Proving array properties using data abstraction

Julien Braine, Laure Gonnord

Proving Array Properties using Data Abstraction*

Julien Braine
Univ Lyon, EnsL, UCBL, CNRS, Inria,
LIP, F-69342, LYON Cedex 07, France
julien.braine@ens-lyon.fr

Laure Gonnord
Univ Lyon, EnsL, UCBL, CNRS, Inria,
LIP, F-69342, LYON Cedex 07, France
laure.gonnord@ens-lyon.fr

Abstract

This paper presents a framework to abstract data structures within Horn clauses that allows abstractions to be easily expressed, compared, composed and implemented. These abstractions introduce new quantifiers that we eliminate with quantifier elimination techniques.

We study the case of arrays and our experimental evaluation show promising results on classical array programs.

CCS Concepts: • **Theory of computation** → *Verification by model checking*; *Abstraction*; *Type structures.*

Keywords: abstraction, data structures, Horn clauses, array properties

ACM Reference Format:
Julien Braine and Laure Gonnord. 2020. Proving Array Properties using Data Abstraction. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3427762.3430179

1 Introduction

Static analysis of programs containing non-bounded data-structures is a challenging problem as most interesting properties require quantifiers. Even stating that all elements of an array are equal to 0 requires it. A common way to reduce the complexity of such problems is abstraction using program transformation [13] or abstract interpretation [8].

In this paper, we suggest a new technique that we name *data abstraction* that takes advantage that we are abstracting data-structures. Inspired by previous work on arrays [4, 14], we combine quantifier instantiation with abstract interpretation. We obtain a transformation from Horn clauses to Horn clauses, a format with clear semantics to which programs with assertions can be reduced. The goal is to provide a framework in which abstractions on data structures can be easily expressed, compared, composed and implemented and decorrelate them from the back-end solving. Example 1 will be our motivating and running example. Proving this program is challenging as it mixes the difficulty of finding universally quantified invariants with modulo arithmetic.

In Section 2, we introduce Horn clauses, the transformation of our running example, and Galois connections, in Section 3, we formally give our data abstraction technique, in Section 4, we give an instance of such an abstraction on arrays and in Section 5 we give the experimental results of our tool and compare it with the Vaphor tool [14].

Example 1. Running example: the following program initializes an array to even values, then increases all values by one and checks that all values are odd. We wish to prove that the assertion is verified.

```
for(k=0; k<N; k++) //Program point For1
    a[k] = rand()*2;
for(k=0; k<N; k++) //Program point For2
    a[k] = a[k]+1;
for(k=0; k<N; k++) //Program point For3
    assert(a[k] % 2 == 1);
```

2 Preliminaries

2.1 Horn Clauses

A Horn clause is a logical formula over free variables and predicates. The only constraint is that Horn clauses are "increasing", that is, there can be at most one positive predicate in the clause. Horn clauses are written in the following form: $P_1(\overrightarrow{exprs}_1) \wedge \ldots \wedge P_n(\overrightarrow{exprs}_n) \wedge \phi \rightarrow P'(\overrightarrow{exprs'})$ where:

- $\overrightarrow{exprs}_1, \ldots, \overrightarrow{exprs}_n, \phi, \overrightarrow{exprs'}$ are expressions possibly containing free variables.
- $P_1, \ldots, P_n$ are the "negative" predicates
- P' is the positive predicate or some expression

The semantics of such a Horn clause is the following: $\forall vars, P_1(\overrightarrow{exprs}_1) \wedge \ldots \wedge P_n(\overrightarrow{exprs}_n) \wedge \phi \Rightarrow P'(\overrightarrow{exprs'})$ where $vars$ are the free variables of the expressions. We say a set of Horn clauses is satisfiable if and only if there exist values (sets) for each predicate that satisfy all the Horn clauses.

Programs with assertions can be transformed into Horn clauses using tools such as SeaHorn [2] or JayHorn [12], and Example 2 gives the transformation of Example 1 into Horn clauses by creating a predicate per program point and expressing the constraints on each program point using clauses.

