Deciding Array Formulas
with Frugal Axiom Instantiation

Amit Goel\textsuperscript{1}, Sava Krsti\textsuperscript{c}\textsuperscript{1}, and Alexander Fuchs\textsuperscript{2}

\textsuperscript{1} Strategic CAD Labs, Intel Corporation
\textsuperscript{2} The University of Iowa

Abstract. How to efficiently reason about arrays in an automated solver based on decision procedures? The most efficient SMT solvers of the day implement “lazy axiom instantiation”: treat the array operations read and write as uninterpreted, but supply at appropriate times appropriately many—not too many, not too few—instances of array axioms as additional clauses. We give a precise account of this approach, specifying “how many” is enough for correctness, and showing how to be frugal and correct.

1 Introduction

Suppose a theory $T$ talks about some operations $f_1, \ldots, f_n$, specified axiomatically. If a query (set of quantifier-free formulas) $\Phi$ of this theory is unsatisfiable, then there exists a finite set $L$ of axiom instances such that $\Phi \cup L$ is unsatisfiable in the “empty theory” where nothing is assumed about the $f_i$ except that they are functions of the appropriate type. This existence is generally non-constructive (classical undecidability), but for some theories we are in luck. Kapur and Zarba \cite{13} show that, barring integers, theories of the commonly used data types allow equisatisfiable “reduction” of this kind. In particular, a simple reduction to the theory of uninterpreted functions works for the McCarthy theory of arrays:

\[
\begin{align*}
\text{read(write}(a, i, x), j) &= \begin{cases} 
  x & \text{if } i = j \\
  \text{read}(a, j) & \text{otherwise}
\end{cases} \quad (\text{read-over-write}) \\
\end{align*}
\]

\[
\begin{align*}
a \neq b \Rightarrow \exists i. \text{read}(a, i) \neq \text{read}(b, i) \quad (\text{extensionality})
\end{align*}
\]

The array reduction procedure in \cite{13} is simple to describe and verify, but unsuitable for direct implementation because of a large number of unnecessary axiom instances it introduces. Of the few existing solvers that handle array benchmarks, the most efficient ones (\textit{Yices} \cite{10} and \textit{Z3} \cite{11}) are indeed careful to generate axiom instances in a lazy fashion. However, their exact instantiation algorithms have not been published. We fill this gap by contributing a fully specified decision procedure with frugal axiom instantiation and proving it correct.

A proper setup to state and prove results of this kind requires a precise enough concept of a theory solver. This is a non-trivial issue in itself, since modern SMT solvers—to which we would like our results to be clearly applicable—combine a SAT solver and several theory solvers in architectures with complex
flow of data. In §2, we define a view of solvers that captures axiom instantiation, abstracting away other features that, even if indispensable for combination, can be made opaque without loss of precision and generality. In §3, we define a theory of updatable functions, a simple extension of the “theory of uninterpreted functions” into which array formulas readily translate. Then we give rule-based descriptions of three progressively finer solvers for this theory and briefly discuss their frugality. Correctness proofs are moved to the appendix and implementation is discussed in §4.

Related Work. The interaction between a SAT solver and a theory solver that creates axiom instances (and other lemmas) was first formalized by Barrett et al. [4]; our alternative in §2.2 is closer to the implementation level and directly applicable. Solving formulas about arrays started with the venerable Simplify [9] and its sophisticated quantifier instantiation mechanism; it is generic and thus suitable for any axiomatic theory, but it does not guarantee completeness. UCLID is complete for the non-extensional array fragment [8]. The first complete procedures were implemented in solvers of the CVC family [3], based on the theoretical work by Stump et al. [20]. Their rule-based system is more elaborate and a quick comparison of frugality is hardly possible. Armando et al. [2] construct a complete procedure from axioms by using superposition-based rewriting techniques, but do not look for optimizations. Dutertre and de Moura’s solver Yices [10] and de Moura and Bjørner’s Z3 [11] deal with arrays by controlled axiom instantiation, the details of which are not published. Recent papers by Bradley et al. [7] and Ghilardi et al. [12] on array procedures focus on extending the language beyond the standard read/write; they do use axiom instantiation, just without emphasis on frugality.

2 Solvers

We adopt the typed setting as in [14], where we refer for precise definitions of signatures; of types, terms, formulas associated with each signature; and of theories over a given signature. Let us recall here that parametric types (involving type variables) are allowed; that the concrete type Bool and logical symbols including equality are implicitly present in every signature, and that formulas are just terms of type Bool. Given a theory $T$ and a set of formulas $\Phi$ over its signature, we say that $\Phi$ is satisfiable if it has a model. The $T$-validity $\models_T \Phi$ means that the negation of $\Phi$ is unsatisfiable. Again, refer to §2 of [14] for full definitions.

Define $\phi \ll \psi$ to mean $\models \phi \iff (\exists \bar{y})\psi$, where $\bar{y}$ is the string of variables that occur in $\psi$, but not in $\phi$. This equisatisfiable expansion relation is a strong form of equisatisfiability: every model of $\psi$ restricts to a model of $\phi$, and every model of $\phi$ extends to a model of $\psi$. Generalize to $\phi \ll_T \psi$, standing for $\models_T \phi \iff (\exists \bar{y})\psi$.

Define a defset (shorthand for “definitional set of equations”) to be a set of the form $\{u_1 = e_1, \ldots, u_n = e_n\}$, where the $u_i$ are distinct variables and the $e_i$ are terms such that $u_i$ occurs in $e_j$ only if $j > i$. We call $u_i$ the proxy for $e_i$. 
For every quantifier-free formula $\phi$ over the union signature $\Sigma_1 + \cdots + \Sigma_n$, we have $\phi \ll \phi_0 \land \phi_1 \land \cdots \land \phi_n$, where $\phi_0$ is a set of propositional clauses and each $\phi_i$ ($i \geq 1$) is a defset over $\Sigma_i$. Purification algorithms as in [19, 14] compute $\phi_0, \ldots, \phi_n$ from a given $\phi$.

### 2.1 Combined SMT Solvers

Figure 1 gives a high-level picture of a modern SMT solver, showing the key state components of the abstract system NODPLL [14], which itself is a generalized, simplified, and elaborated version of the abstract DPLL($T$) framework [18]. It comprises the solvers $T_B$ and $T_E$ for booleans and for equality respectively, and solvers $T_1, T_2, \ldots$ for other theories. The communication between solvers goes mainly by exchange of *interface literals*: propositional literals and (dis)equalities between variables.

