

Numéro National de Thèse: 2022LYSEN014

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée par
l'École Normale Supérieure de Lyon

École Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 11 mai 2022, par:

Julien BRAINE

The Data-abstraction Framework: abstracting unbounded data-structures in Horn clauses, the case of arrays

La Méthode Data-abstraction : une technique d'abstraction de structures de données non-bornées dans des clauses de Horn, le cas des tableaux

---

Devant le jury composé de :

SIGHIREANU Mihaela, Professeure des universités, ENS Paris Saclay	Rapportrice
MINÉ Antoine, Professeur des universités, Sorbonne Université	Rapporteur
BODIN Martin, Chargé de recherche, Inria Grenoble	Examinateur
FERET Jérôme, Chargé de recherche, ENS Paris	Examinateur
POUS Damien, Directeur de recherche, École Normale Supérieure de Lyon	Examinateur
GONNORD Laure, Professeure des universités, Grenoble INP	Directrice de thèse
MONNIAUX David, Directeur de recherche, CNRS, Vérimag, Grenoble	Co-encadrant de thèse

Remerciements

Je souhaite remercier ma directrice de thèse Laure Gonnord pour son accompagnement, ses conseils et sa bienveillance. Je voudrais également remercier mon co-encadrant David Monniaux pour nos échanges, ses conseils et sa relecture technique de ce manuscrit. Je tiens tout particulièrement à remercier Russ Harmer pour son implication, sa disponibilité et la qualité de nos échanges durant l'écriture de ce manuscrit.
Un grand merci à mes collègues Paul, Alexis et Amaury pour leur aide, les moments de partage et nos grandes discussions. Merci à ma famille et mes amis pour leur soutien durant ces quatre années, dont deux de pandémie.
J'aimerais remercier mes rapporteurs Mihaela Sighireanu et Antoine Miné qui ont pris le temps de lire avec soin ce manuscrit. Enfin, merci aux membres de l'équipe CASH et au personnel du LIP pour nos conversations et l'aide apportée dans le labyrinthe administratif.

Abstract

Proving properties of programs using data-structures such as arrays often requires universally quantified invariants, e.g., "all elements below index i are nonzero". Instead of directly manipulating programs, we use Horn formulas which have recently become a popular format to express safety properties of programs.

In this manuscript, we propose a general abstraction scheme operating on Horn formulas for unbounded data-structures. The main idea is to use abstraction to simplify the unbounded data-structures into simpler types such as integers. As such a simplification loses information, not all safety properties can be proven after abstraction. It is thus key to choose the right abstraction for the given problem.

Our contribution is a general framework for which the data-structures and the abstraction to apply are parameterized. The specificity of that framework is the attention spent to ensure that our algorithm exactly implements the simplification induced by the abstraction by using a property called *relative completeness*. This contrasts with previous work that mainly uses benchmarks to prove the validity of the technique instead of analyzing whether the technique verifies a similar property.

Although the proposed framework is general and may theoretically handle data-structures such as trees or graphs, our focus has been on a specific abstraction of arrays that has already been studied in the literature called *cell abstraction*. This abstraction is of particular interest as it can handle most container algorithms.

There are three main contributions in this manuscript. First, a demonstration that our framework in combination with cell-abstraction can express many existing techniques, and thus our framework handles those as well. Secondly, a proof that an existing restricted technique actually satisfies *relative completeness* and an extension of the technique to handle a broader class of programs. Thirdly, the framework gives definitions, algorithms and theorems parameterized by the desired abstraction and, even though we have only used this framework for arrays, it gives the foundation for future abstraction of other data-structures.

Résumé

Prouver des propriétés sur des programmes utilisant des structures de données telles que des tableaux nécessite souvent des invariants universellement quantifiés, par exemple, "tous les éléments avec indice plus petit que i sont non nuls". Au lieu de manipuler directement les programmes, nous utilisons des clauses de Horn, un format de plus en plus courant pour exprimer les propriétés de sécurité des programmes.

Dans ce manuscrit, nous proposons un schéma d'abstraction général opérant sur des formules de Horn pour des structures de données non bornées. L'idée principale est d'utiliser l'abstraction pour simplifier les structures de données en types plus simples tels que les entiers. Comme une telle simplification perd de l'information, toutes les propriétés ne peuvent pas être prouvées après abstraction. Il est donc essentiel de choisir la bonne abstraction pour le problème donné.

Notre contribution est d'avoir créé une méthode générale où les structures de données et l'abstraction à appliquer sont paramétrées. La spécificité de cette méthode est l'attention portée à une propriété que nous nommons *complétude relative* et qui spécifie que l'algorithme implémente exactement la simplification induite par l'abstraction. Cette approche contraste avec les travaux antérieurs qui utilisent principalement des benchmarks pour prouver la validité de la technique au lieu d'analyser si la technique vérifie une propriété similaire.

Bien que le cadre proposé soit général et puisse théoriquement gérer des structures de données telles que des arbres et graphes, nous nous sommes concentrés sur une abstraction spécifique de tableaux, déjà étudiée dans la littérature et appelée *abstraction de cellule*. Cette abstraction est particulièrement intéressante car elle peut gérer la plupart des algorithmes de conteneur.

La contribution de ce manuscrit peut être résumée en trois points. Tout d'abord, nous démontrons que notre méthode, en combinaison avec l'abstraction de cellule, permet d'exprimer, et donc de gérer, de nombreuses techniques existantes. Deuxièmement, nous prouvons qu'une de ces techniques existantes satisfait le propriété de complétude relative et une extension de celle-ci pour gérer une classe plus large de programmes. Troisièmement, même si nous nous sommes focalisés sur les tableaux, les définitions, algorithmes et théorèmes que nous fournissons sont paramétrés avec l'abstraction souhaitée et donne les bases pour de futures abstractions d'autres structures de données.

Contents

1 Introduction **5**
- 1.1 General concerns . . . 5
  - 1.1.1 Purpose to which the PhD contributes . . . 5
  - 1.1.2 Scope of the PhD . . . 8
  - 1.1.3 Philosophy of the PhD . . . 10
- 1.2 Methodology : a problem verification transformation technique . . . 10
  - 1.2.1 Choosing an adequate intermediate representation: Horn clauses . . . 11
  - 1.2.2 Completeness, a key property for theoretical guarantees . . . 12
  - 1.2.3 A Horn clause transformation framework: Data-Abstraction . . . 13
- 1.3 Overview of the contribution and manuscript . . . 13
  - 1.3.1 Overview of the proposed verification scheme . . . 13
  - 1.3.2 Structure of the manuscript . . . 13

Notation Sheet **14**

2 From programs to Horn clauses **17**
- 2.1 Transforming programs into logical constraints . . . 17
  - 2.1.1 The transformation of a simple program: array initialization . . . 17
  - 2.1.2 Verifying safety properties of program points: assertions . . . 19
  - 2.1.3 Verifying and using functions . . . 21
    - 2.1.3.1 Functions as relations between input and output values . . . 21
    - 2.1.3.2 Verifying functions . . . 22
    - 2.1.3.3 Handling function calls . . . 23
  - 2.1.4 Using a theory adapted to logical formulae . . . 26
  - 2.1.5 Large block encoding . . . 27
- 2.2 Horn clauses and Horn problems . . . 28
  - 2.2.1 Syntax: expressions, evaluation contexts and Horn clauses . . . 28
  - 2.2.2 Horn problems . . . 30
    - 2.2.2.1 Defining Horn problems . . . 30
    - 2.2.2.2 Horn clauses encode Horn problems . . . 31
- 2.3 A complex example: merge sort . . . 32

3 Data-Abstraction: expectations **37**
- 3.1 Algorithm based on abstraction: definition & properties . . . 37
  - 3.1.1 Abstraction: definition . . . 37
  - 3.1.2 Solving algorithms: definition, *soundness* and *completeness* . . . 40
  - 3.1.3 Solving algorithms based on abstraction: *relative completeness* . . . 41

3.1.4 Working around undecidability: transformations *implementing the abstraction* . . . 44
3.2 Data-abstraction: a specific subset of abstractions . . . 45
3.3 Cell abstraction: an interesting Data-abstraction for arrays . . . 48
3.3.1 Properties of a good array abstraction . . . 48
3.3.1.1 General properties of a good data-abstraction . . . 48
3.3.1.2 The class of Horn problems a good abstraction should handle . . . 49
3.3.2 Cell-abstraction: possibly the right abstraction . . . 50
3.3.2.1 Cell-abstraction: definition . . . 50
3.3.2.2 Cell-abstraction: extending expressibility with combinators . . . 51
3.3.2.3 Cell abstraction: evaluation . . . 53
Overview and contribution within this chapter . . . 54

4 Data-Abstraction: transformation algorithm 57
4.1 The $abstract$ algorithm . . . 57
4.1.1 Defining the abstracted Horn problem $\mathscr{G}(H)$ . . . 58
4.1.2 $\mathscr{G}(H)$ as an algorithm . . . 59
4.2 The $eliminate$ algorithm . . . 62
4.3 Formalizing *the data-abstraction framework algorithm implements the abstraction* as a condition on calls to $insts$ . . . 67
4.3.1 *Completeness* of $eliminate$, seemingly impossible . . . 68
4.3.2 *Completeness* of $eliminate$ is possible . . . 69
4.3.3 *Completeness of a call* to $insts$ . . . 72
4.3.4 Discussion of a few subtle choices . . . 74
Overview and contribution within this chapter . . . 77

5 Data-Abstraction: instantiation heuristics 79
5.1 Instantiating Cell abstraction . . . 80
5.1.1 Key concepts to construct $insts_{Cell}(a,ctx)$ . . . 80
5.1.2 Instantiation heuristics based on literature . . . 81
5.1.2.1 The transformations from programs of [MA15; MG16] . . . 81
5.1.2.2 The array formulae decision procedure of [BMS06] . . . 89
5.1.2.3 The instantiation of [BMR13], Quantifiers on demand [GSV18] . . . 92
Conclusion of the litterature overview . . . 94
5.1.3 Our instantiation heuristic for cell abstraction . . . 94
5.1.3.1 Property the instantiation set must verify: *relevant Cells*. . . . 95
5.1.3.2 The $relevant$ and $insts_P^{abs}(expr,ctx)$ algorithms . . . 100
5.2 Instantiation for the foundation of the data-abstraction framework . . . 104
5.2.1 Instantiating σ_{id} and other finite abstractions . . . 106
5.2.2 Instantiating $\sigma_1 \bullet \sigma_2$ . . . 106
5.2.3 Instantiating the $\odot$ combinator . . . 109
5.2.4 Instantiating the $\otimes$ combinator . . . 113
5.2.5 Instantiating the $\oplus$ combinator . . . 114
Overview and contribution within this chapter . . . 114

6 Theoretical contribution: impact on program verification 117
6.1 Classifying the current handling of Horn clauses . . . 118
6.2 Classifying the current handling of program instructions . . . 119

6.3 Extending the current verification scheme . . . 121
6.3.1 Improving the *relevant* algorithm . . . 122
6.3.1.1 Improvements targeting cell abstraction . . . 123
6.3.1.2 Improvements targeting other abstractions . . . 126
6.3.1.3 Discussion of the extensions to the *relevant* algorithm . . . 127
6.3.2 Modifying the verification scheme . . . 128
6.3.2.1 Passing additional information to the context . . . 128
6.3.2.2 Pre-transforming the Horn clauses . . . 129
6.4 Summary of the impact on the verification of programs . . . 129

7 Data-Abstraction: experiments **131**
7.1 Toolchain . . . 131
7.2 Experimental results . . . 133
7.3 Improving the results . . . 136

8 Conclusion **139**

Bibliography **141**

1 Introduction

1.1 General concerns

1.1.1 Purpose to which the PhD contributes

Have you ever encountered a program without any bugs? Probably not. There are several reasons that can explain why programs are written with bugs: lack of programmer concentration, takeover of old code by a new programmer who may not know the specifics, subtle behavior of programming languages, ... However, not all bugs are equal: while most of them only impact the usability of the software, others can have much larger negative impacts: imagine an airplane software segfaulting or the guiding system of a missile failing. Historically, bugs were considered critical only in very specific locations: airplanes, submarines, banking servers, military devices, ... However, more recently, with the rise of data-analysis and thus data-storage, more and more servers can be viewed as critical: a bug leading to an attack within a Google server might lead to huge amounts of data being gathered, with possibilities like large-scale identity theft, attacks on other servers by using the data gathered, ransomware, ... Even worse, with the current number of connected devices, even bugs that enable attacks in phones or computers of individuals can be considered critical: apps are deployed on such a scale that targeting individuals through their browser, operating system, or app can lead to disastrous consequences.

Although techniques to ensure protection of devices have increased – sandboxes, containers, virtual machines, higher control over scripts that are executed within apps, ... – programs are also more and more complex and ensuring that there is no critical bug, especially in these protection tools, is very important. Whereas for non-critical bugs, aiming to remove as many bugs as possible from the software before release, and then, as bugs appear, correct them during updates is a sensible approach; for these critical bugs, this approach needs to be changed: the goal is not to remove as many bugs as possible before release, but to ensure that there is no critical bug.

For most programs, most of the debugging, such as unit testing, is done through testing execution paths of the program and checking that the program behaves correctly on those examples; however, in the case of critical bugs, one usually uses static verification to ensure that the program behaves correctly. Static verification of a program consists in stating a behavior that the program should have, that is, a mathematical property, and then, using formal semantics of code, that is, a mathematical description of how code behaves, provide a proof that the behavior of the code respects the behavior that we stated the program should have. This ensures that for any execution path, even those that we have not tested, the program behaves correctly, or at least as we defined it should. In order to apply static verification of programs, one needs three components: *a formal*

definition of code semantics, *a specification of correct behavior of code*, and *a proof that the code obeys the specifications*.

Formal definition of code semantics. This is non-trivial work and the formalization of code has been tackled for several languages in different proof frameworks. The difficulties lie in several parts: the first issue is correctly understanding the semantics of code according to a definition which is written in English in many many pages. For example, the *standard*, that is, the description of the behavior of code, for C++ is around 1500 pages [ISO17]! The second issue is that many languages have constructions that are hard to formalize, such as exceptions, dynamic typing, memory layout, binary compatibility, ... Finally, and this is the hardest part, which explains why the standard is so long, is the very subtle interaction between language constructions, for example the interaction between polymorphism and references in Ocaml[1] [Ler+21], but such examples can be found in almost all programming languages. Just to give an idea of how difficult it is: most compilers have bugs initially from not handling correctly the standard (and not only due to optimization issues). The writing of a formally certified C compiler [Ler20] (which is a fairly simple language), which is not at all as optimized as current state-of-the-art compilers was a huge project undertaken only recently!

In this PhD, I do not aim to provide formal semantics of code. Instead, I will be using a transformation from code to a logical formula which we will assume correct and through that transformation, the semantics of code is given. In practice, for a fully-verified system, one should adapt that transformation for the chosen programming language and prove the correctness of that transformation using the formal semantics of code.

Specification of correct behavior of code. In practice, asking programmers to write a formal specification of *there are no critical bugs in the program* is extremely hard. Of course, some bugs, such as illegal memory accesses, division by zero, integer overflow ... are fairly easy to specify, and one may even have automated tools add such specifications; other bugs, such as *eventually my thread will be executed* or *my app can recover from loss of connection or from hardware failure* are much easier to overlook. Even for very small functions, it is hard to perfectly describe the intended behavior of the code! For example, the correct (functional) summary of a function that sorts a container is that the output is sorted and is a permutation of the input; however, one often forgets to state the latter and the function is thus under-specified.

In practice, code is highly layered: from built-in functions to general libraries, to application-specific libraries and finally the application itself which links all these libraries together. Because writing complete specifications of the application is extremely hard, one has better chances of writing complete specifications by also writing the specifications of each function on which the application and libraries are built. Furthermore, by doing so, proofs of each function are decoupled: to prove that function f which uses function g is correct, one only needs to check that g verifies its specifications and that f is correct using the specifications of g instead of the code of g. The consequence of such an approach is that static program verification schemes can scale better, that is, handle much larger code at lower cost.

Another important part of writing specifications is the formal language in which we write them. Some of the most expressive languages for properties about programs are temporal logics [RU71].

[1] Naive polymorphism and references lead to a breach within the type system. Thus, when references are used, weak polymorphism is introduced. A thorough example can be found on https://ocamlverse.github.io/content/weak_type_variables.html

An important subset of temporal logics can be divided into safety properties, that is, which program states are valid, and liveness properties, that is, properties that ensure a program eventually/repeatedly goes through a given state. To stress how important both properties are, let us give an example of each. Safety properties are similar to assertions in programs; the aim is to restrict the set of values that are valid at a given program point. For example, one may need to $assert(n < size(a))$ before the expression $a[n]$, or $assert(sorted(a))$ before doing a binary search. Unlike safety properties, liveness properties are useful to ensure that the program will react. For example, one may want to say that whenever the pilot of an airplane moves the control joystick, the program will eventually (one may also wish to ensure time constraints) move the wings of the airplane. In other words, to guarantee that the program will not get stuck doing something else or continuously have new things to do and forever procrastinate the task of moving the wings.

In this PhD, I focused on safety properties because I believe safety properties constitute the core of library function specifications. An example of such a safety property in a very simple array initialization function is given in Listing 1.1.

Listing 1.1 – Array initialization

```
void array_init(Array<int> a)
{
  unsigned i=0;
  while(i<a.size())
  {
    /* loop invariant to discover:
       ∀k∈[0,i[,a[k]=0∧i<a.size() */
   /* Note that ∀k∈[0,i[,a[k]=0 is the core,
      but i<a.size() is also necessary */
    a[i] <- 0;
    i <- i+1;
  }
  //Safety property
  assert(∀k∈[0,a.size()[,a[k]=0);
}
```

A proof that the code obeys the specifications. In other words, provide a proof that the program is correct. In general, this problem is undecidable [Ric53]. Worse, even for simple programs and simple safety properties, the problem is undecidable. The main difficulty lies in finding loop invariants, that is, a property that is satisfied in all loop iterations and is enough to ensure the correctness of the program. For example, in the array initialization program depicted in Listing 1.1, the difficulty is to guess the property $\forall k \leq i, a[k] = 0$: unrolling the loop would just give $(i = 1 \Rightarrow a[0] = 0) \wedge (i = 2 \Rightarrow a[0] = 0 \wedge a[1] = 0) \wedge \ldots$; and finding the invariant of Listing 1.2 is even harder! However, checking that a program is correct with given loop invariants is much easier: it consists in deciding the satisfiability of a formula that uses program operations and the given loop invariants, but without any fixpoints; an SMT[2] problem! Even in the case of integer programs, finding the right generalizing property, that is, loop invariant, from loop unrolling is extremely hard: it is akin to finding the right property in proofs by recursion, and any student who has gone through a math bachelor can attest to how hard this can be! Whereas checking that a program is correct given loop invariants is akin to checking that the given recursion hypothesis works, which

[2] Satisfiability Modulo Theory

is usually much easier! In the case of simple programs and loop invariants this problem is in the NP complexity class [Sch78], an extremely simple class considering that most problems in static verification are undecidable. However, the undecidability of finding loop invariants should not make us abandon it and restrain us from tackling it as there are several factors that may help us cope with this issue.

First, programmers do not write random programs: they know the intended behavior of their program and have at least an intuition of why they believe the code they wrote is correct. Thus, one may rely on the programmer's help when automated tools fail to find a correct proof. Tools such as FramaC and theorem provers such as Coq and Isabelle [Cuo+12; Fil+97; NWP02] do so extensively as the programmer may directly give invariants or prove manually the given specifications, but most techniques also do so in a much lighter fashion: techniques are usually only successful for some types of programs or have parameters to handle different types of programs and it is up to the user to choose the correct technique and/or parameter.

Secondly, many bugs are due to cases overlooked by the programmer, updates in some part of the code whose impact on another part of code was not thought of, typing and concentration errors, in very simple functions whose proof of specifications would seem easy to humans; whereas complex algorithmic computations – think of an algorithm which colors a planar graph and the specification asserts that it is done in at most 4 colors – for which the proof requires complex math is usually restricted only within very specific functions. For example, common cases overlooked by programmers include: what if this function throws an exception? What if the previous result was 0 and thus my division is invalid (for example, the average of an empty container)? What if my array was empty when computing max, making the first initialization value $a[0]$ invalid? And common programmer mistakes may be: forgetting that array indices start at zero in this language, that the bound of the loop was $n-1$ and not n, using the wrong variable k instead of n, ... This means that for many functions, ensuring that they are correct should be much easier than providing proofs of non-trivial math problems and one may hope to do so.

Tools to tackle this problem can mainly be divided into two types: proof assistants such as Coq and Isabelle [Fil+97; NWP02], where the entire proof that the program is correct is checked by an automated system, and non-certified tools such as FramaC, Z3, Astrée, ... [Cuo+12; MB08; Cou+05] who provide a proof of correctness based on proven techniques, usually a research paper whose proofs have not been implemented in a proof assistant. On one hand, proof assistants are much more demanding: the level of formalism required for the behavior of code, specifications and proofs of theorems that are used to prove the correctness of a program is extremely high: it must be understandable by an automated system based on type or set theory. On the other hand, proof assistants provide the most solid proofs and allow one to reason with the full expressivity of type/set theory.

In this PhD, the technique I produce aims to be mostly automated: the user chooses only a few input parameters and should not have to provide full invariants. The automated technique is proven on paper and is implemented within a non-certified tool – unlike Verasco [Jou16]; hopefully, the task of transferring the theorems and algorithms to a proof assistant may be done as future work.

1.1.2 Scope of the PhD

Before going into depth as to how I suggest to statically verify programs, let us briefly discuss the scaling of writing and verifying programs. Programs are executed as binary code which is usually generated from an assembly language which is itself usually written from a language similar to C

or let us say an LLVM[3] intermediate representation. For most people, it would seem much easier to make sense of a program written in C than in assembly and thus it would be much easier to check the correctness of a program in its C version than in its assembly version. We believe we should expect the same for algorithms and therefore program verification algorithms should strive to work on high-level, rather than low-level, languages where it is extremely hard to *make sense of the program*.

In practice, the levels of abstraction that a language such as C offers, that is, mainly functions and libraries, allows us to apply techniques such as function summaries to verify programs, which allows for effective scaling. This would be much harder in a language not offering functions. However, as seen by the history of current programming languages, the levels of abstraction provided by C are not sufficient to write programs efficiently and we should not expect it to be different for the goal of verifying programs.

This leads me to my topic of focus: the verification of programs containing unbounded data-structures, that is, data-structures that use a statically unknown amount of memory, or simply said, an unbounded amount of memory. However, we assume that data-structures are already recognizable, because given by the programming language, and thus, it is not necessary to use techniques such as separation logic [Rey02] to recover the data-structures from the memory layout. A simple example of the type of functions I aim to verify is the array initialization of Listing 1.1 and a slightly more complex one is given in Listing 1.2.

Listing 1.2 – Binary search

```
//summary : ∀a, v, (sorted(a) ∧ binary_search(a, v).first() ⇒ a[binary_search(a, v).second()] = v)
pair<bool,unsigned> binary_search(Array<int> a, int v)
{
  unsigned min = 0;
  unsigned max = a.size();
  while(max>=min)
  {
    /* loop invariant to discover:
       ∀k ∈ [0, min[, a[k] < v ∧ ∀k ∈]max, a.size()[, a[k] > v */

    /* Note that if one wanted to prove the liveness
       property that the function terminates, one needs to find
       that max−min is strictly decreasing */
    unsigned mid <- (max+min)/2;
    if(a[mid]>v)
      min <- mid+1;
    else if(a[mid]==v)
      return pair(true,mid);
    else
      max <- mid-1;
  }
  return pair(false, 0);
}
//Did you spot the under-specification in the summary?
```

A key aspect of unbounded data-structures compared to integer programs is that we introduce another difficulty: the number of locations is now unbounded. Although both problems are undecidable, the unbounded data-structures case seems much harder. This intuition can be justified by what happens with bounded model checking [VW86] on linear programs with quantifiers. In

[3] LLVM is not an acronym and denotes a set of compiling projects, more information on https://llvm.org/

the case with only integers and no unbounded data-structure, the problem is decidable [Len83], whereas with a simple unbounded data-structure such as arrays, the problem becomes undecidable [BMS06].

The approach that I considered tried to separate concerns: use existing techniques to handle the integer part of the problem and provide an approach whose sole purpose is to handle the unboundedness of the space locations. The key idea to handle the unbounded space is to aggregate information so that the result may speak about the whole data-structure in a condensed form. An example of an aggregation tool is quantifiers: instead of writing a property such as $a[0] = 0 \wedge a[1] = 0 \wedge \ldots \wedge a[n-1] = 0$ we can aggregate this information as $\forall i < n, a[i] = 0$. However, aggregation of values means that we lose information and thus we are applying a kind of abstraction. It is crucial that we do not lose the information which is necessary to prove the correctness of our program and thus we need to choose the correct aggregation method, in other words, the correct abstraction! For example, using quantifier aggregation is not adequate to speak about the sum of an array; however, it is adequate to speak about the relation between indices and values of an array.

1.1.3 Philosophy of the PhD

When considering a new method, two main concerns stand out. The first is the use cases of that method. In other words, because of the undecidability problem stating that no method can solve all cases, I believe that the use of a method should be sufficiently framed that a user (i.e. a programmer) can understand whether this method should be used for their program verification problem. The second concern consists in being able to evaluate if a method is good in the use cases for which it is considered relevant. There are two main ways to evaluate a method: an experimental one, which requires a large number of relevant benchmarks and an experimental comparison with other methods; and a theoretical one, that is, a theoretical property of this method that ascertains that it is good.

In my PhD, I do not consider that I provide a method that can already be used for real world programs and thus, I do not strive to make an experimental evaluation which is enough to judge the quality of my method. Although there are synthetic small benchmarks and competitions [Bey12] created by the community on which we can apply my method, I have witnessed too much variance to be convinced by these: we will see later on that it is easy to tweak methods so that they satisfy the benchmarks and thus, it is hard to compare experimental results between two tools as we would have to know how much tweaking has been done or use a set of unknown benchmarks.

Therefore, I aim to provide a solid theoretical proof of the quality of my method through theorems and I believe the best way to frame and prove the quality of a theoretical result is having a result of the form *If the problem is in this form then applying this method makes this problem strictly easier*. I also believe that the *form* in question needs to be easy to identify for the user, so that it is fairly easy to determine if this method should be used. Note that I aim to *make the problem strictly easier* and not to *solve the problem*. This is mainly because any *interesting* static verification problem is undecidable and I believe reducing the difficulty of the problem, as discussed in the aggregation of the space locations, is the best achievable goal.

1.2 Methodology : a problem verification transformation technique

Inspired by several papers [GRS05; CCL11; BMR13; MA15; MG16], we noticed that the problem of data-structure aggregation/abstraction methods is fairly independent of the handling of the rest

of the program: control flow graph, integers, ... In order to fully dissociate the data-structure aspects and the integer program aspects, I aim to provide a transformation method to handle the abstraction of data-structures and have existing methods handle the data-structureless verification problem.

1.2.1 Choosing an adequate intermediate representation: Horn clauses

Several papers handle program verification of data-structures through transformations. Perhaps one of the simplest approaches is program transformation as in [MA15], in which the array variables of a program are replaced by two integers and each array operation is transformed into operations on those two integers. One of the biggest drawbacks of this paper is demonstrated by [MG16], in which the technique of [MA15] was refined by using the additional possibilities that Horn clauses offer [Bjø+15]: the transformation of [MG16] transforms programs with arrays into Horn clauses over integers. In many ways, the technique is similar, but the authors have leveraged the expressivity of Horn clauses to achieve a better transformation. In other words, by using Horn clauses, the authors were not limited by what is expressible by programs and could achieve better results. However, one of the drawbacks of [MG16] is that transformations can't be chained: one can't compose the technique of [MG16] with itself, simply because it does not take as input Horn clauses. Another approach to handle arrays is described in [BMR13] in which arrays are removed from Horn clauses by using logical operations. This transformation is thus from and to Horn clauses, but it is unclear whether it makes sense to limit the main technique used behind it, quantifier instantiation, to Horn clauses instead of generic logical formulae.

The importance of choosing the right input and output representations for transformations has been demonstrated by research: SSA form [Cyt+91], LLVM passes [LA04], the hierarchy of compiling representations as in Compcert [Ler20]. The common pattern consists in dividing an important problem, such as program verification or compilation, into a series of intermediate representations with different levels of abstraction, with usually at most a single transformation from one intermediate representation to another and multiple *passes* that go from and to a given same intermediate representation. With that in mind, one may see how the transformation from [MG16] is ill-suited: it compounds the tasks of transforming a program verification problem into Horn clauses and eliminating the data-structures. I believe a transformation aimed at abstracting data-structures should have an output compatible with its input: a transformation that can be re-applied on its output.

