Data Abstraction: A General Framework to Handle Program Verification of Data Structures

Julien Braine, Laure Gonnord, David Monniaux

To cite this version:

Julien Braine, Laure Gonnord, David Monniaux. Data Abstraction: A General Framework to Handle Program Verification of Data Structures. [Research Report] RR-9408, Inria Grenoble Rhône-Alpes; VERIMAG UMR 5104, Université Grenoble Alpes, France; LIP - Laboratoire de l'Informatique du Parallélisme; Université Lyon 1 - Claude Bernard; ENS Lyon. 2021, pp.1-29. hal-03214475v2

HAL Id: hal-03214475
https://inria.hal.science/hal-03214475v2

Submitted on 18 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Data Abstraction: A General Framework to Handle Program Verification of Data Structures

Julien BRAINE, Laure GONNORD, David MONNIAUX

RESEARCH REPORT
N° 9408
May 2021
Project-Team CASH

ISSN 0249-6399 ISRN INRIA/RR--9408--FR+ENG

Data Abstraction: A General Framework to Handle Program Verification of Data Structures

Julien BRAINE[*], Laure GONNORD[*][†], David MONNIAUX[‡]

Project-Team CASH

Research Report n° 9408 — May 2021 — 30 pages

Abstract:
Proving properties on programs accessing data structures such as arrays often requires universally quantified invariants, e.g., “all elements below index i are nonzero”. In this research report, we propose a general data abstraction scheme operating on Horn formulas, into which we recast previously published abstractions. We show our instantiation scheme is relatively complete: the generated purely scalar Horn clauses have a solution (inductive invariants) if and only if the original problem has one expressible by the abstraction.

Key-words: Program Analysis, Data Structures, Abstractions, Horn Clauses

[*] University of Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA),F-69000 Lyon, France
[†] This work was partially supported by the ANR CODAS Project
[‡] CNRS, Verimag, Grenoble;, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l’Europe Montbonnot
38334 Saint Ismier Cedex

Data Abstraction: A General Framework to Handle Program Verification of Data Structures

Résumé : Pour prouver des propriétés de programmes qui manipulent des structures de données comme des tableaux , nous avons besoin de savoir résoudre des formules comportant des quantificateurs universels: par exemple, “tous les éléments d’index inférieur à i sont différents de 0”. Dans ce rapport de recherche, nous proposons une technique générale d’abstraction opérant sur des Clauses de Horn, qui permet de reformuler un certain nombre d’abstractions déjà publiées. Nous montrons que notre schéma d’abstraction est relativement complet: le système de clauses purement scalaires a une solution (sous forme d’invariants inductifs) si et seulement si le problème initial a une solution exprimable dans la logique de l’abstraction.

Mots-clés : Analyse de Programmes, Structures de données, Abstractions, Clauses de Horn

1 Introduction

Static analysis of programs containing unbounded data structures is challenging, as most interesting properties require quantifiers. Even stating that all elements of an array are equal to 0 requires them ($\forall i \; a[i] = 0$), let alone more complex cases such as Example 1. In general, the satisfiability of arbitrary quantified properties on unbounded data structures is undecidable [BMS06], thus there is no algorithm for checking that such properties are inductive, nor inferring them.

The first step is to select an abstract domain to search for invariants, e.g., properties of the form $\forall i, P(a[i])$ for some predicate P and array a. In this paper, we describe a transformation from and to Horn clauses such that these properties may be expressed in the transformed Horn clauses without using quantifiers. For the array data structure, our sheme can optionally completely remove arrays from the transformed Horn clauses. This transformation is *sound* and *relatively complete*: the resulting problem, to be fed to a solver for Horn clauses, such as Eldarica or Z3, has a solution if and only if the original one has one within the chosen abstract domain. In short, we reduce problems involving quantifiers and arrays to problems that do not, with no loss of precision with respect to the abstract domain.