*Partially funded by French ANR CODAS project (ANR-17-CE23-0004-01)

NSAD '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8187-1/20/11…$15.00
https://doi.org/10.1145/3427762.3430179

Example 2. Running example in Horn clauses where all predicates For_i have arity 3 (1 array and 2 integer parameters). Clause (4) in bold, will be used throughout the paper.

$$For1(a, N, 0) \quad (1)$$
$$For1(a, N, k) \wedge k < N \rightarrow For1(a[k \leftarrow r * 2], N, k + 1) \quad (2)$$
$$For1(a, N, k) \wedge k \geq N \rightarrow For2(a, N, 0) \quad (3)$$
$$\boldsymbol{For2(a, N, k) \wedge k < N \rightarrow For2(a[k \leftarrow a[k] + 1], N, k + 1)} \quad (4)$$
$$For2(a, N, k) \wedge k \geq N \rightarrow For3(a, N, 0) \quad (5)$$
$$For3(a, N, k) \wedge k < N \wedge a[k]\%2 \neq 1 \rightarrow false \quad (6)$$
$$For3(a, N, k) \wedge k < N \rightarrow For3(a, N, k + 1) \quad (7)$$

2.2 Galois Connection

A Galois connection [6] is a way of expressing a general abstraction. In our case, we abstract predicates (*i.e.* sets of possible values) on a concrete set $\mathscr{C}$ to an abstract set $\mathcal{A}$.

A Galois connection is defined by

- $\alpha : \mathcal{P}(\mathscr{C}) \rightarrow \mathcal{P}(\mathcal{A})$: the abstraction of a predicate
- $\gamma : \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathscr{C})$ gives the concrete values an abstracted predicate represents.

Two properties are required:1. $S \subseteq \gamma(\alpha(S))$ for soundness. 2. $\forall S^\#, \alpha(\gamma(S^\#)) \subseteq S^\#$ for minimal precision loss.

3 Data Abstraction

In this section, we present our main contribution: *data abstraction*. We abstract the Horn clauses, and then show how to remove the added quantifiers.

3.1 Data Abstraction in Horn Clauses

We propose the notion of *data abstractions* whose goal is to reduce the complexity of elements (such as arrays) by a set of simpler values (such as integers).

Definition 1. Data abstraction (σ, F_σ).

Let $\mathscr{C}$ and $\mathcal{A}$ be sets . A data abstraction is a couple (σ, F_σ) where σ is a function from $\mathscr{C}$ to $\mathcal{P}(\mathcal{A})$ and F_σ is a formula encoding its inclusion relation : $F_\sigma(a^\#, a) \equiv a^\# \in \sigma(a)$[1].

It defines a Galois connection from $\mathcal{P}(\mathscr{C})$ to $\mathcal{P}(\mathcal{A})$ by :

- $\alpha_\sigma(S \subseteq \mathscr{C}) = \bigcup_{a \in S} \sigma(a)$
- $\gamma_\sigma(S^\# \subseteq \mathcal{A}) = \{a \in \mathscr{C} | \sigma(a) \subseteq S^\#\}$

Example 3. $Cell_1$ abstraction of an array: abstracting an array by the set of its cells (*i.e.* couples of index and value).

$\sigma_{Cell_1}(a) = \{(i, a[i])\}$ $\quad$ $F_{\sigma_{Cell_1}}((i, v), a) \equiv v = a[i]$

Remark: abstracting a single array does not lose information, but abstracting a set of arrays (using α) does.

In Algorithm 1 we give the implementation of such abstractions in Horn clauses and Example 4 unrolls its execution. The key idea consists in replacing a predicate $P(expr)$, that is, $expr \in P$, by $expr \in \gamma(P^\#)$ for a new predicate $P^\#$ which is created to represent $\alpha(P)$. Soundness of Algorithm 1, that is, that if a Horn problem H has a counter exemple, so does the result of Algorithm 1 on H, follows from $P \subseteq \gamma(\alpha(P))$.

Algorithm 1. Abstracting in Horn clauses.

Input :

1. H: Horn problem
2. P: predicate to abstract.
3. $P^\#$: unused predicate.
4. F_σ.

Computation : *for each* clause C of H, *for each* $P(expr)$ in C, replace $P(expr)$ by $\forall a^\#, F_\sigma(a^\#, expr) \rightarrow P^\#(a^\#)$, where $a^\#$ is a new unused variable.

Example 4. Execution of Algorithm 1 with $Cell_1$.