In our case, we need an intermediate representation which abstracts us from *low-level* issues such as memory and from high-level constructions such as classes, exceptions, RAII[4] – a C++ construct to help ressource management and class invariants – which are not related to our issue, but keeps the core of the verification problem: we keep track of possible variable values at each program point and we can encode a safety property. Note that we do not need program execution order, but we need to be able to express function summaries to have a scalable verification scheme. Furthermore, we also need a format expressive enough to encode safety properties and allow us to apply techniques such as [MG16]. These constraints explain why we have not chosen one of the following representations. Low-level program representations such as the C language were not chosen mainly because we have the issues caused by syntax, preprocessor, memory and execution order. Functional and high-level program languages such as OcamL were avoided mainly because of constructions such as modules and higher order functions, polymorphic functions ... The LLVM representation was dismissed mainly because of memory and execution order. Why3

[4] RAII stands for Resource Acquisition Is Initialization

[FP13] and theorem prover representations [Fil+97; NWP02] were ill-suited mainly because of the extensive possible constructions. Automata and control flow graph [OG86] representations were avoided mainly because they encode the execution order which is not necessary for the verification of safety properties. Furthermore, logical transformations such as [BMR13; MG16; BMS06] are not easily expressible within that framework.

However, Horn clauses seem like the perfect fit. They encode a fixpoint relation on the possible values at each program point and allow to specify that some values are not within it. In other words, they exactly encode safety properties. Although one can usually recover the execution order from Horn clauses, it is not a key part of the specification, whereas logical transformations are easier: Horn clauses are a subset of logical formulae. Furthermore, the encoding of data-structures in Horn clauses meets our constraints: a recognizable type with a set of simple operations and axioms. Moreover, Horn clauses already have front-ends from programs and back-ends to solve them [Gur+15; Kah+16; MG16], thus, implementing a verification scheme using them does not need to be done from scratch. On all aspects, Horn clauses seem the perfect fit! Note that this only applies for safety properties as liveness properties cannot be encoded using Horn clauses.

1.2.2 Completeness, a key property for theoretical guarantees

While using existing tools from [MB08; BMR13; MG16] to solve Horn clauses, a problem arose. Slight modifications within the analyzed program could shift the result from *program certified* to *timeout* and it was unclear where the problem was located: was it the transformation that handled the data-structures that had issues with the modification, or, was it the back-end solver? As no theoretical guarantee was given for these transformation methods, finding out where the issue was for each example was extremely tiresome; and there began the long quest to prove the effectiveness of transformations.

A way to measure the theoretical effectiveness of verification methods is *completeness*. In general logics, a technique is said *complete* for a set of logical formulae if it can decide it. For example, a subset of first-order theory over integers and arrays is decidable [BMS06] and a subset of second order logic with trees is decidable [Cou90].

As mentioned previously, our focus is aggregation/abstraction methods. For these methods, one can look at how completeness is stated in the abstract interpretation community. A technique is said to be *complete relative to* an abstract domain D, for a class of programs $\mathscr{P}$ and a class of safety properties $\mathscr{S}$, if for any $P \in \mathscr{P}$ and $\phi \in \mathscr{S}$, the technique can decide the existence of an inductive invariant I in D guaranteeing that P satisfies ϕ. The existence of a technique complete relative to D for a $\mathscr{P}$ and $\mathscr{S}$ has been called *the Monniaux problem* in [Fij+19]. In practice, techniques complete relative to fixed polyhedral abstract domains exist for programs (resp. properties) with polyhedral transitions (resp. statements) [Mon19]. However, to the best of our knowledge, it is still unknown for the general polyhedral abstract domain.

In our case, our transformation method aims to handle the data-structures and leaves an integer problem to a back-end solver. However, the back-end problem of handling integers is already undecidable and thus, we cannot hope to satisfy a form of completeness for the scheme consisting in transforming and back-end solving. Instead, we adapt the definition of completeness for transformations so that it does not depend on the decidability of the transformed problem, which leads us to formulations of the form: *A transformation $\mathscr{T}$ is complete for a class of Horn clauses $\mathscr{H}$ if and only if $H \in \mathscr{H}$ is equivalent to $\mathscr{T}(H)$* has a solution. Then, introducing abstraction leads us to *A transformation $\mathscr{T}$ is said complete relative to an abstract domain D for a class of Horn clauses $\mathscr{H}$ if and only if $H \in \mathscr{H}$ has a solution in D is equivalent to $\mathscr{T}(H)$ has a solution.*

1.2.3 A Horn clause transformation framework: Data-Abstraction

The aim is to lay ground-work to provide transformation techniques that allow data-structures to be abstracted in Horn clauses, with a way to analyze the completeness of those transformations. This work leads to *The Data-Abstraction Framework*, which is the main contribution of this PhD.

The *The Data-Abstraction Framework* is parametrized by the chosen abstraction and gives algorithms to implement the abstraction as a Horn clause transformation. It provides the tools to combine abstractions and analyze the completeness of the generated transformation. We published a first version of the Data-Abstraction Framework without the proofs of completeness in [BG20] and a second version with the analysis of completeness in [BGM21].

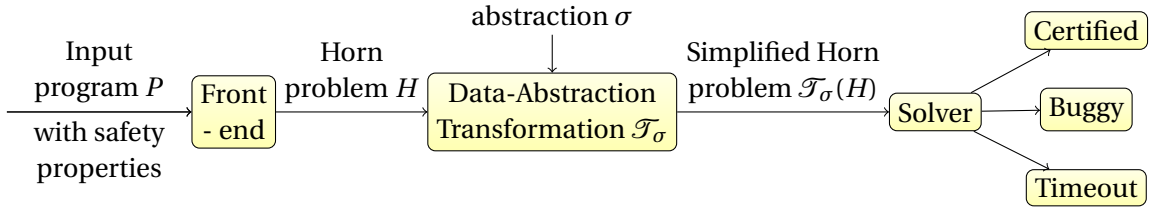
1.3 Overview of the contribution and manuscript

1.3.1 Overview of the proposed verification scheme

The overall method presented in this PhD is divided into several steps as depicted in Figure 1.1. The process takes as input a program with properties to verify and outputs whether these properties are verified for any possible run of the program. The process is divided into three main steps:

1. A front-end which transforms a program with safety properties into Horn clauses encoding whether these are verified. The semantics of our programs and safety properties is given by this transformation.
2. Our main contribution is the Data-Abstraction framework which simplifies the data-structures in Horn clauses using abstraction. The key aspect is being able to predict and prove the effectiveness of this framework.
3. A back-end solver which then solves the resulting problem which does not have the added difficulty of unbounded data-structures.

Figure 1.1 – Overall program verification scheme



1.3.2 Structure of the manuscript

The main content of the manuscript is divided into 6 chapters.

1. Chapter 2 introduces Horn clauses and Horn problems and how the problem of verifying safety properties on programs are transformed into them.
2. Chapter 3 explains what is expected of the data-abstraction framework: to *implement data-abstractions* and presents within that framework Cell abstraction [MG16], an important abstraction for arrays.
3. Chapters 4, 5 define the algorithms used in the data-abstraction framework with their analysis of completeness. Chapter 4 describes the main algorithm and Chapter 5 describes how to tune the main algorithm to make it complete.
4. Chapter 6 discusses the theoretical impact of our results on Horn clauses with respect to the problem of verifying programs.
5. Chapter 7 describes our implementation and practical results using the full toolchain.

Notation Sheet

These pages summarize the different notations used in this manuscript. We advise the reader to refer to it whenever a doubt arises.

	Notation	Description
Booleans	$b_1 \wedge b_2$	b_1 and b_2
	$b_1 \vee b_2$	b_1 or b_2
	$b_1 \rightarrow b_2$; $b_1 \Rightarrow b_2$	b_1 implies b_2. We usually use the latter notation for expressions that do not have evaluation contexts.
	$=$; $\equiv$	b_1 equals b_2. We usually use the latter for proof steps or definitions or when the former is already used.
	$\neg b$	not b
	$\text{ite}(b, e_1, e_2)$	If-then-else: if b then e_1 otherwise e_2
Arrays	$a[i]$	Returns the value of the array a at index i
	$a[i \leftarrow v]$	Returns an array a' such that $\forall k, a'[k] = \text{ite}(k = i, v, a[k])$
	$ConstArray(val)$	Returns an array such that all cells have value val
	$sorted(a, s, e)$	Returns whether an array is sorted for indices in $[s, e[$
	$BoundEq(a, b, s, e)$	Returns whether an array a is equal to b for indices in $[s, e[$
Sets & Functions	$\emptyset$	The empty set
	$\mathbb{Z}$	Signed integers
	$S_1^{S_2}$	The set of functions from S_2 to S_1
	$[a, b]$; $]a, b]$; $[a, b[$; $]a, b[$	Integer values between a and b. The inclusion of values a and b depend on the direction of the braces.
	$\{x \mid expr\}$	The set of values x such that $expr$ is verified.
	$f \circ g$	Is the composition of f and g: $\forall x, f \circ g(x) = f(g(x))$
	lfp f	The smallest fixpoint of f
Types	Int	A type for Horn clauses representing $\mathbb{Z}$.
	`Array<T>`	Are program arrays with index type unsigned integers and value type T.
	$Arr(Ind, Val)$	A type for Horn clauses representing the set of functions from Ind to Val
	$T_1 + T_2$; $T_1(v)$; $T_2(v)$; $match\ val\ f_i$	$T_1 + T_2$ represents the or type. $T_1(v)$ and $T_2(v)$ are the two ways to construct that type. $match\ val\ f_i$ destroys the type: $match\ T_i(v)\ f_i$ returns $f_i(v)$
Ocaml ≀	`fun a -> expr`	Is a function that to a value a returns b.
	`map f l`	Is the list obtained by calling f on each element of the list l
	`concat_map f l`	Here f is a function that to an element returns a list. Calls `map f l` and flattens the result: instead of a list of lists, one obtains a list.
	`filter p l`	Returns the list obtained by removing the element of the list l not satisfying p
C++ ≀	`sorted(a)`	Returns whether an array is sorted
	`a.push_back(v)`	Adds value v at end of array a
	`sub_array(a, s, e)`	Returns a reference to the subpart of the array with indices in $[s, e[$

Eval Contexts	P	A typed predicate named P. Stands for the name of a program point or function relation.
	$\mathscr{M}$	A function that to a predicate returns a set of values.
	$vars$	A function that to a variable name returns a value.
	$[\![expr]\!]^{vars}_{\mathscr{M}}$	Evaluates an expression $expr$ in the contexts $\mathscr{M}$ for predicates and $vars$ for its free variables. This can be used on any kind of expressions, including sets. Definition 2, page 29
	$[\![expr]\!]^{\forall}_{\mathscr{M}}$	Stands for $\forall vars, [\![expr]\!]^{vars}_{\mathscr{M}}$
Horn	C, goal, premises normalized, extended linear, assertion	Horn clause C and syntax. Definition 3, page 29
	$\mathfrak{C}$ satisfiable	A set of Horn clauses $\mathfrak{C}$ is satisfiable if and only if $\exists \mathscr{M}, \forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathscr{M}}$
	$H, f_H, \mathscr{U}_H$	H is a Horn problem, that is, a pair $(f_H, \mathscr{U}_H)$ where f_H is a function and $\mathscr{U}_H$ is a model. Definition 5, page 31
	$H(\mathscr{M})$	$f_H(\mathscr{M}) \leq \mathscr{M} \wedge \mathscr{M} \leq \mathscr{U}_H$. Definition 6, page 31
	H satisfiable	One of the two following equivalent definitions $\mathrm{lfp}\, f_H \leq \mathscr{U}_H$ or $\exists \mathscr{M}, H(\mathscr{M})$
	$H_{\mathfrak{C}}$	The Horn problem encoded by $\mathfrak{C}$. Definition 7, page 31
Generic (data-)abstraction	$\mathscr{G}; \alpha_{\mathscr{G}}; \gamma_{\mathscr{G}}$	The Galois connection $\mathscr{G}$. $\alpha_{\mathscr{G}}$ and $\gamma_{\mathscr{G}}$ are its abstraction and concretization functions. Definition 8, page 38
	$e \triangleleft \mathscr{G}$	e is expressible by the abstraction $\mathscr{G}$. Definition 11, page 42
	$\sigma(a); \alpha_\sigma; \gamma_\sigma$	σ is a data-abstraction, $\sigma(a)$ is the set of abstract values for a. α_σ and γ_σ represent the abstraction and concretization functions of the abstraction of sets induced by σ. Definition 15, page 46
	$\mathscr{G}^{\sigma^{abs}}$	Is the abstraction that abstracts each predicate P using the data-abstraction $abs(P)$. This is the abstraction our framework must implement. Definition 15, page 46
	$\alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})$	This is a consequence of the previous notations. $\forall P, \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^\#) = \alpha_{abs(P)}(\mathscr{M}(P))$
	$\mathscr{G}(H)$	Horn problem induces by the abstraction $\mathscr{G}$. Definition 19, page 58
	F_σ	The relation encoding $a^\# \in \sigma(a)$. Definition 20, page 60
	$F_\sigma[q]$	F_σ with existential quantifiers set to q. Definition 21, page 63
Abs Instances	id, σ_{id}	Is the identity data-abstraction. Definition 16, page 48
	$\sigma_1 \bullet \sigma_2$	Is the data-abstraction combinator for pairs. Definition 16, page 48
	$\sigma_1 \odot \sigma_2$	Is the combinator for composition. Definition 16, page 48
	$Cell, \sigma_{Cell}$	Is the cell data-abstraction for arrays. Definition 17, page 50
	$\sigma_1 \oplus \sigma_2$	Is the sum data-abstraction combinator. Definition 18, page 53
	$\sigma_1 \otimes \sigma_2$	Is the product data-abstraction combinator. Definition 18, page 53
	σ^n_{Cell}	$\sigma_{Cell} \otimes \ldots \otimes \sigma_{Cell}$, n times
Properties	*Sound, complete, relative completeness, implements the abstraction.* Definition 14, page 45	
	Per clause completeness is a local completeness property. Definition 22, page 68	
	Per clause relative completeness is its relative variant. Definition 23, page 71	
	Complete call to $insts$ enables completeness of $eliminate$. Definition 24, page 73	
	Relevant cells enables completeness of $insts$ for cell abstraction. Definition 27, page 98	
Algorithms	$abstract$ is Algorithm 1 of page 60.	
	$eliminate$ is Algorithm 2 of page 66.	
	$insts$ when $abs_P = \sigma_{Cell}$ and $relevant$ are in Algorithm 3 of page 102.	
	$insts$ when abs_P is finite is Algorithm 4 of page 106.	
	$insts$ when abs_P is a combinator are Algorithms 5, 6, 7, 8, pages 107, 110, 113, 114.	

2 From programs to Horn clauses

As discussed in the introduction, we use Horn clauses as an intermediate representation in our verification scheme. Horn clauses were chosen because they are general enough to encode safety properties of programs and are written in a syntax with clear semantics. Chapters 3, 4 and 5 mainly talk about Horn clauses and it is thus key that the reader fully understands the link between Horn clauses and programs. This is especially true for the discussions of Chapters 5 and 7 which aim to show how what we do with Horn clauses impacts programs.

Therefore, the goal of this chapter is not to explain the transformation of a full-fledged programming language into Horn clauses, as a tool such as SeaHorn [Gur+15] may do, but mainly to show how the safety properties of simple programs can be converted into Horn clauses, and the choices that impact our solving of Horn clauses. Note that a full transformation should not only handle complex programming language constructions such as exceptions, memory layout, ..., but also uncover the data-structures expressed through memory pointers: writing the memory as a huge array in Horn clauses will not lead to tractable Horn clauses. This is however not the purpose of this chapter and this chapter should not be viewed as a scientific contribution, but rather as a preliminary chapter.

In this chapter, we first give in Section 2.1 an informal explanation of the transformation through examples and show how some choices impact the underlying Horn clauses. In that section, Horn clauses are not well defined and should be viewed as logical formulae. We then give a formal definition of Horn clauses and Horn problems which sets up notations for the rest of this PhD. Finally, we consider our running example, merge sort.

2.1 Transforming programs into logical constraints

In this section, we discuss how the problem of verifying safety properties in programs can be expressed as logical formulae. These logical formulae will encode what we define as *Horn problems* in the next section, but for now, this is not necessary.

2.1.1 The transformation of a simple program: array initialization

Safety properties define sets of variable values that are not allowed at given program points. For example, `assert(a[0] =0);` states that at the program point defined by the assertion, any array value for a in $\{a \mid a[0] \neq 0\}$ is not allowed. Thus, we need a definition of the set of possible variable values at the program point of the safety property.

The set of possible variable values at a given program point can usually be defined by the operation and the program point that precedes it: for example in `/*Program point` P_1 `*/ i=i+1; /*Program point` P_2 `*/`, the set of possible values for i at program point P_2 is simply $\{v+1 \mid v \text{ is a possible value for } i \text{ at } P_1\}$. Of course, the notion of *program point that precedes it* is defined by the control flow graph [OG86]. For example `/*Program point` P_1 `*/ while(i<n) {i=i+1; /*Program point` P_2 `*/}`, has P_1 preceding P_2, but also P_2 preceding P_1. In the case of while loops, this creates a mutual relation between the possible variable values at program points: in our example, the possible variable values for P_2 depend on those for P_1 which depends on those for P_2 and so on.

To compute the possible variable values at each program point, we define for each program point P_i, the function $P_i(vars)$. The aim is to write constraints on $P_i(vars)$ such that $P_i(vars)$ returns whether the variable values $vars$ are possible at program point P_i. In other words. $\{vars \mid P_i(vars)\}$ should be the set of possible variable values at program point P_i. In Example 1 we show, on an array initialization function, how to write the minimal constraints that ensure that, for all possible variable values $vars$ at program point P_i, we have $P_i(vars)$. Because these constraints are minimal, to retrieve the exact possible variable values at each program point, one just needs to find $P_i(vars)$ satisfying these contraints and such that $\{vars \mid P_i(vars)\}$ is as small as possible. However, this is not our primary goal: we do not wish to find the exact set of possible variable values at each program point, and such that $\{vars \mid P_i(vars)\}$ is as small as possible, we only need to find $P_i(vars)$ satisfying these contraints and satisfying the safety property.

A more abstract view which is not yet of great importance but will form the basis of what we call *Horn problems* is: the program constraints we write on each function P_i define the set of possible variable values at each program point by mutual induction. As we only need to check that the least fixpoint of the induction satisfies the safety property, it is sufficient to check the existence of a postfixpoint satisfying the safety property.

The constraints created by the array initialization program of Listing 1.1 without any specified safety property, that is, without the assertion, are given in Example 1. To add the safety property corresponding to the assertion `assert(`$\forall k \in [0, a.size()[, a[k] = 0$`);` that states that, *for any input array a, at the end of the function, we have* $\forall k \in [0, a.size()[, a[k] = 0$, where $[0, a.size()[$ is the interval from 0 to $a.size()$ containing 0 but not $a.size()$, we need two additional constraints:

1. the constraint $\forall a, P_1(a)$ expresses that the array initialization function can be called with any input array, and thus, we should consider any value of the array a at program point P_1.
2. the constraint $\forall a, i, k, (P_8(a,i) \wedge 0 \leq k < a.size()) \rightarrow a[k] = 0$ expresses that, at the end of the function, the array is initialized.

The problem of whether the array initialization program verifies that safety property is now reduced to the question of **existence** of $P_1, P_3, \ldots$ verifying all the constraints.

Example 1 (Program point constraints for the array initialization program)**.** *In this example, P_i represents the program point at the end of the line numbered i, $\wedge$ is the boolean and operand, $\rightarrow$ is the boolean implies operand, $size(a)$ is the size of the array a, $a[i \leftarrow 0]$ is a new array a' defined as $a'[i] = 0$ and $a'[k] = a[k]$, for $k \neq i$.*

```
1  void array_init(Array<int>& a)
2  {
3    unsigned i=0;
4    while(i<a.size())
5    {
6      a[i] <- 0;
7      i <- i+1;
8    }
9  }
```

$$\forall a, i, P_1(a) \rightarrow P_3(a, 0)$$
$$\forall a, i, P_3(a, i) \wedge i < size(a) \rightarrow P_4(a, i)$$
$$\forall a, i, P_4(a, i) \rightarrow P_6(a[i \leftarrow 0], i)$$
$$\forall a, i, P_6(a, i) \rightarrow P_7(a, i+1)$$
$$\forall a, i, P_7(a, i) \rightarrow P_3(a, i)$$
$$\forall a, i, P_3(a, i) \wedge \neg(i < a.size()) \rightarrow P_8(a, i)$$

2.1.2 Verifying safety properties of program points: assertions

The basic principle of how safety properties on programs are transformed into logical constraints has been explained in Section 2.1.1. We separated two types of constraints: those due to program constraints and those expressing the safety property. Among safety properties, we mainly distinguish properties of program points and properties linking program points. The former are mainly assertions, whereas the latter may for example be relations between input and output values of a function. In this section, we explain how assertion properties can be transformed into logical constraints.

Program assertions. The most basic type of safety property is simply an assertion written in the program language. The way to handle such safety properties as a verification constraint is fairly straightforward: simply write the constraint *if we have values vars at the program point of the assertion, then they must verify the assertion.* For example, one could have added `/* Program point P_8 */ if(a.size() > 0) assert(a[0]=0);` at the end of the array initialization function of Example 1. This safety property would have added the constraints $\forall a, i, P_8(a, i) \wedge a.size() > 0 \rightarrow P_9(a, i)$ and $\forall a, i, P_9(a, i) \rightarrow a[0] = 0$.

In practice, one would want to verify that the array is properly initialized for all values, not only 0. If we are limited to program assertions, one should do so by going through a loop `for(unsigned i=0; i< a.size(); i++) assert(a[i]=0);` and verifying all indices. This may be seen as cumbersome and explains why one may wish to use more expressive type of assertions. The code corresponding to this loop verification is available in Listing 2.1.

Extending assertions: quantifiers. In the case of the array initialization example, it is tempting to use the assertion `/* Program point P_8 */ assert(`$\forall k \in [0, a.size()[, a[k] = 0)$`;` to express the desired safety property as in Listing 2.2. This is an extension with quantifiers to program assertions and can be handled by our transformation into logical constraints. To handle the assertion `/* Program point P_8 */ assert(`$\forall k \in [0, a.size()[, a[k] = 0)$`;`, we simply add the constraint $\forall a, i, P_8(a, i) \rightarrow (\forall k, 0 \leq k < a.size() \rightarrow a[k] = 0)$ which is equivalent to the one discussed in Section 2.1. In practice, a proper language for safety properties such as ACSL [Bau+08] would consider other extensions, but these are not of much interest to us in this PhD.

The method we suggest to handle assertions with quantifiers is fairly straightforward. However, we wish to discuss another method that has been used to prove quantified assertions [MA15; MG16]: using a random index. In this method, instead of checking that all indices of the array are

Listing 2.1 – Using program assertions

```
void array_init(Array<int>& a)
{
  unsigned i=0;
  while(i<a.size())
  {//Program point P
    a[i] <- 0;
    i <- i+1;
  }
  i=0;
  while(i<a.size())
  {
    assert(a[i] =0);
    i++;
  }
}
```

Listing 2.2 – Using quantified assertions

```
void array_init(Array<int>& a)
{
  unsigned i=0;
  while(i<a.size())
  {//Program point P
    a[i] <- 0;
    i <- i+1;
  }
  assert(∀k ∈ [0, a.size()[, a[k] = 0);
}
```

Listing 2.3 – Using start random index

```
void array_init(Array<int>& a)
{
  unsigned k=rand()%a.size();
  unsigned i=0;
  while(i<a.size())
  {//Program point P
    a[i] <- 0;
    i <- i+1;
  }
  assert(a[k]=0);
}
```

Listing 2.4 – Using end random index

```
void array_init(Array<int>& a)
{
  unsigned i=0;
  while(i<a.size())
  {//Program point P
    a[i] <- 0;
    i <- i+1;
  }
  unsigned k=rand()%a.size();
  assert(a[k]=0);
}
```

Listing 2.5 – Variation of the array initialization program

```
bool array_init_and_check(Array<int>& a)
{
  unsigned i=0;
  while(i<a.size())
  {//Program point P
    a[i] <- 0;
    i <- i+1;
  }
  bool b=true;
  i=0;
  while(i<a.size())
  {
    if(a[i] ≠ 0)
      b=false;
    i++;
  }
  //Checking a single random index will not work to prove that b=true
  return b;
}
```

initialized, the idea is to introduce a new variable whose value is random, and check that the array is initialized at that value. Since that value is random, this means that the array is initialized for all values. The code corresponding to this method is given in Listings 2.3 and 2.4.

One of the key differences between Listings 2.1 and 2.2 and the Listing 2.3 is the loop invariant required to prove correctness of the program. In the cases of Listings 2.1 and 2.2, the invariant at program point P needs to be $\forall k \in [0, i[, a[k] = 0$, whereas for Listing 2.3, the invariant $k < i \rightarrow a[k] = 0$ suffices. As for Listing 2.4, it does formally require a quantified invariant; however, due to its syntactical proximity with Listing 2.3, we are unsure whether current solving techniques would not reorder without our knowledge, thus making our next argument hold for it as well.

We argue that the random index method is perhaps best to verify programs as it allows for simpler invariants – i.e. not quantified – and thus has the best chance of succeeding. However, we do not believe it should be used to demonstrate the effectiveness of techniques handling universal properties of arrays for two reasons. First the resulting program does not need quantified invariants and thus, the success of the verification technique on it does not prove anything about its handling of quantified invariants. Second, and this is highly tied to the previous reason, a technique that succeeds on Listing 2.3, but fails on Listings 2.1 and 2.2 cannot handle variations of these programs such as the one of Listing 2.5.

Thus, as our goal for now is to prove the effectiveness of our verification technique and not verify as many programs as possible, as well as making it reliable to variations and syntactical changes, we will use either the method of Listing 2.1 or the one of Listing 2.2. Our method can later be combined with a method that transforms some quantifiers into random indices.

2.1.3 Verifying and using functions

2.1.3.1 Functions as relations between input and output values

Unlike properties of program points, properties of functions usually link the input and the output value of a function. Thus, just like we encoded the set of possible variable values at each program point using logical constraints, we suggest to encode the relation between input and output values by logical constraints. The main idea is to consider a program point that has both the initial variable values and the returned values so that we can extract the relationship from it.

In the case of the binary search program of Listing 1.2, because it does not modify its input and does not have any side-effect, the program points of the return statements contain all the information. If we call $BS((a, v), (b, i))$ the relation between the input values (a, v) and the output values (b, i) of the binary search program, we add two constraints corresponding to the two return statements.

1. The return statement `/* Program point P_10*/return pair(true,mid);` yields the constraint $\forall a, v, min, max, mid, P_{10}(a, v, min, max, mid) \rightarrow BS((a, v), (true, mid))$.
2. The return statement `/* Program point P_16*/return pair(false,0);` yields the constraint $\forall a, v, min, max, P_{16}(a, v, min, max) \rightarrow BS((a, v), (false, 0))$.

The full transformation of the binary search program into logical constraints can be found in Example 2.

However, in the case of functions with side-effects, it is not as easy: we want to capture the side-effects within the relation and thus we consider the side-effects as a new returned value. In the case of the array initialization function, this means that the final value of the array should be viewed as a return value and thus the relation we consider for it has as input a, the initial value of the array, and as output a', the value of the array at the end of the function. Let us call the relation for the array initialization $AI(a, a')$.

There is still another problem with the array initialization example: we do not have a program point at which we have both the initial value of a and the final value of a. We solve this problem by proceeding as if we had copied the initial arguments in a variable that is kept unchanged throughout execution. In Example 3, we show how this is done for the array initialization program.

We now need to use the constrained relation either to verify that the function verifies a given safety property or to encode function calls, that is instructions of the form `/* Program point P_s*/ out = f(in); /* Program point P_e*/`.