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);
```

Contributions (i) an abstraction framework for Horn clauses using unbounded data structures, that we call *data abstraction*; (ii) the analysis of a property we call *relative completeness* in this framework (iii) and the use of that framework to handle programs with arrays and its experimental evaluation.

Contents Section 2 introduces Horn Clauses' concepts and notations. Sections 3 and 4 expose our data abstraction framework and its *relative completeness analysis*. Section 5 considers a data abstraction for arrays. Finally, Section 6 proposes a full algorithm to analyze programs with arrays and its experimental evaluation. The appendix contains the proofs of theorems.

2 Preliminaries: Horn clauses

2.1 Solving Programs with assertions using Horn Clauses

Programs with assertions can be transformed into Horn clauses using tools such as SeaHorn [GKKN15] or JayHorn [KRS19]. The syntax of Horn clauses is recalled in Def. 2. A basic transformation consists in associating a predicate to each point of the control flow graph ; control edges are inductive relations (clauses), and assertions A are clauses $\neg A \rightarrow \mathit{false}$.[1]

Example 2 (Example 1 transformed into Horn clauses)**.** *All predicates For_i have arity 3 (1 array and 2 integer parameters) and Clause (4) in bold, will be used throughout the paper.*

$$For1(a, N, 0) \tag{1}$$

$$For1(a, N, k) \wedge k < N \rightarrow For1(a[k \leftarrow r * 2], N, k + 1) \tag{2}$$

$$For1(a, N, k) \wedge k \geq N \rightarrow For2(a, N, 0) \tag{3}$$

$$\boldsymbol{For2(a, N, k) \wedge k < N \rightarrow For2(a[k \leftarrow a[k] + 1], N, k + 1)} \tag{4}$$

$$For2(a, N, k) \wedge k \geq N \rightarrow For3(a, N, 0) \tag{5}$$

$$For3(a, N, k) \wedge k < N \wedge a[k]\%2 \neq 1 \rightarrow false \tag{6}$$

$$For3(a, N, k) \wedge k < N \rightarrow For3(a, N, k + 1) \tag{7}$$

Variables are local: the a of Clause 1 is not formally related to the a of Clause 4.

[1] Tools such as SeaHorn [GKKN15] handle richer languages such as LLVM bytecode, but the generated clauses are more complex and further removed from their initial semantics. Such clauses fall within our data abstraction framework, but not within the scope of the experimental evaluation of this paper.

A solution to such a system of Horn clauses is a set of inductive invariants suitable for proving the desired properties. Horn clause systems can be solved by tools such as Z3, Eldarica, ... A "Sat" answer means that the inductive invariants exist and thus the program is correct.[2] "Unsat" means that a counterexample was found, leading to an assertion violation. "Unknown" means the tool fails to converge on a possible invariant. Finally, the tool may also timeout.

2.2 Horn Clauses and Horn problems

In our setting, a Horn clause is a boolean expression over free variables and predicates with at most one positive predicate.

Definition 1 (Expressions $expr$, positive and negative predicates P, models $\mathcal{M}$ and semantics $expr(vars)$, $[\![expr]\!]_{\mathcal{M}}$)**.** *In this paper we do not constrain the theory on which expressions are written, the only constraint being that the backend solver must handle it. An expression may contain quantifiers, free variables and predicates. A predicate is a typed name which will be a set when interpreted in a model.*
Expression evaluation *: There are two evaluation contexts for expressions:*

1. ***Models****, written $\mathcal{M}$: map each predicate to a set of the corresponding domain*
2. ***Environments****, written vars, that to each free variable of the expression associates a value of the corresponding type's domain*

$[\![expr]\!]_{\mathcal{M}}$ denotes the evaluation of an expression expr in a model $\mathcal{M}$, $expr(vars)$ denotes its the evaluation in the environment vars, $[\![expr(vars)]\!]_{\mathcal{M}}$ denotes joint evaluation. Furthermore, if an expression value is independent of the model or environment, we may use it directly as its evaluation.
Positive and negative predicates *: A predicate instance in a boolean expression expr is deemed* negative *(resp.* positive*) if and only if there is a negation (resp. no negation) in front of it when expr is in negative normal form.*

Definition 2 (Horn Clauses, extended, normalized, satisfiability)**.** *A* Horn clause *is simply any expression without quantifiers (but with free variables) containing at most one positive predicate.*
Extended Horn clause *: a Horn clause which may use quantifiers.*
Normalized Horn clause *: Normalized Horn clauses are in the form $P_1(e_1) \land \ldots \land P_n(e_n) \land \phi \to P'(e')$ where:*

- *$e_1, \ldots, e_n, \phi, e'$ are expressions without predicates but with free variables.*
- *$P_1, \ldots, P_n$ are the "negative" predicates*
- *P' is the positive predicate or some expression*

Satisfiability*: A set of Horn clauses $\mathfrak{C}$ is said to be* satisfiable *if and only if $\exists \mathcal{M}, \forall C \in \mathfrak{C}, [\![\forall vars, C(vars)]\!]_{\mathcal{M}}$.*
In this paper, we will denote clauses in capital letters: C, sets of clauses in Fraktur: $\mathfrak{C}$, and models in calligraphic: $\mathcal{M}$.

Definition 3 (Notations $ite, f[a \leftarrow b]$)**.** *For a boolean expression b and expressions e_1, e_2, we define the expression "if-then-else", written $ite(b, e_1, e_2)$, evaluating to e_1 when b and to e_2 when $\neg b$.*

*For a function f (*i.e.* an environment) or an array, we define $f[a \leftarrow b]$ as $f[a \leftarrow b](x) = ite(x = a, b, f(x))$*

Example 3 (Satisfiability of Example 2)**.** *The following model satisfies Example 2 with $(a, N, k) \in \mathbb{N}^3$.*

1. $\mathcal{M}(For1) = \{(a, N, k) | k < N \land \forall i < k, a[i]\%2 = 0\}$
2. $\mathcal{M}(For2) = \{(a, N, k) | k < N \land \forall i < k, a[i]\%2 = 1 \land \forall k \le i < N, a[i]\%2 = 0\}$
3. $\mathcal{M}(For3) = \{(a, N, k) | k < N \land \forall i < N, a[i]\%2 = 1\}$

Horn clauses constrain models in two ways: those with a positive predicate force the model to be a post-fixpoint of an induction relation; those without are assertions that force the model to not contain elements violating the assertion.

Horn clauses are the syntactic objects we use to write *Horn problems.* Theorem 2 formalizes the link between Horn problems and Horn clauses.

[2]Z3 is both a SMT solver and a tool for solving Horn clauses. In SMT, a "Sat" answer often means a counterexample trace invalidating a safety property. In contrast, in Horn solving, "Sat" means a safety property is proved.

Definition 4 (Horn Problem H, defines f_H,$\mathcal{U}_H$)**.** *A Horn problem H is a pair $(f_H,\mathcal{U}_H)$ where (i) f_H is a monotone function over models with order $\mathcal{M}_1 \leq \mathcal{M}_2 \equiv \forall P, \mathcal{M}_1(P) \subseteq \mathcal{M}_2(P)$. (ii) $\mathcal{U}_H$ is a model. It is said to be* satisfiable *if and only if* $\mathrm{lfp}\, f_H \leq \mathcal{U}_H$ *(where* lfp *is the least fixpoint operator).*

Theorem 1 (Horn problems as a condition on models, defines $H(\mathcal{M})$)**.** *A Horn problem H is satisfiable if and only if $\exists \mathcal{M}, f_H(\mathcal{M}) \leq \mathcal{M} \wedge \mathcal{M} \leq \mathcal{U}_H$, also written $\exists \mathcal{M}, H(\mathcal{M})$ with $H(\mathcal{M}) =_{def} f_H(\mathcal{M}) \leq \mathcal{M} \wedge \mathcal{M} \leq \mathcal{U}_H$.*

Theorem 2 (Horn clauses as Horn problems, defines $H_{\mathfrak{C}}$)**.** *Let $\mathfrak{C}$ be a set of Horn clauses. There exists a Horn problem $H_{\mathfrak{C}}$, such that for any model $\mathcal{M}$, $H_{\mathfrak{C}}(\mathcal{M}) = \forall C \in \mathfrak{C}, [\![\forall vars, C(vars)]\!]_{\mathcal{M}}$. Thus, $satisfiable(\mathfrak{C}) \equiv satisfiable(H_{\mathfrak{C}})$.*

2.3 Horn problem induced by an abstraction

Static analysis by abstract interpretation amounts to searching for invariants (*i.e.*, models of Horn clauses in our setting) within a subset of all possible invariants called an *abstract domain*; elements of that subset are said to be *expressible by the abstraction.* We formalize abstraction as a Galois connection [CC77], that is, a pair (α, γ) where α denotes abstraction (*i.e.* simplification) and γ denotes the semantics (*i.e.* what the abstraction corresponds to) of abstract elements.

Definition 5 (Models expressible by the abstraction $\mathcal{G}$)**.** *We say that a model $\mathcal{M}$ is expressible by an abstraction $\mathcal{G}$ if and only if $\mathcal{M} = \gamma_{\mathcal{G}} \circ \alpha_{\mathcal{G}}(\mathcal{M})$ or equivalently $\exists \mathcal{M}^\#, \mathcal{M} = \gamma_{\mathcal{G}}(\mathcal{M}^\#)$.*

Example 4 (Models expressible by an abstraction)**.** *Consider the model $\mathcal{M}$ from Example 3. This model is expressible by the abstraction $\mathcal{G}$ such that $\forall P \in \{For1, For2, For3\}$:*

1. $\alpha_{\mathcal{G}}(\mathcal{M})(P) = \{(\boldsymbol{i}, a[i], N, k) | (a, N, k) \in \mathcal{M}(P)\}$
2. $\gamma_{\mathcal{G}}(\mathcal{M}^\#)(P) = \{(a, N, k) | \forall i, (\boldsymbol{i}, a[i], N, k) \in \mathcal{M}^\#(P)\}$

but not by the abstraction $\mathcal{G}'$ such that $\forall P \in \{For1, For2, For3\}$:

1. $\alpha_{\mathcal{G}'}(\mathcal{M})(P) = \{(a[i], N, k) | (a, N, k) \in \mathcal{M}(P)\}$
2. $\gamma_{\mathcal{G}'}(\mathcal{M}^\#)(P) = \{(a, N, k) | \forall i, (a[i], N, k) \in \mathcal{M}^\#(P)\}$

The idea is that the abstraction $\mathcal{G}$ keeps the relationships between indices and values $(i, a[i])$, which is all that is needed for our invariants, whereas $\mathcal{G}'$ forgets the indices and only keeps information about the values, which is insufficient. Section 5 details what each abstraction expresses on arrays.

Definition 6 (Abstraction of a Horn problem $abs(\mathcal{G}, H)$)**.** *The abstraction of Horn problem H by a Galois connection $\mathcal{G}$ noted $abs(\mathcal{G}, H)$, is defined by (i) $f_{abs(\mathcal{G},H)} = \alpha_{\mathcal{G}} \circ f_H \circ \gamma_{\mathcal{G}}$ (ii) $\mathcal{U}_{abs(\mathcal{G},H)} = \alpha_{\mathcal{G}}(\mathcal{U}_H)$.*

Theorem 3 (Definition $abs(\mathcal{G}, H)$ is correct)**.** *For all $\mathcal{M}^\#$, the following statements are equivalent (with the notation $H(\mathcal{M})$, where H is a Horn problem and $\mathcal{M}$ a possible model, from Theorem 1): (i) $abs(\mathcal{G}, H)(\mathcal{M}^\#)$ (ii) $H(\gamma_{\mathcal{G}}(\mathcal{M}^\#))$ (iii) $abs(\mathcal{G}, H)(\alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#))$.*

Remark 1. *From this theorem, it follows that:*

1. *$abs(\mathcal{G}, H)$ corresponds to the desired abstraction as: H is satisfiable by a model expressible by the abstraction ($\gamma_{\mathcal{G}}(\mathcal{M}^\#)$) iff $abs(\mathcal{G}, H)$ is satisfiable.*
2. *$abs(\mathcal{G}, H)(\mathcal{M}^\#)$ is constructible from H and $\gamma_{\mathcal{G}}$. This will be used in Th. 5.*

2.4 Horn clauses transformations

A transformation is *sound* if it never transforms unsatisfiable Horn clauses (incorrect programs) into satisfiable ones (correct programs), *complete* if it never transforms satisfiable Horn clauses into unsatisfiable ones. A transformation is *complete relative to* an abstraction if it never transforms Horn clauses satisfiable in the abstract domain into unsatisfiable ones. **Together, soundness and relative completeness state that the transformation implements exactly the abstraction.**

Definition 7 (Soundness, Completeness, Relative completeness)**.** *A transformation alg from Horn clauses to Horn clauses is said to be:*

- sound *if and only if* $\forall \mathfrak{C}, H_{alg(\mathfrak{C})}$ *satisfiable* $\Rightarrow H_{\mathfrak{C}}$ *satisfiable.*
- complete *if and only if* $\forall \mathfrak{C}, H_{\mathfrak{C}}$ *satisfiable* $\Rightarrow H_{alg(\mathfrak{C})}$ *satisfiable .*

- complete relative to $\mathcal{G}$ *iff* $\forall \mathfrak{C}, abs(\mathcal{G}, H_{\mathfrak{C}})$ *satisfiable* $\Rightarrow H_{alg(\mathfrak{C})}$ *satisfiable* .

Theorem 4 (Soundness with relative completeness is *abs*)**.** *If a transformation alg is sound and complete relative to $\mathcal{G}$, then $\forall \mathfrak{C}, H_{alg(\mathfrak{C})}$ satisfiable $\equiv abs(\mathcal{G}, H_{\mathfrak{C}})$ satisfiable.*

Relative completeness is rarely ensured in abstract interpretation; examples include some forms of policy iteration, which compute the least inductive invariant in the abstract domain. Widening operators, very widely used, break relative completeness. Previous works on arrays do not analyze relative completeness.

In this paper, we present a framework to define abstractions on data such that the relative completeness of transformations is analyzed, and proved when possible. To do so, our abstraction scheme is divided into two algorithms: (i) one that computes $abs(\mathcal{G}, H_{\mathfrak{C}})$ and thus is sound and complete relative to $\mathcal{G}$ but uses extended Horn clauses (*i.e.* Horn clauses with additional quantifiers); (ii) another which transforms these extended Horn clauses back into Horn clauses and ensures soundness and strives to ensure completeness—and is shown to ensure it in the setting of our tool. When the second algorithm ensures completeness, the framework provides an abstraction algorithm from Horn clauses to Horn clauses which is both sound and complete relative to the abstraction.

3 Data Abstraction: Abstracting Horn Clauses

3.1 Implementing Horn Clauses Abstraction

The abstraction on the syntax of Horn clauses is done by choosing a predicate to abstract. This approach can then be successively used on several predicates.

In this paper, we consider a subset of abstractions we call *data abstractions*. The semantics of a predicate is a set of unknown values and an abstraction is simply a relation in the form of a Galois connection between that set of unknown values to a “simpler” set of unknown values. The key particularity of data abstractions is that the abstraction of this set of unknown values is defined by the abstraction of its individual elements, the “data”. This allows us to take advantage of the syntax of Horn clauses because the “data” is simply the expressions passed as parameters to a predicate. Furthermore, as our goal is to syntactically modify the Horn clauses, we require that the abstraction can be encoded by an explicit formula that will be used syntactically during the transformation.

Definition 8 (Data abstraction σ, defines $F_\sigma, \alpha_\sigma, \gamma_\sigma, \mathcal{G}_\sigma^P$)**.** *Let $\mathscr{C}$ and $\mathcal{A}$ be sets . A data abstraction σ is a function from $\mathscr{C}$ to $\mathcal{P}(\mathcal{A})$ and we write F_σ the formula encoding its inclusion relation : $F_\sigma(a^\#, a) \equiv a^\# \in \sigma(a)$*[3]*.*

It defines a Galois connection from $\mathcal{P}(\mathscr{C})$ to $\mathcal{P}(\mathcal{A})$ as follows: for $S \subseteq \mathscr{C}$, $S^\# \subseteq \mathcal{A}$, $\alpha_\sigma(S) = \bigcup_{a \in S} \sigma(a)$ and $\gamma_\sigma(S^\#) = \{a \in \mathscr{C} | \sigma(a) \subseteq S^\#\}$.

This Galois connection can be applied to a predicate P, thus yielding the Galois connection $\mathcal{G}_\sigma^P$ defined by $\alpha_{\mathcal{G}_\sigma^P}(\mathcal{M})(P') = ite(P' = P, \alpha_\sigma(\mathcal{M}(P)), \mathcal{M}(P'))$ and $\gamma_{\mathcal{G}_\sigma^P}(\mathcal{M}^\#)(P') = ite(P' = P, \gamma_\sigma(\mathcal{M}^\#(P)), \mathcal{M}^\#(P'))$.

Example 5 ($Cell_1$ abstraction of an array)**.** *$Cell_1$ abstracts an array by the set of its cells (*i.e. *pairs of index and value).*

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

Remark 2. *This data abstraction σ_{Cell_1} essentially abstracts a function a from an arbitrary index type I to an arbitrary value type by its graph $\{(i, a(i)) \mid i \in I\}$. As such, it does not lose information on individual arrays: two functions are identical if and only if they have the same graph (functional extensionality).*

However, the associated $\alpha_{\sigma_{Cell_1}}$ is not injective and loses information. This is essentially because when one takes the superposition of the graphs of two or more functions, there is no way to recover which part of the graph corresponds to which function. Consider for example $a_0 : 0 \mapsto 0, 1 \mapsto 1$ and $a_1 : 0 \mapsto 1, 1 \mapsto 0$. Then, $\alpha_{\sigma_{Cell_1}}(\{a_0, a_1\}) = \{0, 1\} \times \{0, 1\}$; and thus $\alpha_{\sigma_{Cell_1}}(\{a_0, a_1\})$ contains not only a_0 and a_1, but also the constant arrays 0 and 1.

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

We now give the syntactic transformation on Horn clauses, so that the Horn problem induced by the transformed clauses corresponds to the abstraction of the desired predicate by the given data abstraction. We rely on Theorem 3, which states how the abstract Horn problem must be constructed and find its syntactical counterpart. Thus, if P is the predicate to be abstracted by σ, $\mathcal{M}(P)$ must be replaced by $(\gamma_\sigma(\mathcal{M}(P^\#)))(expr)$, where $\mathcal{M}(P^\#)$ is the abstracted set and $P^\#$ the "abstract predicate". Syntactically, this amounts to replacing any instance of $P(expr)$ by $\forall a^\#, F_\sigma(a^\#, expr) \to P^\#(a^\#)$.

Algorithm 1 (dataabs($\mathfrak{C}, P, P^\#, F_\sigma$))**.**
Input *: $\mathfrak{C}$: Horn clauses; P: predicate to abstract; $P^\#$: unused predicate; F_σ.*
Computation *: for each clause $C \in \mathfrak{C}$, for each $P(expr)$ in C, replace $P(expr)$ by $\forall a^\#, F_\sigma(a^\#, expr) \to P^\#(a^\#)$, where $a^\#$ is a new unused variable.*

Example 6 (Using Algorithm 1 to abstract array a of Example 2 with $Cell_1$)**.** *Let us define the data abstraction $F_{\sigma_{Cell_1} \cdot \sigma_{id}^2}$ (discussed in Section 3.2) by :*
$F_{\sigma_{Cell_1} \cdot \sigma_{id}^2}((i, v, N^\#, k^\#), (a, N, k)) \equiv v = a[i] \wedge N^\# = N \wedge k^\# = k$. *And let us execute* ***dataabs****(Clauses of Example 2,* ***For2, F***
The result for Clause 4: $For2(a, N, k) \wedge k < N \to For2(a[k \leftarrow a[k] + 1], N, k + 1)$ is

$$\begin{aligned}&(\forall (i^\#, v^\#, N^\#, k^\#), v^\# = a[i^\#] \wedge N^\# = N \wedge k^\# = k \to For2^\#(i^\#, v^\#, N^\#, k^\#)) \wedge k < N \\ &\to (\forall (i'^\#, v'^\#, N'^\#, k'^\#), v'^\# = a[k \leftarrow a[k] + 1][i'^\#] \wedge N'^\# = N \wedge k'^\# = k + 1 \\ &\to For2^\#(i'^\#, v'^\#, N'^\#, k'^\#)) \qquad (8)\end{aligned}$$

where $a^\#$ from Algorithm 1 is named $(i^\#, v^\#, N^\#, k^\#)$ in the first replacement and $(i'^\#, v'^\#, N'^\#, k'^\#)$ in the second.

Theorem 5 (Algorithm 1 is correct)**.** *If $P^\#$ unused in $\mathfrak{C}$,*
$\forall \mathcal{M}^\#, H_{dataabs(\mathfrak{C}, P, P^\#, F_\sigma)}(\mathcal{M}^\#[P^\# \leftarrow P]) = abs(\mathcal{G}_\sigma^P, H_\mathfrak{C})(\mathcal{M}^\#)$. Thus, the *dataabs* algorithm is complete relative to $\mathcal{G}_\sigma^P$.

When given a set of Horn clauses, one can abstract several predicates (*i.e.* several program points), perhaps all of them, by applying the abstraction algorithm to them, not necessarily with the same abstraction.

3.2 Combining data abstractions

In Example 6, we had to manually adapt the abstraction $Cell_1$ to the predicate $For2$ which contained three variables. We define combinators for abstractions such that those adapted abstractions can be easily defined, and later, analyzed.

Definition 9 ($\sigma_{id}, \sigma_\perp, \sigma_1 \cdot \sigma_2, \sigma_1 \circ \sigma_2$)**.** *These abstractions and combinators are defined by*

1. $\sigma_{id}(x) = \{x\}$; $F_{\sigma_{id}}(x^\#, x) \equiv x^\# = x$.
2. $\sigma_\perp(x) = \{\perp\}$; $F_{\sigma_\perp}(x^\#, x) \equiv x^\# = \perp$
3. $\sigma_1 \cdot \sigma_2(x_1, x_2) = \sigma_1(x_1) \times \sigma_2(x_2)$;$F_{\sigma_1 \cdot \sigma_2}((x_1^\#, x_2^\#), (x_1, x_2)) \equiv F_{\sigma_1}(x_1^\#, x_1) \wedge F_{\sigma_2}(x_2^\#, x_2)$
4. $\sigma_1 \circ \sigma_2(x) = \bigcup_{x_2^\# \in \sigma_2(x_2)} \sigma_1(x_2^\#)$; $F_{\sigma_1 \circ \sigma_2}(x^\#, x) \equiv \exists x_2^\# : F_{\sigma_1}(x^\#, x_2^\#) \wedge F_{\sigma_2}(x_2^\#, x)$

where σ_{id} is the "no abstraction" abstraction, $\sigma_\perp$ abstracts into the unit type (singleton $\perp$) and is used with the $\cdot$ combinator to project a variable out, $\sigma_1 \cdot \sigma_2$ abstracts pairs by the cartesian product of their abstractions, and $\sigma_1 \circ \sigma_2$ emulates applying σ_1 after σ_2.

We have given in this section a general scheme for abstracting Horn clauses using *data abstraction* and shown its correctness. This scheme transforms Horn clauses into extended Horn clauses: new quantifiers ($\forall a^\#$) are introduced which makes current solvers [dMB08, HR18] struggle. We shall now see how to get rid of these quantifiers while retaining relative completeness.

4 Data Abstraction: Quantifier Removal

4.1 A Quantifier Elimination Technique Parametrized by *insts*

Contrarily to other approches that use general-purpose heuristics [BMR13], we design our quantifier elimination from the abstraction itself, which allows us to analyze the completeness property of the quantifier elimination algorithm.

The quantifiers introduced by the abstraction scheme are created either by :

1. The $\forall a^{\#}$ of Algorithm 1, which is handled in this paper.
2. Quantifiers within F_σ (*i.e.* when abstraction composition is used). In this paper, we only handle existential quantifiers in prenex position of F_σ which is sufficient for the abstractions of this paper[4].

Quantifiers are generated for each instance of the abstracted predicate, and to remove the quantifiers, we separate these instances into two classes :

1. The case when the predicate instance is positive. This case is handled by replacing the quantified variables by free variables, possibly renaming them to avoid name clashes. This is correct as these quantifiers would be universal quantifiers when moved to prenex position and thus have same semantics as free variables when considering the satisfiability of the clauses.
2. The case when the predicate instance is negative. In this case, when moved to prenex position, the quantifiers would be existential and thus can not be easily simplified. We use a technique called *quantifier instantiation* [BMR13], which replaces a quantifier by that quantifier restricted to some finite set of expressions I called *instantiation set* (*i.e.* $\forall a^{\#}, expr$ is replaced by $\forall a^{\#} \in I, expr$), which, as I is finite, can be unrolled to remove that quantifier.

Therefore, our quantifier elimination algorithm takes a parameter *insts* which returns the instantiation set for each abstracted negative predicate instance; and eliminates quantifiers according to their types (negative or positive).

Definition 10 (Instantiation set heuristic $insts(F_\sigma, a, ctx)$)**.** *insts is said to be an instantiation set heuristic if and only if:*

1. *insts takes three parameters : F_σ, the abstraction ; a, the variable that was abstracted ; ctx, the context in which the quantifiers are removed.*
2. *$insts(F_\sigma, a, ctx)$ returns an instantiation set for the pair of quantifiers $a^{\#}$ (the quantified variable corresponding to the abstraction of a) and q (the prenex existential quantifiers of F_σ).*

Thus its type is a set of pairs of expressions where the first expression is for $a^{\#}$ and the second for the prenex existential quantifiers of F_σ.

To ease the writing of the algorithm, we assume that the input clauses to the quantifier elimination algorithm have been created using the abstraction scheme at most once for each predicate (one may use abstraction composition to emulate applying it twice) on Horn clauses that where initially normalized. Furthermore, as we will be manipulating existential quantifiers within F_σ, we will define $F_\sigma[q]$ such that $F_\sigma(a^{\#}, a) = \exists q, F_\sigma[q]a^{\#}, a)$. In order words, $F_\sigma[q]$ is F_σ where the prenex existential quantifiers have been replaced by the value q. We will use () as value for q when F_σ has no prenex existential quantifiers.

Algorithm 2 (Quantifier elimination algorithm *eliminate*)**.**

Input:

- *C, an (extended) clause of the form $e_1 \wedge \ldots \wedge e_n \to e'$*
- *insts an instantiation heuristic as in Definition 10*

Computation:

```
1. //We transform quantifiers from the positive instance $e'$ into free variables
   $e'_{res} := free\_var\_of\_positive\_quantifiers(e')$
2. For i from 1 to n
   (a) //We look if $e_i$ is the abstraction of a predicate, if it is not, $e_{res_i} = e_i$
       Let $(F_{\sigma_i}, a_i, P_i^{\#})$ such that $e_i = \forall a^{\#}, F_{\sigma_i}(a^{\#}, a_i) \to P_i^{\#}(a^{\#})$
       If impossible, $e_{res_i} = e_i$ and go to next loop iteration.
   (b) //We compute the context for that instance
       Let $ctx_i = e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge e_{i+1} \wedge \ldots \wedge e_n \to e'_{res}$
   (c) //We compute the instantiation set for that abstraction.
       Let $I_i = insts(F_{\sigma_i}, a_i, ctx_i)$
   (d) //We finally compute $e_i$ after instantiation
       Let $e_{res_i} = \bigwedge_{(a^{\#},q) \in I_i} F_{\sigma_i}[q](a^{\#}, a_i) \to P_i^{\#}(a^{\#})$
3. Return $e_{res_1} \wedge \ldots \wedge e_{res_n} \to e'_{res}$
```

[4]In practice this can be expanded by analyzing what the quantifiers within F_σ would give when moved to prenex position.

Example 7 (Eliminating quantifiers of Clause 8 of Example 6)**.** *Let us apply eliminate on:*

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

In this extended clause, $n = 2$ *an can be decomposed into* e_1*,* e_2 *and* e'*.*

The instantiation algorithm then follows the following steps:

1. *Step 1, computes* e'_{res} *as given in Clause 9*
2. *We enter Step 2 with* $i = 1$ *and it matches the pattern.*
3. *We compute the context and call insts (call of Equation 11). Let us assume it returns* $\{(k, a[k], N, k), (i'^\#, a[i'^\#], N, k)\}$ *(which is the value returned by the instantiation set heuristic we construct later in this paper).*
4. *We get* e_{res_1} *as given in Clause 9*
5. *We enter Step 2 with* $i = 2$ *and it does not match the pattern. Thus,* $e_{res_2} = e_2$

The final clause is thus

$$(a[k] = a[k] \wedge N = N \wedge k = k \rightarrow For2^\#(k, a[k], N, k) \wedge$$
$$a[i'^\#] = a[i'^\#] \wedge N = N \wedge k = k \rightarrow For2^\#(i'^\#, a[i'^\#], N, k)) \wedge k < N$$
$$\rightarrow v'^\# = a[k \leftarrow a[k]+1][i'^\#] \wedge N'^\# = N \wedge k'^\# = k+1 \rightarrow For2^\#(i'^\#, v'^\#, N'^\#, k'^\#) \quad (9)$$

where $v'^\#, i'^\#, N'^\#, k'^\#$ *are new free variables of the clause.*

Simplifying this clause for readability yields:

$$For2^\#(k, a[k], N, k) \wedge For2^\#(i'^\#, a[i'^\#], N, k) \wedge k < N$$
$$\rightarrow For2^\#(i'^\#, a[k \leftarrow a[k]+1][i'^\#], N, k+1) \quad (10)$$

The call to** insts**, which will be studied in Examples 8 and 10, was:

$$insts(F_{\sigma_{Cell_1} \cdot \sigma^2_{id}}, (a, (N, k)), e_2 \rightarrow e'_{res})$$
$$= insts(F_{\sigma_{Cell_1} \cdot \sigma^2_{id}}, (a, (N, k)), k < N \rightarrow v'^\# = a[k \leftarrow a[k]+1][i'^\#]$$
$$\wedge N'^\# = N \wedge k'^\# = k+1 \rightarrow For2^\#((i'^\#, v'^\#), (N'^\#, k'^\#))) \quad (11)$$

Theorem 6 (*eliminate* sound)**.** $\forall C, insts, \mathcal{M}, [\![eliminate(C, insts)]\!]_{\mathcal{M}} \Rightarrow [\![C]\!]_{\mathcal{M}}$

4.2 Constructing a good heuristic *insts*

To ensure relative completeness, and thus the predictability, of our overall abstraction (multiple calls to *dataabs* with different predicates followed by a call to *eliminate*), we need *eliminate* to be complete. The completeness of *eliminate* is highly tied to each call to *insts*, and we therefore define *completeness of a call to insts* such that whenever all calls are complete, *eliminate* is complete.

Definition 11 (Completeness of a call to *insts*)**.** *We say that a call* $insts(F_\sigma, a, ctx)$ *is* complete *if and only if, for any* $\mathcal{M}$*, and any set* E *of elements of the types of* a*, 12 implies 13.*

$$\forall vars, (\forall (a^\#, q), F_\sigma[q](a^\#, a(vars)) \Rightarrow a^\# \in \alpha_\sigma(E)) \Rightarrow [\![ctx(vars)]\!]_{\mathcal{M}} \quad (12)$$

$$\forall vars, (\forall ((a^\#, q)) \in insts(F_\sigma, a, ctx)(vars),$$
$$F_\sigma[q](a^\#, a(vars)) \Rightarrow a^\# \in \alpha_\sigma(E)) \Rightarrow [\![ctx(vars)]\!]_{\mathcal{M}} \quad (13)$$

Remark 3. *We always have 13 implies 12; soundness is based on this.*

Remark 4. *12 should be understood as the clause before* $(a^\#, q)$ *is instantiated:* $\alpha_\sigma(E)$ *represents* $\alpha(\mathcal{M})(P)$*. Therefore, 12 is similar to* $e_i \rightarrow ctx$ *(of algorithm eliminate) which is the currified current state of the clause in the loop. 13 should be understood as* $e_{res_i} \rightarrow ctx$

Theorem 7 (Completeness of *insts* implies that of *eliminate*)**.** *For any* $C, insts, \mathcal{M}$*, if during execution of* $eliminate(C, insts)$ *all calls to insts are complete, then* $[\![C]\!]_{\alpha_{\mathcal{G}}(\mathcal{M})} = [\![eliminate(C, insts)]\!]_{\alpha_{\mathcal{G}}(\mathcal{M})}$ *where* $\mathcal{G}$ *is such that* $\forall i, \gamma_{\mathcal{G}}(\mathcal{M}(P_i^\#)) = \gamma_{\sigma_i}(P_i^\#)$*, with* $i, P_i^\#, \sigma_i$ *as defined in eliminate.*

Remark 5. *We only consider abstract models, that is, $\alpha_{\mathcal{G}}(\mathcal{M})$ where $\mathcal{G}$ represents the galois connection after multiple calls to dataabs. The result is then a consequence of Remark 4.*

Although our previous completeness definition of *insts* correctly captures the necessary properties for our instantiation algorithm to keep equisatisfiability, it is too weak to reason on when using combinators (see Section 3.2). The desired property of the instantiation heuristic is what we call *strong completeness.*

Strong Completeness The definition of *completeness* only applies in the context of boolean types, as required by the Algorithm *eliminate*. However, when handling the impact of the instantiation of a quantifier, one wishes to handle that impact with respect to an arbitrarily typed expression. For example, in the case of combinator $\sigma_1 \cdot \sigma_2$, the instantiation of the quantifiers generated by the abstraction σ_1 must be aware of its impact on the variables to be abstracted by σ_2. This leads to a definition of *strong completeness* that allows any expression type as context parameter of *insts*, and replaces the satisfiability requirement of the context by an equality requirement.

Definition 12 (Strong completeness of $insts(F_\sigma, a, ctx)$)**.** *$insts(F_\sigma, a, ctx)$ is said strongly complete if and only if, for any $E, vars, \mathcal{M}$,*

$$\begin{aligned}\forall((a^\#, q)) \in insts(F_\sigma, a, ctx)(vars), F_\sigma[q](a^\#, a(vars)) \Rightarrow a^\# \in \alpha_\sigma(E) \\ \Rightarrow \exists vars', (\forall(a^\#, q), F_\sigma[q](a^\#, a(vars')) \Rightarrow a^\# \in \alpha_\sigma(E)) \\ \wedge [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}\end{aligned}$$

Remark 6. *This definition is constructed by contraposing that of completeness.*

Theorem 8 (Strong completeness implies completeness)**.** *If ctx is of boolean type,*
$insts(F_\sigma, a, ctx)$ strongly complete $\Rightarrow insts(F_\sigma, a, ctx)$ complete

We give now some results that enable to modularly design instantiation heuristics while remaining (strongly) complete.

Algorithm 3 (*insts* for finite abstractions)**.** *When $\sigma(a)$ finite and F_σ has no existential quantifiers* $insts(F_\sigma, a, ctx) = \{(a^\#, ()), a^\# \in \sigma(a)\}$

Thus $insts(F_{\sigma_{id}}, a, ctx) = \{(a, ())\}$ and $insts(F_{\sigma_\perp}, a, ctx) = \{(\perp, ())\}$

Algorithm 4 (*insts* for combinators)**.** *We will use @ for tuple concatenation.*

```
$insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx) =$
  let $I_1 = insts(F_{\sigma_1}, a_1, (ctx, a_2))$ in        // We want $I_2$ to keep the values of $I_1$ unchanged
  let $I_2 = insts(F_{\sigma_2}, a_2, (ctx, a_1, I_1))$ in    //We return $I_1 \times I_2$ with the right ordering
  // and the abstracted value at the top of the list. .
  $\{((a_1^\#, a_2^\#), q_1 @ q_2) | (a_1^\#, q_1) \in I_1 \wedge (a_2^\#, q_2) \in I_2\}$