In a simple scenario, the set of literals is fixed at the initialization time, when the solvers are given their defsets $\Psi_i$ obtained by purifying the mixed input formula, and their *literal stacks* $M_i$ are all empty. If $T_i$ can infer a new literal as a consequence of $\Psi_i$ and $M_i$, it can put it on its stack $M_i$, and then the same literal can be *propagated* onto the other stacks $M_j$ as well. When no $T_i$ can make progress by such inference, the SAT solver $T_B$ will put a new literal on its stack speculatively (*decision*), initiating thus a new round of inferences by itself and other solvers (*new decision level*). When one of the solvers detects that its logical state $M_i \land \Psi_i$ is inconsistent, it sets the *conflict set* $C$ to a subset of $M_i$ that contradicts $\Psi_i$. At this point we know that the sequence of speculative decisions is inconsistent and we need to backtrack. A sequence of *explanation* calls to the solvers ensues, where at each step $C$ is modified into another set of literals that is inconsistent with the union of the $\Psi_i$. The set $C$ remains a subset of the union of the $M_i$ but gets progressively pushed back in time. This *conflict analysis* process results in finding an earlier decision level that is logically inconsistent with our last speculative decision. Then every solver *backjumps* to that decision level, a new literal (the negation of some literal from the current level) is deduced, and we continue as before. We are done either when a conflict is detected at level 0 (inconsistency), or there are no more literals to assert speculatively (model exists).

Completeness of the combined solver is a complex matter. By a classical result of Nelson and Oppen, completeness is guaranteed if all participating theories are convex and their solvers propagate every implied equality between shared non-boolean variables [17]. It is guaranteed also if we introduce proxy boolean variables for every equality between shared non-boolean variables. This is the method of *delayed theory combination* of Bozzano et al. [6, 19]. It relieves the
solvers from the obligation to propagate equalities and the theories do not need to be convex, but it comes with the cost of extensive proxying which forces the SAT solver to split over too many variables. The *splitting-on-demand* framework of Barrett et al. [4] allows theory solvers to introduce proxied equalities as needed and, more generally, to introduce new variables and communicate facts about them (“lemmas”) to the system. Solving by axiom instantiation fits naturally into this framework, with axiom instances being the lemmas.

### 2.2 Theory Solvers

Starting to pin down the requirements on theory solvers that include mechanisms for adding new variables and lemmas, let us view the state of a solver abstractly as a sextuple:

| $V$ | a set of variables; $V_{\text{Bool}}$ denoting its subset of boolean variables |
| $\Psi$ | a defset with all variables in $V$ |
| $M$ | a partial assignment sequence over $V_{\text{Bool}}$ |
| $L$ | a set of clauses over $V_{\text{Bool}}$ such that $\Psi \models_T \phi$ for every $\phi \in L$ |
| status | $\text{NO}_CFLCT$ or $\text{CFLCT}$ |
| local | solver-specific state |

We will use the record notation $s.V$, $s.\Psi$, . . . , $s.\text{local}$ to refer to components of a particular state $s$. In addition, we assume there is a function $\Lambda$ that defines the *logical state* $\Lambda(s)$—the formula represented by the state $s$. It is the conjunction of $s.\Psi$, $s.M$, and a formula derived in the solver-specific manner from $s.\text{local}$.

Each solver is specified for a fragment $F$ of a specific theory $T$. The theory-specific heart of the solver is abstracted in the $\text{local}$ component of the state and in the state changes that modify it. We make only these general requirements:

1. *(i)* if a state-to-state transition modifies $M$, then all it does is addition of a literal, exactly as described by the rule Literal in Figure 2 below;
2. *(ii)* given any defset $\Psi_0$ in the fragment $F$, there exists a well-defined initial state $s_{\text{init}}(\Psi_0)$ such that $\Psi_0 \ll T \Lambda(s_{\text{init}}(\Psi_0))$;
3. *(iii)* $\Lambda(s) \ll T \Lambda(s')$, for transitions $s \rightarrow s'$ of the solver that do not modify $M$.

Regarding *(i)*, notice that actual solvers (as conveyed in §2.1) get to know new literals either by being told, or by their own inference; the abstract view we adopt here conveniently obscures the difference between the two. Notice also that our abstract model goes astray from §2.1 by not allowing (dis)equalities as interface literals (members of $M$). This is done for convenience and the loss of generality is small; all proofs would extend without difficulty to the more general setting. Finally, the solver’s activities related to conflict analysis and backtracking are all ignored as not pertaining to this formalization.

A state $s$ will be called *conflicting* or *non-conflicting* depending on whether $s.\text{status}$ is $\text{CFLCT}$ or $\text{NO}_CFLCT$. A state is *final* if it is reachable from an initial state and there are no transitions from it. A solver is *sound* if the existence of a run that begins with $s_{\text{init}}(\Psi_0)$ and ends in a conflicting final state...
s implies that $\Psi_0 \land s.M$ is $T$-unsatisfiable. A solver is complete if the existence of a run that begins with $s_{\text{init}}(\Psi_0)$ and ends in a non-conflicting final state $s$ with $s.L \land s.M$ (propositionally) satisfiable, implies that $\Psi_0 \land s.M$ is $T$-satisfiable. The following lemma expressing soundness and completeness in terms of final states is a consequence of assumptions (i-iii).

**Lemma 1.** (a) A solver is complete if $\Lambda(s)$ is $T$-satisfiable for every non-conflicting final state $s$ for which $s.L \land s.M$ is (propositionally) satisfiable.

(b) A solver is sound when $\Lambda(s)$ is $T$-unsatisfiable for every conflicting state $s$.

The underlined phrases indicate an important subtlety. If we remove them from the text, we will still have a formally correct definition of a solver’s completeness, and (in Lemma 1(a)) a sufficient condition for it. But that condition could be too strong! Our formulation expresses a natural expectation by the solver of its environment: heed the lemmas you are getting from us so when you give us a new literal we can trust that none of our lemmas is violated. The completeness proof of the array solver described in §3 below will go by checking the condition in Lemma 1(a) and would not work with the stronger condition.

Having weakened the concept of individual solvers’ completeness, we need to justify that it is still strong enough to guarantee completeness of the combined solver. This requires the additional assumption that the SAT solver does add all the generated lemmas to its clause base. A complete argument would need a precise definition of the combined solver and can be formulated without difficulty within the NODPLL system [14].

### 3 A Solver for Arrays as Updatable Functions

The parametric theory of uninterpreted functions [15] has a signature consisting of the type constructor $\Rightarrow$, interpreted as the function space operator, and the function symbol $\lambda([\alpha \Rightarrow \beta, \alpha] \rightarrow \beta)$ interpreted as function application. The theory $U$ of updatable functions is obtained by adding to this the symbol $U([\alpha \Rightarrow \beta, \alpha, \beta] \rightarrow ([\alpha \Rightarrow \beta])$ whose meaning is the update operator: $U(a, i, x)$ is the function $b$ such that $b@i = x$ and $b@j = a@j$ for $j \neq i$.\(^3\) Since arrays can be viewed as functions, with operations read and write seen as $@$ and $U$, solvers for $U$ are solvers for the array read/write theory as well. The application symbol will be omitted; we will simply write $ai$ instead of $a@i$.