Input :

1. Clauses of Example 2.
2. $For2$
3. $For2^\#$
4. σ_{Cell_1} applied to a.

Output : Consider Clause 4 from the example on page 2. After applying Algorithm 1 and naming the introduced quantified variables $(i^\#, v^\#)$ and $(i'^\#, v'^\#)$, we obtain:

$$\begin{aligned}(\forall i^\#, v^\# : v^\# = a[i^\#] \rightarrow For2^\#(i^\#, v^\#, N, k)) \wedge k < N \\ \rightarrow (\forall i'^\#, v'^\# : v'^\# = a[k \leftarrow a[k] + 1][i'^\#] \\ \rightarrow For2^\#(i'^\#, v'^\#, N, k + 1))\end{aligned}$$

In this section, we have a general scheme to abstract Horn problems with a *data abstraction*, however, new quantifiers ($\forall a^\#$) are introduced that solvers [7, 10] have trouble solving.

3.2 Removing the Introduced Quantifiers : Instantiation

Our abstraction has introduced new quantifiers in our Horn clauses. Here, we give an algorithm to remove those quantifiers using a technique called *quantifier instantiation* [4] which consists in replacing a universal quantifier, *i.e.* a possibly infinite conjunction, by a conjunction over some finite set S. In other words, an expression of the form $\forall q, expr(q)$ is transformed into an expression of the form $\bigwedge_{q \in S} expr(q)$.

Algorithm 2 removes the quantifiers in two steps :

1. Remove useless quantifiers: $expr \rightarrow (\forall q, exp')$ with $(q \notin exp)$ becomes $expr \rightarrow exp'$, with q a free variable.
2. Instantiate the other $\forall$ thanks to a heuristic *insts*.

Algorithm 2. Instantiation algorithm.

Input :

- C, a clause (after abstraction).
- *insts*, a function that to a quantifier of C and the abstracted value $expr$, returns an instantiation set S.

Computation :

1. Remove universal quantifiers in the head of the clause.
2. For each instance of $\forall a^\#, F_\sigma(a^\#, e) \rightarrow P^\#(a^\#)$, replace it by $\bigwedge_{a^\# \in insts(a^\#, e)} F_\sigma(a^\#, e) \rightarrow P^\#(a^\#)$

[1] Classically, we denote abstracts elements ($\in \mathcal{A}$) with sharps (#).

Example 5. Example of instantiation from Example 4

$$(\forall i^\#, v^\# : v^\# = a[i^\#] \rightarrow For2^\#(i^\#, v^\#, N, k) \wedge k < N) \rightarrow (\forall \boldsymbol{i'}^\#, \boldsymbol{v'}^\# : v'^\# = a[k \leftarrow a[k]+1][i'^\#] \rightarrow For2^\#(i'^\#, v'^\#, N, k+1))$$

After the first step (*i.e.* removing $\forall \boldsymbol{i'}^\#, \boldsymbol{v'}^\#$), we obtain:

$$(\forall i^\#, v^\# : v^\# = a[i^\#] \rightarrow For2^\#(i^\#, v^\#, N, k)) \wedge k < N \rightarrow ((v'^\# = a[k \leftarrow a[k]+1][i'^\#] \rightarrow For2^\#(i'^\#, v'^\#, N, k+1)))$$

Using $insts((i^\#, v^\#), a) = \{(k, a[k]), (i'^\#, a[i'^\#])\}$ (this choice is explained in Section 4.2) and slight simplifications, we get:

$$(\boldsymbol{For2^\#(k, a[k], N, k)} \wedge \boldsymbol{For2^\#(i'^\#, a[i'^\#], N, k)} \wedge k < N) \rightarrow For2^\#(i'^\#, a[k \leftarrow a[k]+1][i'^\#], N, k+1)$$

which is clause without quantifiers equivalent to the clause before instantiation due to our good choice of *insts*. However, for any choice of *insts*, soundness of Algorithm 2 is ensured as Step 2 only happends on premises of the clause (*i.e.* negative predicates) and $\forall q, expr(q) \rightarrow \bigwedge_{q \in S} expr(q)$.

In this Section, we have given a data abstraction technique that from a abstraction formula F_σ and an instantiation heuristic *insts* transforms predicates on variables of the concrete domain into predicates over the abstract domain. The abstraction is always sound and its precision depends on *insts*. We show in Section 5 using array abstraction that the precision loss does not impact our experiments.

4 Abstracting Arrays : Cell Abstractions

To illustrate our *data abstraction* technique, we show how to handle the cell abstractions of Monniaux and Gonnord [14].

4.1 Cell Abstractions

Cell abstractions consist in viewing arrays by (a finite number of) their cells. However, instead of abstracting arrays by specific cells such as the first, the last or the second cell, ..., we use parametric cells (*i.e.* cells with a non fixed index). $Cell_1$ of Example 3 corresponds to one parametric cell.