Example 2 (Transformation of the binary search function into logical constraints)**.** *The free variables of each constraint (i.e. each line) are assumed universally quantified.*

```
1  pair<bool,unsigned>
       binary_search(Array<int> a,
       int v)
2  {
3    unsigned min = 0;
4    unsigned max = a.size();
5    while(max>=min)
6    {
7      unsigned mid = (max+min)/2;
8      if(a[mid]>v)
9        min <- mid+1;
10     else if(a[mid]=v)
11       return pair(true,mid);
12     else
13       max <- mid-1;
14
15
16   }
17   return pair(false, 0);
18 }
```

$P_1(a,v) \rightarrow P_3(a,v,0)$
$P_3(a,v,min) \rightarrow P_4(a,v,min,size(a))$
$P_4(a,v,min,max) \wedge max \geq min \rightarrow P_5(a,v,min,max)$

$P_5(a,v,min,max) \rightarrow P_7(a,v,min,max,(max+min)/2)$
$P_7(a,v,min,max,mid) \wedge a[mid] > v \rightarrow P_8(a,v,min,max,mid)$
$P_8(a,v,min,max,mid) \rightarrow P_9(a,v,mid+1,max,mid)$
$P_7(a,v,min,max,mid) \wedge \neg a[mid] > v \wedge a[mid] = v \rightarrow P_{10}(a,v,min,max,mid)$
$P_{10}(a,v,min,max,mid) \rightarrow BS((a,v),(true,mid))$
$P_7(a,v,min,max,mid) \wedge \neg a[mid] > v \wedge a[mid] \neq v \rightarrow P_{12}(a,v,min,max,mid)$
$P_{12}(a,v,min,max,mid) \rightarrow P_{13}(a,v,min,mid-1,mid)$
$P_9(a,v,min,max,mid) \rightarrow P_{15}(a,v,min,max,mid)$
$P_{13}(a,v,min,max,mid) \rightarrow P_{15}(a,v,min,max,mid)$
$P_{15}(a,v,min,max,mid) \rightarrow P_4(a,v,min,max)$
$P_4(a,v,min,max,mid) \wedge \neg max \geq min \rightarrow P_{16}(a,v,min,max)$
$P_{16}(a,v,min,max) \rightarrow BS((a,v),(false,0))$

Example 3 (Array initialization program as a function.)**.** *We add a new variable a_I which stays constant throughout execution and do as if we returned the modified array. Thus, we encode the relation AI for function calls to $array_init$. The free variables of each constraint (i.e. each line) are assumed universally quantified.*

```
1  void array_init(Array<int>& a)
2  {
3    unsigned i=0;
4    while(i<a.size())
5    {
6      a[i] <- 0;
7      i <- i+1;
8    }
9  }
10
```

$P_1(a_I) \rightarrow P_3(a_I,a_I,0)$
$P_3(a_I,a,i) \wedge i < size(a) \rightarrow P_4(a_I,a,i)$

$P_4(a_I,a,i) \rightarrow P_6(a_I,a[i \leftarrow 0],i)$
$P_6(a_I,a,i) \rightarrow P_7(a_I,a,i+1)$
$P_7(a_I,a,i) \rightarrow P_3(a_I,a,i)$
$P_3(a_I,a,i) \wedge \neg(i < a.size()) \rightarrow P_8(a_I,a,i)$
$P_8(a_I,a,i) \rightarrow AI(a_I,a)$

2.1.3.2 Verifying functions

A safety property of a function is simply a property of the relation between the input and output values; and thus, amounts to adding a logical constraint for the relation corresponding to that property.

For example, in the case of the array initialization function, one may wish to specify the safety property stating that *at the end of the function, all cells of the array are equal to zero and the size of the array is equal to its initial size.* This is simply done by adding the constraint

$\forall a_I, a, AI(a_I, a) \rightarrow (\forall k, 0 \leq k < a.size() \rightarrow a[k] = 0) \wedge a.size() = a_I.size())$ to the constraints of Example 3.

This can be reproduced for the binary search function: consider the property $\forall a, v, ((sorted(a) \wedge binary_search(a, v).first()) \Rightarrow a[binary_search(a, v).second()] = v)$ of Listing 1.2, which states that, for any input sorted array a and value v, if $binary_search$ says it has found the element, then a read of the array a at the index returned by $binary_search$ returns v. This is translated by adding the following logical constraint to the constraints of Example 2: $\forall a, v, b, i, (BS((a, v), (b, i)) \wedge sorted(a) \wedge b) \rightarrow a[i] = v$.

However, this idea is not sufficient as we have not constrained the input values of the function. Currently, all sets of possible values can be picked empty, as we have no constraint on the set of possible values at the input of the function. Thus, we also need to add the logical constraints $\forall a, P_1(a)$ for the array initialization function and $\forall a, v, P_1(a, v)$ for the binary search function.

In this approach, we consider that both the array initialization function and the binary search function are defined for all inputs. However, intuitively, the binary search function should only be used on sorted array inputs. We have already handled this in the way we specified the safety property: $\forall a, v, b, i, (BS((a, v), (b, i)) \wedge \boldsymbol{sorted(a)} \wedge b) \rightarrow a[i] = v$ only ensures that the return value makes sense when the input array is sorted.

While this method does work for our examples, if we consider a slight modification of the binary search function by adding, for example at program point P_3, the assertion `/*Program point P_3*/ assert(sorted(a));`, this creates a problem: we return that the function is buggy as we cannot find $P_1, P_3, \ldots$ verifying all the constraints anymore: we have $\forall a, v, P_1(a, v)$, which implies $\forall a, v, P_3(a, v, 0)$, which does not imply $sorted(a)$ and thus breaks the assertion. Yet, we consider such modifications of the binary search function still correct and thus, we suggest a modification of the approach.

Instead of considering that functions are defined for all inputs but check the relation only for relevant inputs, we suggest to verify partial functions, that is, functions that are only defined for inputs verifying a given condition. Classically [FL11], we name such a condition a *precondition*. By doing so, the *full* safety property of binary search can be simplified so that it does not need to check that the array is sorted. By doing so, we say that we have divided the *full* safety property of binary search into a *precondition* and a *postcondition*. This leads to the following logical constraints:

1. The precondition `sorted(a)` leads to the constraint $\forall a, v, sorted(a) \rightarrow P_1(a, v)$ instead of $\forall a, v, P_1(a, v)$.
2. The constraint $\forall a, v, b, i, (BS((a, v), (b, i)) \wedge b) \rightarrow a[i] = v$ is created by the postcondition $\forall a, v, (binary_search(a, v).first() \Rightarrow a[binary_search(a, v).second()] = v)$.

This allows for the adding of assertions of the form `assert(sorted(a));` at any point of the binary search function without any issues.

2.1.3.3 Handling function calls

In this section, we do not aim to verify functions, but to handle function calls when verifying safety properties of either program points or other functions. This amounts to creating logical constraints for the instruction `/* Program point P_s*/ out = f(in); /* Program point P_e*/`, or `/* Program point P_s*/ f(in); /* Program point P_e*/`, where f is a function; and we suggest three approaches.

Modular handling of function calls This approach consists in using the already constrained relation $F(in, out)$ to express the logical constraints linking P_s and P_e. However, it is important to

also constrain the input program point of the function f: it must now include in. Thus, with possible slight variations depending on whether f has side effects and the variables defined at program points P_s and P_e, we need two following logical constraints:

1. $\forall in, out, P_s(in) \wedge F(in, out) \rightarrow P_e(in, out)$, where F is the relation of the function f.
2. $\forall in, P_s(in) \rightarrow P_f(in)$, where P_f is the input program point of the function f.

For example, the following call to the binary search function `/* Program point P_s*/ (b, i) = binary_search(a, 3); /* Program point P_e*/` yields constraints $\forall a, b, i, P_s(a) \wedge BS((a,3),(b,i)) \rightarrow P_e(a,(b,i))$ and $\forall a, P_s(a) \rightarrow P_1(a,3)$. As for the instruction `/* Program point P_s*/ array_init(a); /* Program point P_e*/` we have the constraints $\forall a, a', P_s(a) \wedge AI(a,a') \rightarrow P_e(a')$ and $\forall a, P_s(a) \rightarrow P_1(a)$.

This approach is extremely general, fairly simple and scales well. However, the generated logical constraints may be hard to solve: the relation $F(in, out)$ is constrained not only by the definition of the function, but also by all calls to the function. Thus, solving the constraints on $F(in, out)$ cannot be done locally.

Using verified functions to handle function calls: summaries Another technique to handle function calls is by using properties we have already verified on functions: instead of using the relation $F(in, out)$, which is defined by the code of the function, to specify the behavior of the function call, we may use preconditions and postconditions that have already been proven about the function.

With this approach, the instruction `/*Program point P_s*/ out = f(in);/* Program point P_e*/`, where f has a precondition $pre(in)$ and a postcondition $post(in, out)$, creates two constraints:

1. $\forall in, out, P_s(in) \wedge post(in, out) \rightarrow P_e(in, out)$ stating that we use the postcondition for our knowledge of the relation between in and out.
2. $\forall in, P_s(in) \rightarrow pre(in)$ stating that the input must verify the precondition.

Of course, we assume that the logical constraints stating that $post(in, out)$ is a valid postcondition and for the precondition $pre(in)$ have already been added as discussed in Section 2.1.3.2.

For example, on the following instruction searching for the value 3 in the array a, `/* Program point P_s*/ (b, i) = binary_search(a, 3); /* Program point P_e*/`, if we consider the already discussed precondition $sorted(a)$ and postcondition $b \Rightarrow a[i] = v$, where (a, v) is the input expression and (b, i) the output expression, this yields the constraints $\forall a, b, i, (P_s(a) \wedge (b \rightarrow a[i] = v)) \rightarrow P_e(a,(b,i))$ and $\forall a, P_s(a) \rightarrow sorted(a)$, in addition to the already mentioned constraints stating that $sorted(a)$ is a precondition and $b \Rightarrow a[i] = v$ a postcondition.

This method has two drawbacks and one major advantage compared to the modular handling of function calls. The first drawback is that we need the user, or a tool, to supply a precondition and a postcondition for the function we wish to call. Second, even if the postcondition can be verified, it may be too imprecise to prove the safety property we are considering. For example, consider the following program `a = random_array; (b, i) = binary_search(a, 3); assert(`$\neg b \rightarrow (\forall i \in [0, a.size()[, a[i] \neq 3)$`);`. If we use the already mentioned precondition and postcondition for binary search, we cannot prove that the assertion is verified: the postcondition $b \Rightarrow a[i] = v$ does not state anything interesting when b evaluates to $false$. We say that the postcondition is under-specified.

However, the main advantage of this method is that it separates the problem of handling function calls and verifying functions: we ensure that functions verify the postcondition for the given precondition, and then use the precondition and postcondition in the calls, instead of adding constraints to the unknown relation $F(in, out)$. By doing so, the verification problem becomes sim-

pler as the unknown relation $F(in, out)$ is only constrained locally, and more robust to changes: a change to the function f that does not break its postcondition does not affect the verification of the rest of the code.

In many ways, this is very similar to what is already done in libraries and large code-bases: instead of using the code of the function as the specification for the function, one uses its documentation! Therefore, in many cases, the drawbacks of this technique are small: the user already specified the behavior of the function in the documentation, and that behavior is the one the programmer should use and should therefore not be under-specified; the main difference is that the documentation needs to be written in a language understood by the verification process. This is why we believe summaries is the most important function call technique to handle.

Inlining: removing function calls Another way to handle function calls in programming languages is by replacing a call to a function by the code of the called function. Example 4 is a simple example of how this can be done. This technique is called *inlining* and suffers from two major drawbacks. First, this technique may generate huge duplication of code as the function is copied for each calling context. Secondly, this technique is unable to handle general recursion as the function needs to unroll itself recursively: a recursive function f calls f and thus requires to embed the code of f at that point, but that code also calls f and requires to embed the code of f and so on and so forth.

Our handling of function calls for verification purposes can also be done using inlining: inline the desired function calls and only then transform the resulting program into logical constraints. On the one hand, this may make the problem huge; on the other hand, the logical constraints are simpler: we do not need to use a set describing the relation between input and output values of the function. Thus, this approach is extremely relevant for functions that only have one or two calling contexts.

Comparison of these three techniques We suggested three different ways of how one can write the constraints due to function calls. In Table 2.1, we summarize the benefits, the drawbacks and the use cases of each one. Note that each call of a same program can be handled by a different method according to what seems best suited.

Table 2.1 – Comparison of how function calls are handled

Method	Benefits	Drawbacks	When to use
Inlining	Extremely simple and amounts to not handling function calls.	Produces duplication of the whole function at each call and cannot handle recursive functions	If the function is small or called from only 1 or 2 places.
Summaries	No duplication, possible abstraction by the user.	Requires pre/postconditions to be provided.	If the function is used a lot.
Modular	No duplication and no added information.	Harder to check: the relation needs to be inferred.	If we have a verification technique that can handle it.

Example 4 (Inlining of the factorial function in the computation of $\binom{n}{k}$).

```
unsigned factorial(unsigned n)
{
  unsigned res=1;
  while(n>0)
  {
    res=res*n;
    n--;
  }
  return res;
}
```

```
unsigned choose(unsigned n,
    unsigned k)
{
  unsigned num=factorial(n);
  unsigned bot1=factorial(k);
  unsigned bot2=factorial(n-k);
  return num/(bot1*bot2);

}
```

If we inline the calls to `factorial` *in* `choose`*, this yields the program:*

```
unsigned choose(unsigned n,
    unsigned k)
{
  //First inline call
  unsigned n_1=n;
  unsigned res_1=1;
  while(n_1>0)
  {
    res_1=res_1*n_1;
    n_1--;
  }
  unsigned num=res_1;

  //Second inline call
  unsigned n_2=k;
  unsigned res_2=1;
  while(n_2>0)
  {
    res_2=res_2*n_2;
    n_2--;
  }
  unsigned bot1=res_2;

  //Third inline call
  unsigned n_3=n-k;
  unsigned res_3=1;
  while(n_3>0)
  {
    res_3=res_3*n_2;
    n_3--;
  }
  unsigned bot2=res_3;

  return num/(bot1*bot2);
}
```

2.1.4 Using a theory adapted to logical formulae

Consider the array merge sorted program of Listing 2.6 which merges two sorted arrays. In this program we see not only the *size* instruction, but also the *push_back* instruction which adds an element to the end of the array. Our current transformation from programs to logical constraints uses the same theory for the logical formulae as the used programming language. Thus, we would use *push_back* in the logical constraints.

The problem with that approach is that the theory of a programming language is usually ill-suited for logical formulae: first, the programming language usually has many elements in its theory so that one can program fast. Our logical formulae are generated automatically and this is thus not a benefit and it introduces multiple syntactical elements that one needs to handle in algorithms, making them longer. Secondly, the types of a programming language are usually constrained by execution problems: for example, integers are usually bound so that they fit in the processor. This is not an interesting property for types in our logical formulae and the better approach is to use infinite integers, and add bound constraints during the transformation. This makes algorithms more general as they need to handle general bound constraints while making the type system easier.

The theory we use in our logical formulae can be changed but usually has the following specificities which corresponds to what already exists in the Horn solver we use. This theory is basic and may be extended according to need.

1. An integer type Int which represents any unbounded signed integer, that is, an element of $\mathbb{Z}$. The operations on the Int type are as usual: addtion, multiplication, comparison...
2. A boolean type $Bool$ with usual operations and the if-then-else instruction $ite(b, v_1, v_2)$ which return v_1 when b is true and v_2 otherwise.
3. And, as in this PhD we consider arrays, an array type Arr parametrized by an index type $\mathscr{I}$ and a value type $\mathscr{V}$. When $\mathscr{I}$ is unspecified, one should assume Int. These arrays represent functions from $\mathscr{I}$ to $\mathscr{V}$, however, the operations on the type Arr are closer to arrays: we have the write operation $a[i \leftarrow v]$, the read operation $a[i]$ and the = operation.
4. Additionally, we use $ConstArray(val), sorted(a, beg, end)$ and $BoundEq(a, b, beg, end)$ which are not usually handled within Horn solvers, where $ConstArray(val)$ is the constant array equal to val, $sorted(a, beg, end)$ is equivalent to $\forall i, j, beg \leq i < j < end \rightarrow a[i] \leq a[j]$ and $BoundEq(a, b, beg, end)$ denotes partial equality and is equivalent to $\forall i, beg \leq i < end \rightarrow a[i] = b[i]$.
   In practice, one can either consider $sorted$ and $BoundEq$ to be aliases for their respective quantified expressions or as new elements of the theory. The difference is that in one case one needs to handle these quantified expressions whereas in the other case, one needs to handle additional theory constructors but not quantifiers. In this manuscript, this difference is only of importance for the algorithm of Listing 5.6.

To demonstrate how a transformation from program to logical constraints may change the theory, we show how the array merge sorted program of Listing 2.6 can be transformed into logical constraints using the above theory for arrays: thus we eliminate the $size$ and $push_back$ operations. For readability, we do not give the transformation into logical constraints, but how this program can be transformed into the equivalent program of Listing 2.7 which uses the theory of logical constraints for arrays. To deduce the final constraints, one should simply transform the program of Listing 2.7 into logical constraints as we have done before. Note that in this transformation we only simplify the theory of arrays and we leave the theory for integers untouched.

2.1.5 Large block encoding

The transformation we described creates, for each program point P, the unknown set of possible variable values at program point P. In practice, this is not necessary: for example, the two instructions `i++;/*Program point P*/ i=i*2;` are semantically equivalent with respect to safety properties to the single instruction `i= (i+1)*2;`, thus why introduce the set of possible values at program point P?

This idea is well-known in the compiling community and is called large block encoding [Bey+09]: instead of encoding each instruction separately, we can encode blocks of instructions together. This allows further simplification within the logical constraints. For example, the loop body of Listing 2.7, without the assertion, can be converted into the following logical constraint, where P is the program point at the beginning of the loop and with a few simplifications.

$$\forall b, b_s, c, c_s, res, res_s, bi, ci, res', bi', ci', \big(P(b, b_s, c, c_s, res, res_s, bi, ci) \wedge bi + ci < b_s + c_s - 1 \wedge (res', bi', ci') = ite(b[bi] > c[ci], (res[res_s \leftarrow c[ci]], bi, ci + 1), (res[res_s \leftarrow b[bi]], bi + 1, ci))\big) \rightarrow P(b, b_s, c, c_s, res', res_s + 1, bi', ci')$$

Listing 2.6 – Array merge sorted

```
//Merging of two sorted arrays

Array<int> merge_sorted(Array<int> b,
                        Array<int> c)

{
  Array<int> res;

  unsigned bi=0;
  unsigned ci=0;
  while(bi+ci<b.size()+ c.size()-1)
  {

    if(b[bi] > c[ci])
    {
      res.push_back(c[ci]);

      ci++;
    }
    else
    {
      res.push_back(b[bi]);

      bi++;
    }
  }
  return res;
}
```

Listing 2.7 – Array merge sorted simplified

```
//Merging of two sorted arrays
//Let ArrInt = Arr<unsigned, int>
(ArrInt, unsigned) merge_sorted(
  ArrInt b, unsigned b_size,
  ArrInt c, unsigned c_size)
{
  ArrInt res;
  unsigned res_size;
  unsigned bi=0;
  unsigned ci=0;
  while(bi+ci<b_size+c_size-1)
  {
    assert(bi< b_size && ci< c_size);
    if(b[bi] > c[ci])
    {
      res[res_size] = c[ci];
      res_size++;
      ci++;
    }
    else
    {
      res[res_size] = b[bi];
      res_size++;
      bi++;
    }
  }
  return (res, res_size);
}
```

In practice, this part of the transformation does not need to occur during the transformation from programs to logical constraints but can directly occur on the set of logical constraints: the idea is to merge constraints of the form $\forall vars_1, P_1(in_1) \wedge \phi_1 \rightarrow P_2(out_1)$ and $\forall vars_1, P_2(in_1) \wedge \phi_2 \rightarrow P_3(out_2)$ where ϕ_1 and ϕ_2 are explicit expressions and $vars_1 \cap vars_2 = \emptyset$, into the constraint $\forall vars_1, vars_2, P_1(in_1) \wedge \phi_1 \wedge \phi_2 \rightarrow P_3(out_2)$, thus removing the use of P_2.

2.2 Horn clauses and Horn problems

2.2.1 Syntax: expressions, evaluation contexts and Horn clauses

We use the classical representation of expressions as trees where each node is either a constructor of the theory or logic in which we work or a variable. Our expressions are assumed well-typed and the only specificity is that we divide the set of symbols into two subsets: the predicates usually named $P, P', P_i, \ldots$ and the variables of a given theory.

Definition 1 (Expressions)**.** *An expression is simply a tree where each node is either:*

1. *a typed variable.*
2. *the application of a typed predicate to arguments of the correct type.*
3. *a constructor of the theory applied to correctly typed expressions.*

4. *a quantified typed variable applied to an expression.*

We define two evaluation contexts for expressions: models, usually written $\mathcal{M}, \mathcal{M}', \mathcal{M}_i$, which are functions that to a predicate P associate a set of the correct type; and an evaluation context for the free variables, usually written $vars, vars', vars_i$, which are functions associating mathematical values to the free variables. We use denotational semantics for expressions and the evaluation of an expression e in the evaluation context $vars$ for the free variables and the evaluation context $\mathcal{M}$ for the predicates, written $[\![e]\!]^{vars}_{\mathcal{M}}$, as described in Definition 2. If the evaluation is independent of a context, one can remove $\mathcal{M}$ or $vars$ and simply write, $[\![e]\!]_{\mathcal{M}}$ or $[\![e]\!]^{vars}$ or even $[\![e]\!]$. Furthermore, for a boolean expression e, we define $[\![e]\!]^{\forall}$ and $[\![e]\!]^{\forall}_{\mathcal{M}}$ which returns a boolean corresponding to whether e is true for all values of the free variables.

Definition 2 (Evaluation of expressions: $[\![e]\!]^{vars}_{\mathcal{M}}$)**.** $[\![e]\!]^{vars}_{\mathcal{M}}$ *is recursively defined by:*

1. $[\![v]\!]^{vars}_{\mathcal{M}} \equiv vars(v)$*, where* v *is a variable.*
2. $[\![P(e_1,\ldots,e_n)]\!]^{vars}_{\mathcal{M}} \equiv ([\![e_1]\!]^{vars}_{\mathcal{M}},\ldots,[\![e_n]\!]^{vars}_{\mathcal{M}}) \in \mathcal{M}(P)$*, where* P *is a predicate.*
3. $[\![C(e_1,\ldots,e_n)]\!]^{vars}_{\mathcal{M}} \equiv C([\![e_1]\!]^{vars}_{\mathcal{M}},\ldots,[\![e_n]\!]^{vars}_{\mathcal{M}})$*, where* C *is a constructor.*
4. $[\![\forall v, e]\!]^{vars}_{\mathcal{M}} \equiv \forall vval, [\![e]\!]^{vars_{vval}}_{\mathcal{M}}$ *where* $vars_{vval}(v) = vval$ *and* $vars_{vval}(v') = vars(v')$*. Existential quantifiers are evaluated in similar manner.*

The expressions on which we work are usually over a theory containing boolean, tuples, integers and arrays. However, our algorithms and proofs are general enough to account for any theory. Thus, we do not wish to restrict the theory we consider and one may assume we use the same theory for expressions as for formulae. Doing so enables us to go through broader theories to simplify our writing of expressions: for example, using set theory and considering that $\{\ldots\}$ is a constructor makes $[\![\{e_1,\ldots,e_n\}]\!]^{vars}_{\mathcal{M}}$ well-defined and this can be used to simplify $\{[\![e_1]\!]^{vars}_{\mathcal{M}},\ldots,[\![e_n]\!]^{vars}_{\mathcal{M}}\}$ while staying formal.

Although we may not constrain the expressions on which we work, the logical constraints generated from programs as described in Section 2.1 must be in the subset of expressions called Horn clauses. This subset is mainly defined by how predicates are used: the goal is to allow at most one *positive* predicate instance in each clause, that is at most one predicate which would not have a negation preceding it after expanding boolean operations. *Normalized* Horn clauses directly enforce such a syntax whereas *general* Horn clauses are simply expressions that can be reduced to normalized Horn clauses through boolean manipulations using the Tseitin transformation [Tse83]. We also define *extended* Horn clauses that correspond to Horn clauses with an unbounded number of predicate instances; *linear* Horn clauses which correspond to transitions from a single predicate to another and correspond to non-function call program instructions; *assertion* Horn clauses which encode safety properties. The satisfiability of a set of Horn clauses is simply the existence of a model – corresponding to the existence of $P_1,\ldots,P_n$ of Section 2.1.1 – that satisfies each of its clauses for all values of its free variables – as in our examples of Section 2.1 which have universal quantifiers around each clause.

Definition 3 (Horn clauses)**.** ***Normalized*** *Horn clauses are expressions of the form* $e_1 \wedge \ldots \wedge e_n \rightarrow e'$ *where:* $e_1,\ldots,e_n$ *are either a predicate application or an expression without predicates. We call* $e_1,\ldots,e_n$ *the* ***premises*** *and* e' *the* ***goal****.*

General *Horn clauses are clauses that can be rewritten into a conjunction of normalized Horn clauses by using only boolean manipulations. This transformation can always be done efficiently (i.e. in linear time) by introducing intermediate predicates by using the Tseitin transformation [Tse83].*

*Normalized **extended** Horn clauses allow $e_1, \ldots, e_n$ to be expressions of the form $\forall v, cond(e) \rightarrow P(e')$ where e, e' are expressions that may use the variable v. Normalized **linear** Horn clauses have at most one predicate instance in $e_1 \wedge \ldots \wedge e_n$. The linear Horn clauses that have no predicate instance in $e_1 \wedge \ldots \wedge e_n$ are said to be **start** Horn clauses. Normalized **assertion** Horn clauses are such that e' is an expression without predicates.*

Definition 4 (Satisfiability of Horn clauses)**.** *A set of Horn clauses $\mathfrak{C}$ is said to be satisfiable by $\mathcal{M}$ if and only if $\forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathcal{M}}$. It is said to be satisfiable if there exists a model such that $\mathfrak{C}$ is satisfiable by that model.*

2.2.2 Horn problems

Horn problems consist in taking an abstract view of the syntax of Horn clauses that is more suited for non-syntactical proofs. In this section, we first show how the problem of verifying safety properties on programs naturally redefines predicates and models, and create what we call *Horn problems*. We then explain that Horn clauses are a natural syntax for Horn problems and formally show in Theorem 2 that Horn clauses encode Horn problems.

2.2.2.1 Defining Horn problems

In this section, we work without Section 2.2.1 and we informally redefine predicates and models using a more semantical view. We urge the reader to understand that these redefinitions are equivalent and we will use the formal definitions of Section 2.2.1 in the rest of this manuscript.

To understand the formalization of Horn problems, let us define an abstract view of what kind of problem *the program verifies the safety properties* actually is instead of defining it through the syntax of the program or through the syntax of the logical formulae generated by the transformation of programs into logical constraints.

To formally define what a safety property is, we first need to define what a program point is and what a set of values for program points are. We define program points as simply typed identifiers, usually written $P, P_n, P', \ldots$, where the intuition is that the type of a program point P matches the type of the tuple of variables defined at program point P.

Sets of variable values for program points are called models and are written $\mathcal{M}, \mathcal{M}_n, \mathcal{M}', \mathcal{U}, \mathcal{U}', \ldots$ They are simply functions from program points to set of values. For example, $\mathcal{M}(P) = \{(0,2,4),(1,2,4)\}$ is a set of values for program points $\mathcal{M}$ such that the values of variables at program point P is 0 or 1 for the first variable, 2 for the second and 4 for the third.

Safety properties encode that the set of possible variable values at each program point must belong in a set of *allowed* values. Thus, it corresponds to the property $\forall P, \mathcal{M}_{prg}(P) \subseteq \mathcal{U}_{safety}(P)$, also written $\mathcal{M}_{prg} \leq \mathcal{U}_{safety}$ using a custom ordering, where $\mathcal{M}_{prg}$ is the model that corresponds to the set of possible values at each program point of the considered program and $\mathcal{U}_{safety}$ is a model that specifies the allowed values for each program point.

We have captured the notion of *a safety property for the set of possible variable values at each program point*. We now need to capture what it means that *that set of possible values is constructed by a program*. In other words, we need to capture the properties of $\mathcal{M}_{prg}$.

The set of possible values at each program point can be computed through a possibly unbounded symbolic execution of the program. Thus, if we call f_{prg} the function that executes one step of that unbounded symbolic execution, one may say that these sets and relations correspond to the least fixpoint of f, written lfp f. Thus, $\mathcal{M}_{prg}$ should in fact be defined as lfp f_{prg}.

Thus, we define Horn problems as pairs $(f_{prg}, \mathcal{U}_{safety})$ and we say that they are satisfiable whenever $\text{lfp}\, f_{prg} \leq \mathcal{U}_{safety}$. This was already briefly discussed in Section 2.1.1.

Definition 5 (Horn Problem H, defines $f_H, \mathscr{U}_H$)**.** *A Horn problem H is a pair $(f_H, \mathscr{U}_H)$ where:*

1. *f_H is a monotone function over models with order $\mathscr{M}_1 \leq \mathscr{M}_2 \equiv \forall P, \mathscr{M}_1(P) \subseteq \mathscr{M}_2(P)$.*
2. *$\mathscr{U}_H$ is a model.*

It is said to be satisfiable *if and only if* $\text{lfp}\, f_H \leq \mathscr{U}_H$ *(where* lfp *is the least fixpoint operator).*

2.2.2.2 Horn clauses encode Horn problems

Horn clauses encode Horn problems without using the least fixpoint operator. To do so, we formalize our discussion of Section 2.1.1, which states that we only need to check for the existence of what we call a postfixpoint.