For every $U$-defset $\Psi$ there is a defset $\Psi'$ such that $\Psi \lll \Psi'$ and $\Psi'$ is “flattened” so that all its equations have one of the following forms:

$$\begin{align*}
p & \triangleq (u = v) & x & \triangleq ai & b & \triangleq U(a, i, x)
\end{align*}$$

(1)

Read $\triangleq$ here just as $=$; the little triangle is just to remind us that the equality is definitional. We will assume that $\text{Bool}$ is the only concrete type used, and that its use is limited to range positions (no type of a variable is allowed to contain

\(^3\) The theory $U$ can be easily extended with the constant function symbol $K^{\beta - ([\alpha \Rightarrow \beta])}$ interpreted as $K(x) = \lambda i.x$. Only for simplicity, we restrict ourselves to $U$.\(^3\)
\[ \text{Boo} \Rightarrow \sigma \text{ as a subexpression}. \] It is well-known that without the last assumption, or some similar condition, even the congruence closure algorithm would not be complete. (See the discussion about cardinality constraints in [15].)

We proceed to describe three related solvers for \( \mathcal{U} \), named \( \text{UPD}_0 \), \( \text{UPD}_1 \) and \( \text{UPD}_2 \), that conform to the requirements set in §2.2. The solvers’ defsets are of the form (1), and their only local state component is an equivalence relation \( \sim \) on the set \( V - V_{\text{Bool}} \) of non-boolean variables. The \( \sim \)-equivalence class of \( u \) will be denoted \([u]\). We will write \( \sim_u \) when referring to \( \sim \) at a given state \( s \). By definition, the theory-specific contribution to the logical state function \( \Lambda(s) \) is the conjunction of all equalities of the form \( u = v \), where \( u \sim_v \). Recall that, in addition to this, \( \Lambda(s) \) contains \( s.M \) and \( s.\Psi \) as conjuncts. Intuitively, \( u \sim_v v \) means “\( u \) and \( v \) are known to be equal in the state \( s \)”, i.e., it is an invariant that \( u \sim_v v \) implies \( \Lambda(s) \models \mathcal{U} \ u = v \).

Define \( a \wedge_i b \) to mean that for some \( x \), the equation \( a \triangleq U(b, i, x) \) is in \( \Psi \). The equivalence relation generated by \( \sim \) together with all relations \( \wedge_i \) will be denoted \( \wedge \). The relations \( \wedge_i \) will not change from state to state, but \( \wedge \) will. Intuitively, if \( a, b \in V^{\Psi \Rightarrow \tau} \), then \( a \wedge_{s} b \) implies that in the state \( s \) we know that \( a \) and \( b \) agree on all but finitely many arguments.

Define \( \text{proxied}_s(e) \) to mean that \( e \) occurs in some proxy equation \( (u \triangleq e) \in s.\Psi \), in which case we also write \([(e)]_s = u\). Generalizing this to arbitrary terms, define \([(e)]_s \) to be the term obtained from \( e \) by recursively replacing subterms with their \( s.\Psi \)-proxies until there are no \( \text{proxied}_s \)-subterms anymore. The set \( s.\Psi \) of proxy equations will monotonically increase from state to state. We will suppress the subscript \( s \) from \( \text{proxied}_s \) and \([(e)]_s \) when it is clear from the context.

The transitions for our solvers are given by the rules in Figure 2. Only the rules \( \text{Eq} \) and \( \text{Congr} \) modify the local state \( \sim \); together with \( \text{Conflict} \), they give an abstract presentation of the congruence-closure algorithm. The state-transforming function \( \text{proxy}(e) \) used in the remaining rules introduces a proxy variable for every application and equality subterm of \( e \), if it does not already exist, and returns the final boolean combination of propositional variables. For example, the action \( L := L + \text{proxy}(i \neq j \Rightarrow a j = b j) \) in \( \text{RoW}_{0,1,2} \) means addition of some of the equations \( x \triangleq a j, y \triangleq b j, p \triangleq (i = j) \), and \( q \triangleq (x = y) \) to \( \Psi \) with fresh variables \( x,y,p,q \), followed by addition of the clause \( p \lor q \) to \( L \). The criterion for adding \( x \triangleq a i \) to \( \Psi \) (and \( x \) and \( y \) to \( V \)) is that \( \Psi \) does not already contain a proxy for \( a i \); if it does, then \( x \) just denotes that proxy. In \( \text{Ext}_\text{arg} \), the action \( \text{proxy}(a = b) \) means adding \( p \triangleq (a = b) \) to \( \Psi \) if the proxy for \( a = b \) does not exist in \( \Psi \). The occurrences of \( \text{proxy} \) in \( \text{Ext}_{0,1,2} \) are to be understood analogously.

We use the convention that the same rule with the same parameters cannot fire twice. For \( \text{Literal} \) and \( \text{Conflict} \) this is already true, but guards of the other rules need an additional progress condition. For \( \text{Eq} \) and \( \text{Congr} \) these conditions are \( u \neq v \) and \([ai] \neq [bj] \) respectively. For \( \text{RoW}_{0,1,2} \), the progress means that \( L \) does not contain \([i \neq j' \Rightarrow a j' = b j'] \) for any \( j' \sim j \), and for \( \text{Ext}_{1,2} \) the condition is that \( L \) does not contain \([a' \neq b' \Rightarrow a' i \neq b'i] \) for any \( i \) and any \( a' \sim a \) and \( b' \sim b \). Finally, the progress condition for \( \text{Ext}_\text{arg} \) is \( \neg\text{proxied}(a' = b') \)
Fig. 2: Above and below the line in each rule are the rule’s guard and action respectively. The guard is the enabling condition on the state, and the action is the state change. The rules in the top box describe a congruence-closure algorithm. These rules are part of systems \textsc{upd}_0, \textsc{upd}_1, \textsc{upd}_2 as well. The lower three boxes present the read-over-write and extensionality rules that are specific for each of the three systems. The notation $V_{\text{init}}^{\sigma \Rightarrow \tau}$ in \textsc{Ext}_0 is for the initial set of variables of type $\sigma \Rightarrow \tau$, and $V^{\sigma \Rightarrow \tau}$ in \textsc{RoW}_0 is for the set of variables of type $\sigma \Rightarrow \tau$ that exist in the current state.
for any $a' \sim a$ and $b' \sim b$. Progress conditions are left implicit in Figure 2 in order to reduce clutter, but they are indispensable.