Definition 2. Cell abstractions $Cell_n$.

$\sigma_{Cell_n}(a) = \{(i_1, a[i_1], \ldots, i_n, a[i_n])\}$ and
$F_{\sigma_{Cell_n}}((i_1, v_1, \ldots, i_n, v_n), a) \equiv v_1 = a[i_1] \wedge \ldots \wedge v_n = a[i_n]$.

Cell abstractions are of great interest because of their expressivity: many interesting concrete properties can be expressed as abstract properties. Furthermore, our *data abstraction* framework allows us to formalize other existing array abstractions using compositions from cell abstractions.

Example 6 gives examples of expressible properties by cell abstractions and Example 7 shows how to construct some common abstractions from cell abstraction.

Example 6. Properties expressed with cell abstractions.

For each concrete property in the table, we give a cell abstraction that allows to capture it with an abstract property.

Concrete	Abs	Abstract property
$a[0] = 0$	$Cell_1$	$i_1 = 0 \Rightarrow v_1 = 0$
$a[n] = 0$	$Cell_1$	$i_1 = n \Rightarrow v_1 = 0$
$a[0] = a[n]$	$Cell_2$	$(i_1 = 0 \wedge i_2 = n) \Rightarrow v_1 = v_2$
$\forall i, a[i] = 0$	$Cell_1$	$v_1 = 0$
$\forall i, a[i] = i^2$	$Cell_1$	$v_1 = i_1^2$
$\forall i, a[n] \geq a[i]$	$Cell_2$	$i_2 = n \Rightarrow v_2 \geq v_1$

Example 7. Array abstractions from cell abstractions.

Array smashing. : $\sigma_{smash}(a) = \{a[i]\}$. This abstraction keeps the set of values reached but loses all information linking indices and values. It is the composition of $Cell_1$ and "forgetting i_1", that is, the data abstraction $\sigma_{forget}(i_1) = \top$

Array slicing. There are several variations, and for readability we present the one that corresponds to "smashing each slice" **??** and pick the slices $]-\infty, i[, [i, i],]i, \infty[$

$$\sigma_{slice}(a) = \{(a[j_1], a[i], a[j_3]), j_1 < i \wedge j_3 > i\}$$

It is the composition of $Cell_3$ and knowing if i_1, i_2, i_3 are in the slice: $\sigma_{rm}(i_1, i_2, i_3) = \{i_1 < i \wedge i_2 = i \wedge i_3 > i\}$. This creates a Boolean which, after simplification, can be removed.

4.2 Instantiating Cell Abstractions

The *data abstraction* framework, requires an instantiation heuristic *insts*. Inspired by [5, 14], we create the heuristics $insts_{Cell_n}$ of Definition 3. The idea is that relevant indices for clause instantiation are those that are read and this is how the instantiation set in Example 5 was constructed.

Definition 3. Instantiation heuristic for $Cell_n$.

Let C be a clause after the step 1 of Algorithm 2.

$insts_{Cell_n}(q, expr) =$

$\{(e, expr[e]) | \exists e', e'[e] \in C\}^n$ if this set is non empty

$\{(\_, expr[\_])\}^n$ with _ being any value otherwise

4.3 Entirely Removing Arrays: Ackermanisation[1]

Motivation. Although predicates do not have arguments of array types after abstraction, clauses still use the arrays to express the transition relation. Removing those arrays is a theoretically solved issue as we do not have any quantifiers in our clauses [5]. However, we experimentally noticed that doing so in our preprocessing improves the solver's results.

Technique. The axiom $a[i \leftarrow v][j] \equiv ite(i = j, v, a[j])$ is applied to remove array writes (*ite* denotes if-then-else). Then, for each index *expr* at which an array a is read, we create a fresh variable v_{expr} and replace $a[expr]$ by v_{expr} in the clause, then, for each pair of indices $expr_1, expr_2$ added, we generate the constraint $expr_1 = expr_2 \rightarrow v_{expr_1} = v_{expr_2}$.

Example 8. Ackermanisation of arrays.