$insts(F_{\sigma_1 \circ \sigma_2}, a, ctx) =$
  //We first instantiate $\sigma_2$
  let $I_2 = insts(F_{\sigma_2}, a, ctx)$ in $I_{tmp} := I_2$;   $I_f := \emptyset$
  //For each instantiation of $\sigma_2$ we instantiate with $\sigma_1$
  while $I_{tmp} \neq \emptyset$
    //All orders for picking $(q_0, q_2)$ are valid
    let $(q_0, q_2) \in I_{tmp}$ in $I_{tmp} := I_{tmp} - \{(q_0, q_2)\}$
    //We keep the other instantiation sets unchanged
    //We also keep "$q_0$ is an abstraction of $a$" and the global context unchanged
    let $I_{(q_0,q_2)} = insts(F_{\sigma_1}, q_0, (I_2 - \{(q_0, q_2)\}, I_f, F_{\sigma_2}[q_2](q_0, a), ctx))$ in
    //We combine $I_{(q_0,q_2)}$ with $(q_0, q_2)$
    $I_f := I_f \cup \{(a^\#, (q_0 @ q_1 @ q_2)) | ((a^\#, q_1)) \in I_{(q_0,q_2)}\}$
  return $I_f$
```

//Note : $(a^\#, (q_0@q_1@q_2)) \in I_f \equiv (q_0, q_2) \in I_2 \wedge (a^\#, q_1) \in I_{(q_0,q_2)}$
//Note : $I_{(q_0,q_2)}$ depends on I_f and thus on the picked order

Theorem 9 (Strong Completeness of *insts* of Algorithms 3 and 4)**.** *If σ is finite, $insts(F_\sigma, a, ctx)$ is strongly complete. If its recursive calls are strongly complete, $insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx)$ is strongly complete. If σ_1, σ_2 are compatible: $\forall E \neq \emptyset, \alpha_{\sigma_2} \circ \gamma_{\sigma_2} \circ \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E) = \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)$ and its recursive calls are strongly complete, then $insts(F_{\sigma_1 \circ \sigma_2}, a, ctx)$ is strongly complete.*

Remark 7. *The compatibility condition is true for our abstractions (Theorem 10).*

Example 8 (Using combinator instantiation)**.** *In Example 7, we assumed the result of Call 11:*

$$insts(F_{\sigma_{Cell_1} \cdot \sigma^2_{id}}, (a, (N, k)), k < N \rightarrow v'^\# = a[k \leftarrow a[k] + 1][i'^\#] \\ \wedge N'^\# = N \wedge k'^\# = k + 1 \rightarrow For2^\#((i'^\#, v'^\#), (N'^\#, k'^\#)))$$

Let us now expand this call further using our combinator construction for insts.

1. *We enter the call $insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx)$ with $\sigma_1 = \sigma_{Cell_1}, \sigma_2 = \sigma^2_{id}, a_1 = a, a_2 = (N, k)$ and $ctx = k < N \rightarrow v'^\# = a[k \leftarrow a[k] + 1][i'^\#] \wedge N'^\# = N \wedge k'^\# = k + 1 \rightarrow For2^\#((i'^\#, v'^\#), (N'^\#, k'^\#))$*
2. *We compute $I_1 = insts(F_{\sigma_1}, a_1, (ctx, a_2))$ As we do not have yet an instantiation heuristic for σ_{Cell_1}, let us assume this call returns $I_1 = \{(k, a[k]), (i'^\#, a[i'^\#])\}$. This call is further expanded in Example 10.*
3. *We now compute $I_2 = insts(F_{\sigma_2}, a_2, (ctx, a_1, I_1))$ But $\sigma_2 = \sigma^2_{id}$ thus yielding an embedded call to the $\cdot$ combinator*
   (a) *We enter the call $insts(F_{\sigma_{id} \cdot \sigma_{id}}, (N, k), (ctx, a_1, I_1))$*
   (b) *We call $insts(F_{\sigma_{id}}, N, ((ctx, a_1, I_1), k))$, yielding $\{N\}$*
   (c) *We call $insts(F_{\sigma_{id}}, k, ((ctx, a_1, I_1), N, \{N\}))$, yielding $\{k\}$*
   (d) *We return $\{N, k\}$*
4. *We return the final result: $\{((k, a[k]), (N, k)), ((i'^\#, a[i'^\#]), (N, k + 1))\}$*

Note that if the call to the instantiation using $Cell_1$ is strongly complete, then our final instantiation set is as well. The following call to the instantiation of $Cell_1$ is studied in Example 10.

$$insts(F_{\sigma_1}, a, (ctx, a_2)) = \big(insts(F_{\sigma_{Cell_1}}, a, (k < N \rightarrow v'^\# = a[k \leftarrow a[k] + 1][i'^\#] \\ \wedge N'^\# = N \wedge k'^\# = k + 1 \rightarrow For2^\#((i'^\#, v'^\#), (N'^\#, k'^\#)), (N, k)))\big) \quad (14)$$

5 Cell Abstraction: a complete Data Abstraction

To illustrate our *data abstraction* technique, we show how to handle the cell abstractions of Monniaux and Gonnord [MG16].

5.1 Cell Abstractions

Cell abstractions consist in viewing arrays (maps from an index type to a value type) by a finite number of their cells. However, instead of using cells at specific fixed indices, such as the first or the last, we use parametric cells (*i.e.* cells with a non fixed index). $Cell_1$ of Example 5 corresponds to one parametric cell. In Definition 13, we extend $Cell_1$ to $Cell_n$.

Definition 13. *Cell abstractions $Cell_n$.*
$\sigma_{Cell_n}(a) = \{(i_1, a[i_1]), \ldots, (i_n, a[i_n]))\}$ *and*
$F_{\sigma_{Cell_n}}(((i_1, v_1), \ldots, (i_n, v_n)), a) \equiv v_1 = a[i_1] \wedge \ldots \wedge v_n = a[i_n]$.

Many interesting properties can be defined by cell abstractions (Tab. 1). Furthermore, our data abstraction framework allows formalizing other existing array abstractions by compositions of cell abstractions (Example 9).

Example 9. *Array abstractions from cell abstractions.*

Table 1: Properties specified by cell abstractions

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

Array smashing*: $\sigma_{smash}(a) = \{a[i]\}$. This abstraction keeps the set of values reached but loses all information linking indices and values. It can be constructed using $Cell_1$ abstraction in the following way : $\sigma_{smash} \equiv (\sigma_\perp \cdot \sigma_{id}) \circ \sigma_{Cell_1}$*[5]

Array slicing*: [CCL11, GRS05, HP08] There are several variations, and for readability we present the one that corresponds to "smashing each slice" and picking 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 can be constructed using $Cell_1$ abstraction in the following way : $\sigma_{slice} \equiv (\sigma_{slice_1} \cdot \sigma_{slice_2} \cdot \sigma_{slice_3}) \circ \sigma_{Cell_3}$*[6] *with $\sigma_{s_k}(j, v) = ite(j \in slice_k, v, \perp)$*

Theorem 10. *The abstractions of Example 9 have strongly complete instantiation heuristics when $Cell_n$ has.*

Remark 8. *These abstractions are of the form $\sigma \circ \sigma_{Cell_n}$. Their strong completeness is proven because σ is finite and σ_{Cell_n} always verifies the compatibility condition of Theorem 9 when left-side composed.*

5.2 Instantiating Cell Abstractions

The *data abstraction* framework requires an instantiation heuristic *insts* for $Cell_n$. To achieve the strong completeness property, we first compute all "relevant" indices of the abstracted array, that is, indices which when left unchanged ensure that *ctx* keeps its value.

Algorithm 5 ($relevant(a, expr)$)**.** *We compute the set of relevant indices of the array a for expr. This set may contain $\top$ which signifies that there are relevant indices that the algorithm does not handle (and thus the final algorithm does not ensure strong completeness).*

In this algorithm $arrayStoreChain(b, I, V)$ denotes an array expression equal to $b[i_1 \leftarrow v_1][\ldots][i_n \leftarrow v_n]$ with $i_1, \ldots, i_n \in I$ and $v_1, \ldots, v_n \in V$. n may be 0.

```
$relevant(a, expr) =$
  //For a a variable avar, return the indices that are "read"
  let $read\ avar\ expr = match\ expr\ with$
                    $|arrayStoreChain(avar, I, V)[i] \rightarrow \{i\} \cup \bigcup_{(j,v)\in(I,V)} read(avar, j) \cup read(avar, v)$
    $|\forall q, e|\exists q, e \rightarrow map\ (fun\ x \rightarrow ite(q \in x, \top, x))\ read(avar, e)$                    in
    $|Cons(exprs) \rightarrow \bigcup_{expr\in exprs} read(avar, exprs)$        $|avar \rightarrow \{\top\}$
                                        //with Cons an expression constructor or predicate
  //Reducing a to a variable avar
  $match\ a\ with$
    $|arrayStoreChain(avar, I, V)\ when\ is\_var(avar) \rightarrow I \cup read(avar, expr)$
    $|\_ \rightarrow \{\top\}$
```

Remark 9. *For readability, the algorithm is kept short, but may be improved to return $\top$ in fewer cases using techniques from [BMS06]*

[5]This is a semantic equivalence, not a strict equality : formally $(\sigma_\perp \cdot \sigma_{id}) \circ \sigma_{Cell_1}(a)$ returns $\{(\perp, a[i]\}$ instead of $\{a[i]\}$

[6]As for smashing, this is a semantic equivalence

Theorem 11 ($relevant(a, expr)$ is correct)**.** *If* $\top \notin relevant(a, expr)$ *then*

$$\forall \mathcal{M}, vars, a', (\forall i \in relevant(a, expr), a'[i] = a(vars)[i]) \Rightarrow \exists vars', [\![expr(vars)]\!]_{\mathcal{M}} = [\![expr(vars')]\!]_{\mathcal{M}} \wedge a(vars') = a'$$

We use our relevant indices to construct an instantiation heuristic for $Cell_n$.

Algorithm 6 (Instantiation heuristic for $Cell_n$)**.**

```
$insts(F_{\sigma_{Cell_n}}, a, ctx) =$
    $let\ R = relevant(a, ctx)\ in$    $//Compute\ relevant\ set$
    $let\ Ind = ite(R = \emptyset, \_, R - \{\top\})\ in$    $//Make\ it\ non\ empty\ and\ remove\ \top$
    $//\_\ can\ be\ chosen\ as\ any\ value\ of\ the\ index\ type$
    $let\ I = \{(i, a[i]) | i \in Ind\}\ in (I^n, ())$    $//\ \ make\ it\ a\ pair\ index\ value,\ make\ n\ copies$
```

Theorem 12 (Strong Completeness for cell Abstraction)**.** *Any call to* $insts(F_{\sigma_{Cell_n}}, a, ctx)$ *is strongly complete whenever* $\top \notin relevant(a, ctx)$.

Example 10 (Instantiation of $Cell_1$)**.** *In Example 8, we assumed the result of the Call 14:*

$$insts(F_{\sigma_1}, a, (ctx, a_2)) = insts(F_{\sigma_{Cell_1}}, a, (k < N \rightarrow v'^{\#} = a[k \leftarrow a[k]+1][i'^{\#}] \wedge N'^{\#} = N \wedge k'^{\#} = k+1 \rightarrow For2^{\#}((i'^{\#}, v'^{\#}), (N'^{\#}, k'^{\#})), (N, k)))$$

Let us use our instantiation heuristic for $Cell_1$ *to compute the result of that call.*

1. *We first compute the relevant set, we obtain* $R = \{k, i'^{\#}\}$
2. *It is already non empty and does not contain* $\top$*, so* $Ind = R = \{k, i'^{\#}\}$
3. *We add the value part, yielding* $I = \{(k, a[k]), (i'^{\#}, a[i'^{\#}])\}$
4. $n = 1$*, therefore the result is* $I^1 = I = \{(k, a[k]), (i'^{\#}, a[i'^{\#}])\}$

And the call is complete as the relevant set did not contain $\top$.

6 Implementation and Experiments

In Sections 3 and 4, we constructed the building blocks to verify programs using an abstraction technique that strives to ensure relative completeness and in Section 5, we gave a powerful abstraction for arrays. We now combine these sections to create a tool for the verification of programs with arrays. We benchmark this tool and compare it to other tools and then analyze the results.

6.1 The full algorithm

Our tool uses Algorithm 7 which has for only parameter n. The abstraction used consists in abstracting each array of each predicate by $Cell_n$.

Algorithm 7 (Full n-Cell Abstraction Algorithm $trs(\mathfrak{C}, n)$)**.**
Input*: A set of Horn clauses* $\mathfrak{C}$*,* n *the number of cells*
Computation:

```
1. Abstraction : For each predicate $P$ of $C$ over types $t_1, \ldots, t_k$
    (a) Let $\sigma^P = \prod_{1 \le i \le k} ite(isArray(t_k), \sigma_{Cell_n}, \sigma_{id})$
    (b) Let $P^{\#}$ be an unused predicate.
    (c) $\mathfrak{C} := dataabs(\mathfrak{C}, F_{\sigma}^P, P, P^{\#})$
2. Quantifier elimination : For each clause $C \in \mathfrak{C}$
    $C := eliminate(C, insts)$ //insts is the heuristic discussed in this article
3. Simplification (optional): For each clause $C \in \mathfrak{C}$
    (a) $C := simplify(C, insts)$ //we simplify as in Clause 10
```

Table 2: Experimental results

	#prg	n	Noabs				VapHor				Dataabs				Dataabs acker			
			👍	◌	?	≥ 1	👍	◌	?	≥ 1	👍	◌	?	≥ 1	👍	◌	?	≥ 1
Buggy	4	1	4	0	0	4	4	0	0	4	4	0	0	4	4	0	0	4
Buggy	4	2	–	–	–	–	4	0	0	4	0	4	0	0	3	1	0	3
NotHinted	12	1	0	11.5	0.5	0	1	11	0	1	0	12	0	12	0	11.83	0.17	0
NotHinted	12	2	–	–	–	–	0	12	0	0	0	12	0	12	0	12	0	0
Hinted	12	1	0	11	1	0	4	8	0	7	8.99	2.84	0.17	11	8.83	3.17	0	**12**
Hinted	12	2	–	–	–	–	0	12	0	0	0	12	0	0	5.83	6.17	0	6

Columns: corresponds to the abstraction tool. 1. Noabs corresponds to no abstraction 2. Vaphor is the tool by Monniaux & Gonnord 3. Dataabs and Dataabs acker represent our implementation, with and without Ackermannisation.
Lines: files grouped by category and number of cells
Values:👍(respectively ◌, ?) correspond to the number of files that where correct (respectively timeout, unknown), averaged over the 3 naming options and the 2 random seeds). ≥ 1 represents the number of files for which at least one of the naming options and random seeds returned the correct result.

(b) $C := Ackermannize(C)$ */*This step removes possible array writes by applying read over write axioms. Then if an array a is only used in read expressions, we eliminate it at quadratic cost [KS08, §3.3.1]. Note that this is is not possible before abstraction because a is a parameter of predicates. */*

We implemented this algorithm, and used it on clauses generated by the Vaphor converter from mini-Java programs to Horn Clauses. The implementation is publicly available [7]. Overall the toolchain ensures soundness and relative completeness due to the strong completeness of our building blocks. To ensure that within the computation of $insts(F_{\sigma_{Cell_n}}, a, ctx)$, $relevant(a, ctx)$ does not contain $\top$, we rely on the form of clauses generated by the Vaphor converter, among which: 1. There are no array equalities. 2. There is at most one negative predicate per clause. Theorem 14 available in the appendix states the exact requirements on the form of the clauses to ensure relative completeness.

6.2 Experimental setting and results

We modified the converter to take into account optional additional invariant information given by the programmer, in the form of "hints". These hints are additional assertion clauses so that the solver may converge more easily to the desired model. These additional clauses are also abstracted by our abstraction.

Our initial Horn clauses are generated by the converter from programs in three categories: 1. incorrect programs. 2. correct programs without hints. 3. correct programs with hints. We then compare our implementation of Algorithm 7 with no abstraction and the abstraction tool of Vaphor on the modified benchmarks of [MG16]. Note that the latter tool implements a variant of $Cell_n$ that always Ackermannizes in the process. We modified the benchmarks of [MG16, DDA10] such that properties are checked using a loop instead of a random access, e.g., they take i random and assert $t[i] = 0$ to verify $\forall i\ t[i] = 0$, thus enforcing the use of quantified invariants. This is why Vaphor behaves much more poorly on our examples than in [MG16]. The modified benchmarks are available at `https://github.com/vaphor/array-benchmarks`.

We use as backend the state of the art solver Z3 version 4.8.8 - 64 bit with timeout 40s on a laptop computer. Because of issue `https://github.com/Z3Prover/z3/issues/909` that we have witnessed, we launch each benchmark with 3 different naming conventions for predicates in addition to using 2 random seeds. The results are presented in Table 2. We did not show columns for incorrect results or errors as there are none.

6.3 Analysis

Abstraction & Solving timings Our abstraction is fast: most of it is search & replace; most of the time is spent in the hand-written simplification algorithm, which is not the purpose of this paper. The solving time in Z3 is much higher. It seems that, for correct examples, Z3 either converges quickly (<5sec) or fails to

[7]`https://github.com/vaphor`

converge regardless of timeout—only in 18 cases out of 1176 does it solve the problem between 5s and the timeout.

Soundness and relative completeness There are no incorrect results, confirming that all implementations are sound. There are cases where Z3 cannot conclude "correct" on its own but with enough help (*i.e.* hints and Ackermannization, different predicate names, and random seeds) all files are solved as shown by the hinted, $Cell_1$ line, column ≥ 1 of Dataabs acker. This confirms that our implementation is relatively complete, as proved in theory.

Tool comparison Z3 without abstraction is unable to solve any correct example, even with hints. We mostly behave better than Vaphor for $Cell_1$ on hinted examples, perhaps because we create smaller instantiation sets: Vaphor handles complex clauses greedily, using all indices read as instantiation set.

$Cell_1$ versus $Cell_2$ All our correct examples have invariants expressible by $Cell_1$, and thus also by $Cell_2$. However, the clauses generated using $Cell_2$ are bigger and the instantiation set sizes are squared, thus complexifying the clauses and probably stopping Z3 from converging easily on an invariant. Vaphor, in contrast, generates fewer instantiations for $Cell2$ by using $\sigma_{Cell_2}(a) = \{(i, a[i], j, a[j]), i < j\}$, which explains its better performance on $Cell_2$ buggy examples.

Ackermannizing or not Ackermannization completely removes arrays from the clauses, but changes neither the invariants nor the space in which invariants are sought. Z3 is supposed to handle the theory of arrays natively and should thus be more efficient than our eager Ackermannization; yet the latter improves results on non buggy examples.

Overall results and Z3 Our tool transforms a Horn problem with arrays requiring quantified invariants which lie within the $Cell_n$ abstraction into an equivalently satisfiable Horn problem which needs neither quantifiers nor arrays. The non-hinted examples show that this is insufficient to make automatic proofs of programs with arrays practical, mainly because Z3 struggles to handle these array free Horn problems. In addition, Z3 sometimes succeeds or not, or even returns unknown, depending on details in the input (e.g., predicate naming, randon, Ackermannisation). Further work is needed on integer Horn solvers for better performance and reduced brittleness.

Benchmarks and future work The current benchmarks are generated from a toy Java langage and have invariants expressible in $Cell_1$. Future work includes (i) adding new challenging examples which require $Cell_2$ such as sortedness or even more complex invariants such as "the array content is preserved as a multiset" [MG16]; (ii) tackling challenging literature examples [Bey19] in real langages, perhaps using a front end such as SeaHorn [GKKN15].

7 Related Work

Numerous abstractions for arrays have been proposed in the literature, among which array slicing [CCL11, GRS05, HP08]. In Example 9 we showed how they are expressible in our framework. Similarly to Monniaux and Alberti [MA15] we think that disconnecting the array abstraction from other abstractions and from solving enables using back-end solvers better. Like Monniaux and Gonnord [MG16] we use Horn Clauses to encode our program under verification, but we go a step further by using Horn Clauses as an intermediate representation to chain abstractions. Furthermore, our formalization is cleaner for multiple arrays and proves relative completeness.

Our instantiation method is inspired from previous work on solving quantified formulae [BMR13, BMS06, GSV18]. [BMS06] does not consider Horn clauses, that is, expressions with unknown predicates, but only expressions with quantifiers. [BMR13] has an approach very similar to ours, but without casting it within the framework of *data abstractions*; they use trigger-based instantiation. Both instantiation methods of [BMR13, BMS06] lead to bigger instantiation sets than the one we suggest, yet, contrary to us, they do not prove completeness. Finally, the technique used in [GSV18] creates instantiation sets not as pre-processing, but during analysis. Although more general, it is highly likely that the technique suffers from the same unpredictability that Horn solvers have. In our case, we believe that we can tailor the instantiation set to the abstraction and analyze its precision.

Finally, other recent approaches focus on more powerful invariants through proofs by induction [ISIRS20]. However, as stated by their authors, their approach is complementary to ours: theirs is less specialized, and thus has trouble where our approach may easily succeed, but enables other invariants: our data abstraction framework may allow abstracting within their induction proofs.

8 Conclusion

We have proposed an approach for the definition, combination and solving of data abstractions for Horn Clauses. The framework also provides sufficient conditions for (relative) completeness, and prove the result for a large class of array abstractions. We propose an implementation and experimental evaluation on classical examples of the literature. Future work include extending the applicability of the framework for other data structures such that trees.

References

[Bey19] Dirk Beyer. Automatic Verification of C and Java Programs: SV-COMP 2019. In *TACAS*, 2019.

[BMR13] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. On Solving Universally Quantified Horn Clauses. In *SAS*, 2013.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, 2006.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. *SIGPLAN Not.*, 46(1):105–118, January 2011.

[DDA10] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In A. D. Gordon, editor, *Proceedings of the 19th European Symposium on Programming*, volume 6012 of *LNCS*, pages 246–266. Springer, 2010.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[GKKN15] Arie Gurfinkel, Themesghen Kahsai, Anvesh Komuravelli, and Jorge Navas. The SeaHorn Verification Framework. In *CAV*, 2015.

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. A Framework for Numeric Analysis of Array Operations. In *PLDI*, 2005.

[GSV18] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on Demand. In *ATVA*, 2018.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering Properties About Arrays in Simple Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, New York, NY, USA, 2008. Association for Computing Machinery.

[HR18] Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn Solver. In *FMCAD*, 2018.

[ISIRS20] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Putting the Squeeze on Array Programs: Loop Verification via Inductive Rank Reduction. In *VMCAI*, 2020.

[KRS19] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. JayHorn: A Java Model Checker: (Competition Contribution), 2019.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008.

[MA15] David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *SAS*, 2015.

[MG16] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 361–382, 2016.

[RY20] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis An Abstract Interpretation Perspective*. MIT Press, 2020.

A Appendix: proofs

A.1 Horn Clauses and Galois connections

Definition 14 (Galois connection $\mathcal{G}$, defines $\alpha_\mathcal{G}, \gamma_\mathcal{G}$)**.** *A Galois connection $\mathcal{G}$ between two lattices is defined by*

- $\alpha_\mathcal{G} : \mathscr{C} \to \mathcal{A}$ *gives the abstraction of a value.*
- $\gamma_\mathcal{G} : \mathcal{A} \to \mathscr{C}$ *gives the concrete value of an abstract element.*

where: (i) $\alpha_\mathcal{G}, \gamma_\mathcal{G}$ are monotone (ii) $a \leq \alpha_\mathcal{G}(c) \equiv \gamma_\mathcal{G}(a) \leq c$

Theorem 13 (Further properties of Galois connections)**.** *3. $S \leq \gamma_\mathcal{G}(\alpha_\mathcal{G}(S))$ for soundness.*
4. $\forall S^\#, \alpha_\mathcal{G}(\gamma_\mathcal{G}(S^\#)) \leq S^\#$ for minimal precision loss.
5. $\alpha_\mathcal{G} \circ \gamma_\mathcal{G} \circ \alpha_\mathcal{G} = \alpha_\mathcal{G}$ and $\gamma_\mathcal{G} \circ \alpha_\mathcal{G} \circ \gamma_\mathcal{G} = \gamma_\mathcal{G}$: abstracting an already abstracted element yields that element.

Proof. These proofs can be found in [RY20]. □

Theorem 1 (Horn problems as a condition on models, defines $H(\mathcal{M})$)**.** *A Horn problem H is satisfiable if and only if $\exists \mathcal{M}, f_H(\mathcal{M}) \leq \mathcal{M} \wedge \mathcal{M} \leq \mathcal{U}_H$*