For a given input set $\Psi_0$ of proxy equations, the initial state $s_{\text{init}}(\Psi_0)$ is the tuple $\langle V, \Psi, [], \emptyset, \text{NO\_CFLCT}, \sim \rangle$, where

- $\Psi$ is obtained by adding new proxy equations to $\Psi_0$ if necessary so that for every $b \triangleleft U(a,i,x)$ in $\Psi$, we have $\text{proxied}(bi)$ and—if $x$ is of type $\text{Bool}$—$\text{proxied}(ai)$ too;
- $V$ is the set of all variables in $\Psi$;
- $\sim$ is generated by $[bi] \sim x$, where $b \triangleleft U(a,i,x)$ is in $\Psi$.

It is straightforward to check that conditions (i-iii) are satisfied and so the systems $\text{UPD}_{0,1,2}$ are theory solvers in the sense of §2.2.

**Theorem 1.** $\text{UPD}_0$, $\text{UPD}_1$, and $\text{UPD}_2$ are terminating, sound, and complete.

We can offer here only some high-level remarks about the proof of Theorem 1. The completeness part is the most difficult and, like other authors (e.g., [20, 7, 13]), we prove it by constructing a syntactic model for $\Lambda(s)$ when $s$ is a final non-conflicting state. The amount of the “syntactic material” available in a final state directly affects the difficulty of the model construction. The system $\text{UPD}_0$ generously provides a syntactic witness for disequality of any two arrays $a,b$ of the same type, as well as the information whether $ai$ and $bi$ are equal or not, for any index $i$. This information suffices to build a model. Indeed, when used with the strategy that applies $\text{Ext}_0$ exhaustively, then $\text{RoW}_0$ exhaustively, and then lets $\text{cc}$ finish the job, $\text{UPD}_0$ is much like the reduction procedure of [13].

The conditions $c \nLeftarrow a$ and $a \nLeftarrow b$ in the guards of $\text{RoW}_1$ and $\text{Ext}_1$ greatly reduce the number of created lemmas. Intuitively, we do not lose completeness by imposing these restrictions because if two array variables are not in the same $\nLeftarrow$-class, then we can rely on the (assumed) infinite cardinality of the index type to ensure a witness for their disequality. The additional guard constraints $i \sim j$ and $a \sim b$ in $\text{RoW}_1$ and $\text{Ext}_1$ are justified by the observation that the relation $\sim$ expresses what the system at a given state knows about implied equalities between variables. They further curtail the generation of lemmas and also suggest strategies that prioritize the use of $\text{cc}$ rules over the lemma-generating ones.

In our final system $\text{UPD}_2$, creation of $\text{Ext}$-lemmas is not done until positively necessary: when an array disequality is explicitly asserted (put on the stack $M$). Unfortunately, this extremely frugal $\text{Ext}_2$ leads to an incomplete system: even if coupled with $\text{RoW}_0$, it would not refute the unsatisfiable query $\{i \neq j, b = U(a,i,x), b = U(a,j,y), fa \neq fb\}$ (example in §6.2 of [20]), because it would not generate the necessary extensionality lemma for $a = b$. A minimal remedy is provided by the rule $\text{Ext}_{\text{arg}}$; it will introduce a proxy variable for $a = b$ so that the SAT solver will eventually split on it, enabling the necessary firing of $\text{Ext}_2$.

---

4 If we allowed concrete types other than $\text{Bool}$, we would need to forbid the finite ones to occur as index types, and we would need to impose this condition $\text{proxied}(ai)$ whenever the type of $x$ is finite.

5 For the full proof, see [www.cs.uiowa.edu/~fuchs/PAPERS/array_solver.pdf](www.cs.uiowa.edu/~fuchs/PAPERS/array_solver.pdf).
4 Experiments

We have extended the congruence-closure module of the SMT solver DPT, written in OCaml, with the rules of the system upd2 [1]. Performance of our initial implementation⁶ is compared in Figure 3 with CVC3 [3] and Yices [10] on the set of qf_auflia benchmarks whose arithmetical content is expressed in the difference logic [5]. (Of the three participants in the qf_auflia category at the SMT-COMP 2007 competition, Yices was the winner, and CVC3 was the only open-source solver.) DPT proved competitive with Yices and significantly better than CVC3. With 100 sec timeout, DPT used 2787 sec and timed out on 6 benchmarks, CVC3 used 30729 sec and timed out on 230 benchmarks, and Yices used 1385 sec and timed out on 5 benchmarks.⁷

To emphasize the benefit of ⋊-related frugality, we have also compared the three solvers on ten benchmarks Φ₁₀₀, Φ₂₀₀, ..., Φ₁₀₀₀, where Φₙ is the satisfiable formula

\[ U((\cdots U(a₀, i₁, x₁), \ldots), iₙ, xₙ) \neq U((\cdots U(b₀, i₁, x₁), \ldots), iₙ, xₙ) \]

expressing disequality of the results of the same sequence of n updates applied to a₀ and b₀. DPT takes less than a second even for Φ₁₀₀₀, while CVC3 times out (after 100 sec) on Φ₄₀₀, and Yices times out on Φ₅₀₀.

5 Conclusion

This paper is part of our ongoing project to design a high-performance SMT solver, open-sourced and with strong theoretical foundations [1, 15, 14]. We have given a rule-based axiom-instantiating solver for a parametric theory of arrays, where formulas can refer to multiple and arbitrarily nested array types. Our description lends itself to implementation in a fairly direct manner.

We have clarified the conditions that solvers of the axiom-instantiating and similar kinds must meet in order to correctly participate in an SMT combination

⁶ We ran DPT 1.2 with the --q option and the variable OCAMLRUNPARAM set to 51200000.

⁷ The performance on these benchmarks depends not only on the array decision procedure, but also on the linear arithmetic procedure, the combination method, pre-processing, and the implementation language.
framework, and we have proved that our array solver meets the conditions. Competitiveness of our implementation proves that we have struck a balance between frugality and completeness. In this, we have likely not achieved the optimum, but we have provided a setup for checking correctness of better algorithms to be found. We expect to use the same setup for building an axiom-instantiating solver for the basic theory of sets. We suspect there are other theories that could use specific frugal instantiation algorithms, perhaps the theory developed for verifying heap-manipulating programs in [16].

Acknowledgments. Thanks to Clark Barrett, Jim Grundy, Albert Rubio, and Cesare Tinelli with whom we discussed ideas about the array solver and received comments, criticism, or alternative solutions.

References


Appendix

A Comments On Rules and Frugality

Read-over-write. In each update equation \( a = U(b, i, x) \) in \( \Psi \), the variables \( a, b \) are of the same function type \( \sigma \Rightarrow \tau \). This equation triggers creation of lemmas of the form \( i \neq j \Rightarrow aj = bj \) for some “index” variables \( j \in V^\sigma \). We write \( a \times_i b \) instead of \( (a = U(b, i, x)) \in \Psi \) in the guard of the RoW rules to emphasize the irrelevance of \( x \) for lemma creation. The rules RoW_0, RoW_1, RoW_2 differ in the number of index variables \( j \) selected. For RoW_0, the only constraint on \( j \) is that it occurs as argument of a function \( c \) of type \( \sigma \Rightarrow \tau \). For example, since \( \text{proxied}(ai) \) holds by the assumption on the system initialization, RoW_0 will fire with \( c \equiv a \) and \( j \equiv i \); the only useful effect of the corresponding action will be the creation of a proxy for the term \( bi \). More generally, the lemma \( i \neq j \Rightarrow aj = bj \) will be useless whenever \( i \sim j \) holds, because (it is the system’s invariant that) \( i \sim j \) implies \( \Psi_{\text{init}}, M \models_U i = j \).