Let us encode the property *H is satisfiable* as a logical formula without using the least fixpoint operator. Instead of saying that $\text{lfp}\, f_H \leq \mathscr{U}_H$, we instead write $\exists \mathscr{M}, \text{lfp}\, f_H \leq \mathscr{M} \leq \mathscr{U}_H$; and, using properties of the least fixpoint, this simplifies to $\exists \mathscr{M}, f_H(\mathscr{M}) \leq \mathscr{M} \leq \mathscr{U}_H$. Such a model is called a postfixpoint in the literature. Because this is the main property we will be using, we define $H(\mathscr{M})$ stating that the model $\mathscr{M}$ verifies the property $f_H(\mathscr{M}) \leq \mathscr{M} \leq \mathscr{U}_H$.

Definition 6 ($H(\mathscr{M})$)**.**

$$H(\mathscr{M}) \equiv f_H(\mathscr{M}) \leq \mathscr{M} \wedge \mathscr{M} \leq \mathscr{U}_H$$

Theorem 1 (Horn problems as a condition on models)**.** *A Horn problem H is satisfiable if and only if $\exists \mathscr{M}, H(\mathscr{M})$.*

Proof. We divide the proof in two parts: the main part H satisfiable $\equiv \exists \mathscr{M}, H(\mathscr{M})$ and the auxiliary lemma $f_H(\mathscr{M}) \leq \mathscr{M} \Rightarrow \text{lfp}\, f_H \leq \mathscr{M}$ used by the main part.

Proof that H satisfiable $\equiv \exists \mathscr{M}, H(\mathscr{M})$

H satisfiable
$\equiv \text{lfp}\, f_H \leq \mathscr{U}_H$
$\equiv \exists \mathscr{M}, \text{lfp}\, f_H \leq \mathscr{M} \leq \mathscr{U}_H$
($\Rightarrow$) By taking $\mathscr{M}$ equals lfp f_H
($\Leftarrow$) is proven in the auxiliary lemma
$\equiv \exists \mathscr{M}, f_H(\mathscr{M}) \leq \mathscr{M} \leq \mathscr{U}_H$
$\equiv \exists \mathscr{M}, H(\mathscr{M})$

Proof that $f_H(\mathscr{M}) \leq \mathscr{M} \Rightarrow \text{lfp}\, f_H \leq \mathscr{M}$

We prove that $x \leq \mathscr{M}$ implies $f_H(x) \leq \mathscr{M}$
lfp $f_H \leq \mathscr{M}$ follows by induction
$x \leq \mathscr{M}$
$\equiv f_H(x) \leq f_H(\mathscr{M})$ by monotonicity of f_H
$\equiv f_H(x) \leq \mathscr{M}$ by assumption $f_H(\mathscr{M}) \leq \mathscr{M}$

□

We now show that Horn clauses encode, for some H, $H(\mathscr{M})$, and thus, the satisfiability of Horn clauses encode $\exists \mathscr{M}, H(\mathscr{M})$. The main idea is that assertion Horn clauses define $\mathscr{U}_H$ whereas non-assertion Horn clauses define f_H. This view enables us to understand the restriction of the syntax of Horn clauses: f_H must be a monotone function, and this is why we cannot have several predicates instances in the goal.

Definition 7 (Horn problem encoded by a set of Horn clauses $H_{\mathfrak{C}}$)**.** *For a set of possibly extended Horn clauses $\mathfrak{C}$, we call $H_{\mathfrak{C}}$ a Horn problem encoding them*[1]*. That is $H_{\mathfrak{C}}$ verifies:*

$$\forall \mathscr{M}, H_{\mathfrak{C}}(\mathscr{M}) \equiv \forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathscr{M}}$$

Theorem 2 (Definition 7 is correct)**.** *Let $\mathfrak{C}$ be a set of possibly extended Horn clauses.*

$$\exists H, \forall \mathscr{M}, H(\mathscr{M}) \equiv \forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathscr{M}}$$

[1] The Horn problem is in fact unique, but we have not proved it as this is not useful in our manuscript.

Proof. Assume $\mathfrak{C}$ normalized (otherwise, simply normalize it using Tseitin). Let us first construct f_H and $\mathcal{U}_H$.

1. Let $\mathfrak{C} = \mathfrak{C}_1 \cup \mathfrak{C}_2$ where $\mathfrak{C}_1$ are the assertion clauses and $\mathfrak{C}_2$ are the non-assertion clauses.
2. Let us define $\mathcal{U}_H$ as the biggest model satisfying $Prop(\mathcal{M})$ where $Prop(\mathcal{M}) \equiv \forall C \in \mathfrak{C}_1, [\![C]\!]^{\forall}_{\mathcal{M}}$.
   Formally[2], $\mathcal{U}_H = \bigsqcup\{\mathcal{M} \mid \forall C \in \mathfrak{C}_1, [\![C]\!]^{\forall}_{\mathcal{M}}\}$, where $\bigsqcup$ is the join of the lattice. In other words, $\forall E, P, (\bigsqcup E)(P) = \bigcup\{E(P)\}$.
3. Let us define f_H such that $\forall P, f_H(\mathcal{M})(P) = \mathcal{M}(P) \cup \mathcal{M}_n(\mathcal{M})(P)$ where $\mathcal{M}_n(\mathcal{M})$ is the smallest model verifying $cond(\mathcal{M}, \mathcal{M}_n(\mathcal{M}))$ where $cond(\mathcal{M}, \mathcal{M}') \equiv \forall C \in \mathfrak{C}_2, \forall vars, [\![e_1 \wedge \ldots \wedge e_n]\!]^{vars}_{\mathcal{M}} \Rightarrow [\![e']\!]^{vars}_{\mathcal{M}'}$ using the notations of Definition 3: $e_1 \wedge \ldots \wedge e_n$ are the premises of C and e' its goal.
   Formally[3], $\mathcal{M}_n(\mathcal{M}) = \bigsqcap\{\mathcal{M}' \mid \forall C \in \mathfrak{C}_2, \forall vars, [\![e_1 \wedge \ldots \wedge e_n]\!]^{vars}_{\mathcal{M}} \Rightarrow [\![e']\!]^{vars}_{\mathcal{M}'}\}$, where $\bigsqcap$ is the meet of the lattice. In other words, $\forall E, P, (\bigsqcap E)(P) = \bigcap\{E(P)\}$.

We need to show that f_H is monotone and $\forall \mathcal{M}, H(\mathcal{M}) \equiv \forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathcal{M}}$. The monotonicity of f_H is simply because $cond$ is monotone in its first argument.

Proof of the monotonicity of f_H.

1. Let $\mathcal{M}_1 \leq \mathcal{M}_2$
2. Thus, $\forall C \in \mathfrak{C}_2, \forall vars, [\![e_1 \wedge \ldots \wedge e_n]\!]^{vars}_{\mathcal{M}_1} \Rightarrow [\![e_1 \wedge \ldots \wedge e_n]\!]^{vars}_{\mathcal{M}_2}$
3. Thus, for any $\mathcal{M}'$, $cond(\mathcal{M}_2, \mathcal{M}') \Rightarrow cond(\mathcal{M}_1, \mathcal{M}')$
4. Thus, $\mathcal{M}_n(\mathcal{M}_1) \leq \mathcal{M}_n(\mathcal{M}_2)$
5. Adding $\mathcal{M}_1 \leq \mathcal{M}_2$, we obtain $f_H(\mathcal{M}_1) \leq f_H(\mathcal{M}_2)$.

Proof that H encodes $\mathfrak{C}$. We need to prove $\forall \mathcal{M}, H(\mathcal{M}) \equiv \forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathcal{M}}$.

1. Introduce $\mathcal{M}$.
2. $H(\mathcal{M})$ can be rewritten into $f_H(\mathcal{M}) \leq \mathcal{M} \wedge \mathcal{M} \leq \mathcal{U}_H$.
3. Let us prove $\mathcal{M} \leq \mathcal{U}_H \equiv \forall C \in \mathfrak{C}_1, [\![C]\!]^{\forall}_{\mathcal{M}}$
   (a) $Prop$ is decreasing as $C \in \mathfrak{C}_1$ only uses predicate instances in the premises.
   (b) Reason by implication and assume $\mathcal{M} \leq \mathcal{U}_H$. We have $Prop(\mathcal{U}_H)$ as $\mathcal{U}_H$ is the biggest model satisfying $Prop$, thus, as $Prop$ decreasing, we have $Prop(\mathcal{M})$ our desired result.
   (c) Now assume $Prop(\mathcal{M})$. As $\mathcal{U}_H$ is the biggest model satisfying $Prop$, we have $\mathcal{M} \leq \mathcal{U}_H$.
4. Let us prove $f_H(\mathcal{M}) \leq \mathcal{M} \equiv \forall C \in \mathfrak{C}_2, [\![C]\!]^{\forall}_{\mathcal{M}}$
   (a) $f_H(\mathcal{M}) \leq \mathcal{M}$ is equivalent to $\mathcal{M}_n(\mathcal{M}) \leq \mathcal{M}$
   (b) As $\mathcal{M}_n(\mathcal{M})$ is the smallest model satisfying $cond(\mathcal{M}, \mathcal{M}_n(\mathcal{M}))$, $\mathcal{M}_n(\mathcal{M}) \leq \mathcal{M}$ is equivalent to $cond(\mathcal{M}, \mathcal{M})$
   (c) But $cond(\mathcal{M}, \mathcal{M})$ is exactly $\forall C \in \mathfrak{C}_2, [\![C]\!]^{\forall}_{\mathcal{M}}$
5. Thus, $H(\mathcal{M}) \equiv \forall C \in \mathfrak{C}, [\![C]\!]^{\forall}_{\mathcal{M}}$

□

2.3 A complex example: merge sort

In this section, we consider a more complex example: the merge sort program of Listing 2.9 and Listing 2.10. Listing 2.8 corresponds to the `merge_sorted` of Listing 2.6 with added program point names and safety properties. To our knowledge, sorting algorithms, and especially those with

[2] The proof that this corresponds to the biggest model satisfying $prop$ is due to its continuity.
[3] The proof that this corresponds to the smallest model satisfying $cond$ is due to its continuity.

Listing 2.8 – Array merge sorted

```
//pre: sorted(b) ∧ sorted(c)
//post sorted(res)
Array<int> merge_sorted(Array<int> b, Array<int> c)
{
  //Program point P_ms_start or P_ms_start_inl when inlined.
  Array<int> res;
  unsigned bi=0, ci=0; //Program point P_loop
  while(bi+ci<b.size()+ c.size()-1)
  {
    if(b[bi] > c[ci])
      { res.push_back(c[ci]); ci++; }
    else
      { res.push_back(b[bi]); bi++; }
  }
  //Program point P_ms_end or P_ms_end_inl when inlined.
  return res;
}
```

Listing 2.9 – Merge sort recursive

```
Array<int> merge_sort_rec(Array<int> a)
{
  if(a.size() >1)
  {
    unsigned mid=a.size()/2; //Program point P_b
    Array<int> b = sub_array(a, 0, mid);//Program point P'_b
    Array<int> c = sub_array(a, mid, a.size());
    b = merge_sort_rec(b); c = merge_sort_rec(c); //Program point P_call
    Array<int> res= merge_sorted(b, c); //Program point P'_call
    assert(sorted(res)); return res;
  }
  else
    return a;
}
```

Listing 2.10 – Merge sort non-recursive

```
Array<int> merge_sort_not_rec(Array<int> a)
{
  for(unsigned step=1; step<a.size();step*=2){
    unsigned i=0;
    Array<int> b, c;
    while(i<a.size())
    {
      unsigned istart=i, j;
      for(j=i;j<min(i+step, a.size());j++)
        b.push_back(a[j]);
      i=j;
      for(j=i;j<min(i+step, a.size());j++)
        c.push_back(a[j]);
      i=j;
      Array<int> res=merge_sorted(b, c);
      a.sub_array(istart, j) = res;
    }
  }
  assert(sorted(a));
  return a;
}
```

$nlogn$ complexity [MG16] have barely been tackled by automated tools in the literature, and merge sort is one of them.

This example is of particular interest as it manipulates arrays, requires complex universally quantified invariants, has a recursive and a non-recursive form and has function calls. The only interest of the non-recursive form is to show that such a program can be fully inlined and this will serve for the discussion of Chapter 6. Furthermore, we use the `sub_array(a, beg, end)` operation that we assume to be a language primitive extracting the subpart of the array corresponding to indices in $[beg, end[$. The Horn clauses we write for this example are created using Section 2.1 and the theory we use is that of Section 2.1.4. For readability reasons, our transformation assumes program integers to be infinite.

Instead of transforming the full program into Horn clauses, we divide program operations into several categories and select a single well-chosen program operation to transform for each category. For each category, we discuss whether the generated clauses are *linear, non-linear, assertion, start*[4] and whether the expressions on arrays they use are *trivial, basic, complex*. We say that clauses are trivial when there is no array operation, basic when the only operations are non-quantified array reads and writes, and complex otherwise. Furthermore, our discussion holds for the two ways of implementing $sorted$ and *BoundEq* discussed in Section 2.1.4: the semantics of both versions are identical and the difference will only be witnessed in a very specific algorithm[5].

The classification is given in Table 2.3 and the generated Horn clauses are given in Example 5. The important conclusion of this classification, which will be key for our discussion of Chapter 6, is that:

1. We must handle clauses which are *linear, basic*.
2. We must handle clauses that are *linear, assert, complex*.
3. We must handle clauses that are *linear, start, complex*.
4. For the non-recursive merge sort of Listing 2.10, using the inlining technique, we only need to additionally handle clauses that are *linear, trivial*. The inlining technique cannot be used for Listing 2.9 and this is why we provided Listing 2.10.
5. If we want to use function summaries, we need to additionally handle clauses that are *linear, complex*.
6. If we want to use modular function calls, we need to additionally handle clauses that are *non-linear, trivial*.

Example 5 (Horn clauses generated by merge sort)**.**

$$\begin{aligned}&\big(P_{loop}(b,b_s,c,c_s,res,res_s,bi,ci) \wedge bi+ci < b_s+c_s-1 \wedge \\ &(res',bi',ci') = ite(b[bi] > c[ci], (res[res_s \leftarrow c[ci]], bi, ci+1), (res[res_s \leftarrow b[bi]], bi+1, ci))\big) \\ &\qquad\qquad \rightarrow P_{loop}(b,b_s,c,c_s,res',res_s+1,bi',ci') \end{aligned} \tag{2.1}$$

$$\big(P_b(a,a_s,mid) \wedge BoundEq(b,a,0,mid) \wedge b_s = mid\big) \rightarrow P'_b(a,a_s,mid,b,b_s) \tag{2.2}$$

$$P'_{call}(a,a_s,mid,b,b_s,c,c_s,res,res_s) \rightarrow sorted(res,0,res_s) \tag{2.3}$$

$$(sorted(b,0,b_s) \wedge sorted(c,0,c_s)) \rightarrow P_{ms_start}(b,b_s,c,c_s) \tag{2.4}$$

[4] Note that none of the clauses are *extended* as the quantifiers are not over predicates.
[5] More precisely, the algorithm of Listing 5.6.

Table 2.3 – Classification of the clauses of Example 5 generated by program operations

Category	Example	Clauses	Type
Simple prg transitions[1]	Loop block of `merge_sorted`	2.1	linear, basic
Complex prg transitions[2]	`b = sub_array(a, 0, middle)`	2.2	linear, complex
Program point verification[3]	`assert(sorted(res));`	2.3	linear, assert, complex
Function verification[4]	Verifying `merge_sorted`	2.4	linear, start, complex
		2.5	linear, assert, complex
Inline function call[5]	`res= merge_sorted(b, c);`	2.6	linear, trivial
		2.7	linear, trivial
Modular function call[6]	`res= merge_sorted(b, c);`	2.8	linear, trivial
		2.9	non-linear, trivial
Summary function call[7]	`res= merge_sorted(b, c);`	2.10	linear, assert, complex
		2.11	linear, complex

[1] Program block transitions that use only simple operations on arrays that we illustrate with the complex loop block of `merge_sorted` already considered in Section 2.1.5.

[2] Program block transitions that use more complex operations on arrays that we illustrate with the call `b = sub_array(a, 0, middle)` of Listing 2.9. Note that this is very similar to using a function summary.

[3] The verification of program points that we illustrate with the call `assert(sorted(res));` of Listing 2.9.

[4] The verification of a function that we illustrate by checking that the precondition $\forall b, c, sorted(b) \wedge sorted(c)$ and the postcondition $\forall b, c, sorted(merge_sorted(b, c))$ is valid for `merge_sorted` .

[5] The handling of inline function calls that illustrated by the call `Array<int> res= merge_sorted(b, c);` of Listing 2.9.

[6] The modular handling of function calls that we illustrate with the call `Array<int> res= merge_sorted(b, c);` of Listing 2.9.

[7] The handling of function calls using summaries that we illustrate with the call `Array<int> res= merge_sorted(b, c);` of Listing 2.9 using the already verified precondition and postcondition.

$$P_{ms_end}((b, b_s, c, c_s), (res, res_s)) \rightarrow sorted(res, 0, res_s) \tag{2.5}$$

$$P_{call}(a, a_s, mid, b, b_s, c, c_s) \rightarrow P_{ms_start_inl}(a, a_s, mid, b, b_s, c, c_s) \tag{2.6}$$

$$P_{ms_end_inl}(a, a_s, mid, b, b_s, c, c_s, res, res_s) \rightarrow P'_{call}(a, a_s, mid, b, b_s, c, c_s, res, res_s) \tag{2.7}$$

$$P_{call}(a, a_s, mid, b, b_s, c, c_s) \rightarrow P_{ms_start}(b, b_s, c, c_s) \tag{2.8}$$

$$\big(P_{call}(a, a_s, mid, b, b_s, c, c_s) \wedge P_{ms_end}((b, b_s, c, c_s), (res, res_s))\big) \rightarrow P'_{call}(a, a_s, mid, b, b_s, c, c_s, res, res_s) \tag{2.9}$$

$$P_{call}(a, a_s, mid, b, b_s, c, c_s) \rightarrow (sorted(b, 0, b_s) \wedge sorted(c, 0, c_s)) \tag{2.10}$$

$$\big(P_{call}(a, a_s, mid, b, b_s, c, c_s) \wedge sorted(res, 0, res_s)\big) \rightarrow P'_{call}(a, a_s, mid, b, b_s, c, c_s, res, res_s) \tag{2.11}$$

3 Data-Abstraction: expectations

The Data-Abstraction framework is the key contribution of this manuscript. The goal of this framework is to simplify Horn problems that require models with unbounded data-structures – the equivalent of candidate invariants for programs – into Horn problems that do not require such models. Not all data-structures of all programs can be simplified in the same manner and our framework takes as parameter the user specifications for this simplification.

Obviously, our framework does not target all kinds of simplifications: its goal is to transform invariants with unbounded data-structures into invariants without, which usually is a simplification that loses information. We believe the right formalism to specify such simplification is *abstraction*: the transformation can be specified through semantics instead of through syntactic operations and the information loss is, at least partially, quantified. In our case, we only handle a subset of abstractions: abstractions that target unbounded data-structures and the formalism we choose for these abstractions, called *data-abstraction*, reflects that.

This chapter does not give an algorithm; instead, its purpose is to lay out the formalism, the core properties and abstractions the data-abstraction algorithm must handle, and to show that by doing so, the data-abstraction framework handles the unbounded data-structures of a broad class of programs. The work of actually finding the data-abstraction framework's algorithm such that it verifies these properties is left to Chapter 4.

This chapter is organized as follows: Section 3.1 states the properties of algorithms based on abstractions; Section 3.2 restricts the abstraction parameter to unbounded data-structures and gives the base of the *data-abstraction* formalism; finally, Section 3.3 shows the relevance and motivation of these properties and formalism by studying an important abstraction for arrays called *cell abstraction* [MG16].

3.1 Algorithm based on abstraction: definition & properties

3.1.1 Abstraction: definition

Abstraction in computer science is very similar to a natural process in everyday life: its a form of approximation. For example, instead of saying *I cut down the Oak and the Acacia of my garden*, one may instead say *I cut down the Trees of my garden*. In this example, one may say that *Oak and Acacia* was abstracted by *Trees* and one may see how the latter is less precise: unless the audience of the sentence already knows that I only have *Oak and Acacia* in my garden, they have less information.

While we abstract objects in sentences in everyday life, we rarely define how objects should be

abstracted: *Oak and Acacia* could also have been abstracted by *two things* in another sentence. In static verification of programs, our goal is to abstract the *concrete* properties of variables at a given program point - the candidate invariants - by simpler *abstract* objects, with the intention that finding the properties of interest is simpler with these abstract objects. In our case, the *concrete* properties will contain unbounded data-structures whereas the *abstract* objects will not. These concrete properties are unknowns of our program verification problem and thus we must not state how an individual concrete property needs to be abstracted, as was done for *Oak and Acacia*; but, instead, give a general rule for how they must be abstracted.

Such a rule is called an *abstraction* and, formally, is a function α that expresses by what something is abstracted. This abstraction function α is defined on $\mathscr{C}$, the set of things to abstract called *the concrete domain*, whose elements are called *concrete*, and α has value in $\mathscr{A}$, the set of things into which we approximate, called *the abstract domain*, whose elements are called *abstract* and will be noted with a # suffix. In our example, the abstraction which abstracts *Oak and Acacia* by *Trees* is different from the abstraction that abstracts *Oak and Acacia* by *two things*; the former may be viewed as the abstraction that abstracts a set of objects by the class of objects that contains them, whereas the latter may be viewed as the abstraction that approximates with the number of instances contained in that set.

The abstraction function α can't be just any function: it must have the properties of an approximation, that is, the approximation of a precise property such as *the value of that variable at this program point is* 2 must be contained within the approximation of an imprecise property containing it such as *the value of that variable at this program point is even*. Formally, the concrete domain $\mathscr{C}$ and the abstract domain $\mathscr{A}$ are given an order $<$ which represents this notion of precision of statements: $a < b$ means that b is less precise than a, or equivalently, that the behaviors of a are included in the behaviors of b. Furthermore, this order must have lattice properties: one must be able to take the least upper bound written $\sqcup$ and the greatest lower bound written $\sqcap$ of any subset of objects. A canonical example of lattices is sets where $<$ is represented by $\subset$, $\sqcup$ by $\cup$ and $\sqcap$ by $\cap$. For example, the property *the value of that variable at this program point is* 2 is simply represented by the set $\{2\}$ and the property *the value of that variable at this program point is even* is simply represented by the set $\{i | i \ mod \ 2 = 0\}$. These two properties are comparable and their comparison yields $\{2\} < \{i | i \ mod \ 2 = 0\}$.

The abstraction function α states how concrete objects are abstracted but not how abstract objects are related to concrete objects. In other words, it is not obvious what concrete properties an abstract object represents. Classically, we use a concretization function γ which, to an abstract object, gives the most precise concrete object to which it corresponds. Obviously, there is a link between α and γ and given one of them, the other one is unique. Such a couple (α, γ) is called a Galois connection [CC77], has a composition function $\circ$ which consists in chaining abstractions, and obeys many classical properties [RY20]. Here we recall the properties that we will use throughout this manuscript.

Definition 8 (Galois connection $\mathscr{G} = (\alpha, \gamma)$ and $\mathscr{G}_1 \circ \mathscr{G}_2$)**.** *A Galois connection $\mathscr{G} = (\alpha, \gamma)$ is defined between a concrete domain $\mathscr{C}$ and an abstract domain $\mathscr{A}$.*

- *$\alpha : \mathscr{C} \to \mathscr{A}$ gives the abstraction of a value.*
- *$\gamma : \mathscr{A} \to \mathscr{C}$ gives the concrete value of an abstract element.*

where:

1. *$\mathscr{C}$ and $\mathscr{A}$ are complete lattices*
2. *α, γ are monotone, a necessary property for approximations*
3. *$a^\# \leq \alpha(c) \equiv \gamma(a^\#) \leq c$, states the key property linking α and γ*
4. *$\gamma(a^\#) = \sqcup\{c | \alpha(c) \leq a^\#\}$, states that γ returns the most precise concrete element*

5. $\alpha \circ \gamma \circ \alpha = \alpha$ *and* $\gamma \circ \alpha \circ \gamma = \gamma$ *states that repeating the same abstraction does not change the result.*

Note that properties 1, 2 and 3 are sufficient to deduce the others.

The composition of two Galois connections $\mathscr{G}_1 = (\alpha_1, \gamma_1)$ *and* $\mathscr{G}_2 = (\alpha_2, \gamma_2)$ *is written* $\mathscr{G}_1 \circ \mathscr{G}_2$ *is* $(\alpha_1 \circ \alpha_2, \gamma_2 \circ \gamma_1)$.

To illustrate how the formalism of Definition 8 works on properties of variable values at program points, let us consider a few examples. Example 6 abstracts a property for a program point with only an integer variable by the signs that integer variable may have and is a classic didactic abstraction [CC77]. Example 7 abstracts a property for a program with a single integer variable by the best interval to which that variable belongs and is an efficient abstract domain used in abstract interpretation [CC76]. Example 8 abstracts a property for a program with a single array variable by the set of values the first cell of that array may have and is a didactic data-structure abstraction for this manuscript.

Example 6 (Integers to sign abstraction $\mathscr{G}_{sign} = (\alpha_{sign}, \gamma_{sign})$)**.** *We abstract a property over an integer variable, that is, a set of integers, thus* $\mathscr{C} = \mathscr{P}(\mathbb{Z})$*, by one of* $\{+, -, None, Both\} = \mathscr{A}$*, where:*

1. $+$ *states that the variable is always positive*
2. $-$ *states that the variable is always strictly negative*
3. $None$ *states that the variable has no value, this corresponds to an unreached program point*
4. $Both$ *states that the variable may be positive or negative*
5. *The lattice of* $\mathscr{C} = \mathscr{P}(\mathbb{Z})$ *is simply the set lattice.*
6. *The lattice of* $\{+, -, None, Both\} = \mathscr{A}$ *has partial order:* $None < +, - < Both$ *where* $+$ *and* $-$ *are not comparable.*

Formally, this corresponds to the following (α, γ) *Galois connexion.*

$$\begin{array}{lll|lll} \alpha_{sign}(S) = + & & if \forall x \in S, x \geq 0 & \gamma_{sign}(+) & = \{x | x \geq 0\} \\ \quad - & & if \forall x \in S, x < 0 & \gamma_{sign}(-) & = \{x | x < 0\} \\ \quad None & & if S = \emptyset & \gamma_{sign}(None) & = \emptyset \\ \quad Both & & otherwise & \gamma_{sign}(Both) & = \mathbb{Z} \end{array}$$

Example 7 (Integers to intervals abstraction)**.** *We abstract a property over an integer variable, that is, a set of integers, thus* $\mathscr{C} = \mathscr{P}(\mathbb{Z})$*, by the pair* (a, b) *representing the best interval* $[a, b]$ *that contains all those values, thus* $\mathscr{A} = \mathbb{Z}^2$*. Formally, this corresponds to the following* (α, γ) *Galois connexion.*

$$\alpha(S) = (min\ S, max\ S) \quad \Big| \quad \gamma(a, b) = [a, b]$$

Example 8 (Arrays to first cell abstraction)**.** *We abstract a property over an array variable, that is, a set of arrays, thus* $\mathscr{C} = \mathscr{P}(\mathbb{Z}^{\mathbb{Z}})$*, by only the first cell, thus* $\mathscr{A} = \mathscr{P}(\mathbb{Z})$*. Formally, this corresponds to the following* (α, γ) *Galois connexion.*

$$\alpha(S) = \{a[0] | a \in S\} \quad \Big| \quad \gamma(S^\#) = \{a | a[0] \in S^\#\}$$

The previous examples can be used to abstract program points. For example, consider the property $\{i | 2 \leq i \leq 12 \wedge i\%2 = 0\}$ for variable values at program point P in the program `int i = (rand()%5)*2; /*Program point Start*/ i <- i+2;/*Program point P*/` and apply the interval abstraction of Example 7. The result is the abstract value $(2, 12)$ representing the interval $[2, 12]$ and one can see we have lost the information $i\%2 = 0$.

In practice, one does not only abstract a single program point: in our example, it would not make sense to abstract the program point P without abstracting the program point $Start$, as P and $Start$ are intrinsically linked and this explains our choice to abstract all program points of

a program. However, the abstraction for each program point may be different, mainly because variables may come in and out of scope, and thus the abstraction, which involves all variables in scope, must change; but also because some specific locations of the program, such as an algorithmic function, may require a specific abstraction. Therefore, when considering a program, one should not consider a single abstraction, but a function that to a program point associates an abstraction for that program point.

The Horn problem counterpart to this program point abstraction scheme is an abstraction of models, the equivalent of the candidate invariant property for all program points. However, we only focus on abstractions of models that are defined by a function f_{abs} that to a predicate, the equivalent of a program point, associates the abstraction for that predicate. These abstractions $\mathcal{G}^{f_{abs}}$ define a subset of all models abstractions: the abstractions that abstract each predicate independently.