Removing array writes on the **running clause** from Example 5 yields :

$$(For2^{\#}(k, a[k], N, k) \wedge For2^{\#}(i'^{\#}, a[i'^{\#}], N, k) \wedge k < N) \\ \rightarrow For2^{\#}(i'^{\#}, ite(k = i'^{\#}, a[k] + 1, a[i'^{\#}]), N, k + 1)$$

and removing array reads with $a_{i'^{\#}}, a_k$ new variables:

$$(For2^{\#}(k, a_k, N, k) \wedge For2^{\#}(i'^{\#}, a_{i'^{\#}}, N, k) \wedge k < N \\ \wedge (k = i'^{\#} \rightarrow a_k = a_{i'^{\#}})) \\ \rightarrow For2^{\#}(i'^{\#}, ite(k = i'^{\#}, a_k + 1, a_{i'^{\#}}), N, k + 1)$$

5 Experiments

Benchmarks. We used the mini-java benchmarks [14]. We modified them to add loop invariants as optional hints, increased readability by reducing the number of intermediate variables, and assertions are now checked through a loop instead of checking a random index (*i.e.* instead of checking that $a[k]$ verifies the property for a random k, we iterate with a loop $0 \leq k < N$ and check that $a[k]$ verifies the property). We divided our experiments in several categories:

1. Our running example with and without hints
2. The mini-java benchmarks [14] without hints
3. The mini-java benchmarks [14] with hints
4. The buggy (the assertion is wrong) mini-java benchmarks [14] to check for soundness of our tool.

Toolchain. We used the following toolchain :

1. The mini-java to Horn converter used [14] to convert programs into Horn clauses with an added option to handle hints. It also contains options to handle the syntactical output of the clauses without changing their semantics (*i.e.* such as naming conventions).
2. One of the following abstraction method from Horn clauses to Horn clauses:
   - No abstraction: we keep the original file.
   - The Vaphor abstraction [14] (*i.e.* excluding the part that converts mini-java to Horn clauses) tool.
   - Our data abstraction tool (removing arrays in predicates using $Cell_1$ abstraction).
   - Our data abstraction tool with ackermanisation.
3. The Z3[2] Horn solver with a 30s timeout.

The code for all tools and benchmarks is available on github[3]. The version used of each tool is tagged with "NSAD20".

Results. Our experimental results are summarized in Table 1. It contains, for each different toolchain and each category of example, the number of examples for which:

- The solver computed the desired result (👍) (*i.e.* sat if the example is not buggy, unsat otherwise) with default syntax options
- The solver returned an undesired result (👎) (*i.e.* unsat when the example was not buggy and sat otherwise) with default syntax options
- The solver returned unknown (*i.e.* the solver abandoned) or timed-out (⌛), that is took more than 30s seconds with default syntax options
- The solver computed the desired result in at least one of the syntax options (≥ 1)

We have no case of problems in the toolchain and results are identical with a timeout of 120 seconds.

Analysis. The experimental results show that

1. The tool seems sound (without bugs) : no buggy example becomes not buggy.
2. $Cell_1$ abstraction with our instantiation heuristic is expressive enough that the solver never returns that there is a bug when there was not one initially. Even better, we know that the invariant is expressible in the abstract domain as the column ≥ 1 for $Cell_1$ ackermanised on hinted examples is equal to #exp.
3. Data abstraction behaves better than Vaphor.
4. The Z3 solver is not yet good enough on integers to find the necessary invariants without hints.
5. The Z3 solver is dependant on syntax as the column ≥ 1 is not equal to the column 👍.
6. Increasing the timeout does not seem to help the solver converge as results at timeout=30s are equal to results at timeout=120s.
7. Completely removing arrays helps.
8. Non-hinted or non-abstracted versions timeout.

Discussion. Points 1 and 2 show that the tool achieves its purpose, that is, reducing invariants on arrays requiring quantifiers to invariants without quantifiers on integers by using the $Cell_1$ abstraction without losing precision (*i.e.* that the invariants are expressible in the abstract domain). Future work should use more array programs benchmarks [3] and possibly use another front-end to handle them [2, 12].

Point 3 can be explained by several reasons. First, [14] does not give an explicit technique on how to abstract multiple arrays and the effective transformation in the tool seems less expressive than applying $Cell_1$ abstraction to each array. Furthermore, Horn solvers based on Sat Modulo Theory (SMT) are very sensible to the SMT proofs. Our data-abstraction tool implements several simple expression simplifying techniques, which may lead to better convergence of the solver by reducing the noise in SMT proofs.