To explain the requirement in rule RoW_1 that the variable \( c \) be from the same \( \bowtie \)-class that contains \( a \) and \( b \), recall first that the relations \( \bowtie_i \) are not modified by any transitions. Fix a \( \bowtie \)-class \( A \) and let \( I \) be the set of indexes \( i \) such that \( a \bowtie_i b \) holds for some \( a, b \in A \). In a sense, \( I \) is the set of relevant index variables for the class \( A \). (We know that in any interpretation, all functions in \( A \) must agree on all arguments that are not represented by elements of \( I \).) The guard \( c \bowtie a \) in RoW_1 ensures that applications of functions to “irrelevant variables” are not unnecessarily introduced in lemmas.

In RoW_2, the “relevant” indexes for \( a, b \) are further restricted to arguments of functions \( c \) that are directly \( \sim \)-related to either \( a \) or \( b \). This restriction is

\(8\) Note that \( i \not\sim j \) is just the negation of \( i \sim j \) and does not mean that \( i \neq j \) follows from the current logical state of the solver.
important, but not as severe as it may seem, because of the “relevance propagation”. To see this, consider the situation where \( a \times_i b, a' \sim a, \) and \( c \times_k a' \). If \( \text{proxied}(c_j) \) holds and \( j \neq k \), then an application of \( \text{RoW}_2 \) will first generate the lemma \( k \neq j \Rightarrow a'j = c_j \), which will make \( \text{proxied}(a'j) \) true, so that the lemma \( i \neq j \Rightarrow a_j = b_j \) will also be generated by \( \text{Ext}_2 \), even if not immediately possible. Note, however, that if \( j \equiv k \), then these two lemmas will not be created by \( \text{Ext}_2 \).

The rule \( \text{Ext}_2 \), unlike \( \text{Ext}_1 \), recognizes the irrelevance of \( k \) for \( a, b \).

**Extensionality.** The presence of the extensionality lemma \( a \neq b \Rightarrow ai \neq bi \) ensures a syntactic witness of disequality of functions \( a \) and \( b \). The rules \( \text{Ext} \) use a fresh index variable for each lemma created. For \( \text{Ext}_1 \), the restrictions are that the types of \( a \) and \( b \) agree and that \( a \) and \( b \) are present at initialization time. The guard of \( \text{Ext}_1 \) strengthens the first conditions and omits the second because it is essentially implied. Indeed, since the relations \( \times_i \) are the same in all states, \( a \neq b \) and \( a \not\equiv b \) imply that there exist \( a', b' \in V_{\text{init}} \) such that \( a' \sim a \) and \( b' \sim b \); in this situation, the extensionality lemmas for pairs \( a, b \) and \( a', b' \) have the same effect. The guard of \( \text{Ext}_2 \) is obviously stronger than that of \( \text{Ext}_1 \).

As mentioned in the main text, the purpose of \( \text{Ext}_{\text{arg}} \) is just to introduce a proxy propositional variable for a function equality \( a = b \), so that \( \text{Ext}_2 \) can later create the extensionality lemma for \( a, b \) if needed.

**Counting the Lemmas.** For illustration, let us use the formulas \( \Phi_n \) from §4, to estimate the number of lemmas created by systems \( \text{UPD}_0, \text{UPD}_1 \), and \( \text{UPD}_2 \).

The flattened equisatisfiable expansion of \( \Phi_n \) is \( \neg p \land \Psi_n \), where \( \Psi_n \) is the defset

\[
(p \triangleq (a_n = b_n)) \land \bigwedge_{\nu=1}^{n}(a_{\nu} \triangleq U(a_{\nu-1}, i_{\nu}, x_{\nu}) \land b_{\nu} \triangleq U(b_{\nu-1}, i_{\nu}, x_{\nu})),
\]

with \( n \) array variables \( a_{\nu}, b_{\nu} \), and \( n \) index and value variables \( i_{\nu} \) and \( x_{\nu} \).

- **[UPD0]** The rule \( \text{Ext}_0 \) applies for every pair of array variables, introducing \( \approx 2n^2 \) boolean proxy variables for array (dis)equalities, and the same number of witness variables. Since there are \( 2n^2 \times \)-related pairs and \( \approx 2n^2 \) index variables, the number of \( \text{RoW}_0 \) lemmas generated is \( \approx 4n^3 \).

- **[UPD1]** In \( \Phi_n \), there are two \( \times \)-classes \( (a_{\nu} vs. b_{\nu}) \), and a little calculation shows that, in the worst case, the number of \( \text{Ext}_1 \) lemmas generated is \( \approx n^2 \) and the number of \( \text{RoW}_1 \) lemmas is \( \approx n^3 \).

- **[UPD2]** Since the only proxied array equality in \( \Psi_n \) is \( a_n = b_n \) and these two variables are not \( \times \)-related, there will be no \( \text{Ext} \)-lemmas generated when \( \text{UPD}_2 \) is run on \( \Phi_n \)!

**B Proofs**

2 Proof of Lemma 1

Consider any run \( \rho : s_{\text{init}}(\Psi_0) = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \). We claim that \( s_n, M \land A(s_i) \models \vdash s_n, M \land A(s_{i+1}) \) holds for every for every step \( s_i \rightarrow s_{i+1} \) of \( \rho \). This follows immediately from property (iii) of §2.2, except for steps \( s_i \rightarrow s_{i+1} \) based on the rule Literal. And for Literal-based steps we have \( A(s_{i+1}) = A(s_i) \land s_{i+1}, M \), which, together with the generally true fact \( \models s_n, M \land s_i, M \iff s_n, M \), implies
the claim. The claim just proved, together with the transitivity of $\ll \mathfrak{U}$ and property (ii) of $\S 2.2$, implies $s_n. M \land \Psi_0 \ll \mathfrak{U} \Lambda(s_n)$. Thus, $s_n. M \land \Psi_0$ and $\Lambda(s_n)$ are equisatisfiable, and both statements of the lemma follow.

3 Proof of Theorem 1

The guards of rules $\text{RoW}_0$, $\text{RoW}_1$, $\text{RoW}_2$ are progressively stronger, and the actions of all three rules are the same. The same can be said of the three rules $\text{Ext}_0$, $\text{Ext}_1$, and $\text{Ext}_2$ (see Comments on Rules above). Finally, the guard of $\text{Ext}_\text{arg}$ is stronger than the guard of $\text{Ext}_1$, and the action of $\text{Ext}_\text{arg}$ is part of the corresponding action of $\text{Ext}_1$.