Definition 9 (Abstraction of models: $\mathcal{G}^{f_{abs}}$)**.** *For a function f_{abs} that to each predicate P associates an abstraction $(\alpha_P^{f_{abs}}, \gamma_P^{f_{abs}})$, one can construct the abstraction over models $\mathcal{G}^{f_{abs}} = (\alpha, \gamma)$ by:*

$$\alpha(\mathcal{M})(P^\#) = \alpha_P^{f_{abs}}(\mathcal{M}(P)) \qquad \gamma(\mathcal{M}^\#)(P) = \gamma_P^{f_{abs}}(\mathcal{M}^\#(P^\#))$$

where:

1. *The lattices over models has order $\mathcal{M}_1 \leq \mathcal{M}_2 \equiv \forall P, \mathcal{M}_1(P) \subseteq \mathcal{M}_2(P)$*
2. *The concrete domain of $\mathcal{G}^P$ is the set of models over predicates without an additional suffix #.*
3. *The abstract domain of $\mathcal{G}^P$ is the set of models over predicates with an additional suffix #.*

3.1.2 Solving algorithms: definition, *soundness* and *completeness*

The purpose of the Data-abstraction framework verification scheme is to give an algorithm that aims to determine if a given Horn problem, usually created from a program, is satisfiable or unsatisfiable.

Formally, we call these *solving algorithms* and, because they may produce incorrect answers, that is, output that a satisfiable (resp unsatisfiable) Horn problem is unsatisfiable (resp satisfiable), we avoid confusion between the satisfiability of a Horn problem and the output of the algorithm by naming the outputs *Certified* and *Buggy*. Thus, a solving algorithm for a set of Horn clauses $\mathfrak{C}$ is an algorithm from $\mathfrak{C}$ to $\{Certified, Buggy\}$.

One of our goals in this manuscript is to have a theoretical evaluation of our solving algorithms, and thus, we need properties to express that a solving algorithm is *good* or *bad*. The expectation is that a good algorithm answers *Certified* on satisfiable Horn problems and *Buggy* on non-satisfiable ones. From this expectation one can create a measure for how good a solving algorithm is: the ratio of correct to incorrect answers, inclusion of sets on which the algorithm returns correct answers, ... These natural measures handle the *Certified* and *Buggy* answers in symmetric manners, but, in static verification of programs, errors on the *Certified* answer are quite different from errors on the *Buggy* answer.

Because in the field of static verification of programs our goal is to *ensure the absence of bugs* and not *catch as many bugs as possible*, errors on the *Certified* answer are back-breaking: there is no more way to ensure the absence of bugs. This property of solving algorithms is called *soundness* and formally, it states that if the algorithm returns *Certified* then the input problem was satisfiable.

Errors on the *Buggy* answer are much less problematic. An algorithm that may return *Buggy* even though the input Horn problem is satisfiable just changes the meaning of the *Buggy* answer from *there is a bug* to *there may be a bug*. Thus, an algorithm with such errors, but with the soundness property, still allows us to ensure the absence of bugs for some programs: those on which it returns *Certified*.

However, a solving algorithm that returns *Buggy* on all inputs verifies our mandatory *soundness* property but is not a good algorithm. Thus, the *soundness* property is not enough, and as it is mandatory, the evaluation of solving algorithms must be based on the cases it answers *Buggy* on satisfiable Horn problems. It may be tempting to evaluate by using the size of the sets on which errors occur, but the drawback is that each of these errors appears to have identical weight which may be quite far from reality: some patterns may occur extremely frequently in programs whereas others may not, thus an algorithm which fails on many rare patterns but succeeds on a few common patterns may be better than an algorithm doing the reverse. A solution may be to add weights to the cases, but that would require a distribution for Horn problems which must be obtained experimentally and offers the same drawbacks as an experimental evaluation. Thus, as for now we do not have any further information about our Horn problems, we restrict ourselves to the property called *completeness* that states that there are no cases where the algorithm returns *Buggy* on satisfiable Horn problems. The *completeness* property will be refined once abstractions are introduced.

Definition 10 (Solving algorithm S, soundness, completeness)**.** *A solving algorithm S for a class of sets of Horn clauses $\mathbb{C}$, is an algorithm that for any set of Horn clauses $\mathfrak{C}$ of $\mathbb{C}$ returns either Certified or Buggy.*

Note that algorithms operate on syntax and this is why, this and future definitions use Horn clauses instead of Horn problems. However, in practice, these definitions hold for many other syntactical objects, mainly extended Horn clauses and programs.

Furthermore, for pratical reasons in formulae, we identify Certified with the boolean $True$ and Buggy with the boolean $False$.

The soundness and completeness of a solving technique are formally stated as:

1. *S is sound on $\mathbb{C}$ if and only if $\forall \mathfrak{C} \in \mathbb{C}, S(\mathfrak{C}) \Rightarrow H_{\mathfrak{C}}$ satisfiable.*
2. *S is complete on $\mathbb{C}$ if and only if $\forall \mathfrak{C} \in \mathbb{C}, \neg S(\mathfrak{C}) \Rightarrow H_{\mathfrak{C}}$ unsatisfiable, or equivalently, $\forall \mathfrak{C} \in \mathbb{C}, H_{\mathfrak{C}}$ satisfiable $\Rightarrow S(\mathfrak{C})$*

3.1.3 Solving algorithms based on abstraction: *relative completeness*

The definition of abstraction as a Galois connection indicates how the unknowns of our problems, that is the candidate properties for each predicate, are simplified. However, this definition does not directly relate to how solving algorithms, that aim to decide if a set of Horn clauses is satisfiable, must behave with respect to an abstraction and the properties they may satisfy. In this section, we formalize this link by stating that a solving algorithm is *based on abstraction* and we adapt the *completeness* property for such algorithms.

Abstractions define a loss of information and a solving algorithm based on an abstraction is simply a solving algorithm that reasons without that loss of information. In practice, this means that if the proof of correctness of a program depends on information that is lost by the abstraction, then a solving algorithm based on that abstraction cannot return *Certified*. For example, consider the program `a[3] <- 1;/*Program point P*/ assert(a[3] = 1);` and the program `a[0] <- 1;/*Program point P*/ assert(a[0] = 1);` and the first cell abstraction of Example 8 applied to program point P. After abstraction, the first program should not be provable whereas the second should. The first cell abstraction loses all information about cells with index different from 0, thus proving the first program requires information about $a[3]$ which is lost, whereas proving the second requires information about $a[0]$ which is kept.

The formalization of why the property $a[3] = 1$ is lost whereas the property $a[0] = 1$ is kept can be done by looking at what their abstractions represent. The abstraction of $a[3] = 1$ by the first cell abstraction of Example 8 is $\alpha(\{a|a[3] = 1\}) = \mathbb{Z}$ and the concrete representation of the abstract value $\mathbb{Z}$ is $\gamma(\mathbb{Z}) = \mathbb{Z}^{\mathbb{Z}}$. The concrete value $\mathbb{Z}^{\mathbb{Z}}$ represents the property $true$ and not the property $a[3] = 1$ and thus this property has been *lost*. However, the concrete representation of the abstraction of the property $a[0] = 1$ is $\gamma(\alpha(\{a|a[0] = 1\})) = \gamma(\{1\}) = \{a|a[0] = 1\}$ and the information has not been lost.

More generally, the information loss of a given concrete element e by an abstraction $\mathcal{G} = (\alpha, \gamma)$ can be measured by computing $\gamma \circ \alpha(e)$ as shown in Example 9. If $\gamma \circ \alpha(e) = e$, then there is no information lost and we say that e is *expressible by the abstraction* and is written $e \triangleleft \mathcal{G}$. In the literature, the set of elements *expressible by the abstraction* form a Moore family linked to the abstraction [CC79].

Solving algorithms based on an abstraction $\mathcal{G} = (\alpha, \gamma)$ are simply algorithms that can only return *Certified* on an input $\mathfrak{C}$ if $\mathfrak{C}$ is satisfiable by a model expressible by $\mathcal{G}$. In other words, algorithms based on an abstraction can only attempt to prove the correctness of a problem by using elements that are expressible by the abstraction.

Example 9 (Information loss of abstractions)**.**

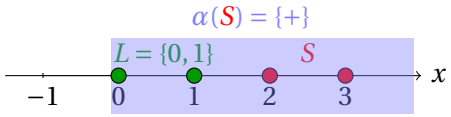


Figure 3.1 – **Information loss L for sign abstraction on $S = [2, \infty[$**: we lose information that $\{0, 1\} \notin S$, thus S is not expressible by the abstraction.

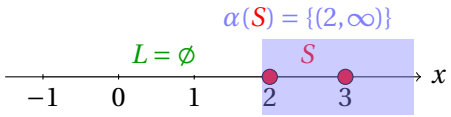


Figure 3.2 – **Information loss L for interval abstraction on $S = [2, \infty[$**: there is no information loss, S is expressible by the abstraction.

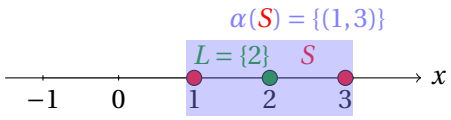


Figure 3.3 – **Information loss L for interval abstraction on $S = \{1, 3\}$**: we lose the information $\{2\} \notin S$. Thus, S is not expressible by the abstraction.

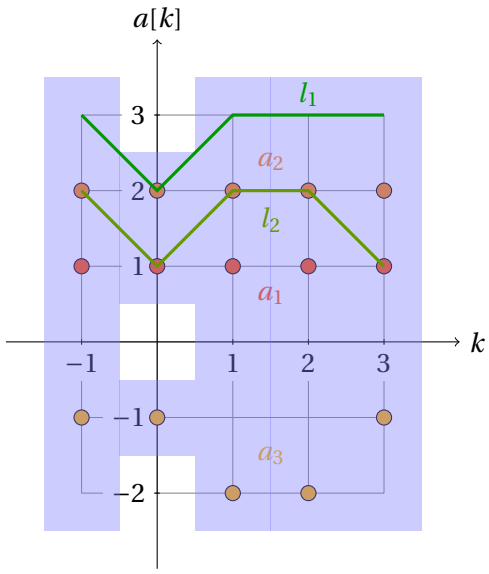


Figure 3.4 – **Information loss for first-cell abstraction on $S = \{a_1, a_2, a_3\}$:** the information loss is hard to depict, but arrays l_1 and l_2 belong to the information loss. Array l_1 shows the information loss due to added cells and array l_2 shows the information loss due to *forgetting the array from which the cell comes.*

Definition 11 (Element expressible by the abstraction)**.** *A concrete element e is said expressible by an abstraction $\mathcal{G} = (\alpha, \gamma)$ if and only if $\exists e^{\#}, e = \gamma(e^{\#})$, or equivalently, $e = \gamma \circ \alpha(e)$. This property is written $e \triangleleft \mathcal{G}$.*

Definition 12 (Solving algorithm based on an abstraction $\mathcal{G}$)**.** *A solving algorithm S is said based on abstraction $\mathcal{G}$ if and only if $\forall \mathfrak{C} \in \mathbb{C}, S(\mathfrak{C}) \Rightarrow (\exists \mathcal{M} \triangleleft \mathcal{G}, H_{\mathfrak{C}}(\mathcal{M}))$*

The consequence of the information loss of abstraction is that completeness of a solving algorithm based on a interesting abstraction cannot be achieved: a sound algorithm which is also complete decides the non-abstracted problem, and thus, no relevant information has been lost. We thus adapt the completeness property that we use for solving algorithms to solving algorithms based on abstraction.

The adaptation is straightforward: we follow the same methodology and add the information that our algorithm is abstraction based and thus can only return *Certified* on problems that have models expressible by the abstraction. Thus, *no errors* becomes *there are no cases of Horn problems satisfiable by a model expressible by the abstraction on which the algorithm returns Buggy* instead of considering all Horn problems. This property is called *relative completeness* and we say that a solving algorithm is complete relative to an abstraction $\mathscr{G}$.

Definition 13 (Relative completeness of a solving algorithm)**.** *A solving algorithm S is said complete relative to an abstraction $\mathscr{G}$ if and only if* $\forall \mathfrak{C} \in \mathbb{C}, (\exists \mathscr{M} \lhd \mathscr{G}, H_{\mathbb{C}}(\mathscr{M})) \Rightarrow S(\mathfrak{C})$.

In many solving algorithms to handle static verification of programs, completeness, relative completeness or any other kind of theoretical measure of effectiveness of solving algorithms is not considered. Thus, the proof of effectiveness of these algorithms relies on benchmarks, which, considering the quantity of heuristics used for these undecidable problems is often unconvincing: it is extremely difficult to judge if the algorithm is successful thanks to specific tuning of the heuristics for these benchmarks or if these algorithms would fare well in real world applications: in many cases, the benchmarks are small, proof-of-concept programs and not fully grown applications.

Perhaps one of the main reasons these properties of solving algorithms, *soundness, completeness and relative completeness*, are not considered is that they are hard to satisfy. These algorithms are required to be sound and thus, should they be complete, they would decide the static analysis problem. However, this is impossible for all interesting classes of programs: even the class of programs with only integers and loops is undecidable [Ric53]! As for relative completeness, we believe the question has long been in the mind of the community, however, the only name we have found in the literature [Fij+19] for it is the *Monniaux problem*.

For example, solving algorithms complete relative to the interval abstraction, or even any fixed polyhedral abstraction – here, we mean the polyhedral abstract domains from abstract interpretation [CH78] –, exist [MS04], mainly because the abstract domain is simple enough that the existence of an abstract invariant can be expressed in first-order logic over integers; and thus, is decidable. In the case of the general polyhedral abstract domain [CH78], it is still unknown whether the problem is decidable: the main difference compared to the fixed polyhedral domains is that the number of faces of the polyhedral is unbounded, thus, convergence is no longer assured; however, it is as yet unknown whether this added difficulty makes the problem undecidable.

In our case, our goal is to extend current solving algorithms so that they may handle unbounded data-structures by using a parametrized abstraction for them. The problem is that even the non-extended solving algorithm cannot be *complete* on non-datastructure problems as they handle integers, and, thus, our extended solving algorithm cannot be complete relative to the chosen abstract domain for data-structures: consider an extension of a solving algorithm to handle arrays with the *first cell array abstraction* of Example 8 and a class of programs with integers, arrays and loops. Even if the extension handles the arrays correctly, the leftover problem is still undecidable and thus, the extended solving algorithm cannot be complete relative to the *first cell array abstraction*. Therefore, properties of solving algorithms are not the right properties to measure the effectiveness of an extension.

3.1.4 Working around undecidability: transformations *implementing the abstraction*

Attempting to extend a solving algorithm such that it may handle unbounded data-structures and satisfy properties such as *relative completeness* is the wrong way to go: integer problems are already too hard, thus, even if our extension handles the unbounded data-structures optimally with respect to the chosen abstraction, the extended algorithm will not satisfy *relative completeness* as it would require deciding problems at least as hard as integer problems.

However, we do not believe that evaluating extensions is vain: consider the *first cell array abstraction* of Example 8. This abstraction is extremely simple and seems straightforward to handle: as we only care about the first cell of the array, let us consider arrays as just a single integer value representing the first cell and replace all array writes (resp reads) such that when the index is the first cell, the integer is modified (resp read) and if not, the operation is removed (resp returns a random value). This transformation on the array initialization program of Listing 1.1 is explained in Example 10 and seems perfect: there is no information loss except for the one due to the abstraction and one can use another solving algorithm to handle the resulting integer problem. We would like to say that this transformation *implements the abstraction*, but, for now, we lack the formalism.

Example 10 (Transformation of the array initialization program of Listing 1.1 by the *first cell array abstraction* of Example 8)**.**

Listing 3.1 – Before abstraction

```
void array_init(Array<int> a)


{
  unsigned i=0;
  while(i<a.size())
  {
    a[i] <- 0;
    i <- i+1;
  }
  //Safety property
  i=0;
  while(i<a.size())
  {
    assert(a[i]=0);

    i <- i+1;
  }
}
```

Listing 3.2 – After abstraction

```
void array_init(
    int a/*the abstract array*/,
    unsigned n /*the size of a*/)
{
  unsigned i=0;
  while(i<n)
  {
    if(i=0) then a <- 0;
    i <- i+1;
  }
  //Safety property
  i=0;
  while(i<n)
  {
    if(i=0) then assert(a=0);
    else {int rnd=rand();assert(rnd=0);}
    i <- i+1;
  }
}
```

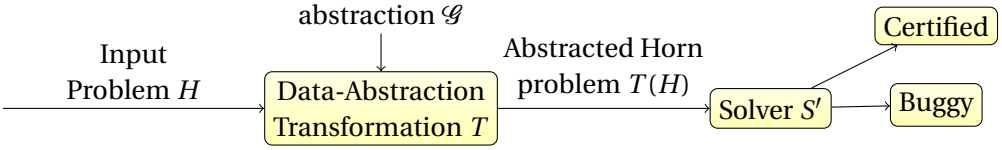
After abstraction, for an array with at least two cells, the program is incorrect. This is to be expected as the necessary invariant after the initialization loop $\forall k < a.size(), a[i] = 0$ *is not expressible by the abstraction: this invariant cares about multiple cells whereas the abstraction only cares about the first.*

Let us take a closer look at what we did in Example 10: we considered a solving algorithm based on the *first cell array abstraction* of Example 8 that first transforms our program with arrays into an integer program and then uses an algorithm of our choice to solve the integer program. In other words, our solving algorithm S is the composition of a transformation T with a back-end solving

algorithm of our choice S'; we write $S = S' \circ T$. Furthermore, in Example 10 the transformation verifies a *relative completeness* property adapted to transformations: the output program is satisfiable if and only if the initial program is satisfiable by a model expressible by the abstraction.

Inspired by this example, our Data-abstraction framework aims at being an extension of solving algorithms for unbounded data-structures by being a transformation as shown in Figure 3.5. The effectiveness of the transformation can be stated by properties linking the satisfiability of the output problem and the satisfiability of the input problem. This is unlike approaches that attempt to improve current solving algorithms, such as interpolation or abstract interpretation based algorithms, for which it seems extremely hard to correctly formalize a property stating *the extension does its job well.*

Figure 3.5 – Solving algorithm S induced by the Data-abstraction framework



We adapt the soundness and completeness properties to transformations and we will say that a transformation algorithm satisfying both soundness and relative completeness *implements the abstraction.* Let us stress that a transformation algorithm that *implements an abstraction* is a transformation whose information loss is precisely the abstraction. In other words, the approximation made by an algorithm implementing an abstraction is **exactly** described by the abstraction and is the best one available for abstraction based transformations. The property of *implementing the abstraction* **is the target property of our work**.

Definition 14 (Transformation algorithm and their properties)**.** *A transformation algorithm* $T : \mathbb{C} \to \mathbb{C}'$ *is an algorithm that takes an input a set of Horn clauses* $\mathfrak{C} \in \mathbb{C}$ *and outputs a set of Horn clauses* $\mathfrak{C}' \in \mathbb{C}'$.

As with solving algorithms, we state the formal definition for sets of Horn clauses, but this definition can be adapted to other syntactical objects, mainly extended Horn clauses and programs.

Such a transformation T *is said:*

1. *Sound on* $\mathbb{C}$ *if and only if:* $\forall \mathfrak{C} \in \mathbb{C},\ H_{T(\mathfrak{C})}$ *satisfiable* $\Rightarrow\ H_{\mathfrak{C}}$ *satisfiable.*
2. *Complete on* $\mathbb{C}$ *if and only if:* $\forall \mathfrak{C} \in \mathbb{C},\ H_{T(\mathfrak{C})}$ *unsatisfiable* $\Rightarrow\ H_{\mathfrak{C}}$ *unsatisfiable.*
3. *Based on an abstraction* $\mathcal{G}$ *for* $\mathbb{C}$ *if and only if:* $\forall \mathfrak{C} \in \mathbb{C}, H_{T(\mathfrak{C})}$ *satisfiable* $\Rightarrow \exists \mathcal{M} \lhd \mathcal{G}, H_{\mathfrak{C}}(\mathcal{M})$.
4. *Complete on* $\mathbb{C}$ *relative to an abstraction* $\mathcal{G}$ *if and only if:* $\forall \mathfrak{C} \in \mathbb{C}, \exists \mathcal{M}, \mathcal{M} \lhd \mathcal{G} \wedge H_{\mathfrak{C}}(\mathcal{M}) \Rightarrow H_{T(\mathfrak{C})}$ *satisfiable.*
5. *To implement the abstraction* $\mathcal{G}$ *on* $\mathbb{C}$ *if and only if the transformation* T *is based the abstraction* $\mathcal{G}$ *and is sound and complete relative to* $\mathcal{G}$*. This yields the formula:* $\forall \mathfrak{C} \in \mathbb{C}, H_{T(\mathfrak{C})}$ *satisfiable* $\equiv \exists \mathcal{M}, \mathcal{M} \lhd \mathcal{G} \wedge H_{\mathfrak{C}}(\mathcal{M})$.

3.2 Data-abstraction: a specific subset of abstractions

The purpose of this manuscript is to tackle the problem of finding models for Horn problems with unbounded data-structures. To do so, we suggest a solving technique which first consists in a transformation to abstract these data-structures and then use a back-end solver to handle the abstracted problem. Abstraction of unbounded data-structures is a specific subset of abstractions

and we need to correctly frame its specificity in order to write a transformation algorithm that implements the abstraction. In this section, we capture and formalize that specificity.

In program verification we abstract properties of variable values at program points, that is, **sets** of values. However, the intuition for abstractions of data is that one should only need to specify how a **single** value is abstracted: one expects to specify how an array is abstracted, and, from that, deduce how a property over arrays is abstracted. The sentence above captures exactly one of the specificities of unbounded data-structure abstraction: the abstraction is defined for each value of a data-structure independently, whereas general abstraction expresses how a set of data-structure values is abstracted. With this idea in mind, one can revisit Examples 6, 7 and 8.

Inherently, the *sign abstraction* of Example 6 could be defined by how a single integer value is abstracted by its sign. In other words, each integer of the set is abstracted by its sign. The same is true of the *first cell abstraction* of Example 8, where each array is abstracted by its first cell. However, this is not the case of Example 7, one cannot define the interval abstraction of a set from its individual elements: the abstraction of any of the single elements would be itself; it is the fact of abstracting several of them at the same time which gives meaning to the abstraction.

We thus distinguish two types of abstractions: abstractions of data that can be defined as the union of how each individual element is abstracted, and *shape abstractions* that do not aim to abstract individual values but rather usually aim to give a shape, such as intervals, to a set of values. Our Data-Abstraction framework targets abstractions of data and thus, the abstraction for sets of values will be defined from it. The model abstraction for a predicate can then be deduced using Definition 9.

A second specificity is that we do not target just any data-structure, but unbounded or infinite data-structures. Thus, the *sign abstraction* of Example 6 is not entirely relevant to our framework, but the *first cell abstraction* of Example 8 is. In the latter example, we abstract an array by a single cell, the cell with index zero. However, this has drastic expressivity limitations and unless the safety property only cares about $a[0]$, the process is doomed to fail. Natural extensions of such an idea could be to abstract a not only by $a[0]$, but perhaps by a specific finite subset $a[0], a[x], a[y], \ldots$. In fact, one may push the idea further and say that in an unbounded data-structure, perhaps all elements of this unbounded data-structure are relevant, thus no finite subset may suffice. We drop the finite limitation and say that an unbounded data-structure abstraction abstracts a given element, such as an array a, by a set of abstract elements, for example $\{a[0], a[1], \ldots\}$.

Formally, an unbounded data-structure abstraction is defined by a function $\sigma : C \to \mathscr{P}(A)$ which describes how a single data-value is abstracted into multiple abstract values. From the function σ, one may deduce the abstraction of set of values $\mathscr{G}_\sigma$, that is, how a single predicate is abstracted. In our Horn clauses setting, our final objective is to abstract entire models and not individual predicates. Thus, we use Definition 9 with the function $\sigma^{abs} = \text{fun}\, P \to \mathscr{G}_{abs(P)}$, where abs is a function that, to a predicate, returns the data-abstraction to use for that predicate.

Definition 15 (Data abstraction $\sigma, \mathscr{G}_\sigma, \sigma^{abs}$)**.** *Let $\mathscr{C}$ and $\mathscr{A}$ be sets. A data abstraction σ is a function from $\mathscr{C}$ to $\mathscr{P}(\mathscr{A})$. It defines a Galois connection $\mathscr{G}_\sigma = (\alpha_\sigma, \gamma_\sigma)$ with concrete domain $\mathscr{P}(\mathscr{C})$ and abstract domain $\mathscr{P}(\mathscr{A})$ by:*

$$\alpha_\sigma(S) = \bigcup_{a \in S} \sigma(a) \qquad \gamma_\sigma(S^\#) = \{a \in \mathscr{C} \mid \sigma(a) \subseteq S^\#\}$$

A data-abstraction of models is a function abs that to each predicate P associates a data-abstraction $abs(P)$. It defines a model abstraction function $\sigma^{abs} = \text{fun}\, P \to \mathscr{G}_{abs(P)}$; and using Definition 9, an abstraction of models $\mathscr{G}^{\sigma^{abs}} = (\alpha_{\mathscr{G}^{\sigma^{abs}}}, \gamma_{\mathscr{G}^{\sigma^{abs}}})$ with:

1. $\alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^\#) = \alpha_{abs(P)}(\mathscr{M}(P)) = \bigcup_{a \in \mathscr{M}(P)} abs(P)(a)$
2. $\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)(P) = \gamma_{abs(P)}(\mathscr{M}^\#(P^\#)) = \{a \mid abs(P)(a) \subseteq \mathscr{M}^\#(P^\#)\}$

To illustrate the formalism of Definition 15, we suggest to revisit a few classical examples. Example 11 revisits our *first cell abstraction* of Example 8 within the data-abstraction formalism. Example 12 considers the classical *array smashing abstraction* [GRS05] which consists in abstracting an array by the set of values contained in that array. Properties of the values contained in the array, such that $\forall i, 0 \leq a[i] \leq 10$, are expressible with this abstraction, but not properties linking indices to values, such as $\forall i, a[i] \leq i$. Example 13 considers a simple variant of the widely used *slice abstraction* [GRS05; CCL11]. There are many variants of the slice abstraction, but the foundation of this abstraction is to divide an array into several parts which are treated uniformly. One of the most common uses is for loops of the form `for(i=0; i<n;i++)` : one divides the array into three slices $[0, i[$, the part of the array that has already gone through the loop; $\{i\}$, the slice representing only the current index and thus allows strong updates; and $]i, n[$ the part of the array that has not already gone through the loop. With this abstraction, properties which are different for the first slice $[0, i[$, that is, the part of the array that has already been passed through the loop and the combination of the second and third slice $[i, n[$, that is, the part of the array that has not been yet handled, can be stated separately. In our simple variant, each slice will be handled as if we had done the smashing abstraction. Such an abstraction can prove properties such as $\forall k < i, a[k] < 50 \wedge \forall k \geq i, a[k] < 100$ which could be necessary for a loop which divides array values by two.

Example 11 (Arrays to first cell data-abstraction)**.** *The first cell abstraction of Example 8 can be defined as $\mathcal{G}_{\sigma_{[0]}}$ where $\sigma_{[0]}$ is the data-abstraction defined by*

$$\sigma_{[0]}(a) = \{a[0]\}$$

Example 12 (Array smashing data-abstraction)**.** *Array smashing [GRS05] consists in abstracting an array by the values of all of its cells values, thus*

$$\sigma_{smashing}(a) = \{a[i] \mid i \in \mathbb{Z}\}$$

Example 13 (Simple array slicing data-abstraction)**.** *Array slicing partitions the array in several slices, that is, continuous subarrays and then, in our simple case, we handle each of these subarrays as if we had done smashing.*

Finding a good partitioning of the array is a non-trivial problem but a common case is for loops such as `for(i=0; i<n;i++)` and an array a partitioned into three slices: the slice of already handled indices $]0, i[$; the slice of the current index $\{i\}$; the slice of the future indices of the loop to handle $]i, n[$. Thus, the abstraction is dependent on the variables i and n and can be defined as:

$$\sigma_{slice}(a, i, n) = \{((a[k_1], a[k_2], a[k_3]), i, n) \mid 0 \leq k_1 < i \wedge k_2 = i \wedge i < k_3 < n\}$$

The previous examples explain how a data-structure is abstracted. However, in practice, one wishes to abstract values that have the type of the tuple of all defined variables at the given program point, or, in Horn terms, values that match the predicate type. First, most of these variables are non-data-structure values that do not need to be abstracted; secondly, there might be several data-structure values that need to be abstracted with possibly different abstractions. Our solution to these problems is to give an identity abstraction for variables that do not need to be abstracted and an *abstraction combinator* written $\bullet$ which takes two abstractions and creates an abstraction of pairs where each element is abstracted by its abstraction.

We also add other abstraction combinators which can be used to create new interesting abstractions from existing abstractions. The most important of them is the composition combinator which takes two abstractions and returns the abstraction that corresponds to chaining the two abstractions. Other combinators will added later on in Definition 18.