Points 4 to 7 show that the Z3 tool is not yet mature enough to handle the Horn clauses we have after abstraction. One possible reason may be that the Z3 Horn solver heuristics were optimized for Horn clauses directly constructed from programs and not for the type of Horn clauses we generate after abstraction. A possible solution to improve predictability and reduce the impact of syntax could be to solve the

[2] version 4.8.8 - 64 bit

[3] https://github.com/vaphor

Table 1. Experimental results

	#exp	Noabs				VapHor				$Cell_1$				$Cell_1$ ackermanised			
		👍	👎	◌	≥ 1	👍	👎	◌	≥ 1	👍	👎	◌	≥ 1	👍	👎	◌	≥ 1
Running	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0
RunningHinted	1	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	1
NotHinted	11	0	0	11	0	1	0	10	1	0	0	11	0	0	0	11	0
Hinted	11	0	0	11	0	5	0	6	5	6	0	5	10	8	0	3	11
Buggy	4	4	0	0	4	4	0	0	4	4	0	0	4	4	0	0	4

Horn clauses using abstract interpretation. However, this would require relational invariants which may be expensive.

Point 8 shows that the proposed technique can not be used to automatically generate invariants on Horn clauses containing arrays, however, it succeeds to reduce the problem of finding quantified invariants on arrays to solving integer Horn clauses. It seems the latter is still too hard and this may change in the near future, possibly by using another solver.

6 Related Work

Numerous abstractions for arrays have been proposed in the literature, among which array slicing [8]. In Example 7 we showed how they are expressible in our framework. Similarly to [13] we think that disconnecting the array abstraction from other abstractions and from solving enables to better use back-end solvers. Like [14] we use Horn Clauses to encode our program under verification, but we go a step further in the use of Horn Clauses as an intermediate representation useful to chain abstractions. Furthermore, our formalization is cleaner when multiple arrays are involved.

Our instantiation method had been inspired from previous work on solving quantified formula [4, 5, 9]. The paper [5] does not consider Horn clauses, that is, expressions with unknown predicates but only expressions with quantifiers. The paper [4] does a very similar approach to ours, however, they do not suggest the notion of *data abstractions* in a goal to analyze them and they use trigger based instantiation. Both instantiation methods of [4, 5] lead to bigger instantiation sets than the one we suggest, and yet, we proved through benchmarks that our instantiation set was sufficient for the types of programs used [4]. Finally, the technique used in [9] creates instantiation sets not as a prepossessing, but while the solver is analyzing. This technique seems possibly best for a universal way of handling quantifiers, however, it is likely that the technique suffers of the same unpredictability that Horn solvers have. We believe that we can tailor the instantiation set to the abstraction and analyze its precision.

Finally, other recent techniques focus on more powerful invariants through proofs by induction[11]. However, both techniques are complementary: their technique is less specialized and thus has trouble where our approach may easily succeed but enables other invariants: data abstraction may allow to abstract within their induction proofs.

7 Conclusion

In this paper we gave an abstraction framework for data using Horn clauses. Using this framework, we successfully described the cell abstractions[14] in a simple manner and some other common array abstraction using composition. The method has been implemented and shows interesting preliminary experimental results.

Experiments show that the chosen solver Z3 seems to be very unpredictable for the kind of Horn clauses we generate and further investigation needs to be done. Another direction is to experiment with other Horn clauses solving techniques.

Moreover, we plan to improve our implementation by parametrizing it with the desired data-abstraction, and on the theoritical side, work on isolating a fragment on which the $Cell_n$ heuristic is complete.

References

[1] W. Ackermann. 1957. Solvable cases of the decision problem. *Journal of Symbolic Logic* (1957).

[2] Arie Arie Gurfinkel, Themesghen Kahsai, Anvesh Komuravelli, and Jorge Navas. 2015. The SeaHorn Verification Framework. In *CAV*.

[3] Dirk Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *TACAS*.

[4] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS*.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *VMCAI*.

[6] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

[8] Denis Gopan, Thomas Reps, and Mooly Sagiv. 2005. A Framework for Numeric Analysis of Array Operations. In *PLDI*.

[9] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2018. Quantifiers on Demand. In *ATVA*.

[10] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *FMCAD*.

[11] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2020. Putting the Squeeze on Array Programs: Loop Verification via Inductive Rank Reduction. In *VMCAI*.

[12] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. 2019. *JayHorn: A Java Model Checker: (Competition Contribution)*.

[13] David Monniaux and Francesco Alberti. 2015. A simple abstraction of arrays and maps by program translation. In *SAS*.

[14] David Monniaux and Laure Gonnord. 2016. Cell morphing: from array programs to array-free Horn clauses. In *SAS*.