Thus, the system $\text{upd}_0$ simulates $\text{upd}_1$, and $\text{upd}_1$ simulates $\text{upd}_2$, so it suffices to prove that $\text{upd}_0$ is terminating and sound, and that $\text{upd}_2$ is complete.

Termination of $\text{upd}_0$. Consider any run $\rho$: $s_{\text{init}}(\Psi_0) = s_0 \rightarrow s_1 \rightarrow \ldots$ of $\text{upd}_0$. Since $\text{Ext}_0$ can fire only once for any given pair $a, b \in s_0.V$, there are only finitely many steps in $\rho$ that are based on the rule $\text{Ext}_0$.

Consider the state-dependent set $V_{\text{arg}}$ of variables $j$ such that $\text{proxied}(c_j)$ holds for some $c$. The only transition that can increase $V_{\text{arg}}$ is $\text{Ext}_0$. We have just observed that there are only finitely many $\text{Ext}_0$-steps in $\rho$, so $V_{\text{arg}}$ stabilizes along $\rho$. Now, the rule $\text{RoW}_0$ is parametrized by quadruples $\langle a, b, i, j \rangle$ of which we know that $a, b, i \in s_0.V$ (because of $a \stackrel{\Delta}{=} b$) and $j \in V_{\text{arg}}$. Since there are only finitely many eligible quadruples, the number of $\text{RoW}_0$-steps in $\rho$ must be finite too.

Now we know that from some point on, the rules $\text{RoW}_0$ and $\text{Ext}_0$ do not apply, and so the set $s_n.V$ stabilizes. Let $V^*$ be its finite limit. The equivalence relation $\sim$ over $V^*$ is non-decreasing along the run $\rho$, so it must stabilize too, which implies that only finitely many steps in the run are based on rules $\text{Eq}$ and $\text{Cong}$. The rule $\text{Literal}$ can apply only finitely many times because it extends $M$ with a literal over $V^*$ that is not already in $M$; and $\text{Conflict}$ can apply only once. Thus, all rules apply only finitely many times, so $\rho$ is finite.

Soundness of $\text{upd}_0$. Referring to Lemma 1(b), it suffices to prove that $\Lambda(s)$ is unsatisfiable for every conflicting final state $s$. Indeed, any run that leads to such $s$ must use rule $\text{Conflict}$ at some step, so there exist $u, v, p$ such that $\neg p$, $p \triangleq (u = v)$, $u = v$ are all in $\Lambda(s)$.

Completeness of $\text{upd}_2$. In view of Lemma 1(a), it suffices to prove that the formula $\Lambda(s)$ has a model for every final non-conflicting state $s$ such that $s. L \land s. M$ is satisfiable. We fix such a state $s$ and just write $V, \Psi, \ldots$ for its components $s.V, s.\Psi, \ldots$. To reduce clutter, let us also write just $\text{RoW}$ and $\text{Ext}$ instead of $\text{RoW}_2$ and $\text{Ext}_2$. Define $\text{func}(a)$ to be the set of all pairs $\langle [i], [x] \rangle$ such that $x \triangleq a' i$ for some $a' \sim a$. Define $\text{dom}(a)$ to be the set of first components of
pairs that are in \( \text{func}(a) \). The following properties are vital.

\[
\begin{align*}
\text{if } a \sim b & \text{, then } \text{func}(a) = \text{func}(b) & (2) \\
M & \text{ is a complete assignment to variables in } V_{\text{Bool}} & (3) \\
\text{func}(a) & \text{ is a partial function } V/\sim \rightarrow V/\sim & (4) \\
\text{if } a \not\sim b & \text{, then } \text{dom}(a) = \text{dom}(b) \cup \{i\} & (5) \\
\text{if } a \not\sim b & \text{ and } a \not\equiv b, \text{ then } \text{func}(a) \neq \text{func}(b) & (\ast)
\end{align*}
\]

Property (2) follows from the definition of \( \text{func} \). Properties (3), (4), (5) follow from inapplicability of the rules \text{Literal}, \text{Congr}, \text{and RoW} respectively. Property (\ast) does not hold in general, but let us proceed assuming it does. When we finish the proof that \( A(s) \) is satisfiable under this assumption, we will show in the next subsection that, for any non-conflicting final state \( s \), there is another non-conflicting final state \( s' \) such that (\ast) holds at \( s' \) and such that \( A(s') \) contains \( A(s) \) as a conjunct. Then the proof will be complete.

Let \( T \) be the set of types of variables occurring in \( \Psi \) and let \( \sigma \leq \tau \) mean that \( \sigma \) is a subexpression of \( \tau \). Recall the three forms of equations in \( \Psi \), shown in (1). Partition \( \Psi \) into subsets \( \Psi^\tau \), where for each \( \tau \in T \), \( \Psi^\tau \) contains the equations \( p \triangleq (u = v) \) with \( \text{type}(u) = \tau \), and the equations \( x \triangleq a i \) and \( b \triangleq U(a, i, x) \) with \( \text{type}(a) = \tau \). Without loss of generality, \( T \) is closed under \( \leq \). Assuming \( T \) has \( n \) elements, put these elements in a linear ordering, starting with \( \text{Bool} \), and making sure that simpler types (with respect to \( \leq \)) precede the more complex ones. Let \( T_i \) be the set of the first \( i \) elements of \( T \). By assumption, \( T_1 = \{\text{Bool}\} \) and every \( T_i \) is closed under \( \leq \). Let \( \Psi_i \) be the union of all \( \Psi^\tau \) such that \( \tau \in T_i \) and let \( V_i \) be the set of non-boolean variables occurring in \( \Psi_i \).

Fix an interpretation \( \llbracket \tau \rrbracket \) of types \( \tau \in T \) that associates an infinite set \( \llbracket \alpha \rrbracket \) to each type variable \( \alpha \in T \). We need to define a variable assignment \( v \mapsto [v] \) that makes \( M \wedge \Psi \) satisfied. (If \( v \) has type \( \tau \), then \( [v] \) must be an element of \( \llbracket \tau \rrbracket \).

For every propositional variable \( p \) we know that either \( p \in M \) or \( \neg p \in M \), so we set \( [p] = \text{true} \) in the first case, and \( [p] = \text{false} \) in the second.