Definition 16 ($\sigma_{id}, \sigma_1 \bullet \sigma_2, \sigma_1 \odot \sigma_2$)**.** *We define*

1. *The no abstraction abstraction by* $\sigma_{id}(x) = \{x\}$
2. *The pair abstraction by* $\sigma_1 \bullet \sigma_2(x_1, x_2) = \sigma_1(x_1) \times \sigma_2(x_2)$
3. *The composition abstraction by* $\sigma_1 \odot \sigma_2(x) = \bigcup_{x^\# \in \sigma_2(x)} \sigma_1(x^\#)$

3.3 Cell abstraction: an interesting Data-abstraction for arrays

In this section, we formalize the *cell-abstraction* of [MG16] in our framework and justify why this abstraction is so important. The contribution in this section is not in the abstraction itself, but in its formalism within the data-abstraction framework, which provides new abstractions by using combinators and Horn problem transformation algorithms with proven properties.

3.3.1 Properties of a good array abstraction

3.3.1.1 General properties of a good data-abstraction

Before we dive in and explain what *cell-abstraction* is, let us first understand the characteristics of a good abstraction.

The expressiveness trade-off property. First, there needs to be a good trade-off between how much an abstraction simplifies the verification problem and the size of the class of programs one may wish to use this abstraction on[1]. For example, the abstraction the *first cell abstraction* of Example 11 simplifies the problem greatly, but is barely useful: it only cares about the first cell of the array. However, the *smashing* abstraction of Example 12 is already much more useful and has the same abstract domain; thus, the problems generated after transformation may not be much harder to solve than with the *first cell abstraction*. As for the *slice* abstraction of Example 13, it is even more expressive and handles a much broader class of programs if the slices are chosen wisely and seems to only linearly complexify *smashing*: it uses three integers instead of one. Under this metric, one may say that slicing is much better than the other two abstractions.

The composability property. Secondly, in our data-abstraction framework, abstractions can be combined; for example by using the composition combinator. This means that abstractions that can be used as a good basis to construct other abstractions, usually with a different expressiveness/simplification trade-off, are of particular interest. For example, the *slice* abstraction of Example 13 can be composed with the abstraction that mixes the three slices, defined by $\sigma(v_1, v_2, v_3) = \{v_1, v_2, v_3\}$, to create the *smashing* abstraction of Example 12. However, one cannot use the *slice* abstraction of Example 13 or the *smashing* abstraction of Example 12 to create the *first cell abstraction* of Example 11. Perhaps, a better abstraction with respect to this framework would do so and thus would allow users to combine abstractions to create a very simple abstraction when all they need is the first cell.

The predictability property. Thirdly, and perhaps more importantly, a good abstraction should make it straightforward for the user to know whether his problem belongs to the class of problems the abstraction targets: even though the user is not expected to know the exact program invariant

[1] The same trade-off appears when considering abstract domains for integers, for example, intervals versus octagons versus convex polyhedra.

– otherwise, why even bother with a tool that helps finding them – he is expected to know enough to correctly chose the input abstractions for the Data-abstraction framework. This means that the class of programs targeted by the abstraction should not depend on details that the user cannot easily predict. For example, in the *slice* abstraction of Example 13, the slices are guided by the syntactical analysis of loop bounds, and thus a programmer that uses $a[i+1]$ instead of $a[i]$ within the loop changes the loss of information of the abstraction from irrelevant to relevant; thus the *slice* abstraction is perhaps not the best in this regard.

3.3.1.2 The class of Horn problems a good abstraction should handle

Example 14 (Properties of container libraries algorithms)**.** *The main container property to find is the set of values restricting the array values at the beginning of the loop of the main algorithm.*

Category	*Algorithm*	*Main container property to find*
Init	*Constant value*(a,v)	$\forall k<i, a[k]=v$
	Identity(a)	$\forall k<i, a[k]=k$
Modifying	$b=$*Copy*(a)	$\forall k<i, b[k]=a[k]$
	$b=$*Insert*(a,pos,v)	$\forall k<i, k<pos \Rightarrow b[k]=a[k]$ $\wedge k=pos \Rightarrow b[k]=v$ $\wedge k>pos \Rightarrow b[k]=a[k-1]$
	$b=$*Rev*(a)	$\forall k<i, b[k]=a[a.size()-1-k]$
Finding	$r=$*Linear search*(a,v)	$(a[r]=v \wedge \forall k<r, a[k]\neq v)\vee$ $((r=-1)\wedge \forall k<i, a[k]\neq v)$
	Binary search(a,v)	$\forall k<lower, a[k]<v \wedge \forall k\geq upper, a[k]>v \wedge$ $s(a,lower,upper)$
	Max(a)	$\forall k<i, a[k]\leq a[max]$
Sorting	*Bubble sort*(a)	$change=false \Rightarrow s(a,0,a.size())$
	$b=$*Insertion sort*(a)	$s(b,0,b.size())\wedge$ $\forall k<lower, b[k]<a[curr] \wedge \forall k\geq upper, b[k]>a[curr]$
	$c=$*Merge sorted*(a,b)	$s(c,0,i_a+i_b)\wedge s(a,i_a,a.size())\wedge s(b,i_b,b.size())$ $\wedge\forall k<i_a+i_b, c[k]\leq a[i_a]\wedge c[k]\leq b[i_b]$
	Merge sort(a)	*Property given by Mergesorted*
	Quick sort(a)	$s(b,0,b.size())\wedge s(c,0,c.size())\wedge$ $\forall k\leq b.size(), b[k]\leq pivot \wedge \forall k\leq c.size(), c[k]>pivot$
Fold	$v=sum(a)$	$v=\sum_0^i a[i]$
	$c=count(a,val)$	$c=card\ \{k \mid k\leq i_a \wedge a[k]=val\}$

1 $s(a,lower,upper)$ *should expand to* $\forall k_1,k_2, lower\leq k_1<k_2<upper \Rightarrow a[k_1]\leq a[k_2]$.
2 $card\ S$ *denotes the number of elements in the set* S.
3 *For conciseness reasons, integer related bounds such as* $0\leq k$ *or* $k\leq a.size()$ *are not always stated.*
4 *Variable notations are the following:* a,b,c *denote array variables,* i,i_a,i_b *denote loop indices,* k,k_1,k_2 *are quantified variables used to express properties. In some algorithms, the main container property uses locally scoped variables such as* $pivot$.
5 *We do not necessarily display the property for the full specification of the algorithm: for example, the full specification of sorting algorithms should also state that the multiset of values has been left unchanged.*
6 *Algorithms that are just slight modifications of already given algorithms have been omitted: for example, the property for* min *can be deduced from the property given for* max.

The predictability property of a good abstraction requires knowing the types of programs and

properties we aim to handle with an array abstraction. In this PhD, we target algorithms that form the basis of container librairies and thus we need an abstraction that makes invariants of such algorithms expressible. Example 14 shows the most common container algorithms with the properties we need to express.

There are a few things to notice in Example 14. First, all properties involve an unbounded number of cells of the array; thus, abstractions that consider only a fixed finite number of cells, such as the *first cell abstraction* of Example 8, are not expressive enough. Secondly, except for fold operations and constant value, these properties care about how indices are linked to values: initialization needs to link the index with its value, copy needs to link the index of the array a with the index of the array b, ... Thus, abstractions based on the *smashing* of all cells cannot work and thus the *slice* abstraction of Example 13 does not work either.

Not all slice based abstractions require each slice to be abstracted as if we had done smashing as in Example 13. The general principle of slices is to divide the array into a finite number of multiple continuous segments that are handled uniformly and the way they are handled can be parameterized [CCL11]. In our data-abstraction framework, slice abstraction can be viewed as an abstraction that divides an array into multiple subarrays – in Example 13, this corresponds to the abstraction $\sigma(a) = \{(a_1, a_2, a_3) | a_1 = a[0 \dots i[\wedge a_2 = a[i \dots i] \wedge a_3 = a] i \dots a.size()[\}$ – composed with an abstraction that handles each of these arrays uniformly, for example smashing. Thus, a proper slice abstraction can be used to prove any of the non-fold algorithms: one can cut the slices such that they correspond to the bounds of the quantifiers k, k_1, k_2 of Example 14 and then use an abstraction that can express the property stated after the quantifier.

The whole problem with the slice abstraction is thus finding a proper slicing of the array and the correct abstraction for each slice that allows to express the desired property. In practice, asking the user to specify the right slices and the right abstraction is much too complex, and no automatic tool currently manages to correctly handle our algorithms; thus, we do not believe slice based abstractions to be the way to go, especially once taking into account the reliance on syntax most of these automatic tools have.

3.3.2 Cell-abstraction: possibly the right abstraction

3.3.2.1 Cell-abstraction: definition

Cell abstraction [MG16] consists in viewing arrays through their relationship between k and $a[k]$. This allows properties such as $\forall k < i, a[k] = v$ and $\forall k < i, a[k] = k$ to be easily stated in the abstract domain, but not properties such as $sorted(a, 0, n) \equiv \forall k_1, k_2, 0 \leq k_1 < k_2 < n \Rightarrow a[k_1] \leq a[k_2]$ as $a[k_1]$ not only depends on k_1, but also on $a[k_2]$. More formally, elements expressible by the abstraction are of the form $\forall k, P(k, a[k])$ where P is some property, and the data-abstraction corresponding to *cell abstraction* is the functional representation of an array $\sigma(a) = \{(k, v) | v = a[k]\}$.

Definition 17. *Cell abstractions* σ_{Cell}.
$\sigma_{Cell}(a) = \{(k, v) | v = a[k]\}$

It might not be obvious why properties such as sortedness are not expressible; after all, an array is abstracted by all its cells and thus, no information seems lost. The loss of information does not happen on individual arrays, but when considering properties about arrays. Let us consider the simple property $a[1] = a[0]$, that is, the set $\{a \mid a[1] = a[0]\}$, and let us show why this property is not expressible by *cell abstraction* in three different manners.

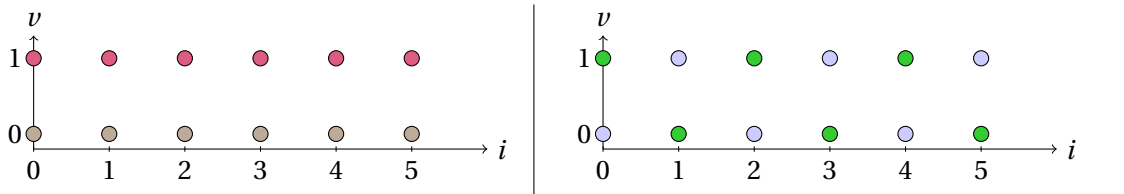
The intuitive way is based on a very simple idea: if this property is expressible by cell abstraction, one needs to differentiate a set that satisfies this property from a set that does not satisfy this

property **in the abstract domain**. We show this is not the case by considering the two sets, one with two constant arrays a_0 and a_1 with constant value 0 and 1 respectively that respects the property $a[1] = a[0]$; and another set with two alternating arrays a_{odd}, a_{even} with value equal to 1 on odd, respectively even, values and 0 otherwise which breaks the property $a[1] = a[0]$. These two sets of arrays are the same after abstraction: they are both equal to $\{(i,v) \mid 0 \leq v \leq 1\}$ as depicted in Figure 3.6; thus, these two sets cannot be differentiated by cell abstraction and the property $a[1] = a[0]$ is not expressible by the abstraction.

The computational way is simply to use Definition 11 of elements expressible by an abstraction. Let us show that $\gamma_{\sigma_{Cell}} \circ \alpha_{\sigma_{Cell}}(\{a \mid a[1] = a[0]\})$ is all arrays and thus that the property is entirely lost. $\alpha_{\sigma_{Cell}}(\{a \mid a[1] = a[0]\}) = \{(i,v) \mid \exists a \in \{a \mid a[1] = a[0]\} \wedge v = a[i]\}$. The problem is that this set is simply $\mathbb{Z}^2$ as for any value of v the constant array equal to v verifies $a[1] = a[0]$ and $v = a[i]$. Thus, $\gamma_{\sigma_{Cell}} \circ \alpha_{\sigma_{Cell}}(\{a \mid a[1] = a[0]\})$ is all arrays.

Finally the quick way is to simply see that this property is not expressible as $\forall k, P(k, a[k])$ for some property P: there are several different cells used in the property simultaneously, in this case, the ones with indices 0 and 1. Note that by simultaneously, we mean that both indices need to be accessed *at the same time* because they need to be compared with one another. This is unlike a property such as $\forall k, a[k] = 0$ where each index k can be accessed independently. This leads to the following rule of thumb to determine if a property is expressible: just look at the number of different array reads in the syntax of the property; if it is 1 or less, then the property is expressible, otherwise, it is not[2].

Figure 3.6 – Unexpressibility of cell abstraction



(a) Abstraction of $\{a_0, a_1\}$ is $\{(i,v) | 0 \leq v \leq 1\}$ (b) Abstraction of $\{a_{odd}, a_{even}\}$ is also $\{(i,v) | 0 \leq v \leq 1\}$

3.3.2.2 Cell-abstraction: extending expressibility with combinators

The main problem about the expressivity of cell-abstraction for the type of algorithm we target is that it cannot express properties such as sortedness which links two array values, that is, properties of the form $\forall k_1, k_2, P(k_1, a[k_1], k_2, a[k_2])$ for some property P. A natural extension to the cell abstraction $\sigma(a) = \{k, a[k]\}$ is to consider the data-abstraction $\sigma(a) = \{k_1, a[k_1], k_2, a[k_2]\}$ which is more expressive and thus would allow us to express exactly properties of the form $\forall k_1, k_2, P(k_1, a[k_1], k_2, a[k_2])$.

However, instead of defining a new abstraction $\sigma(a) = \{k_1, a[k_1], k_2, a[k_2]\}$, that, in many ways is very similar to cell abstraction, we prefer to use combinators which allow, from cell abstraction, to construct $\sigma(a) = \{k_1, a[k_1], k_2, a[k_2]\}$. The abstraction $\sigma(a) = \{k_1, a[k_1], k_2, a[k_2]\}$ can be viewed as *we abstract a twice by cell abstraction such that links between both abstraction instances can be stated.*

[2] This rule of thumb is not entirely exact and only handles properties where arrays are only used in select and store operations.

Example 15 (Properties of container libraries algorithms abstracted with σ^n_{Cell})**.** *We consider the notations of Example 14. The n column denotes the* $\boldsymbol{n}$ *in the abstraction* $\sigma^{\boldsymbol{n}}_{Cell}$ *that is used and the property column states the abstract property P over an array a in the abstract domain, that is, the abstract set* $\{(k_a, v_a) | P(k_a, v_a)\}$ *for* σ^1_{Cell} *and* $\{((k^1_a, v^1_a), (k^2_a, v^2_a)) | P(k^1_a, v^1_a, k^2_a, v^2_a)\}$ *for* σ^2_{Cell}.

Category	*Algorithm*	*n*	*Property of Example 14 in the abstract domain*
Init	*Constant value*(a, v)	1	$k_a < i \Rightarrow v_a = v$
	Identity(a)	1	$k_a < i \Rightarrow v_a = k_a$
Modifying	$b =$*Copy*(a)	1	$k_a = k_b < i \Rightarrow v_a = v_b$
	$b =$*Insert*(a, pos, v)	1	$k_b < i \Rightarrow$ $((k_b < pos \wedge k_a = k_b) \Rightarrow v_b = v_a) \wedge$ $((k_b = pos) \Rightarrow v_b = v) \wedge$ $((k_b > pos) \wedge k_a = k_b - 1 \Rightarrow v_b = v_a)$
	$b =$*Rev*(a)	1	$(k_b < i \wedge k_a = n_a - 1 - k_b) \Rightarrow v_b = v_a$
Finding	$r =$*Linear search*(a, v)	1	$((k_a = r \Rightarrow v_a = v) \wedge k_a < r \Rightarrow v_a \neq v) \vee$ $k_a < i \Rightarrow v_a \neq v$
	Binary search(a, v)	2	$k^1_a < lower \Rightarrow v^1_a < v \wedge$ $k^1_a \geq upper \Rightarrow v^1_a > v \wedge$ $lower \leq k^1_a < k^2_a < upper \Rightarrow v^1_a \leq v^2_a$
	Max(a)	2	$k^1_a < i \wedge k^2_a = max \Rightarrow v^1_a \leq v^2_a$
Sorting	*Bubble sort*(a)	2	$change = false \Rightarrow$ $0 \leq k^1_a < k^2_a < upper \Rightarrow v^1_a \leq v^2_a)$
	$b =$*Insertion sort*(a)	2	$s^\#(b, 0, n_b) \wedge$ $(k^1_b < lower \wedge k^1_a = curr) \Rightarrow v^1_b < v^1_a \wedge$ $(k^1_b \geq upper \wedge k^1_a = curr) \Rightarrow v^1_b > v^1_a$
	$c =$*Merge sorted*(a, b)	2	$s^\#(c, 0, i_a + i_b) \wedge s^\#(a, i_a, n_a) \wedge s^\#(b, i_b, n_b) \wedge$ $(k^1_c < i_a + i_b \wedge k^1_a = i_a \wedge k^1_b = i_b) \Rightarrow (v^1_c \leq v^1_a \wedge v^1_c \leq v^1_b)$
	Merge sort(a)	2	*Property given by* $Mergesorted$
	Quick sort(a)	1	$s^\#(b, 0, n_b) \wedge s^\#(c, 0, n_c)$ $\wedge k^1_b \leq n_b \Rightarrow v^1_b \leq pivot \wedge k^1_c \leq n_c \Rightarrow n^1_c > pivot$
Fold	*sum*	$\emptyset$	*impossible*
	count	$\emptyset$	*impossible*

1 *The* $\bullet$ *combinator is used to abstract multiple variables, with identity for non-array variables and* σ^n_{Cell} *for array variables.*

2 *For a program array a, we write* n_a *the size of that array.*

3 *For space reasons we write* $s^\#(a, start, end)$ *the abstract sortedness property which expands to* $start \leq k^1_a < k^2_a < end \Rightarrow v^1_a \leq v^2_a$.

This leads to the definition of two new combinators: the product combinator written $\otimes$ and the sum combinator written $\oplus$. Both combinators allow one to abstract a data-value by two abstractions; however, the product combinator keeps links between the abstraction instances whereas the sum combinator does not; thus, the product combinator is more expressive and, in the case of cell abstraction, is the one we will use. Note that both combinators extend the expressiveness of each of the individual abstractions.

Definition 18 (Sum and product combinators $\oplus, \otimes$)**.** *The product abstraction expresses any properties of both abstract domains and relations between both abstract domains and is defined by:*

$$\sigma_1 \otimes \sigma_2(x) = \sigma_1(x) \times \sigma_2(x)$$

The sum abstraction expresses properties of both abstract domains, but not properties linking both abstract domains and is defined by

$$\sigma_1 \oplus \sigma_2(x) = \{T_1(x_1) \mid x_1 \in \sigma_1(x)\} \cup \{T_2(x_2) \mid x_2 \in \sigma_2(x)\}$$

where T_1 and T_2 are the constructors for the sum type. View notation sheet for details.

With these new combinators, one can extend cell abstraction to handle our new properties with the abstraction $\sigma_{Cell} \otimes \sigma_{Cell}$. Although in our examples we never need $\sigma_{Cell} \otimes \sigma_{Cell} \otimes \sigma_{Cell}$, which would allow one to express properties relating three cells, we generalize these multiple products with the power notation; thus, $\sigma_{Cell} \otimes \sigma_{Cell}$ is written as σ^2_{Cell}. Let us now consider in Example 15 our desired properties of Example 14 and see how they can be expressed with σ^n_{Cell}.

3.3.2.3 Cell abstraction: evaluation

In Section 3.3.1, we discussed three properties a good array abstraction should have. We now evaluate the cell abstraction with respect to these criteria.

The first is the trade-off between expressivity and simplicity. The cell-abstraction is extremely expressive as it allows, by using the product combinator $\otimes$, to express any property which involves only a finite number of cells at a given time, and thus, to handle all non-fold operations of Example 14. As for simplicity, the abstract domain is barely more complicated than smashing: we only use an additional integer representing the index, and when using the product combinator, the abstract domain is still only a subset of $\mathscr{P}(\mathbb{Z}^n)$. Overall, the cell abstraction seems to offer one of the best trade-offs compared to the abstractions of Examples 11, 12 and 13.

The second is how well cell-abstraction and the data-abstraction framework, mainly combinators, fare together. Several example properties show this is a perfect match. First, cell abstraction is naturally extended by the product combinator which is general and not specifically targeted at cell abstractions. Secondly, many previous abstractions can be expressed, as demonstrated by Example 16, using cell abstraction, combinators, and very simple abstractions, thus these abstractions can be easily handled within our framework if we handle cell-abstractions. Finally, cell-abstraction with combinators can also target specific abstractions that were not properly formalized in previous works, such as ensuring that multisets of elements are kept unchanged during sorting or fold operations, as demonstrated in Example 17.

Finally, the class of properties that are handled by cell-abstraction is well framed and is predictable: for σ^n_{Cell}, this set of properties contains those that use at most n array accesses, possibly universally quantified. Thus, cell abstractions cannot be used to handle properties dependent on the full array, which fall in the category of what we call fold operations. Note that they can still be used as a preprocessing method for other abstractions that do handle specific fold operations as has been done for the count property of Example 17.

Example 16 (Expressing previous abstractions with Cell abstraction and combinators)**.** *The following abstraction can be expressed using cell abstraction, combinators, and simple abstractions:*

1. *The first cell array abstraction of Example 11 can be expressed using the simple abstraction* $\sigma(k,v) = \text{ite}(k \neq 0, \emptyset, \{v\})$*. This yields,* $\sigma_{[0]} = \sigma \odot \sigma_{Cell}$*.*
2. *The smashing abstraction of Example 12 can be expressed using the simple abstraction* $\sigma(k,v) = \{v\}$*. This yields,* $\sigma_{smashing} = \sigma \odot \sigma_{Cell}$*.*
3. *The slicing abstraction of Example 13 can be expressed using the simple abstraction* $\sigma(i,v) = \{v\}$ *and the abstractions* $\sigma_{[a,b]}(k,v) = \text{ite}(a \leq k \leq b, \{k,v\}, \emptyset)$*. Up to parentheses, this yields,* $\sigma_{slicing} = ((\sigma_{[0,i[} \odot \sigma) \bullet (\sigma_{[i,i]} \odot \sigma) \bullet (\sigma_{[i,n[} \odot \sigma)) \odot \sigma^3_{Cell}$*.*

Example 17 (Expressing new abstractions using Cell abstraction and combinators)**.** *Some properties, such as counting the number of elements in a given set, are global properties and cannot directly be tackled by cell abstraction. However, in [MG16], an idea is to consider the map that to a value associates the number of occurences in an array. In such a map, one usually only cares about the relationship between indices and elements, and thus, the property is local.*

We formalize this idea by using a global abstraction $\sigma_{multiset}(a) = \{b \mid \forall j, b[j] = card\{i \mid a[i] = j\}\}$ *and then combining it with cell abstraction. Thus, the right abstraction to prove properties of multisets are* $\sigma_{Cell} \odot \sigma_{multiset}$*.*

For the counting property of Example 14 this is enough; however, for sortedness, one usually wants both the property that ensures sorted with σ^2_{Cell} *and the property that the multiset is preserved. To do so, one can just combine both abstraction with the sum combinator. This yields* $\sigma^2_{Cell} \oplus (\sigma_{Cell} \odot \sigma_{multiset})$*.*

Overview and contribution within this chapter

In this chapter, we introduced the notion of abstraction and formalized how algorithms based on abstractions should behave and the interesting properties they might have. We then constructed a specific framework to formalize and combine abstractions of unbounded data-structures and show that this framework allows us to construct interesting abstractions of arrays. The contribution in this chapter is mostly in what we believe to be the correct setup for abstractions of data-structures and the demonstration that this setup handles existing techniques.

More specifically:

1. The notion of abstraction presented in Section 3.1.1, mainly Galois connections, is already well known work. The only slight deviation is our adaptation to Horn problems and models instead of programs and invariants.
2. In Sections 3.1.2 and 3.1.4, we link abstraction of models and algorithms by defining algorithm properties. We motivate and formalize these properties and justify our choice for a transformation based solving algorithm. Although we do not believe these properties to be ground-breaking, we have not found their formalism stated as such in the literature, except for *soundness* and *completeness*.
3. The full data-abstraction framework formalism introduced in Section 3.2 is a contribution, which can only be fully understood after having read this full manuscript: the formalism may seem simple, but this powerful framework enables abstractions to be assembled like legos, without too much added difficulty, and to formalize existing abstractions. This framework has gone through multiple trial and error steps which may be better understood in the following chapters. We hope the current formalism is the right one, but future work may succeed in improving it as more abstractions – for example abstractions for trees – are added.

4. Finally Section 3.3 aims to demonstrate the power of this framework by tackling container library algorithms. The contribution is formalizing existing abstractions and techniques in our framework.

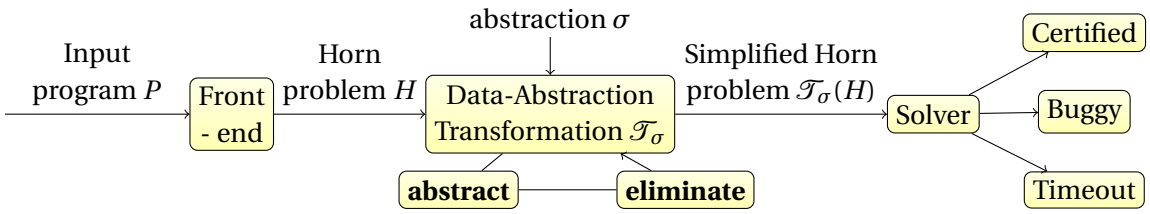
4 Data-Abstraction: transformation algorithm

In Chapter 3 we gave the specifications of the Data-abstraction framework: the input abstractions, the properties of the output, and the capabilities of that framework. However, we have not yet shown that these specifications can be implemented as an algorithm.

In this chapter, we give the data-abstraction framework algorithm and show that meeting the *implementing the abstraction specification* of Chapter 3 depends on an heuristic called $insts$. Later on, in Chapter 5, we construct $insts$ and show that we meet the specifications for a broad class of algorithms that includes most of those discussed in Chapter 3.

The construction of the data-abstraction framework algorithm is divided into two parts. First, we show the existence of a Horn problem that satisfies the right properties and give a constructive definition for it. We adapt this constructive definition in the $abstract$ algorithm which operates on the syntax of Horn problems, that is, Horn clauses. This algorithm returns **extended** Horn clauses, that is, Horn clauses with additional quantifiers around predicates. Then we transform this logical formula back to Horn clauses by removing the additional quantifiers in an algorithm called $eliminate$ that depends on an heuristic $insts$. The full picture of the verification process using these two algorithms is given in Figure 4.1.

Figure 4.1 – Overall program verification scheme

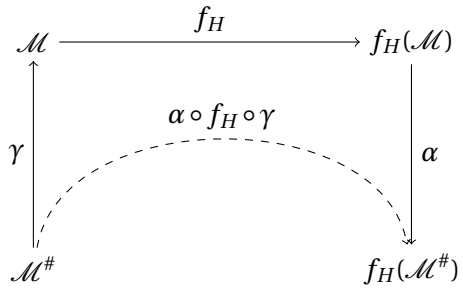


This chapter is organized into three sections: one section for the $abstract$ algorithm, one for the $eliminate$ algorithm and one that translates the property *The data-abstraction framework algorithm verifies the specifications of Chapter 3* into a property that each call to the heuristic $insts$ must verify. The construction of the $insts$ heuristic is left to Chapter 5.

4.1 The $abstract$ algorithm

The goal of the $abstract$ algorithm is to transform a non-abstracted Horn problem into an abstracted Horn problem. The requirement from Chapter 3 is that it *implements the abstraction*,

Figure 4.2 – Constructing the abstract transition of a concrete transition f_H



that is, the abstracted problem is satisfiable if and only if the initial problem is satisfiable by a model expressible by the abstraction as defined in Definition 11.

To construct such an algorithm, we divide the work into two parts. First, we prove the existence of an abstracted Horn problem that verifies the requirement and provide a constructive definition for it. Then, we write this mathematical definition as an algorithm, the $abstract$ algorithm.

4.1.1 Defining the abstracted Horn problem $\mathscr{G}(H)$

Our goal is for any Horn problem H and abstraction $\mathscr{G} = (\alpha, \gamma)$ to prove the existence of and construct a Horn problem, that we call $\mathscr{G}(H)$, such that the transformation that to H returns $\mathscr{G}(H)$ *implements the abstraction*, that is, $\mathscr{G}(H)$ satisfiable $\equiv \exists \mathscr{M} \triangleleft \mathscr{G}, H(\mathscr{M})$ as given in Definition 14. A Horn problem $\mathscr{G}(H)$ that verifies such a property may not be unique, and here we consider a classical construction that achieves these properties.