Proof. Assume $H(\mathcal{M})$. Because $f_H(\mathcal{M}) \leq \mathcal{M}$, $\mathcal{M}$ is a post fixpoint and therefore $\operatorname{lfp} f_H \leq \mathcal{M}$. Finally, we have $\operatorname{lfp} f_H \leq \mathcal{M} \leq \mathcal{U}_H$.

Assume $\operatorname{lfp} f_H \leq \mathcal{U}_H$. We have $H(\operatorname{lfp} f_H)$. □

Theorem 2 (Horn clauses are instances of Horn problems, defines $H_\mathfrak{C}$)**.** *Let $\mathfrak{C}$ be a set of Horn clauses. There exists a Horn problem $H_\mathfrak{C}$, such that for any model $\mathcal{M}$, $H_\mathfrak{C}(\mathcal{M}) = \bigwedge\limits_{C \in \mathfrak{C}} [\![\forall vars, C(vars)]\!]_\mathcal{M}$.*

Proof. $\bigvee$ denotes the join (least upper bound) operator of the lattice of models, $\bigwedge$ the meet operator (greatest lower bound). Assertion clauses are clauses with no positive predicates. For a set of Horn clauses $\mathfrak{C}$ denote $assert(\mathfrak{C})$ the set of assertion clauses of $\mathfrak{C}$.

Let $f_{H_\mathfrak{C}}(\mathcal{M}) = \bigwedge\{\mathcal{M}' | \forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_{pos=\mathcal{M}', neg=\mathcal{M}}\}$
where $[\![X]\!]_{pos=\mathcal{M}', neg=\mathcal{M}}$ interprets the positive predicate of X by $\mathcal{M}'$ and the negative predicates by $\mathcal{M}$.
Let $\mathcal{U}_{H_\mathfrak{C}} = \bigvee\{\mathcal{M} | \forall C \in assert(\mathfrak{C}), [\![\forall vars, C(vars)]\!]_\mathcal{M}\}$.

1. $f_{H_\mathfrak{C}}$ monotonic: consider $\mathcal{M}_1 \leq \mathcal{M}_2$.
   $\{\mathcal{M}' | \forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_{pos=\mathcal{M}', neg=\mathcal{M}_2}\} \subseteq \{\mathcal{M}' | \forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_{pos=\mathcal{M}', neg=\mathcal{M}}$
   Thus, $f_{H_\mathfrak{C}}(\mathcal{M}_1) \leq f_{H_\mathfrak{C}}(\mathcal{M}_2)$.
2. $H_\mathfrak{C}(\mathcal{M}) = \forall C \in \mathfrak{C}, [\![\forall vars, C(vars)]\!]_\mathcal{M}$:
   (a) Assume $H_\mathfrak{C}(\mathcal{M})$.
   We have $\forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_{pos=f_{H_\mathfrak{C}}(\mathcal{M}), neg=\mathcal{M}}$. Thus, because $f_{H_\mathfrak{C}}(\mathcal{M}) \leq \mathcal{M}$, we have
   $\forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_{pos=\mathcal{M}, neg=\mathcal{M}}$
   $= \forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_\mathcal{M}$. The assertion clauses are simply proven by $\mathcal{M} \leq \mathcal{U}_{H_\mathfrak{C}}$
   (b) Assume $\forall C \in \mathfrak{C}, [\![\forall vars, C(vars)]\!]_\mathcal{M}$. First $\mathcal{M} \leq \mathcal{U}_{H_\mathfrak{C}}$ by definition of $\mathcal{U}_{H_\mathfrak{C}}$. Secondly, $\mathcal{M} \in \{\mathcal{M}' | \forall C \in (\mathfrak{C} - assert(\mathfrak{C})), [\![\forall vars, C(vars)]\!]_{pos=\mathcal{M}', neg=\mathcal{M}}\}$, thus $f_{H_\mathfrak{C}}(\mathcal{M}) \leq \mathcal{M}$.

□

Theorem 3 (Definition $abs(\mathcal{G}, H)$ is correct)**.** *For all $\mathcal{M}^\#$, the following statements are equivalent (with the notation $H(\mathcal{M})$, where H is a Horn problem and $\mathcal{M}$ a possible model, from Theorem 1):*

1. *$abs(\mathcal{G}, H)(\mathcal{M}^\#)$*
2. *$H(\gamma_\mathcal{G}(\mathcal{M}^\#))$*
3. *$abs(\mathcal{G}, H)(\alpha_\mathcal{G} \circ \gamma_\mathcal{G}(\mathcal{M}^\#))$*

Remark 10. *From this theorem, it follows that:*

1. *The definition of $abs(\mathcal{G}, H)$ corresponds to the desired abstraction as : H satisfiable by a model expressible by the abstraction $(\gamma_{\mathcal{G}}(\mathcal{M}^\#))$ is equivalent to $abs(\mathcal{G}, H)$ satisfiable.*
2. *$abs(\mathcal{G}, H)(\mathcal{M}^\#)$ is constructible from H and $\gamma_{\mathcal{G}}$. This will be used in Theorem 5.*

Proof. Introduce $\mathcal{M}^\#$. Proof that i is equivalent to ii

$$abs(\mathcal{G}, H)(\mathcal{M}^\#)$$

$$\equiv \alpha_{\mathcal{G}} \circ f_H \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \mathcal{M}^\# \wedge \mathcal{M}^\# \leq \alpha_{\mathcal{G}}(\mathcal{U}_H)$$

We use the item ii of 14

$$\equiv f_H \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \gamma_{\mathcal{G}}(\mathcal{M}^\#) \wedge \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \mathcal{U}_H$$

$$\equiv H(\gamma_{\mathcal{G}}(\mathcal{M}^\#))$$

Proof that i is equivalent to iii

$$abs(\mathcal{G}, H)(\mathcal{M}^\#)$$

$$\equiv \alpha_{\mathcal{G}} \circ f_H \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \mathcal{M}^\# \wedge \mathcal{M}^\# \leq \alpha_{\mathcal{G}}(\mathcal{U}_H)$$

We use the item ii of 14

$$\equiv f_H \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \gamma_{\mathcal{G}}(\mathcal{M}^\#) \wedge \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \mathcal{U}_H$$

We use the item 5 of Theorem 13

$$\equiv f_H \circ \gamma_{\mathcal{G}} \circ \alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \gamma_{\mathcal{G}} \circ \alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \\ \wedge \gamma_{\mathcal{G}} \circ \alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \mathcal{U}_H$$

We use the item ii of 14

$$\equiv \alpha_{\mathcal{G}} \circ f_H \circ \gamma_{\mathcal{G}}(\alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#)) \leq \alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \wedge \\ \alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#) \leq \alpha_{\mathcal{G}}(\mathcal{U}_H)$$

$$\equiv abs(\mathcal{G}, H)(\alpha_{\mathcal{G}} \circ \gamma_{\mathcal{G}}(\mathcal{M}^\#))$$

□

Theorem 4 (Soundness with relative completeness is *abs*)**.** *If a transformation alg is sound and complete relative to $\mathcal{G}$, then*

$$\forall \mathfrak{C}, H_{alg(\mathfrak{C})} \text{ satisfiable } \equiv abs(\mathcal{G}, H_{\mathfrak{C}}) \text{ satisfiable}$$

In other words, when a transformation is sound and complete relative to an abstraction, it exactly does the abstraction.

Proof.

$$H_{\mathfrak{C}} \text{ satisfiable } \Rightarrow abs(\mathcal{G}, H_{\mathfrak{C}}) \text{ satisfiable}$$

using Theorem 3, which combined with soundness yields

$$H_{alg(\mathfrak{C})} \text{ satisfiable } \Rightarrow H_{\mathfrak{C}} \text{ satisfiable}$$

which used with relative completeness gives the equivalence. □

Theorem 5 (Algorithm 1 is correct)**.** *If $P^\#$ is not used in $\mathfrak{C}$, then for any $\mathcal{M}^\#$,*

$$H_{dataabs(\mathfrak{C}, P, P^\#, F_\sigma)}(\mathcal{M}^\#[P^\# \leftarrow P]) = abs(\mathcal{G}_\sigma^P, H_{\mathfrak{C}})(\mathcal{M}^\#)$$

Proof. $H_{dataabs(\mathfrak{C},P,P^\#,F_\sigma)}(\mathcal{M}^\#[P^\# \leftarrow P])$
$=_1 \bigwedge_{C^\# \in dataabs(\mathfrak{C},P,P^\#,F_\sigma)} [\![\forall vars, C^\#(vars)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]}$
$=_2 \bigwedge_{C \in \mathfrak{C}} [\![\forall vars, C(vars)]\!]_{\gamma_{\mathcal{G}_P}(\mathcal{M}^\#)}$
$=_3 H_{\mathfrak{C}} \circ \gamma_{\mathcal{G}_P}(\mathcal{M}^\#)$
$=_4 abs(\mathcal{G}_\sigma^P, H_{\mathfrak{C}})(\mathcal{M}^\#)$
The first and third equalities correspond to unrolling the definition of $H(\mathcal{M})$. The fourth equality is proved by Theorem 3, now let us prove the second equality.
Let $replaced(e) =_{def} e$ where each instance of $P(expr)$ has been replaced by $\forall a^\#, F_\sigma(a^\#, expr) \rightarrow P^\#(a^\#)$. Let us prove the second equality.
$\bigwedge_{C^\# \in dataabs(\mathfrak{C},P,P^\#,F_\sigma)} [\![\forall vars, C^\#(vars)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]}$
$= \bigwedge_{C \in \mathfrak{C}} [\![\forall vars, replaced(C)(vars)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]}$ by the definition of *replaced*.

We now prove equality 2 by proving that $\forall C \in \mathfrak{C}, \forall vars$,
$[\![replaced(C)(vars)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]} = [\![C(vars)]\!]_{\gamma_{\mathcal{G}_P}(\mathcal{M}^\#)}$.

We show it by showing, by structural induction on the expression e, the property $Ind(e)$: $[\![replaced(e)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]} = [\![e]\!]_{\gamma_{\mathcal{G}_P}(\mathcal{M}^\#)}$.
Assume e is built as $Cons(exprs)$ using the $Cons$ term constructor at the root.

1. if $Cons$ is not a predicate (a boolean operator, for instance) then by the induction hypothesis, the evaluation of $exprs$ is unchanged, thus $Ind(e)$
2. if $Cons$ is a predicate P' different from P, then $\gamma_{\mathcal{G}_P}$ does not affect P' and, by the induction hypothesis, the evaluation of $exprs$ is unchanged, thus $Ind(e)$
3. if $Cons$ is P, then $[\![replaced(e)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]}$
$= [\![\forall a^\#, F_\sigma(a^\#, exprs) \rightarrow P^\#(a^\#)]\!]_{\mathcal{M}^\#[P^\# \leftarrow P]}$
$= \forall a^\#, F_\sigma(a^\#, exprs) \rightarrow \mathcal{M}^\#[P^\# \leftarrow P](P^\#)(a^\#)$
$= \forall a^\#, F_\sigma(a^\#, exprs) \rightarrow \mathcal{M}^\#(P)(a^\#)$
$= exprs \in \{c | \sigma(c) \subseteq \mathcal{M}^\#(P)\}$
$= exprs \in \gamma_\sigma(\mathcal{M}^\#(P))$
$= exprs \in \gamma_{\mathcal{G}_P}(\mathcal{M}^\#)(P)$
$= [\![P(exprs)]\!]_{\gamma_{\mathcal{G}_P}(\mathcal{M}^\#)}$

Thus, $Ind(e)$ holds in all cases. □

Theorem 6 (Soundness of *eliminate*)**.** $\forall C, insts, \mathcal{M}, [\![eliminate(C, insts)]\!]_{\mathcal{M}} \Rightarrow [\![C]\!]_{\mathcal{M}}$

Proof. Fix $C, insts, \mathcal{M}$. We have $\forall i, [\![e_i]\!]_{\mathcal{M}} \Rightarrow [\![e_{res_i}]\!]_{\mathcal{M}}$ as the finite conjunction over I_i in e_i is contained in the infinite conjunction of e_i. Then $[\![e_1 \wedge \ldots \wedge e_n]\!]_{\mathcal{M}} \Rightarrow [\![e_{res_1} \wedge \ldots \wedge e_{res_n}]\!]_{\mathcal{M}}$. Finally, as $[\![e']\!]_{\mathcal{M}} \equiv [\![e'_{res}]\!]_{\mathcal{M}}$, we have $[\![eliminate(C, insts)]\!]_{\mathcal{M}} \Rightarrow [\![C]\!]_{\mathcal{M}}$. □

A.2 Completeness and strong complenetess

Theorem 7 (Completeness of calls to *insts* implies completeness of *eliminate*)**.** *For any $C, insts, \mathcal{M}$, if during execution of eliminate$(C, insts)$ all calls to insts are complete, then*

$$[\![C]\!]_{\alpha_{\mathcal{G}}(\mathcal{M})} = [\![eliminate(C, insts)]\!]_{\alpha_{\mathcal{G}}(\mathcal{M})}$$

where $\mathcal{G}$ is such that $\forall i, \gamma_{\mathcal{G}}(\mathcal{M}(P_i^\#)) = \gamma_{\sigma_i}(P_i^\#)$, with $i, P_i^\#, \sigma_i$ as defined in eliminate.

Proof. Let $C_i = e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge e_{res_i} \wedge e_{i+1} \ldots \wedge e_n \rightarrow e'_{res}$. and let $\mathcal{M}' = \alpha_{\mathcal{G}}(\mathcal{M})$. Our algorithm *eliminate* takes as input a clause equivalent to C_0 as step 1 does not change the semantics, and outputs C_n. Let us show by induction on i that $[\![C_i]\!]_{\mathcal{M}'} \equiv [\![C_0]\!]_{\mathcal{M}'}$.

$[\![C_{i+1}]\!]_{\mathcal{M}'}$

$\equiv [\![e_{res_{i+1}} \to e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge e_{res_i} \wedge e_{i+2} \wedge \ldots \wedge e_n \to e'_{res}]\!]_{\mathcal{M}'}$

by currification

$\equiv [\![e_{res_{i+1}} \to ctx_{i+1}]\!]_{\mathcal{M}'}$

assuming step 2 matches

$\equiv \forall vars, (\forall ((a^{\#}, q)) \in insts(F_{\sigma_{i+1}}, a_{i+1}, ctx_{i+1})(vars),$
$F_{\sigma_{i+1}}[q](a^{\#}, a_{i+1}(vars)) \Rightarrow a^{\#} \in \mathcal{M}'(P^{\#}_{i+1})) \to [\![ctx_{i+1}(vars)]\!]_{\mathcal{M}'}$

expanding the match pattern

$\equiv \forall vars, (\forall ((a^{\#}, q)), F_{\sigma_{i+1}}[q](a^{\#}, a_{i+1}(vars))$
$\Rightarrow a^{\#} \in \mathcal{M}'(P^{\#}_{i+1})) \to [\![ctx_{i+1}(vars)]\!]_{\mathcal{M}'}$

using completeness of that call to *insts*, with $\alpha_\sigma(E) = \mathcal{M}'(P^{\#}_{i+1})$

$\equiv [\![e_{i+1} \to e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge e_{res_i} \wedge e_{i+2} \wedge \ldots \wedge e_n \to e'_{res}]\!]_{\mathcal{M}'}$

using the match pattern

$\equiv [\![C_i]\!]_{\mathcal{M}'}$ decurrying

$\equiv [\![C_0]\!]_{\mathcal{M}'}$ by induction assumption

□

Theorem 8 (Strong completeness implies completeness)**.** *If ctx is of boolean type,* $insts(F_\sigma, a, ctx)$ *strongly complete* $\Rightarrow$ $insts(F_\sigma, a, ctx)$ *complete*

Proof. Assume strong completeness of $insts(F_\sigma, a, ctx)$ and fix $E, \mathcal{M}$. By contraposition, completeness of $insts(F_\sigma, a, ctx)$ is equivalent to :

$$\begin{gathered}(\exists vars, \neg [\![ctx(vars)]\!]_{\mathcal{M}} \wedge (\forall ((a^{\#}, q)) \in insts(F_\sigma, a, ctx)(vars), \\ F_\sigma[q](a^{\#}, a(vars)) \Rightarrow a^{\#} \in \alpha_\sigma(E))) \\ \Rightarrow \\ (\exists vars', \neg [\![ctx(vars')]\!]_{\mathcal{M}} \wedge \\ (\forall ((a^{\#}, q)), F_\sigma[q](a^{\#}, a(vars)) \Rightarrow a^{\#} \in \alpha_\sigma(E)))\end{gathered}$$

Now introduce *vars* and assume the first part of the implication.

Therefore we have,
$\forall ((a^{\#}, q)) \in insts(F_\sigma, a, ctx)(vars), F_\sigma[q](a^{\#}, a(vars)) \Rightarrow a^{\#} \in \alpha_\sigma(E)$, and we obtain using strong completeness

$$\exists vars', (\forall (a^{\#}, q), F_\sigma[q](a^{\#}, a(vars)) \Rightarrow a^{\#} \in \alpha_\sigma(E) \wedge \\ [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}.$$

But using $\neg [\![ctx(vars)]\!]_{\mathcal{M}}$ (assumed in the first part of completeness), and $[\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}$ we obtain $\neg [\![ctx(vars')]\!]_{\mathcal{M}}$.