The assignment to non-propositional variables is defined in an inductive type-directed fashion: we proceed to show by induction on \( i \) that \( M \wedge \Psi_i \) has a “diverse” satisfying assignment, in which for every \( u, v \in V_i \)

\[ [u] = [v] \iff u \sim v \quad (6) \]

Our claim is true for \( i = 1 \) because \( \Psi_1 = \emptyset \). For the induction step, let \( \Psi_{i+1} = \Psi_i \cup \{\Psi^\tau\} \). The proof splits into three cases, depending on \( \sigma \).

\[ \text{Case 1: } \sigma \text{ is a type variable.} \]

Let \( X \) be the set of variables of type \( \sigma \) and let \( f : X/\sim \rightarrow [\sigma] \) be an arbitrary injection: it exists because \( [\sigma] \) is infinite. Define \([u] = f([u])^9\). This makes (6) true. It remains to check that all conditions in \( \Psi^\sigma \) are satisfied by this interpretation of variables. Every condition in \( \Psi^\sigma \) is of the form \( p \triangleq (u = v) \), where \( u, v \in X \). By (3), we have that one of \( p, \neg p \) is in \( M \). We

\[ ^9 \text{Read } f([u] \text{ as } f([u]).} \]

14
need to show that \([p] = [u = v]\), which is now equivalent to: \(p \in M\) if \(u \sim v\). 
And indeed, if \(p \in M\), we have \(u \sim v\), because otherwise the rule \textbf{Eq} would apply; also, if \(\neg p \in M\) then we have \(u \not\sim v\), because otherwise the rule \textbf{Conflict} would apply.

Case 2: \(\sigma = (\tau \Rightarrow v)\) and \(v \not\in \textbf{Bool}\). Let \(A, I,\) and \(X\) be the sets of all variables of types \(\tau \Rightarrow v, \tau,\) and \(v\) respectively. We need to define functions \([\sigma] : [\tau] \rightarrow [v]\) for all \(a \in A\) so that: (i) condition (6) holds for them; (ii) the conditions in \(\Psi^\sigma\) are satisfied too.

By induction hypothesis, the interpretation functions \(I : [\tau] \rightarrow [v]\) and \(X : X/\sim \rightarrow [v]\) induce injections \(I/\sim \rightarrow [\tau]\) and \(X/\sim \rightarrow [v]\).

Define the function \(f : A/\sim \rightarrow [\tau] \times [v]\) as the composition

\[
A/\sim \xrightarrow{f_1} 2([\tau]/\sim) \times ([v]/\sim) \xrightarrow{f_2} 2([\tau] \times [v])
\]

where \(f_2\) is induced by the injections mentioned above and \(f_1\) is defined by \(f_1([a]) = \text{func}(a)\). Note that property (2) implies that \(f_1\) is correctly defined.

Thus, \(f[a]\) is the image in \([\tau] \times [v]\) of \(\text{func}(a)\) seen as a subset of \((I/\sim) \times (X/\sim)\). It is a partial function whose domain \(D_a \subseteq [\tau]\) consists of all elements \([j]\) such that \(j \in \text{dom}(a)\).

Let \(D\) be the union of the domains \(D_a\) of the partial functions \(f[a]\), where \(a \in A\), and let \(R\) be the union of these functions’ ranges. Choose elements \(\varepsilon_1 \in [v], \varepsilon_2 \in [v] \setminus R\), and elements \(\delta_a \in [\tau] \setminus D\) so that \(\varepsilon_1 \neq \varepsilon_2\) and, for every \(a, b \in A, \delta_a = \delta_b \iff a \not\sim b\) (8)

Since \(D\) and \(R\) are finite and \([\tau]\) and \([v]\) are infinite, such elements \(\varepsilon_1, \varepsilon_2,\) and \(\delta_a\) clearly exist. For each \(a \in A\), define the interpretation \([a]\) as the extension of the partial function \(f[a]\) that maps \(\delta_a\) to \(\varepsilon_1\), and all other elements outside the domain of \(f[a]\) to \(\varepsilon_2\). We need to prove that, with this definition of functions \([a]\), the condition (6) is satisfied, as well as the conditions in \(\Psi^\sigma\). For (6), we have a chain of implications:

\[
a \sim b \xrightarrow{(1)} [a] = [b] \xrightarrow{(2)} \delta_a = \delta_b \land f[a] = f[b] \xrightarrow{(3)} a \not\sim b \land \text{func}(a) = \text{func}(b) \xrightarrow{(4)} a \sim b
\]

The first two implications follow from the just given definition of \([a]\) and \([b]\), the third follows from (8) and injectivity of \(f_2\) in (7); the fourth is a restatement of condition (6).

It remains to check that the conditions of \(\Psi^\sigma\) are satisfied. For those of the form \(p \not\triangleq (a = b)\) the argument is as in Case 1 above. Then for conditions of the form \(x \not\triangleq ai\), we just need to check \([x] = [a]([i])\). And indeed, we have \([x, x] \in \text{func}(a)\), which implies \([v, x] \in f[a]\), which in turn implies the desired equality by definition of \([a]\).

Consider finally conditions of the form \(b \not\triangleq U(a, i, x)\). We need to show (1) \([b](i) = [x]\), and (2) \([b](t) = [a](t)\) for every \(t \in [\tau]\) such that \(t \not\in [i]\). For
(†), we know that initialization furnishes a proxy equation \( y \triangleq bi \) such that \( y \sim x \). Thus, we have \( [b](\langle i \rangle) = [y] = [x] \), where the first equation is an instance of the fact proved in the previous paragraph, and the second follows by (6) (induction hypothesis). For (†), the proof splits into cases depending on whether or not \( t \in D_a \). If this condition is true, then the guard of the rule RoW is satisfied with \( c \) being the same as \( b \) and with some \( j \) such that \([j] = t\). Thus, \( \neg[i = j] \Rightarrow [aj = bj] \) is in L. From the assumption \([i] \neq [j]\) it follows that \( \neg[i = j] \in M \), because \( M \) is a complete assignment. Since \( M \) is consistent with \( L \), it follows that \([aj = bj]\) is in \( M \) too. This implies \([b](\langle j \rangle) = [a](\langle j \rangle)\), by an argument as in the previous paragraph. To finish the proof, consider the remaining case when \( t \notin D_a \). Now, by definition of \([a]\), we have that \([a](t)\) is either \( \varepsilon_1 \) or \( \varepsilon_2 \), depending on whether \( t = \delta_a \) or not. From (5) and \( t \neq [i] \), we have \( t \notin D_b \). Since \( a \times b \), we have \( \delta_a = \delta_b \), and \([b](t) = [a](t)\) follows.

Case 3: \( \sigma = (\tau \Rightarrow \text{Bool}) \). The proof is much the same as in Case 2. The interpretation function \( a \mapsto [a] \) is defined as before, with the exception that now we set \( \varepsilon_1 = \text{true} \) and \( \varepsilon_2 = \text{false} \). The implication chain (9) holds again, but for the implication \([a] = [b] \Rightarrow f[a] = f[b] \) we need a new argument. It uses the boolean-specific proxying in the initialization phase, thanks to which from the inapplicability of RoW we obtain a stronger form of condition (5): if \( a \times b \), then \( \text{dom}(a) = \text{dom}(b) \). We have that \([a] = [b] \) implies \( \delta_a = \delta_b \) as before, which implies \( a \times b \) by (8). Thus, \([a] = [b] \) implies \( \text{dom}(a) = \text{dom}(b) \), and the two together imply \( f[a] = f[b] \). The rest of the proof is unchanged.