A Horn problem H is a pair of a function f_H, which is similar to the transition relation of a program, and a model $\mathscr{U}_H$ expressing the set of correct states for the program. The abstraction of H by an abstraction $\mathscr{G} = (\alpha, \gamma)$ is a new Horn problem where both f_H and $\mathscr{U}_H$ are abstracted. The abstraction of $\mathscr{U}_H$, the set of correct states, is extremely simple: the abstraction function α can be used and yields $\alpha(\mathscr{U}_H)$. However, f_H is not that easy to handle, but the technique is common knowledge in the abstract interpretation community.

The abstraction $\mathscr{G} = (\alpha, \gamma)$ of the function f_H is a function $f_H^\#$ that to an abstract model $\mathscr{M}^\#$ must return some abstract model $f_H^\#(\mathscr{M}^\#)$; and somehow, the link between $\mathscr{M}^\#$ and $f_H^\#(\mathscr{M}^\#)$ must reflect f_H. The key to compute $f_H^\#(\mathscr{M}^\#)$ and link it to f_H is to pass through the concrete domain so that we can use f_H directly. To do so, we follow the diagram given in Figure 4.2: given $\mathscr{M}^\#$, we first compute what it represents in the concrete domain by using γ. We can now apply f_H. Finally, we can use α to obtain the value in the abstract domain. Thus, the abstraction of a Horn problem $H = (f_H, \mathscr{U}_H)$, written $\mathscr{G}(H)$ is thus $(\alpha \circ f_H \circ \gamma, \alpha(\mathscr{U}_H))$.

Definition 19 (Abstraction of a Horn problem $\mathscr{G}(H)$)**.** *The abstraction of a Horn problem $H = (f_H, \mathscr{U}_H)$ by $\mathscr{G} = (\alpha, \gamma)$ is $\mathscr{G}(H) = (\alpha \circ f_H \circ \gamma, \alpha(\mathscr{U}_H))$.*

Before going onward, one needs to check that Definition 19 indeed satisfies the best properties of an abstraction stated in Chapter 3. In Theorem 3, we prove that Definition 19 is well formed, that is, constructs a Horn problem and that it meets the expectations of Chapter 3.

Theorem 3 (The definition of $\mathscr{G}(H)$ is correct)**.** *Let $H = (f_H, \mathscr{U}_H)$ be a Horn problem and $\mathscr{G} = (\alpha, \gamma)$ be an abstraction.*

1. *$\mathscr{G}(H)$ is a Horn problem.*

2. *The transformation algorithm $\mathcal{T}$ that to Horn clauses encoding H returns Horn clauses encoding $\mathcal{G}(H)$ implements the abstraction $\mathcal{G}$.*

Proof.

1. To prove that $\mathcal{G}(H)$ is a Horn problem one needs to prove that $\alpha \circ f_H \circ \gamma$ is monotonic. This is the case because each function is monotonic and monotonicity is preserved by composition.
2. We recall that $\mathcal{M} \lhd \mathcal{G}$ means that $\mathcal{M}$ is expressible by the abstraction $\mathcal{G}$ and advise to read Definition 11 on Page 42 while reading this proof.
   Let us now prove that $\mathcal{T}$ implements the abstraction $\mathcal{G}$ by proving that $H_{\mathcal{T}(\mathfrak{C})}$ satisfiable $\equiv \exists \mathcal{M} \lhd \mathcal{G}, H_{\mathfrak{C}}(\mathcal{M})$. As $H_{\mathcal{T}(\mathfrak{C})} = \mathcal{G}(H_{\mathfrak{C}})$ by assumption, and let us call $H_{\mathfrak{C}}$ simply H, we prove $\mathcal{G}(H)$ satisfiable $\equiv \exists \mathcal{M} \lhd \mathcal{G}, H(\mathcal{M})$.
   $\mathcal{G}(H)$ satisfiable
   $\equiv \exists \mathcal{M}^\#, \mathcal{G}(H)(\mathcal{M}^\#)$
   $\equiv \exists \mathcal{M}^\#, \alpha \circ f_H \circ \gamma(\mathcal{M}^\#) \leq \mathcal{M}^\# \wedge \mathcal{M}^\# \leq \alpha(\mathcal{U}_H)$
   We use the item 3 of Definition 8
   $\equiv \exists \mathcal{M}^\#, f_H \circ \gamma(\mathcal{M}^\#) \leq \gamma(\mathcal{M}^\#) \wedge \gamma(\mathcal{M}^\#) \leq \mathcal{U}_H$
   $\equiv \exists \mathcal{M}^\#, H(\gamma(\mathcal{M}^\#))$
   ($\Rightarrow$) $\gamma(\mathcal{M}^\#) \lhd \mathcal{G}$
   ($\Leftarrow$) Let $\mathcal{M}^\# = \alpha(\mathcal{M})$, $H(\gamma(\mathcal{M}^\#)) = H(\gamma(\alpha(\mathcal{M}))) = H(\mathcal{M})$ because $\mathcal{M}$ expressible by $\mathcal{G}$
   $\equiv \exists \mathcal{M} \lhd \mathcal{G}, H(\mathcal{M})$

□

4.1.2 $\mathcal{G}(H)$ as an algorithm

In Section 4.1.1 we showed that a transformation algorithm that to Horn clauses encoding H returns Horn clauses encoding $\mathcal{G}(H)$ implements the abstraction $\mathcal{G}$. In this section, we tackle the problem of finding such a transformation algorithm for data-abstractions, the main issue being that we need to write $\mathcal{G}(H)$ in a Horn clause syntax. However, as discussed in Chapter 3, we do not aim to implement as a transformation algorithm any abstraction of models, but only, using the notations of Definition 15, those of the form $\mathcal{G}^{\sigma^{abs}}$, where abs is a function that to a predicate returns the data-abstraction to use.

To construct Horn clauses encoding $\mathcal{G}^{\sigma^{abs}}(H)$, let us recall that the satisfiability of Horn problems can be expressed in two different manners: the definition as lfp $f_H \leq \mathcal{U}_H$ or as $\exists \mathcal{M}, \mathcal{M} \geq f_H(\mathcal{M}) \wedge \mathcal{M} \leq \mathcal{U}_H$, which is also written $\exists \mathcal{M}, H(\mathcal{M})$. As discussed in Chapter 2, the syntax of Horn clauses encode the latter definition, $\exists \mathcal{M}, H(\mathcal{M})$. Therefore, for our abstracted Horn problem, one should compute $\mathcal{G}^{\sigma^{abs}}(H)(\mathcal{M}^\#)$.

In Theorem 4, we show that $\mathcal{G}^{\sigma^{abs}}(H)(\mathcal{M}^\#)$ is equivalent to $H \circ \gamma_{\mathcal{G}^{\sigma^{abs}}}(\mathcal{M}^\#)$. Thus, the intuition behind our algorithm is to compose with $\gamma_{\mathcal{G}^{\sigma^{abs}}}$.

Theorem 4 (Definition of $\mathcal{G}(H)$ as conditions on models)**.** $\forall \mathcal{M}^\#, H \circ \gamma(\mathcal{M}^\#) \equiv \mathcal{G}(H)(\mathcal{M}^\#)$

Proof.

$$
\begin{aligned}
&\mathcal{G}(H)(\mathcal{M}^\#)\\
\equiv&\mathcal{M}^\# \geq \alpha \circ f_H \circ \gamma(\mathcal{M}^\#) \wedge \mathcal{M}^\# \leq \alpha(\mathcal{U}_H)\\
&\text{We use the item 3 of Definition 8}\\
\equiv&\gamma(\mathcal{M}^\#) \geq f_H \circ \gamma(\mathcal{M}^\#) \wedge \gamma(\mathcal{M}^\#) \leq \mathcal{U}_H\\
\equiv&H \circ \gamma(\mathcal{M}^\#)
\end{aligned}
$$

□

The unrolling of Definition 15 and 9 gives us that $\gamma_{\mathcal{G}^{\sigma^{abs}}}$ is defined by $\gamma_{\mathcal{G}^{\sigma^{abs}}}(\mathcal{M}^\#)(P) = \{a | \sigma_P(a) \subseteq \mathcal{M}^\#(P^\#)\}$.

The problem with the latter definition is that it uses sets: in Horn clauses, one does not manipulate sets; instead, one encodes a set through its inclusion relation function. Rewriting $\gamma_{\mathcal{G}^{\sigma^{abs}}}(\mathcal{M}^\#)(P)$ through its inclusion relation yields $a \in \gamma_{\mathcal{G}^{\sigma^{abs}}}(\mathcal{M}^\#)(P) \equiv \forall a^\#, a^\# \in \sigma_P(a) \Rightarrow a^\# \in \mathcal{M}^\#(P^\#)$.

Next, as the definition of the data-abstraction appears within the expression, we need a syntactical expression for that set. Again, we choose to use the definition through the inclusion relation function defined by $F_{\sigma_P}(a^\#, a) \equiv a^\# \in \sigma_P(a)$. Therefore, the abstraction parameter abs for the $abstract$ algorithm will be a function that, to each predicate, associates a formula F_σ. Definition 20 gives the F_σ formula for the abstractions and combinators we have already seen.

Finally, let us recall that, in Horn clauses, $a^\# \in \mathcal{M}^\#(P^\#)$ is written $P^\#(a^\#)$. We obtain $a \in \gamma_{\mathcal{G}^{\sigma^{abs}}}(\mathcal{M}^\#)(P) \equiv \forall a^\#, abs(P)(a^\#, a) \rightarrow P^\#(a^\#)$.

Thus, our algorithm simply consists in replacing everywhere $P(expr)$ by its value once composed with γ, that is, $\forall a^\#, abs(P)(a^\#, expr) \rightarrow P^\#(a^\#)$.

Algorithm 1 ($abstract_{abs}(\mathfrak{C})$)**.**

Input: $\mathfrak{C}$, *the set of Horn clauses.*

Computation: FOR EACH *clause* $C \in \mathfrak{C}$, FOR EACH $P(expr)$ *in* C, REPLACE $P(expr)$ *by* $\forall a^\#, abs(P)(a^\#, expr) \rightarrow P^\#(a^\#)$

Definition 20 (Data-abstractions as formulae F_σ)**.**

$\sigma_{Cell}(a) = \{(i, v) \mid v = a[i]\}$	$F_{\sigma_{Cell}}((i, v), a) \equiv v = a[i]$
$\sigma_{id}(a) = \{a\}$	$F_{\sigma_{id}}(a^\#, a) \equiv a^\# = a$
$\sigma_1 \bullet \sigma_2(a, b) = \sigma_1(a) \times \sigma_2(b)$	$F_{\sigma_1 \bullet \sigma_2}((a^\#, b^\#), (a, b)) \equiv F_{\sigma_1}(a^\#, a) \wedge F_{\sigma_2}(b^\#, b)$
$\sigma_1 \odot \sigma_2(a) = \bigcup_{a'}^{\sigma_2(a)} \sigma_1(a')$	$F_{\sigma_1 \odot \sigma_2}(a^\#, a) \equiv \exists a', F_{\sigma_2}(a', a) \wedge F_{\sigma_1}(a^\#, a')$
$\sigma_1 \otimes \sigma_2(a) = \sigma_1(a) \times \sigma_2(a)$	$F_{\sigma_1 \otimes \sigma_2}((a_1^\#, a_2^\#), a) \equiv F_{\sigma_1}(a_1^\#, a) \wedge F_{\sigma_2}(a_2^\#, a)$
$\sigma_1 \oplus \sigma_2(a) = \begin{array}{l}\{T_1(x_1) \mid x_1 \in \sigma_1(a)\}\\ \cup \{T_2(x_2) \mid x_2 \in \sigma_2(a)\}\end{array}$	$F_{\sigma_1 \oplus \sigma_2}(a^\#, a) \equiv match\ a^\#\ (\text{fun}\ v \rightarrow F_{\sigma_i}(v, a))$

The process used to construct Algorithm 1 exactly encodes $\mathcal{G}^{\sigma^{abs}}(H)$ and thus implements the abstraction as stated by Theorem 5. However, the transformation introduces a new quantifier $\forall a^\#$; and, even though the output is still a formula encoding a Horn problem, it is no longer in the syntax of Horn clauses. Thus, we may not use current Horn solvers – or with poor results – and this leads us the $eliminate$ algorithm that aims to recover from this defect. An example of the output of Algorithm 1 is given in Example 18 and highlights the added quantifiers.

Example 18 (Using the $abstract$ algorithm)**.** *Since* $abstract$ *operates on each clause independently, we exemplify its use on individual clauses. We do not use clauses of Example 5 as these yield huge formulae.*

On a simple Horn clause. *Consider the very simple clause* $P(a) \rightarrow P'(a)$ *and the abstraction* abs *such that* $abs(P) = abs(P') = \sigma_{Cell}$. *We obtain the following extended Horn clause where* (i, v) *and* (i', v') *represents* $a^\#$ *and* $v = a[i]$ *and* $v' = a[i']$ *represents* $F_{\sigma_{Cell}}((i, v), a)$.

$$\big(\forall (i, v), v = a[i] \rightarrow P^\#(i, v)\big) \rightarrow \big(\forall (i', v'), v' = a[i'] \rightarrow P'^\#(i', v')\big)$$

On a non-linear Horn clause. *Consider the non-linear clause* $\big(P(a) \wedge F((a, 0), x)\big) \rightarrow P'(x)$ *which encodes the instruction* `x=f(a, 0);` , *and the abstraction* abs *such that* $abs(P) = \sigma_{Cell}, abs(F) = \sigma_{Cell} \bullet \sigma_{id} \bullet \sigma_{id}$, *and* $abs(P') = \sigma_{id}$. *We obtain the following extended Horn clause using Definition 20 to compute* $F_{abs(F)}$:

$$\begin{aligned}&\big((\forall (i, v), v = a[i] \rightarrow P^\#(i, v)) \wedge \\ &(\forall (((i_f, v_f), z^\#), x^\#), (v_f = a[i_f] \wedge z^\# = 0 \wedge x^\# = x) \rightarrow F^\#(((i_f, v_f), z^\#), x^\#))\big) \\ &\qquad\qquad \rightarrow \big(\forall x'^\#, x'^\# = x \rightarrow P'^\#(x'^\#)\big) \qquad (4.1)\end{aligned}$$

Theorem 5 (The $abstract$ algorithm implements the abstraction)**.**

$$\forall abs, \mathfrak{C}, \mathscr{M}^\#, H_{abstract_{abs}(\mathfrak{C})}(\mathscr{M}^\#) \equiv H_{\mathfrak{C}}(\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#))$$

Thus, $abstract_{abs}$ *implements the abstraction* $\mathscr{G}^{\sigma^{abs}}$ *using Theorem 3 and Theorem 4.*

Proof. Let us introduce $\mathscr{M}^\#$.

1. For an expression e (possibly a clause), let $replaced(e)$ be the expression constructed from e by replacing all instances of $P(expr)$ by $\forall a^\#, abs(P)(a^\#, expr) \rightarrow P^\#(a^\#)$.
2. We need to prove $H_{abstract_{abs}(\mathfrak{C})}(\mathscr{M}^\#) \equiv H_{\mathfrak{C}} \circ \gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)$ Let us reduce this proof goal to proving $\forall C \in \mathfrak{C}, ([\![replaced(C)]\!]^\forall_{\mathscr{M}^\#} \equiv [\![C]\!]^\forall_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)})$. Proof:

   $H_{abstract_{abs}(\mathfrak{C})}(\mathscr{M}^\#) \equiv H_{\mathfrak{C}} \circ \gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)$

   By definition of $H_{\mathfrak{C}}$

   $\equiv (\forall C^\# \in abstract_{abs}(\mathfrak{C}), [\![C^\#]\!]^\forall_{\mathscr{M}^\#}) \equiv (\forall C \in \mathfrak{C}, [\![C]\!]^\forall_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)})$

   By definition of $replaced(e)$ and $abstract_{abs}$

   $\equiv (\forall C \in \mathfrak{C}, [\![replaced(C)]\!]^\forall_{\mathscr{M}^\#}) \equiv (\forall C \in \mathfrak{C}, [\![C]\!]^\forall_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)})$

   $\Leftarrow \forall C \in \mathfrak{C}, ([\![replaced(C)]\!]^\forall_{\mathscr{M}^\#} \equiv [\![C]\!]^\forall_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)})$
3. But we prove by structural induction on the expression e that $\forall vars, ([\![replaced(e)]\!]^{vars}_{\mathscr{M}^\#} = [\![e]\!]^{vars}_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)})$, thus we have $\forall C \in \mathfrak{C}, ([\![replaced(C)]\!]^\forall_{\mathscr{M}^\#} \equiv [\![C]\!]^\forall_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)})$.
   Consider $e = Cons(e_1, \ldots, e_n)$ where $Cons$ is an expression constructor or a predicate and consider the following cases:
   (a) if $Cons$ is not a predicate then by the induction hypothesis, the evaluation of $e_1, \ldots, e_n$ is unchanged, and $Cons$ is unchanged, thus $[\![replaced(e)]\!]^{vars}_{\mathscr{M}^\#} = [\![e]\!]^{vars}_{\gamma_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}^\#)}$
   (b) if $e = P(expr)$ with P a predicate, then

$$
\begin{aligned}
&[\![replaced(e)]\!]^{vars}_{\mathcal{M}^\#} \\
&\quad \text{Expanding the definition of } replaced \\
=&[\![\forall a^\#, abs(P)(a^\#, replaced(expr)) \rightarrow P^\#(a^\#)]\!]^{vars}_{\mathcal{M}^\#} \\
&\quad \text{Using set notation} \\
=&abs(P)([\![replaced(expr)]\!]^{vars}_{\mathcal{M}^\#}) \subseteq \mathcal{M}^\#(P^\#) \\
&\quad \text{By expanding the definition of } \gamma_{\mathcal{G}\sigma^{abs}} \text{ (see Definition 15)} \\
=&[\![replaced(expr)]\!]^{vars}_{\mathcal{M}^\#} \in \gamma_{\mathcal{G}\sigma^{abs}}(\mathcal{M}^\#)(P) \\
&\quad \text{By induction hypothesis} \\
=&[\![expr]\!]^{vars}_{\gamma_{\mathcal{G}\sigma^{abs}}(\mathcal{M}^\#)} \in \gamma_{\mathcal{G}\sigma^{abs}}(\mathcal{M}^\#)(P) \\
&\quad \text{By definition of the semantics} \\
=&[\![P(expr)]\!]^{vars}_{\gamma_{\mathcal{G}\sigma^{abs}}(\mathcal{M}^\#)} \\
=&[\![e]\!]^{vars}_{\gamma_{\mathcal{G}\sigma^{abs}}(\mathcal{M}^\#)}
\end{aligned}
$$

□

4.2 The *eliminate* algorithm

The output produced by the *abstract* algorithm is not in pure Horn clause format: there are additional universal quantifiers. These additional quantifiers make current solving techniques struggle and, in this section, we focus on removing them through an algorithm called *eliminate*. Unlike previous approaches [BMR13; BMS06], we do not believe in a general quantifier elimination technique independent of the abstraction used and we believe that knowledge about the abstraction is important to remove quantifiers without losing too much information.

The principle behind the *eliminate* algorithm is fairly straightforward and is called quantifier instantiation [BMR13]. Usually this technique is applied without any regards for the completeness property and thus a simple explanation usually suffices. However, in this PhD, we tackle the problem of precisely figuring out how this technique should be applied so that the information loss may be minimized and controlled. This requires precision and, in this section, we decompose the problem into several parts.

Two types of quantifiers instances. To understand how to handle the additional quantifiers, we need to understand where they are added in the Horn clause. First, let us consider the case where $abs(P)$ does not contain quantifiers and the only added quantifier is thus $\forall a^\#$. For this purpose, we consider two examples that show that not all added quantifiers should be handled in the same manner.

First, consider the clause $i = 0 \rightarrow P(a)$. After abstraction, this clause yields $i = 0 \rightarrow (\forall a^\#, abs(P)(a^\#, a) \rightarrow P^\#(a^\#))$, which is equivalent to, by moving quantifiers to prenex and decurrifying, $\forall a^\#, (i = 0 \wedge abs(P)(a^\#, a)) \rightarrow P^\#(a^\#)$. However, variables in Horn clauses are implicitly universally quantified as shown by Definition 4; thus, quantifiers that when moved to prenex position would be universal can be removed and replaced by a fresh variable. Therefore, this clause is semantically equivalent to the clause $(i = 0 \wedge abs(P)(a^\#, a)) \rightarrow P^\#(a^\#)$.

Secondly, consider the clause $P(a) \rightarrow i = n$. After abstraction this clause yields $(\forall a^\#, abs(P)(a^\#, a) \rightarrow P^\#(a^\#)) \rightarrow i = n$, which is equivalent to, by moving quantifiers to prenex,

$\exists a^\#, (abs(P)(a^\#, a) \to P^\#(a^\#)) \to i = n$. This quantifier cannot be removed easily and needs to be handled in a specific manner.

The generalization of these two examples is that when the quantifier is introduced by replacing an instance of a predicate in the goal of the clause – the right hand side of the implication – then the quantifier can simply be replaced by a new free variable. One just needs to make sure that that variable is unused to avoid clashing of names. However, when the quantifier is introduced by replacing an instance of a predicate in the premise of a clause – the left hand side of the implication –, then the generated quantifier $\exists a^\#$ cannot simply be replaced by a free variable and we will use a technique called quantifier instantiation [BMR13].

Categorizing quantifiers within $abs(P)$. The formula $F_\sigma(a^\#, a)$ expressing the condition $a^\# \in \sigma(a)$ for a data-abstraction σ may contain quantifiers; albeit one may argue that our only example is the data-abstraction composition combinator. The handling of these quantifiers is again explained by what happens once they are moved to prenex position. The conclusion is that it depends on the type of the quantifier: universal or existential.

If the quantifier would be existential in prenex position, then that quantifier can be handled exactly as the $\forall a^\#$ quantifier and therefore we use the same technique: if the predicate instance is in the goal of the clause, just replace it by a free variable; if it is in the premise of the clause, use quantifier instantiation.

However, if the quantifier would be universal in prenex position of F_σ, then it introduces yet another quantifier alternation when considering a predicate instance in the premises of the clause and our current technique is not equipped to handle it. Thus, our technique only removes quantifiers that would be existential in prenex position of F_σ, and this does not seem like much of a limitation: we have yet to witness a case where this is needed.

Quantifiers that would be existential in prenex position of F_σ act like the quantifier $\forall a^\#$. This can be explained in a much simpler way by introducing the notation $F_\sigma[q]$, where q is a tuple of expressions, assumed well typed, corresponding to the existential quantifiers in F_σ. Definition 21 gives the expressions $F_\sigma[q]$ for the abstractions and combinators of Definition 20. With this notation, one can rewrite $\forall a^\#, abs(P)(a^\#, expr) \to P^\#(a^\#)$, that is, by what $P(expr)$ is replaced as: $\forall (a^\#, q), abs(P)[q](a^\#, expr) \to P^\#(a^\#)$ and thus, clearly show that q is of the same quantifier type as $a^\#$.

Definition 21 ($F_\sigma[q]$ for data-abstractions)**.** *The only data-abstraction combinator that really uses an existential quantifier is $\odot$. The other just pass on the possible existential quantifiers of the abstractions they combine.*

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

$F_{\sigma_{id}}[()](a^\#, a) \equiv a^\# = a$

$F_{\sigma_1 \bullet \sigma_2}[(q_1, q_2)]((a^\#, b^\#), (a, b)) \equiv F_{\sigma_1}[q_1](a^\#, a) \wedge F_{\sigma_2}[q_2](b^\#, b)$

$F_{\sigma_2 \odot \sigma_1}[(i^\#, q_1, q_2)](a^\#, a) \equiv F_{\sigma_2}[q_2](a^\#, i^\#) \wedge F_{\sigma_1}[q_1](i^\#, a)$

$F_{\sigma_1 \otimes \sigma_2}[(q_1, q_2)]((a_1^\#, a_2^\#), a) \equiv F_{\sigma_1}[q_1](a_1^\#, a) \wedge F_{\sigma_2}[q_2](a_2^\#, a)$

$F_{\sigma_1 \oplus \sigma_2}[q](a^\#, a) \equiv match\ (q, a^\#)\ (\text{fun}\ (q', v) \to F_{\sigma_i}[q'](v, a))$

Quantifier instantiation [BMR13]. Quantifier instantiation simply consists in replacing a formula of the form $\forall q, P(q)$, where P denotes a boolean formula, which can also be written as an infinite conjunction $\bigwedge_q P(q)$, by a finite conjunction $\bigwedge_{q \in S} P(q)$ where S is called the *instantiation*

set S. The main problem with this technique is that $\forall q, P(q)$ is not equivalent to $\bigwedge_{q\in S} P(q)$, and thus may stop us from implementing the abstraction.

In practice, when this transformation is used on instances in the premise of our clause, for any instantiation set S, the transformation is sound as $\forall q, P(q) \Rightarrow \bigwedge_{q\in S} P(q)$ and thus if a formula of the form $((\forall q, P(q)) \wedge others) \rightarrow goal$ is unsatisfiable so is the formula $((\bigwedge_{q\in S} P(q)) \wedge others) \rightarrow goal$.

However, *completeness* is not guaranteed and is even rarely achieved or even achievable. How much information is lost is highly dependent on the chosen instantiation set S and, even though this problem will be mainly tackled in Chapter 5, we illustrate its dependence in Example 19. In this example, we abstract using cell abstraction an array on a clause that cares about the value at index 0 of the array. By using an instantiation set that does not consider the index 0, we lose all information but by considering the instantiation set with only element 0 the information is not lost.

Example 19 (The choice of instantiation set is important)**.** *Consider the clause $P(a) \wedge k = a[0] \rightarrow P'(k)$. Let us abstract this clause so that P is abstracted by cell abstraction and P' is abstracted by the identity abstraction. This yields the abstracted clause $(\forall (i,v), v = a[i] \rightarrow P^\#(i,v)) \wedge k = a[0] \rightarrow (\forall k^\#, k^\# = k \rightarrow P'^\#(k^\#))$. Let us simplify this clause for clarity to get $(\forall (i,v), v = a[i] \rightarrow P^\#(i,v)) \wedge k = a[0] \rightarrow P'^\#(k)$, and let us consider two instantiations sets for the quantifier (i,v) and compare what the abstract clause is equivalent to after instantiation.*

Instantiation set $\{(1,a[1]),(2,a[2])\}$	*Instantiation set* $\{(0,a[0])\}$
$(a[1] = a[1] \rightarrow P^\#(1,a[1])) \wedge (a[2] = a[2] \rightarrow P^\#(2,a[2]))$ $\wedge k = a[0] \rightarrow P'^\#(k)$	$(a[0] = a[0] \rightarrow P^\#(0,a[0]))$ $\wedge k = a[0] \rightarrow P'^\#(k)$
$P^\#(1,a[1]) \wedge P^\#(2,a[2]) \rightarrow P'^\#(a[0])$	$P^\#(0,a[0]) \rightarrow P'^\#(a[0])$
$P^\#(1,v_1) \wedge P^\#(2,v_2) \rightarrow P'^\#(v_0)$	$P^\#(0,v_0) \rightarrow P'^\#(v_0)$

The first step consists in using the definition of instantiation. The second is a simplification of redundant equalities and implications. The third replaces the array a by integers as would be done in the removal of uninterpreted functions: replace $a[i]$ by a new variable v_i and if both $a[i]$ and $a[j]$ appear, add the condition $i = j \Rightarrow v_i = v_j$.

When using the instantiation set $\{(1,a[1]),(2,a[2])\}$ there is no link between the parameter of $P'^\#$, and the premise of the clause, thus no information about variable values is conveyed. However, when using the instantiation set $\{(0,a[0])\}$, there the value of the parameter of $P'^\#$ is constrained by its value in $P^\#$ and thus, information is conveyed.

This example shows how the choice of instantiation set highly impacts information loss.

Putting it together. Let us now give a first attempt at our quantifier elimination algorithm. The first version of the $eliminate$ algorithm takes a formula f output by the $abstract$ algorithm, a pointer p to an instance of an expression of the form $\forall (a^\#, q), abs(P)[q](a^\#, expr) \rightarrow P^\#(a^\#)$ within f, and an finite instantiation set S for the quantifier formed by the pair $(a^\#, q)$, where q is the values for the quantifiers within F_σ.

If that instance is within the goal of the clause, we just remove the $\forall$ quantifier and make fresh variables of $a^\#$ and q, possibly renaming to avoid name clashes. If that instance is within the premises of the clause, we use quantifier instantiation and replace that instance of $\forall (a^\#, q), abs(P)[q](a^\#, expr) \rightarrow P^\#(a^\#)$ by $\bigwedge_{(a^\#,q)\in S} abs(P)[q](a^\#, expr) \rightarrow P^\#(a^\#)$.

This algorithm should then be repeatedly applied until all quantifiers are removed.