Therefore, we have both elements of our contraposed completeness and our proof is complete. □

Even though most abstractions for unbounded data structures are fairly complex and involve infinitely many abstract values for some concrete values (*i.e.* $cardinal(\sigma(a))$ is infinite) as is the case for $Cell_1$, many "filler" abstractions such as σ_{id}, or simpler abstractions (see Section 5) are finite. For finite abstractions, one can define a very simple heuristic *insts* which verifies the strong completeness property : return the set of abstract elements!

Theorem 9 (Strong Completeness of $insts$ of Algorithms 3 and 4)**.** *If σ is finite, $insts(F_\sigma, a, ctx)$ is strongly complete. If its recursive calls are strongly complete, $insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx)$ is strongly complete. If σ_1, σ_2 are compatible: $\forall E \neq \emptyset, \alpha_{\sigma_2} \circ \gamma_{\sigma_2} \circ \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E) = \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)$ and its recursive calls are strongly complete, then $insts(F_{\sigma_1 \circ \sigma_2}, a, ctx)$ is strongly complete.*

Proof.

- $f$inite case

  Assume F_σ does not use nested existential quantifiers and $\forall vars, \sigma(a(vars)) \subseteq insts(F_\sigma, a, ctx)(vars)$ and introduce $E, vars, \mathcal{M}$.

$$\begin{aligned}
&\forall((a^\#, ())) \in insts(F_\sigma, a, ctx)(vars),\\
&\qquad F_\sigma[()](a^\#, a(vars)) \Rightarrow a^\# \in \alpha_\sigma(E)\\
&\equiv \forall((a^\#, ())) \in insts(F_\sigma, a, ctx)(vars) \cap \sigma(a(vars)), a^\# \in \alpha_\sigma(E)\\
&\equiv \forall((a^\#, ())) \in \sigma(a(vars)), a^\# \in \alpha_\sigma(E) \text{ by assumption}\\
&\qquad \equiv (\forall(a^\#, ()), F_\sigma[()](a^\#, a(vars')) \Rightarrow a^\# \in \alpha_\sigma(E))\\
&\wedge [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!] \text{ with } vars' = vars
\end{aligned}$$

  As a consequence, we have:

- $\forall a, ctx, insts(F_{\sigma_{id}}, a, ctx)$ as defined in Algorithm 3 is strongly complete.
- $\forall a, ctx, insts(F_{\sigma_{bot}}, a, ctx)$ as defined in Algorithm 3 is strongly complete.
- *inductive cases*: for dot and $\circ$: see below

□

Theorem 9. *Strong completeness for $\cdot$*
$\forall \sigma_1, \sigma_2, a_1, a_2, ctx, insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx)$ as defined in Algorithm 4 is strongly complete when its recursive calls to insts, that is, $insts(F_{\sigma_1}, a_1, (ctx, a_2))$ and $insts(F_{\sigma_2}, a_2, (ctx, a_1, I_1))$ are.

Proof. For $insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx)$, the idea consists in first abstracting a_2 (respectively a_1) thus yielding a $vars_{tmp}$ and then abstracting a_1 yielding a $vars'$. The key thing is that when abstracting a_2, we must not change anything to the rest of the clause, including the instantiation sets. Now, let us do the full proof of strong completeness by working from the assumption we have in strong completeness:

$$\begin{aligned}
&\forall(a^\#, q) \in insts(F_{\sigma_1 \cdot \sigma_2}, a, ctx)(vars),\\
&\qquad F_{\sigma_1 \cdot \sigma_2}[q](a^\#, a(vars)) \Rightarrow a^\# \in \alpha_{\sigma_1 \cdot \sigma_2}(E)
\end{aligned}$$

1) Let us unpack $a = (a_1, a_2)$ and $q = q_1 @ q_2$ As we know the number of identifiers generated by σ_1, we know where to split the list q

$$\begin{aligned}
&\forall((a_1^\#, a_2^\#), q_1 @ q_2) \in insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx)(vars),\\
&\qquad F_{\sigma_1 \cdot \sigma_2}[q_1 @ q_2]((a_1^\#, a_2^\#), (a_1(vars), a_2(vars)))\\
&\qquad\qquad \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)
\end{aligned}$$

2) Using the definition of $insts(F_{\sigma_1 \cdot \sigma_2}, (a_1, a_2), ctx), I_1, I_2$

$$\begin{aligned}
&\forall(a_1^\#, q_1) \in I_1(vars), (a_2^\#, q_2) \in I_2(vars),\\
&\qquad F_{\sigma_1 \cdot \sigma_2}[q_1 @ q_2]((a_1^\#, a_2^\#), (a_1(vars), a_2(vars)))\\
&\qquad\qquad \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)
\end{aligned}$$

3) Using the definition of $F_{\sigma_1 . \sigma_2}$

$$\begin{aligned}
&\forall(a_1^\#, q_1) \in I_1(vars), (a_2^\#, q_2) \in I_2(vars), F_{\sigma_1}[q_1](a_1^\#, a_1(vars))\\
&\qquad \wedge F_{\sigma_2}[q_2](a_2^\#, a_2(vars)) \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)
\end{aligned}$$

4) Now, let us move things around

$$\forall (a_2^\#, q_2) \in I_2(vars), F_{\sigma_2}[q_2](a_2^\#, a_2(vars)) \Rightarrow \forall (a_1^\#, q_1) \in I_1(vars), F_{\sigma_1}[q_1](a_1^\#, a_1(vars)) \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)$$

5) Now, let us prepare to use strong completeness of I_2

$$\forall (a_1^\#, q_1) \in I_1(vars), F_{\sigma_1}[q_1](a_1^\#, a_1(vars)) \Rightarrow \forall (a_2^\#, q_2) \in I_2(vars), F_{\sigma_2}[q_2](a_2^\#, a_2(vars)) \Rightarrow a_2^\# \in \{x | (a_1^\#, x) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)\}$$

6) The set in question is a projection. Using the definition of $\alpha_{\sigma_1 \cdot \sigma_2}$, we have :

$$\forall (a_1^\#, q_1) \in I_1(vars), F_{\sigma_1}[q_1](a_1^\#, a_1(vars)) \Rightarrow \forall (a_2^\#, q_2) \in I_2(vars), F_{\sigma_2}[q_2](a_2^\#, a_2(vars)) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\{x | (a_1^\#, x) \in \alpha_{\sigma_1 \cdot \sigma_{id}}(E)\})$$

7) Let E_t be the latter set, yielding

$$\forall (a_1^\#, q_1) \in I_1(vars), F_{\sigma_1}[q_1](a_1^\#, a_1(vars)) \Rightarrow \forall (a_2^\#, q_2) \in I_2(vars), F_{\sigma_2}[q_2](a_2^\#, a_2(vars)) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(E_t)$$

8) Let us move the first implication to the context.

$$\forall (a_2^\#, q_2) \in I_2(vars), F_{\sigma_2}[q_2](a_2^\#, a_2(vars)) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(E_t)$$

9) Let use strong completeness of $I_2 = insts(F_{\sigma_2}, a_2, (ctx, a_1, I_1))$ with $E = E_t$
We retrieve $vars_t$ such that

$$(\forall (a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, a_2(vars_t)) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(E_t)) \wedge [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars_t)]\!]_{\mathcal{M}} \wedge [\![a_1(vars)]\!]_{\mathcal{M}} = [\![a_1(vars_t)]\!]_{\mathcal{M}} \wedge [\![I_1(vars)]\!]_{\mathcal{M}} = [\![I_1(vars_t)]\!]_{\mathcal{M}}$$

10) Let us unfold E_t using the equalities and reintroducing the implication of step 8

$$(\forall (a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, a_2(vars_t)) \Rightarrow \forall (a_1^\#, q_1) \in I_1(vars_t), F_{\sigma_1}[q_1](a_1^\#, a_1(vars_t)) \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)$$

11) We remember $[\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars_t)]\!]_{\mathcal{M}}$
Now that we have removed I_2, let us repeat steps after 4 for I_1

$$\forall (a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, a_2(vars_t)) \Rightarrow \forall (a_1^\#, q_1) \in I_1(vars_t), F_{\sigma_1}[q_1](a_1^\#, a_1(vars_t)) \Rightarrow a_1^\# \in \alpha_{\sigma_1}(\{x | (x, a_2^\#) \in \alpha_{id \cdot \sigma_2}(E)\})$$

12) Let us name that set E'_t and let use strong completeness of I_1 (moving the first implication in context)
We retrieve $vars'$ such that

$$(\forall (a_1^\#, q_1), F_{\sigma_1}[q_1](a_1^\#, a_1(vars')) \Rightarrow a_1^\# \in \alpha_{\sigma_1}(E'_t)) \wedge [\![ctx(vars_t)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}} \wedge [\![a_2(vars_t)]\!]_{\mathcal{M}} = [\![a_2(vars')]\!]_{\mathcal{M}}$$

13) Let us unfold E'_t using the equalities and context

$$\forall(a_1^\#, q_1), F_{\sigma_1}[q_1](a_1^\#, a_1(vars')) \Rightarrow \forall(a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, a_2(vars')) \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \cdot \sigma_2}(E)$$

14) Reversing the steps before 4
and using $[\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars_t)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}$ (steps 11 and 12)

$$(\forall(a^\#, q), F_{\sigma_1 \cdot \sigma_2}[q](a^\#, a(vars')) \Rightarrow a^\# \in \alpha_{\sigma_1 \cdot \sigma_2}(E)) \wedge [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}$$

15) Our desired result ! $\square$

Theorem 9 (Strong completeness for $\circ$)**.** *$\forall \sigma_1, \sigma_2, a, ctx, insts(F_{\sigma_1 \circ \sigma_2}, a, ctx)$ as defined in Algorithm 4 is strongly complete when*

- *its recursive calls to insts, that is, $I_2, \{I_{(q_0,q_2)}, (q_0, q_2) \in I_2\}$ are.*
- *σ_1, σ_2 are compatible, that is :*
  $\forall E \neq \emptyset, \alpha_{\sigma_2} \circ \gamma_{\sigma_2} \circ \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E) = \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)$

Proof. The case $E = \emptyset$ is proven by definition of string completeness. Let us continue with $E \neq \emptyset$.

$$\forall(a^\#, q) \in insts(F_{\sigma_1 \circ \sigma_2}, a, ctx)(vars), F_{\sigma_1 \circ \sigma_2}[q](a^\#, a(vars)) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

16) Unfolding definition of $insts(F_{\sigma_1 \circ \sigma_2}, a, ctx)$ (using the first note) and $F_{\sigma_1 \circ \sigma_2}$. We remind that $(a^\#, (q_0 @ q_1 @ q_2)) \in insts(F_{\sigma_1 \circ \sigma_2}, a, ctx) \equiv (q_0, q_2) \in I_2 \wedge (a^\#, q_1) \in I_{(q_0,q_2)}$

$$\forall(q_0, q_2) \in I_2(vars), \forall(a^\#, q_1) \in I_{(q_0,q_2)}(vars), F_{\sigma_2}[q_2](q_0, a(vars)) \wedge F_{\sigma_1}[q_1](a^\#, q_0) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

17) We first wish to transform the universal quantification restricted to $I_{(q_0,q_2)}$ into an unrestricted one. As there is an instance of $I_{(q_0,q_2)}$ for each $(q_0, q_2) \in I_2$, we iteratively handle each instance. At each stage of the iteration we have the set of instances that have already been handled I_{tmp}, and the set that is still to handle $I_2 - I_{tmp}$. This yields the following property ***Handled(I_{tmp})***.
$\exists vars'$,

$$\forall(q_0, q_2) \in I_{tmp}(vars'), \forall(a^\#, q_1), F_{\sigma_2}[q_2](q_0, a(vars')) \wedge F_{\sigma_1}[q_1](a^\#, q_0) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\wedge \forall(q'_0, q'_2) \in (I_2 - I_{tmp})(vars'), \forall(a'^\#, q'_1) \in I_{(q'_0,q'_2)}(vars'), F_{\sigma_2}[q'_2](q'_0, a(vars')) \wedge F_{\sigma_1}[q'_1](a'^\#, q'_0) \Rightarrow a'^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\wedge [\![ctx(vars')]\!]_{\mathcal{M}} = [\![ctx(vars)]\!]_{\mathcal{M}}$$

Note that the property is divided in three parts :

- The handled part, where the quantifiers are unrestricted.

- The not yet handled part, where the quantifiers are restricted
- The rest

From Step 16, we have :

$Handled(\emptyset)$

18) The proof that $Handled(\emptyset) \Rightarrow Handled(I_2)$ is done in step 25. Assuming it, we have

$Handled(I_2)$

19) Unfolding the definition of $Handled(I_2)$, thus retrieving a $\exists vars'$ and introducing it as $vars_t$, we get :

$$\forall (q_0, q_2) \in I_2(vars_t), \forall (a^\#, q_1),$$
$$F_{\sigma_2}[q_2](q_0, a(vars_t)) \wedge F_{\sigma_1}[q_1](a^\#, q_0)$$
$$\Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$
$$\wedge [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars_t)]\!]_{\mathcal{M}}$$

20) Let us prepare to apply strong completeness of I_2 and let us remember $[\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars_t)]\!]_{\mathcal{M}}$

$$\forall (q_0, q_2) \in I_2(vars_t), F_{\sigma_2}[q_2](q_0, a(vars_t))$$
$$\Rightarrow q_0 \in \{x | \forall (a^\#, q_1), F_{\sigma_1}[q_1](a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)\}$$

21) Let us name the set
$\{x | \forall (a^\#, q_1), F_{\sigma_1}[q_1](a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)\}$ as E_{I_2}. To apply strong completeness of $I_2 = insts(F_{\sigma_2}, a, ctx)$, we need $\exists E, E_{I_2} = \alpha_{\sigma_2}(E)$. We show that $E_{I_2} = \alpha_{\sigma_2} \circ \gamma_{\sigma_2}(E_{I_2})$ and thus, our need is verified for $E = \gamma_{\sigma_2}(E_{I_2})$.
$E_{I_2} = \{x | \forall (a^\#, q_1), F_{\sigma_1}[q_1](a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)\}$
$= \{x | \forall a^\# \in \sigma_1(x), a^\# \in \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)\}$
$= \{x | \sigma_1(x) \subseteq \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)\}$
$= \bigcup \{z | \alpha_{\sigma_1}(z) \subseteq \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)\}$
$= \bigcup \{z | z \subseteq \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)\}$
$= \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)$
Therefore, $\alpha_{\sigma_2} \circ \gamma_{\sigma_2}(E_{I_2})$
$= \alpha_{\sigma_2} \circ \gamma_{\sigma_2} \circ \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)$
$= \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{\sigma_2}(E)$ by compatibility assumption
$= E_{I_2}$
Let us continue our main proof and let us replace the E_{I_2} by $\alpha_{\sigma_2} \circ \gamma_{\sigma_2}(E_{I_2})$

$$\forall (q_0, q_2) \in I_2(vars_t), F_{\sigma_2}[q_2](q_0, a(vars_t))$$
$$\Rightarrow q_0 \in \alpha_{\sigma_2} \circ \gamma_{\sigma_2}(E_{I_2})$$