**Attaining condition (†).** Define \( a \approx b \) to hold if and only if \( a \times b \) and \( \text{func}(a) = \text{func}(b) \). Clearly, \( \approx \) is an equivalence relation that contains \( \sim \). The relation \( \approx \) is defined for each state and, by its definition, the condition (†) is true in a state if and only if \( \approx \) is the same as \( \sim \). Moreover,

\[
\text{the equivalence relation generated by } \approx \text{ and all relations } \times_i \text{ is } \times \quad (10)
\]

Let \( \text{func}^+ \) be defined in the same way as \( \text{func} \), but with relation \( \approx \) playing the role of \( \sim \).

Let us fix a final non-conflicting state \( s \) of \( \text{UPD}_2 \), referring to \( \sim_s \) and \( \approx_s \) as \( \sim \) and \( \approx \) respectively. We have two important properties:

\[
\text{if } a \approx b \text{ and } \text{proxyed}(a = b), \text{ then } a \sim b \quad (11)
\]

\[
\text{func}^+ \text{ is a partial function} \quad (12)
\]

**Proof of (11).** Since \( a \approx b \) is proxied, we have that either \([a = b]\) or \([a \neq b]\) is in \( M \). If \([a = b] \in M \), then \( a \sim b \) follows because \( \text{Eq} \) does not fire at \( s \). Suppose then that \([a \neq b] \in M \). Since \( a \approx b \) implies \( a \times b \) and the rule \( \text{Ext}_2 \) does not fire at \( s \), there must exist a lemma in \( L \) of the form \([a \neq b] \Rightarrow [ai \neq bi] \). Consistency of \( s.M \land s.L \) and the fact that \( M \) is a complete assignment imply \([ai \neq bi] \in M \), i.e. \([x \neq y] \in M \), where \( x \) and \( y \) are proxies for \( ai \) and \( bi \). Now \( \langle \langle i \rangle, [x] \rangle \in \text{func}(a) \) and \( \langle \langle i \rangle, [y] \rangle \in \text{func}(b) \). Since \( a \approx b \) implies \( \text{func}(a) = \text{func}(b) \) and since \( \text{func}(a) \)}
is a partial function, it follows that \([x] = [y]\), i.e. \(x \sim y\). It follows that Conflict fires at \(s\), which we know is not true because \(s\) is final.

Proof of (12). It suffices to prove that \(a \approx b\), \(i \approx j\), proxied\((ai)\) and proxied\((bj)\) imply \([ai] \approx [bj]\). We claim that \(i \sim j\) holds. If it does not, then from non-applicability of Ext\(arg\) to \(s\) we can derive proxied\((i = j)\). But then (11) implies \(i \sim j\) and the claim is proved. Now, \(([i],[ai]) \in \text{func}(a)\) and \(([j],[bj]) \in \text{func}(b)\). Since \(\text{func}(a)\) and \(\text{func}(b)\) are the same partial function by assumption \(a \approx b\), the functionality property together with \(i \sim j\) implies \([ai] \sim [bj]\).

Let \(s^+\) be the state obtained by changing the relation \(\sim\) to \(\approx\). We claim that \(s^+\) is a final state of UPD\(_2\).

Proof of (13). We need to check that no rule of UPD\(_2\) can fire at \(s^+\). Since \(a \not\approx b\) implies \(a \not\sim b\) and since, as noted in (10), \(\star\) is the same for \(s\) and \(s^+\), we get that Ext\(arg\) and Ext cannot fire at \(s^+\) because they do not fire at \(s\). For the same reason, but more obviously, Literal and Eq are out too. Next, Congr does not fire because the relation \(\approx\) is congruence-closed, which is an equivalent reformulation of the property (12) that we already proved. If Conflict fires at \(s^+\), we have \([u \not= v]\) and \(u \approx v\) for some variables \(u, v\). But then (11) implies \(u \sim v\) and Conflict fires at \(s\), which we know is not true. Finally, suppose RoW fires with, say, the first conjunct of \(c \approx a \lor c \approx b\) true. Thus, func\((c) = func(a)\) and so we have proxied\((c'j')\) for some \(c' \sim a\) and \(j' \sim j\). Since RoW does not fire at \(s\), there is a lemma \(i \not= j'' \Rightarrow aj'' \not= bj''\) in \(L\), with \(j'' \sim j\). The same lemma prevents RoW from firing at \(s^+\) as well.

If property \((\ast)\) holds in \(s^+\), we are done. But again, this property need not hold in \(s^+\). We need to repeat the “plus construction” and get to the right final state in the limit. Precise definitions follow.

Define a sequence of equivalence relations \(\sim_0, \sim_1, \ldots\) so that \(\sim_0\) is just \(\sim\) (the relation \(\sim\) on \(s\)) and

\[a \sim_{n+1} b\] if and only if \(a \star b\) and func\(_n\)(a) = func\(_n\)(b).

Here, func\(_n\) is the function func associated with the relation \(\sim_n\). Note that for every \(n\), the relation generated by \(\sim_n\) and all relations \(\star_i\) is the same \(\star\).

Let \(s_n\) be the state obtained by replacing \(\sim\) with \(\sim_n\). We have \(s_1 = s^+_0\), \(s_2 = s^+_1\), \ldots and by what we proved above, each of the states \(s_n\) is non-conflicting final. Since the just defined sequence of equivalence relations is monotonic \((a \sim_i b\) implies \(a \sim_{i+1} b\)), it must stabilize in a finite number of steps. Thus, for some \(n\), we have \(s^+_n = s_n\), and this means that property \((\ast)\) holds in \(s_n\). Clearly, \(\Lambda(s)\) is a conjunct of \(\Lambda(s_n)\).

Example 1. The sequence \(s_n\) may take arbitrarily long to stabilize. Here is an example that requires two steps. Suppose that \(\Psi = \{b \triangleq U(a, i, x), g \triangleq U(f, j, b), p \triangleq (ai = x), q \triangleq (fj = a)\}\) and \(M = [p, q]\). Then \(a \not\approx_0 b, f \not\approx_0 g, \text{func}_0(a) = \text{func}_0(b) = \{([i],[x])\}, \text{func}_0(f) = \{([j],[a])\}\) and \(\text{func}_0(g) = \{([j],[b])\}\). It follows that \(a \sim_1 b, f \not\sim_1 g,\) and \(\text{func}_1(f) = \text{func}_1(g) = \{([j],[a])\}\). Thus, \(a \sim_2 b, f \not\sim_2 g\).