polynomial number of new terms, using a similar argument, and:

Theorem 17: If the satisfiability problem for T_{core} is NP-complete, then the satisfiability problem for $T_{\mathsf{core}} \oplus T_{\mathsf{CAL}}$ is also NP-complete.

VI. Experimental Results

First, let us describe implementation details that are relevant for reproducing our results. Hence, we describe how the proposed inference rules were implemented in the SMT solver Z3. The rule idx is applied whenever a definition of the form $a \equiv store(b, i, v)$ is created. The rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$ are only used when the input formula contains the combinators K and map_f. In this case, $\epsilon\delta$ is applied when an array constant is created, and $\epsilon_{\not\simeq}$ is delayed. Actually, we use *model-based instantiation* for guiding the application of the rule $\epsilon_{\not\simeq}$. The idea is to build a candidate model M^λ without even using $\epsilon_{\not\simeq}$, if in this model $M^\lambda(\epsilon_\sigma) \neq M^\lambda(i)$ for every i ($\neq \epsilon_\sigma$) used as an index, then we have a valid model. Otherwise, we expand $\epsilon_{\not\simeq}$ and continue. The rule $\mathsf{ext}_{\not\simeq}$ is applied when p is assigned to false, and the application of ext_r is delayed. Before applying ext_r we build the set already-diseq. We use a simple indexing technique, called *use-lists*, for applying the remaining rules. Given a definition $a \equiv \mathsf{C}(\ldots, b, \ldots)$, we say a is a parent of b. The use-list data-structure stores the parents of each array constant a. Recall that we use an union-find data-structure for implementing equivalence classes. Then, whenever the union operation is performed we use the use-lists to find new matches for the remaining inference rules. The sets foreign, non-linear and non-linear-stratified are implemented as mappings from equivalence class representatives to Booleans.

Fig. 7 contains two scatter-graphs [3] comparing the performance of Z3 with and without the rule $\Uparrow_r$ in all 2244 benchmarks in the QF_AUFLIA division of SMT-LIB. Each point on the plots represents a benchmark. On each plot the x-axis is the CPU time, in seconds, taken by Z3 using $\Uparrow_r$, and y-axis in the first graph is for Z3 using $\Uparrow$, and in the second graph is for Z3 delaying the application of $\Uparrow_r$. Points above the diagonal are then benchmarks where Z3 with $\Uparrow_r$ is faster. The scatter-graphs clearly show that rule $\Uparrow_r$ increases Z3's performance in hard instances. Note that delaying the application of $\Uparrow_r$ increases the performance in unsatisfiable benchmarks because most array constants are linear, and, consequently, this rule is not needed. However, in satisfiable instances the rule is eventually applied and performance degrades.

VII. Conclusion

We described efficient and simple decision procedures for the array theory and combinatory array logic. The new combinators admit a simple theory of sets and bags. The theory of sets has already been used in real applications at Microsoft (e.g., the program exploration tool Pex and SpecExplorer). The decision procedure is presented as a set of inference rules on top of a core solver which provides basic capabilities. We also described the implementation techniques used to efficiently implement these inference rules. Moreover, in our approach the index domain of an array can be finite (e.g., bit-vectors). We also described several filters for minimizing the number of times an inference rule needs to be applied while retaining completeness. In particular, our experiments show that rule $\Uparrow_r$ improves the performance of Z3.

[3] Data available at http://research.microsoft.com/~leonardo/fmcad09

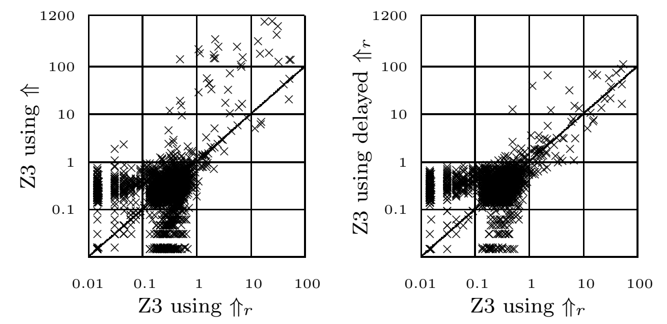

Fig. 7. Z3 on QF_AUFLIA Benchmarks

References

[1] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. (1962) 21–28

[2] Suzuki, N., Jefferson, D.: Verification Decidability of Presburger Array Programs. J. ACM **27** (1980) 191–205

[3] Jaffar, J.: Presburger Arithmetic With Array Segments. Inf. Process. Lett. **12** (1981) 79–82

[4] Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: CAV. (1994)

[5] Manolios, P., Srinivasan, S.K., Vroon, D.: Automatic memory reductions for RTL model verification. In: ICCAD. (2006)

[6] Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS. (2001)

[7] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. **10** (2009)

[8] Lynch, C., Morawska, B.: Automatic decidability. In: LICS. (2002)

[9] Brummayer, R., Biere, A.: Lemmas on Demand for the Extensional Theory of Arrays. In: SMT. (2008)

[10] Kapur, D., Zarba, C.G.: A Reduction Approach to Decision Procedures. Technical Report TR-CS-1005-44, University of New Mexico (2005)

[11] Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV. Volume 4144 of LNCS. (2006)

[12] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. (2008)

[13] Goel, A., Krstic, S., Fuchs, A.: Deciding Array Formula with Frugal Axiom Instantiation. In: SMT. (2008)

[14] Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV. (2007) 519–531

[15] Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodriguez-Carbonell, E., Rubio, A.: A Write-Based Solver for SAT Modulo the Theory of Arrays. In: FMCAD. (2008) 1–8

[16] Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI. (2006) 427–442

[17] Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. Ann. Math. Artif. Intell. **50** (2007) 231–254

[18] Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: FoSSaCS. (2008)

[19] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (2006)

[20] de Moura, L., Bjørner, N.: Model-based Theory Combination. In: SMT. (2007)

[21] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52** (2005) 365–473