22) Let use strong completeness of $I_2 = insts(F_{\sigma_2}, a, ctx)$ with $E = \gamma_{\sigma_2}(E_{I_2})$. We retrieve $vars'$ such that (and reusing $E_{I_2} = \alpha_{\sigma_2} \circ \gamma_{\sigma_2}(E_{I_2})$ and unfolding the definition of E_{I_2}

$$\forall (q_0, q_2), F_{\sigma_2}[q_2](q_0, a(vars'))$$
$$\forall (a^\#, q_1), F_{\sigma_1}[q_1](a^\#, q_0) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$
$$\wedge [\![ctx(vars')]\!]_{\mathcal{M}} = [\![ctx(vars_t)]\!]_{\mathcal{M}}$$

23) Applying the definition of $F_{\sigma_1 \circ \sigma_2}$ in the reverse way as in step 16

$$\forall (a^\#, q),$$
$$F_{\sigma_1 \circ \sigma_2}[q](a^\#, a(vars')) \Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$
$$\wedge [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}$$

24) Our desired result !

25) Let us show $Handled(\emptyset) \Rightarrow Handled(I_2)$ where the property $Handled(I_{tmp})$ is defined in step 17. To do so, we prove that we have $Handled(I)$ for any set I reached by the variable I_{tmp} of Algorithm 4 and prove that $Handled(I) \Rightarrow Handled(I \cup \{(q_0, q_2)\})$ where I and $I \cup \{(q_0, q_2)\}$ are values reached by I_{tmp} in the while loop. It follows by transitivity and finiteness of I_2 that $Handled(\emptyset) \Rightarrow Handled(I_2)$.
The assumption $Handled(I)$ after introducing $vars'$ yields :

$$\forall (q_{0d}, q_{2d}) \in I(vars'), \forall (a_d^{\#}, q_{1d}), \quad F_{\sigma_2}[q_{2d}](q_{0d}, a(vars')) \land F_{\sigma_1}[q_{1d}](a_d^{\#}, q_{0d}) \Rightarrow a_d^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\land \forall (a^{\#}, q_1) \in I_{(q_0,q_2)}(vars'), \quad F_{\sigma_2}[q_2(vars')](q_0(vars'), a(vars')) \land F_{\sigma_1}[q_1](a^{\#}, q_0(vars')) \Rightarrow a^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\land \forall (q_0', q_2') \in (I_2 - I - \{(q_0, q_2)\})(vars'), \quad \forall (a'^{\#}, q_1') \in I_{(q_0',q_2')}(vars'), \quad F_{\sigma_2}[q_2'](q_0', a(vars')) \land F_{\sigma_1}[q_1'](a'^{\#}, q_0') \Rightarrow a'^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\land \quad [\![ctx(vars')]\!]_{\mathcal{M}} = [\![ctx(vars)]\!]_{\mathcal{M}}$$

26) Our goal is to "move" the part in gray from within the part in red into the part in blue. Let us work on the part in gray for now.

$$\forall (a^{\#}, q_1) \in I_{(q_0,q_2)}(vars'), \quad F_{\sigma_2}[q_2(vars')](q_0(vars'), a(vars')) \land F_{\sigma_1}[q_1](a^{\#}, q_0(vars')) \Rightarrow a^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

27) Let us prepare to apply strong completeness of $I_{(q_0,q_2)}$ which is equal to $insts(F_{\sigma_1}, q_0, (I_2 - \{(q_0, q_2)\}, I_f, F_{\sigma_2}[q_2](q_0, a), ctx))$

$$F_{\sigma_2}[q_2(vars')](q_0(vars'), a(vars')) \Rightarrow \forall (a^{\#}, q_1) \in I_{(q_0,q_2)}(vars'), F_{\sigma_1}[q_1](a^{\#}, q_0(vars')) \Rightarrow a^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

28) Let us put $F_{\sigma_2}[q_2(vars')](q_0(vars'), a(vars'))$ into context

$$\forall (a^{\#}, q_1) \in I_{(q_0,q_2)}(vars'), F_{\sigma_1}[q_1](a^{\#}, q_0(vars')) \Rightarrow a^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

29)Let use strong completeness of $I_{(q_0,q_2)}$ with $E = \alpha_{\sigma_2}(E)$. We remind that $I_{(q_0,q_2)} = insts(F_{\sigma_1}, q_0, (I_2 - \{(q_0, q_2)\}, I_f, F_{\sigma_2}[q_2](q_0, a), ctx))$ with $I_f = (I_2 - I - \{(q_0, q_2)\})$. We retrieve $vars_t'$ such that

$$\forall (a^{\#}, q_1), F_{\sigma_1}[q_1](a^{\#}, q_0(vars_t')) \Rightarrow a^{\#} \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\land (I_2 - \{(q_0, q_2)\})(vars') = (I_2 - \{(q_0, q_2)\})(vars_t')$$

$$\land \forall (q_0', q_2') \in (I_2 - I - \{(q_0, q_2)\})(vars'), \quad I_{(q_0',q_2')}(vars') = I_{(q_0',q_2')}(vars_t')$$

$$\land F_{\sigma_2}[q_2(vars')](q_0(vars'), a(vars')) = F_{\sigma_2}[q_2(vars_t')](q_0(vars_t'), a(vars_t'))$$

$$\land [\![ctx(vars')]\!]_{\mathcal{M}} = [\![ctx(vars_t')]\!]_{\mathcal{M}}$$

30) Let us use the equalities and set ourselves in the context of steps 26 and 28

$$\forall (q_{0d}, q_{2d}) \in I(vars'_t), \forall (a^\#_d, q_{1d}),$$
$$F_{\sigma_2}[q_{2d}](q_{0d}, a(vars'_t)) \wedge F_{\sigma_1}[q_{1d}](a^\#_d, q_{0d})$$
$$\Rightarrow a^\#_d \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\wedge \forall (a^\#, q_1),$$
$$F_{\sigma_2}[q_2(vars'_t)](q_0(vars'_t), a(vars'_t))$$
$$\wedge F_{\sigma_1}[q_1](a^\#, q_0(vars'_t))$$
$$\Rightarrow a^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\wedge \forall (q'_0, q'_2) \in (I_2 - I - \{(q_0, q_2)\})(vars'_t),$$
$$\forall (a'^\#, q'_1) \in I_{(q'_0, q'_2)}(vars'_t),$$
$$F_{\sigma_2}[q'_2](q'_0, a(vars'_t)) \wedge F_{\sigma_1}[q'_1](a'^\#, q'_0)$$
$$\Rightarrow a'^\# \in \alpha_{\sigma_1 \circ \sigma_2}(E)$$

$$\wedge \quad [\![ctx(vars'_t)]\!]_{\mathcal{M}} = [\![ctx(vars)]\!]_{\mathcal{M}}$$

31) $Handled(I \cup \{(q_0, q_2)\})$, our desired result ! □

A.3 Cell abstraction results

Theorem 10 (Abstraction compatibility for cell abstractions)**.** *The abstractions of Example 9 have strongly complete instantiation heuristics when $Cell_n$ has.*

Proof. As the abstractions are $Cell_n$ are finite abstractions, the only consideration is compatibility of the abstractions (condition for $\circ$ to have strongly complete instantiation). Let us prove the result for $Cell_1$ and consider the composed abstraction σ_1. We need to prove :$\forall E \neq \emptyset, \alpha_{Cell_1} \circ \gamma_{Cell_1} \circ \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{Cell_1}(E) = \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{Cell_1}(E)$.

Introduce $E \neq \emptyset$ and define $X = \gamma_{\sigma_1} \circ \alpha_{\sigma_1} \circ \alpha_{Cell_1}(E)$. Note that $\exists a, \forall i, (i, a[i]) \in X$ because $E \neq \emptyset$, thus, $\exists a \in E$, thus, $\exists a, \forall i, (i, a[i]) \in \alpha_{Cell_1}(E)$, thus, $\exists a, \forall i, (i, a[i]) \in X$ as $\forall S, S \subseteq \gamma_{\sigma_1} \circ \alpha_{\sigma_1}(S)$.

We now prove $\alpha_{Cell_1} \circ \gamma_{Cell_1}(X) = X$.

$\alpha_{Cell_1} \circ \gamma_{Cell_1}(X) = \{(i, a[i]) | a \in \gamma_{Cell_1}(X)\}$
$= \{(i, a[i]) | a \in \{a | \sigma_{Cell_1}(a) \subseteq X\}\}$
$= \{(i, v) | \exists a \in \{a | \sigma(a) \subseteq X\} \wedge v = a[i]\}$
$= \{(i, v) | \exists a, \sigma_{Cell_1}(a) \subseteq X \wedge v = a[i]\}$
$= \{(i, v) | \exists a, \forall i, (i, a[i]) \in X \wedge v = a[i]\}$
$= \{(i, v) | true \wedge (i, v) \in X\}$ as proved above
$= X$

□

Theorem 11 (Correctness of $relevant(a, expr)$)**.** *If* $\top \notin relevant(a, expr)$ *then*

$$\forall \mathcal{M}, vars, a', (\forall i \in relevant(a, expr), a'[i] = a(vars)[i]) \Rightarrow$$
$$\exists vars', [\![expr(vars)]\!]_{\mathcal{M}} = [\![expr(vars')]\!]_{\mathcal{M}} \wedge a(vars') = a'$$

Remark 11. *Should the relevant algorithm be expanded, one only needs to keep the property of this theorem to have strong completeness of $Cell_n$.*

Proof. Let us first prove a property about $read$: if $read(avar, expr)$ does not contain $\top$, then

$$\forall \mathcal{M}, a, a', (\forall i \in read(avar, expr), a[i] = a'[i]) \Rightarrow \\ \forall vars, [\![expr(vars[avar \leftarrow a]]\!]_{\mathcal{M}} = [\![expr(vars[avar \leftarrow a']]\!]_{\mathcal{M}}$$

This proof is done by induction on $expr$. Now, let us introduce $\mathcal{M}, vars, a'$ and let us assume $\forall i \in$

$relevant(a, expr), a'[i] = a(vars)[i]$
Because $\top \notin relevant(a, expr)$, introduce $avar$ such that $a = arrayStoreChain(avar, I, V)$.
We can use our property on $read$ with $\mathcal{M} = \mathcal{M}, a = a(vars), a' = a', vars = vars$ as :
$\forall i \in read(avar, expr), a(vars)[i] = a'[i]$. (In fact $read(avar, expr) \subseteq relevant(a, expr)$ and $\forall i \in relevant(a, expr), a'[i] = a(vars)[i]$). Therefore, we have :
$[\![expr(vars[avar \leftarrow a(vars)])]\!]_{\mathcal{M}} = [\![expr(vars[avar \leftarrow a'])]\!]_{\mathcal{M}}$
$\equiv [\![expr(vars)]\!]_{\mathcal{M}} = [\![expr(vars[avar \leftarrow a'])]\!]_{\mathcal{M}}$
$\equiv [\![expr(vars)]\!]_{\mathcal{M}} = [\![expr(vars')]\!]_{\mathcal{M}}$
with $vars' = vars[avar \leftarrow a']$, which verifies $a(vars') = a'$ □

Theorem 12 (Strong Completeness for cell Abstraction)**.** *Any call to $insts(F_{\sigma_{Cell_n}}, a, ctx)$ is strongly complete whenever $\top \notin relevant(a, ctx)$.*

Proof. We shall refer to Ind, I etc. as defined in Algorithm. 6. For simplicity, let us write everything as though
$insts(F_{\sigma_{Cell_n}}, a, ctx)$ returned I^n instead of $(I^n, ())$. Introduce $vars, E, \mathcal{M}$ and assume

$$\forall a^{\#} \in insts(F_{\sigma_{Cell_n}}, a, ctx)(vars), F_{\sigma_{Cell_n}}(a^{\#}, a(vars)) \Rightarrow a^{\#} \in \alpha_{\sigma_{Cell_n}}(E)$$

which is equivalent to

$$insts(F_{\sigma_{Cell_n}}, a, ctx)(vars) \cap \sigma_{Cell_n}(a(vars)) \subseteq \alpha_{\sigma_{Cell_n}}(E)$$

which is equivalent to (as $Cell_n$ is just n replication of $Cell_1$)

$$insts(F_{\sigma_{Cell_1}}, a, ctx)(vars) \cap \sigma_{Cell_1}(a(vars)) \subseteq \alpha_{\sigma_{Cell_1}}(E) \tag{15}$$

We need to show (using the same simplifications) :

$$\exists vars', [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}} \wedge \sigma_{Cell_1}(a(vars')) \subseteq \alpha_{\sigma_{Cell_1}}(E)$$

1. Case $E = \emptyset$:
   Let us prove that the assumption 15 is not verified and thus by absurdum, our result stands.
   To prove the assumption if not verified, we use :
   (a) $\alpha_{\sigma_{Cell_1}}(E) = \alpha_{\sigma_{Cell_1}}(\emptyset) = \emptyset$
   (b) $insts(F_{\sigma_{Cell_1}}, a, ctx)(vars) \cap \sigma_{Cell_1}(a(vars)) \neq \emptyset$ As $insts(F_{\sigma_{Cell_1}}, a, ctx)(vars) \neq \emptyset$ by construction of Ind and $insts(F_{\sigma_{Cell_1}}, a, ctx)(vars) \subseteq \sigma_{Cell_1}(a(vars))$ by construction of I (we chose $(a, a[i])$).
2. Continuing with $E \neq \emptyset$
   Let $a_\top \in E$
3. Let a' such that
   $\forall i, a'[i] = ite(i \in relevant(a, ctx), a(vars)[i], a_\top[i])$
4. By assumption $\top \notin relevant(a, ctx)$ and by construction $\forall i \in relevant(a, ctx), a'[i] = a(vars)[i]$, therefore
   $\exists vars', [\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}} \wedge a(vars') = a_\top$ using Theorem 11

5. We already have $[\![ctx(vars)]\!]_{\mathcal{M}} = [\![ctx(vars')]\!]_{\mathcal{M}}$. Thus, we only need
$$\sigma_{Cell_1}(a(vars')) \subseteq \alpha_{\sigma_{Cell_1}}(E)$$
Which is equivalent to
$$\{(i, a'[i])\} \subseteq \alpha_{\sigma_{Cell_1}}(E)$$
Which is verified as
   (a) $\{(i, a'[i]), i \in relevant(a, ctx)\} = \{(i, a(vars)[i]), i \in relevant(a, ctx)\} \subseteq insts(F_{\sigma_{Cell_1}}, a, ctx)(vars)$ which by assumption is included in $\alpha_{\sigma_{Cell_1}}(E)$
   (b) $\{(i, a'[i]), i \notin relevant(a, ctx)\}$
   $= \{(i, a_\top[i]), i \notin relevant(a, ctx)\}$
   $\subseteq \sigma_{Cell_1}(a_\top) \subseteq \alpha_{Cell_1}(\{a_\top\}) \subseteq \alpha_{Cell_1}(\{E\})$
   as $a_\top \in E$.

□

Theorem 14 (Satisfiability preservation of trs)**.** *If the input clauses are normalized Horn clauses such that there is at most one negative predicate in each clause (corresponds to program without function calls) and the theory on which arrays are used contains read, write but no array equalities $a = a'$ and there are no arrays whose index or value type is an array and no array variable appears twice within the same predicate then:*

$$trs(\mathfrak{C}, n) \text{ satisfiable } \equiv \exists \mathcal{M} \text{ expressible by } \mathcal{G}, H_C(\mathcal{M})$$

Where $\mathcal{G}$ is the abstraction used, that is, $\mathcal{G}^{P_1}_{\sigma^{P_1}} \circ \ldots \circ \mathcal{G}^{P_m}_{\sigma^{P_m}}$ with $P_1, \ldots, P_m$ are the predicates of $\mathfrak{C}$ and σ^{P_i} is as defined in Algorithm 7 and $\mathcal{G}^P_\sigma$ is as defined in Definition 8.

Proof. First, note that $\mathcal{G}$ does not use “data abstraction composition combinator”, and can be defined by $\alpha_{\mathcal{G}}(\mathcal{M})(P_i) = \alpha_{\sigma^{P_i}}(\mathcal{M}(P_i))$. As all our other combinators are strongly complete when their parameters are, we only need to prove that all calls to $insts(Cell_n, a, ctx)$ are strongly complete, which is equivalent to $\top \notin relevant(a, ctx)$.

We thus need to examine the cases for $relevant(a, ctx)$ as defined in Algorithm 5:

1. An array store chain of a (named $asc(a)$) is only used in a read. This is equivalent to stating that $asc(a)$ is not used in (i) array equalities (ii) nor within predicates of the context, (iii) nor in a tuple constructor. (i) is true since initial clauses do not contain array equalities initially and we do not introduce any. As all arrays of all predicates are abstracted, predicates do not contain array typed parameters (ii). Only the dot combinator introduce tuples constructors, and no array variable appears twice within the same predicate (iii).
2. ctx does not contain quantifiers. Initially, there are no quantifiers, and the quantifiers introduced by the abstractions are either positive predicates, and thus are transformed into free variables; or a unique negative predicate which is instanciated just afterwards. Thus, there are no quantifiers in the context.

□

Contents

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399