Using instantiation heuristics. The problem with this algorithm is that it requires the user to know the instantiation set S for each predicate instance. This is not desirable and our goal is to create heuristics $insts$ to automatically compute the instantiation set S for each instance [BMR13].

The main problem with automating the instantiation set S is that we need to figure out correct parameters for our heuristics. What is for sure is that the heuristic is highly dependent on the abstraction and should, in fact, be defined with the abstraction. However, what is unclear is how much more context about the current Horn problem should be passed to the instantiation heuristic: if no other context than the abstraction is given, we hit the pitfalls demonstrated by Example 19: the right instantiation set depends on the current Horn problem. However, passing the whole Horn problem as a parameter would make it extremely hard to state properties about those heuristics: a good heuristic could use any technique to attempt to solve the Horn problem, and, if that problem is solved, just return the finite set that works.

Our approach consists in passing the current extended Horn clause as context to the heuristic as clauses in Horn problems are an important unit: the variables of a clause are local and implicitly universally quantified and clauses define a kind of transition relation. Furthermore, the $abstract$ algorithm also operates on each clause independently. Thus, our aim is to pass to the instantiation heuristic $insts$ the abstraction abs that is used, the predicate P and the value $expr$ of the instance that is being instantiated, and the context ctx corresponding to the rest of the current extended Horn clause $C^\#$. Such a call to the instantiation heuristic is written $insts_P^{abs}(expr, ctx)$.

In practice, the current extended Horn clause is not passed as such. To explain this, consider the following extended Horn clause $(a[2] = 3 \wedge (\forall a^\#, abs(P)(a^\#, a) \to P^\#(a^\#)) \wedge a[i] \leq 10) \to i = n$. For this example, the call to the instantiation heuristic will be $insts_P^{abs}(a, (a[2] = 3 \wedge a[i] \leq 10) \to i = n)$. Let us explain: $insts_P^{abs}$ means that the predicate is P and the abstraction used $\mathcal{G}^{\sigma^{abs}}$, a is simply the expression that was abstracted, and $(a[2] = 3 \wedge a[i] \leq 10) \to i = n)$ is the *rest* of the extended clause: the extended clause can be written using currification as $(\forall a^\#, abs(P)(a^\#, a) \to P^\#(a^\#)) \to ((a[2] = 3 \wedge a[i] \leq 10) \to i = n)$. In other words, to reconstruct a formula equivalent to the initial extended clause from a call $insts_P^{abs}(expr, ctx)$ to the instantiation heuristic, one writes $(\forall a^\#, abs(P)(a^\#, expr) \to P^\#(a^\#)) \to ctx$.

The previous example had only a single predicate instance in the premises. Adding multiple predicate instances adds another layer of complexity and yields another question: should all quantifiers induced by all predicates instances be instantiated simultaneously, and thus, the formulae that could be reconstructed from each call to the instantiation heuristic would be the same; or, should we sequentially instantiate, and thus, the formulae that could be reconstructed from each call to the instantiation heuristic would be partially instantiated? We choose the sequential option, and that choice will be explained in the next section. However, this requires us to pick an instantiation order and, for lack of better information, we simply use the order given by the syntax.

Therefore, there are two calls to the instantiation heuristic when handling the extended clause $\forall a^\#, abs(P_1)(a^\#, a[i \leftarrow v]) \to P_1^\#(a^\#) \wedge \forall b^\#, abs(P_2)(b^\#, b) \to P_2^\#(b^\#) \wedge a[i+1] = 2 \to false$. The first is $S_1 = insts_{P_1}^{abs}(a[i \leftarrow v], (\forall b^\#, abs(P_2)(b^\#, b) \to P_2^\#(b^\#) \wedge a[i+1] = 2) \to false)$ and the second is $S_2 = insts_{P_2}^{abs}((b, ((\bigwedge_{(a^\#,q)\in S_1}, abs(P_1)[q](a^\#, a[i \leftarrow v]) \to P_1^\#(a^\#)) \wedge a[i+1] = 2) \to false)$ in which we see the dependence of S_2 on S_1, which demonstrates how our choice is sequential.

The $eliminate$ algorithm. The final $eliminate$ algorithm thus takes a single parameter, the set of extended Horn clauses $\mathfrak{C}$ for which we desire to instantiate the quantifiers: we assume the instantiation heuristics to be used for each abstraction to be in the current context. This algorithm transforms each extended Horn clause independently, first by transforming the quantifiers in the

goal into free variables and then by sequentially instantiating the quantifiers in the premise. Algorithm 2 contains the full algorithm used, and, Example 20 is a tiny application example. For a more complex example, one should refer to Example 21 and the proof of Theorems 13 and 14 may also serve as examples.

The $eliminate$ algorithm is sound for any heuristic $insts$: instantiation is only done in the premise of clauses, thus making clauses less satisfiable rather than more satisfiable. The proof is given in Theorem 6. However, the information loss of the $eliminate$ algorithm, measured through the *completeness* property and not *relative completeness* as $eliminate$ does not perform any abstraction, is highly dependent on the heuristic $insts$ and capturing this dependence is the purpose of the next section.

Algorithm 2 (Quantifier elimination algorithm $eliminate_{abs}(\mathfrak{C}^\#)$)**.** *//In this algorithm, we assume the input to be the output of abstract on normalized Horn clauses*

Input: $\mathfrak{C}^\#$, *the set of extended Horn clauses.* | ***abs***, *the abstraction*

Computation: FOR EACH *clause* $C^\# \in \mathfrak{C}^\#$

1. *Let* $n, e_1, \ldots, e_n$ *and* e' *such that* $C^\# \equiv e_1 \wedge \ldots \wedge e_n \rightarrow e'$
2. *//We transform quantifiers in the goal into free variables*
   e'_{res} := MATCH e' WITH
   //the abstraction of a predicate
   $|\forall a^\#, abs(P)(a^\#, expr) \rightarrow P^\#(a^\#)$ THEN $abs(P)[q](a^\#, expr) \rightarrow P^\#(a^\#)$
   where $a^\#, q$ *are fresh unused variables.*
   //an expression without a predicate
   | _ THEN e'
3. *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 $(a_i, P_i^\#)$ *such that* $e_i = \forall a^\#, F_{abs(P_i)}(a^\#, a_i) \rightarrow P_i^\#(a^\#)$
   If it does not match, $e_{res_i} = e_i$ *and go to next loop iteration.*
   (b) *//We compute the context for that instance.*
   Let $ctx_i = e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge e_{i+1} \wedge \ldots \wedge e_n \rightarrow e'_{res}$ *//Note:if* $n = i = 1$, $ctx_i = true \rightarrow e'_{res}$
   (c) *//We compute the instantiation set for that abstraction.*
   Let $S_i = insts^{abs}_{P_i}(a_i, ctx_i)$
   (d) *//We finally compute* e_i *after instantiation*
   Let $e_{res_i} = \bigwedge_{(a^\#, q) \in S_i} F_{abs(P_i)}[q](a^\#, a_i) \rightarrow P_i^\#(a^\#)$
4. REPLACE $C^\#$ *by* $e_{res_1} \wedge \ldots \wedge e_{res_n} \rightarrow e'_{res}$

Example 20 (Array initialization loop instantiation)**.** *Consider the clause* $P(a, i) \rightarrow P(a[i \leftarrow 0], i+1)$ *and the abstraction* $abs(P) = F_{\sigma_{Cell} \bullet \sigma_{id}}$.

The abstract algorithm transforms this clause into

$$(\forall((j, v), i^\#), (v = a[j] \wedge i^\# = i) \rightarrow P^\#((j, v), i^\#)) \rightarrow (\forall((j, v), i^\#), (v = a[i \leftarrow 0][j] \wedge i^\# = i + 1) \rightarrow P^\#((j, v), i^\#))$$

And the eliminate algorithm proceeds in the following way:

1. *It removes the quantifiers in the goal, thus yielding:*

$$(\forall((j, v), i^\#), (v = a[j] \wedge i^\# = i) \rightarrow P^\#((j, v), i^\#)) \rightarrow ((v' = a[i \leftarrow 0][j'] \wedge i'^\# = i + 1) \rightarrow P^\#((j', v'), i'^\#))$$

2. *It calls* $insts_P^{abs}(a, true \rightarrow ((v' = a[i \leftarrow 0][j'] \wedge i'^{\#} = i + 1) \rightarrow P^{\#}((j', v'), i'^{\#}))$. *Which we will assume returns* $\{((j', a[j']), i)\}$.
3. *Yields the final clause*

$$((a[j'] = a[j'] \wedge i = i) \rightarrow P^{\#}((j', a[j']), i)) \\ \rightarrow ((v' = a[i \leftarrow 0][j'] \wedge i'^{\#} = i + 1) \rightarrow P^{\#}((j', v'), i'^{\#}))$$

Example 21 (Instantiation of a non-linear clause)**.** *We reuse the Clause 4.1 of Example 18 recalled below.*

$$\big((\forall (i, v), v = a[i] \rightarrow P^{\#}(i, v)) \wedge \\ (\forall (((i_f, v_f), z^{\#}), x^{\#}), (v_f = a[i_f] \wedge z^{\#} = 0 \wedge x^{\#} = x) \rightarrow F^{\#}(((i_f, v_f), z^{\#}), x^{\#}))\big) \\ \rightarrow \big(\forall x'^{\#}, x^{\#} = x \rightarrow P'^{\#}(x'^{\#})\big) \quad (4.2)$$

The instantiation of this clause yields the following two calls to the instantiation heuristic $insts_P^{abs}(a, ctx_P)$ *and* $insts_F^{abs}(((a, 0), x), ctx_F)$ *with* ctx_P *and* ctx_F *defined by:*

$$ctx_P \equiv \big(\forall (((i_f, v_f), z^{\#}), x^{\#}), (v_f = a[i_f] \wedge z^{\#} = 0 \wedge x^{\#} = x) \rightarrow F^{\#}(((i_f, v_f), z^{\#}), x^{\#})\big) \\ \rightarrow (x^{\#} = x \rightarrow P'^{\#}(x'^{\#})) \quad (4.3)$$

$$ctx_F \equiv \Big(\bigwedge_{(i,v) \in insts_P^{abs}(a, ctx_P)} v = a[i] \rightarrow P^{\#}(i, v)\Big) \rightarrow (x^{\#} = x \rightarrow P'^{\#}(x'^{\#})) \quad (4.4)$$

Theorem 6 ($eliminate$ sound)**.** *For any abs,* $eliminate_{abs}$ *is a sound transformation. Even better, for all abs,* $\mathcal{M}^{\#}$ *and Horn clauses* $\mathfrak{C}^{\#}$

$$[\![eliminate_{abs}(\mathfrak{C}^{\#})]\!]_{\mathcal{M}^{\#}} \Rightarrow [\![\mathfrak{C}^{\#}]\!]_{\mathcal{M}^{\#}}$$

Proof. This "better" result implies soundness as it means that whenever a clause is satisfiable after instantiation, it was also satisfiable before instantiation **by the same model**.

Let us know prove this "better" result. Let us introduce $\mathcal{M}^{\#}$ and $\mathfrak{C}^{\#}$, and use the notations of Algorithm 2. We can rewrite our proof goal into: $(\forall vars', [\![e_{res_1} \wedge \ldots \wedge e_{res_n} \rightarrow e'_{res}]\!]_{\mathcal{M}^{\#}}^{vars'}) \Rightarrow (\forall vars, [\![e_1 \wedge \ldots \wedge e_n \rightarrow e']\!]_{\mathcal{M}^{\#}}^{vars})$.

The proof of this result if simply because $\forall vars', [\![e']\!]_{\mathcal{M}^{\#}}^{vars'}$ is equivalent to $\forall vars', [\![e'_{res}]\!]_{\mathcal{M}^{\#}}^{vars'}$ and $\forall i, (\forall vars, [\![e_i]\!]_{\mathcal{M}^{\#}}^{vars}) \Rightarrow (\forall vars', [\![e_{res_i}]\!]_{\mathcal{M}^{\#}}^{vars'})$ as an infinite conjunction implies a finite conjunction.

□

4.3 Formalizing *the data-abstraction framework algorithm implements the abstraction* as a condition on calls to $insts$

The data-abstraction framework algorithm is obtained by the composition of two algorithms: $abstract$ and $eliminate$. The $abstract$ transformation implements the abstraction but fails to meet the syntax requirements and the $eliminate$ algorithm aims at fulfilling the syntax requirements, but is only proven *sound* and its loss of information is dependent on the instantiation heuristic $insts$.

We want the data-abstraction framework algorithm to implement the abstraction, and find the instantiation heuristic $insts$ that allows it. To do so, we first show that the data-abstraction framework algorithm implements the abstraction if and only if $eliminate$ is complete in Theorem 7. We then show that $eliminate$ is complete whenever each call to $insts$ in $eliminate$ verifies a property we name *completeness of a call to* $insts$. Finally, the construction of a heuristic $insts$ and the proof that it satisfies the completeness property are given in Chapter 5.

Theorem 7 ($eliminate$ must be complete)**.** *If $\mathscr{T}_1$ is a transformation that implements an abstraction $\mathscr{G}$ and $\mathscr{T}_2$ is a sound transformation, then:*

$$\mathscr{T}_2 \circ \mathscr{T}_1 \text{ implements } \mathscr{G} \equiv \mathscr{T}_2 \text{ complete on the output of } \mathscr{T}_1$$

In this theorem, $\mathscr{T}_1$ represents the $abstract$ algorithm and $\mathscr{T}_2$ the $eliminate$ algorithm. This theorem combined with theorems 5 and 6 shows that the data-abstraction framework algorithm implements the abstraction if and only if $eliminate$ is complete.

Proof. Let us decompose the equivalence into two implications.

1. Assume $\mathscr{T}_2 \circ \mathscr{T}_1$ implements $\mathscr{G}$. Reason by contradiction and assume $\mathscr{T}_2$ not complete on the output of $\mathscr{T}_1$. This translates to: $\exists \mathfrak{C}, H_{\mathscr{T}_1(\mathfrak{C})}$ satisfiable $\wedge$ $H_{\mathscr{T}_2 \circ \mathscr{T}_1(\mathfrak{C})}$ not satisfiable. Introduce $\mathfrak{C}$. We have:
   (a) As $\mathscr{T}_1$ implements $\mathscr{G}$, $H_{\mathscr{T}_1(\mathfrak{C})}$ satisfiable $\equiv \exists \mathscr{M}^\# \triangleleft \mathscr{G}, H_{\mathfrak{C}}(\mathscr{M}^\#)$.
   (b) $\mathscr{T}_2 \circ \mathscr{T}_1$ implements $\mathscr{G}$ thus $H_{\mathscr{T}_2 \circ \mathscr{T}_1(\mathfrak{C})}$ satisfiable.
   (c) But $H_{\mathscr{T}_2 \circ \mathscr{T}_1(\mathfrak{C})}$ not satisfiable by assumption of our reasoning by contradiction which contradicts Item 1b.
2. Assume $\mathscr{T}_2$ complete on the ouput of $\mathscr{T}_1$. Because $\mathscr{T}_2$ is also sound, $\forall \mathfrak{C}$ in the output of $\mathscr{T}_1$, $H_{\mathscr{T}_2(\mathfrak{C})}$ satisfiable $\equiv$ $H_{\mathfrak{C}}$ satisfiable. Let us prove that $\mathscr{T}_2 \circ \mathscr{T}_1$ implements $\mathscr{G}$. Introduce $\mathfrak{C}$.
   $H_{\mathscr{T}_2 \circ \mathscr{T}_1(\mathfrak{C})}$ satisfiable
   Using our deduction
   $\equiv H_{\mathscr{T}_1(\mathfrak{C})}$ satisfiable
   By definition of implementing the abstraction
   $\equiv \exists \mathscr{M}^\# \triangleleft \mathscr{G}, H_{\mathfrak{C}}(\mathscr{M}^\#)$

□

4.3.1 *Completeness* of $eliminate$, seemingly impossible

Because $eliminate$ transforms each clause independently, it is very tempting to believe that $eliminate$ must verify the *per clause completeness* property of Definition 22, that is, for any model $\mathscr{M}^\#$, the input clause should be equivalent to the instantiated clause. Here we show that this reasoning fails and *per clause completeness* is not achievable. We believe that, in [BMR13; MG16], *completeness* for their algorithms may have been discarded due to that reasoning.

Definition 22 (Per clause completeness)**.** *The transformation of $C^\#$ into $C^\#_{insts}$ verifies per clause completeness if and only if*

$$\forall \mathscr{M}^\#, [\![C^\#]\!]_{\mathscr{M}^\#} \Rightarrow [\![C^\#_{insts}]\!]_{\mathscr{M}^\#}$$

To understand why this reasoning fails, let us consider the cell abstraction of an array whose index type is infinite. The quantifiers to instantiate for a predicate in the premises using that

array are $\forall(k,v)$, where k is of the index type, and the condition F_σ is $v = a[k]$. During instantiation, the $\forall(k,v)$ is replaced by a conjunction over a finite set S given by the instantiation set. However, because S must be finite, it cannot handle all indices and there is a part of the array which is not considered after instantiation. The idea to show that we do not have $\forall \mathcal{M}^\#, [\![C]\!]_{\mathcal{M}^\#} \Rightarrow [\![eliminate(C)]\!]_{\mathcal{M}^\#}$ is to construct a model such that what makes the clause satisfiable is hidden in the part of the array not considered by S. Thus making $C^\#$ satisfiable but not $eliminate(C^\#)$ because $C^\#$ can consider the part of the array that makes the clause satisfiable. This problem happens on nearly all clauses, and in Theorem 8 we formalize this reasoning on the abstraction of the very simple clause $P(a) \rightarrow false$.

Theorem 8 (*Completeness* of $eliminate$, seemingly impossible)**.** *Assume the index type of arrays is infinite, for simplicity, let us consider arrays from and to integers. Consider $C^\#$ defined by $(\forall(k,v), v = a[k] \rightarrow P^\#(k,v)) \rightarrow false$ which was constructed using the $abstract$ algorithm with abstraction $abs(P) = \sigma_{Cell}$ on the trivial clause $P(a) \rightarrow false$. There exists no heuristic $insts$, such that:*

$$\forall \mathcal{M}^\#, [\![C^\#]\!]^\forall_{\mathcal{M}^\#} \Rightarrow [\![eliminate_{abs}(\{C^\#\})]\!]^\forall_{\mathcal{M}^\#}$$

Proof.

1. Let us compute $eliminate_{abs}(C^\#)$. We obtain:
   $(\bigwedge_{(k,v)\in S} v = a[k] \rightarrow P^\#(k, a[k])) \rightarrow false$ where S is the instantiation returned by $insts$.
2. Let i be an index not considered by the instantiation set S. This means $\forall v, (i,v) \notin S$. This is possible because S is an instantiation set and thus, must be finite, whereas the index set is supposed infinite.
3. Consider a model $\mathcal{M}^\#$ such that $\mathcal{M}^\#(P^\#) = \{(k,v) \mid k \neq i\}$.
4. Let us compute $[\![eliminate_{abs}(\{C^\#\})]\!]^\forall_{\mathcal{M}^\#}$
   $[\![eliminate_{abs}(\{C^\#\})]\!]^\forall_{\mathcal{M}^\#}$
   $\equiv \forall vars, [\![(\forall(k,v) \in S, v = a[k] \rightarrow P^\#(k,v)) \rightarrow false]\!]^{vars}_{\mathcal{M}^\#}$
   Unrolling the definition of $\mathcal{M}^\#$ and realizing that a is the only free variable
   $\equiv \forall vars, (\forall(k,v) \in S, v = [\![a]\!]^{vars}[k] \rightarrow k \neq i) \Rightarrow false$
   But the condition $k \neq i$ is always satisfied as S does not contain i
   $\equiv true \Rightarrow false \equiv false$
5. Let us compute $[\![C^\#]\!]^\forall_{\mathcal{M}^\#}$
   $[\![C^\#]\!]^\forall_{\mathcal{M}^\#}$
   $\equiv \forall vars, [\![(\forall(k,v), v = a[k] \rightarrow P^\#(k,v)) \rightarrow false]\!]^{vars}_{\mathcal{M}^\#}$
   $\equiv \forall vars, (\forall(k,v), v = [\![a]\!]^{vars}[k] \rightarrow k \neq i) \rightarrow false$
   Consider $(k,v) = (i, [\![a]\!]^{vars}[i])$
   $\equiv false \Rightarrow false \equiv true$
6. Thus, we have $[\![eliminate_{abs}(\{C^\#\})]\!]^\forall_{\mathcal{M}^\#} \equiv false$ and $[\![C^\#]\!]^\forall_{\mathcal{M}^\#} \equiv true$. We do not have the implication.

□

4.3.2 *Completeness* of $eliminate$ is possible

Previously, we proved that $eliminate$ could not verify *per clause completeness* and, because $eliminate$ handles each clause separately, we were led to believe that this meant that $eliminate$

could not satisfy *completeness*. Here, we investigate further and show that the *per clause completeness* property is not the right property as we did not account for abstraction. We name the right property is *per clause relative completeness* and it allows us to simplify the global property of *completeness* into a local (i.e. per clause) property. This brings us one step closer to finding a property for the instantiation heuristic $insts$.

Investigating *per clause completeness*. The idea behind why the completeness of $eliminate$ should imply *per clause completeness* can be formalized. The idea is to consider the use of $eliminate$ on an initial set $\mathfrak{C}^\#$ of clauses plus an additional clause $C^\#$. Because $eliminate$ handles each clause independently, the satisfiability of $C^\#$ can be expressed separately. Thus, by making $eliminate_{abs}(\mathfrak{C}^\#)$ satisfiable and $C^\#$ satisfiable, we enforce that $C^\#$ should be satisfiable by a model that also satisfies $eliminate_{abs}(\mathfrak{C}^\#)$. This reasoning is formally expressed in Theorem 9.

The final idea behind *per clause completeness* is that one may chose $\mathfrak{C}^\#$ so that it is satisfiable by only a single chosen model and thus $eliminate_{abs}(\mathfrak{C}^\#)$ is also only satisfiable by this single chosen model. Using the notations of Theorem 9, this idea means choosing $\mathfrak{C}^\#$ such that $\mathscr{D}^\#$ only contains the given chosen singleton model. If this were true, *per clause completeness* would be a required property for $eliminate$ to be complete. We will see this is not the case as we only considers clauses output by the $abstract$ algorithm.

Theorem 9 (Theorem behind *per clause completeness*)**.** *Let $\mathfrak{C}^\#$ be a set of Horn clauses and $C^\#$ be a clause. Let $\mathscr{D}^\# = \{\mathscr{M}^\# \mid [\![eliminate_{abs}(\mathfrak{C}^\#)]\!]^\forall_{\mathscr{M}^\#}\}$. If $eliminate_{abs}$ is complete then,*

$$(\exists \mathscr{M}^\# \in \mathscr{D}^\#, [\![C^\#]\!]^\forall_{\mathscr{M}^\#}) \Rightarrow (\exists \mathscr{M}^\# \in \mathscr{D}^\#, [\![eliminate_{abs}(\{C^\#\})]\!]^\forall_{\mathscr{M}^\#})$$

Proof.

1. Assume $eliminate_{abs}$ complete and assume $(\exists \mathscr{M}^\# \in \mathscr{D}^\#, [\![C^\#]\!]^\forall_{\mathscr{M}^\#})$. Introduce $\mathscr{M}^\#$.
2. The call to $eliminate_{abs}(\mathfrak{C}^\# \cup \{C^\#\})$ is complete thus:
   $(\exists \mathscr{M}^\#, [\![\mathfrak{C}^\# \cup \{C^\#\}]\!]^\forall_{\mathscr{M}^\#}) \Rightarrow (\exists \mathscr{M}^\#, [\![eliminate_{abs}(\mathfrak{C}^\# \cup \{C^\#\})]\!]^\forall_{\mathscr{M}^\#})$
3. The $\mathscr{M}^\#$ introduced in Item 1 verifies $[\![\mathfrak{C}^\# \cup \{C^\#\}]\!]^\forall_{\mathscr{M}^\#}$ as
   (a) It satisfies the clauses $\mathfrak{C}^\#$ as it is in $\mathscr{D}^\#$ as any model that satisfies the clauses in $eliminate_{abs}(\mathfrak{C}^\#)$ also satisfies the clauses $\mathfrak{C}^\#$ as shown by Theorem 6
   (b) It satisfies $C^\#$ by hypothesis.
4. Thus we have $(\exists \mathscr{M}^\#, [\![eliminate_{abs}(\mathfrak{C}^\# \cup \{C^\#\})]\!]^\forall_{\mathscr{M}^\#})$ using Item 2. Introduce that model as $\mathscr{M}^\#_2$.
5. $\mathscr{M}^\#_2$ is in $\mathscr{D}^\#$ as it satisfies the clauses in $eliminate_{abs}(\mathfrak{C}^\#)$ as $eliminate_{abs}(\mathfrak{C}^\#) \subseteq eliminate_{abs}(\mathfrak{C}^\# \cup \{C^\#\})$.
6. $\mathscr{M}^\#_2$ satisfies the clause $eliminate_{abs}(\{C^\#\})$ as $eliminate_{abs}(C^\#) \subseteq eliminate_{abs}(\mathfrak{C}^\# \cup \{C^\#\})$.
7. $\mathscr{M}^\#_2$ is in $\mathscr{D}^\#$ and satisfies $eliminate_{abs}(\{C^\#\})$, the proof is thus complete.

□

Finding the loophole. The idea that we can set $\mathfrak{C}^\#$ so that it expresses any model is fundamentally correct: in our setting, we do not limit the theory for the syntax of Horn clauses and thus, with the appropriate theory, one could express any model. However, in our setting, it is incorrect: the clauses on which the $eliminate$ algorithm are executed come from the $abstract$ algorithm and these clauses cannot express all models, they can only express abstracted models!

The problem expressed by Theorem 8 was only possible by considering non-abstracted models: the chosen counterexample model $\mathscr{M}^\#(P^\#) = \{(k, v) \mid k \neq i\}$ is not the abstraction of an array as the cell abstraction of a set of arrays is either empty or contains at least a value for **all** indices. Here the index i is omitted.

Therefore the *per clause completeness* property must be modified to take into account the abstraction, and leads to a property we call *per clause relative completeness*. This *per clause relative completeness* property is formalized in Definition 23 and is a modification of the *per clause completeness* property that considers only abstracted models.

Finally, in Theorem 10, we prove that the *completeness* requirement for $eliminate$ can be transformed into a *per clause relative completeness* requirement. This is a great simplification, as $completeness$ is a global property that involves the full Horn problem whereas *per clause relative completeness* is a property local to the clause.

Definition 23 (Per clause relative completeness)**.** *Let $\mathscr{G} = (\alpha, \gamma)$ be an abstraction, the transformation of a clause $C^\#$ into a clause $C^\#_{insts}$ verifies per clause completeness relative to $\mathscr{G}$ if and only if:*

$$\forall \mathscr{M}, [\![C^\#]\!]^\forall_{\alpha(\mathscr{M})} \Rightarrow [\![C^\#_{insts}]\!]^\forall_{\alpha(\mathscr{M})}$$

Theorem 10 (Completeness can be simplified into per clause relative completeness)**.** *If $eliminate_{abs}$ verifies per clause completeness relative to the abstraction $\mathscr{G}^{\sigma^{abs}}$, then $eliminate_{abs}$ verifies completeness on the output of $abstract_{abs}$. Thus, the data-abstraction framework algorithm implements the abstraction.*

Furthermore, per clause relative completeness is also necessary for completeness of $eliminate_{abs}$ on the output of $abstract$ if we allow the syntax of Horn clauses to express any model, that is: for any model $\mathscr{M}$, there exists a set of Horn clauses $\mathfrak{C}$ such that $\mathscr{M}$ is the only model that satisfies $\mathfrak{C}$. In other words, without syntax limitations, per clause relative completeness is the right property.

Proof. Let $\mathscr{G} = \mathscr{G}^{\sigma^{abs}}$ and let us write $eliminate$ and $abstract$ instead of $eliminate_{abs}$ and $abstract_{abs}$.

1. Assume $eliminate$ verifies *per clause relative completeness*. Let $\mathfrak{C}^\#$ be a set of Horn clauses output by $abstract$ on the input $\mathfrak{C}$ and assume $\mathfrak{C}^\#$ satisfiable. Let us prove that $eliminate(\mathfrak{C}^\#)$ is also satisfiable.
   (a) Let $\mathscr{M}^\#$ be a model that satisfies $\mathfrak{C}^\#$
   (b) $\alpha \circ \gamma(\mathscr{M}^\#)$ also satisfies $\mathfrak{C}^\#$. Proof:

$$\begin{aligned}
& H_{\mathfrak{C}^\#}(\alpha \circ \gamma(\mathscr{M}^\#)) \\
& \equiv H_{\mathfrak{C}}(\gamma \circ \alpha \circ \gamma(\mathscr{M}^\#)) && \text{Using Theorem 5} \\
& \equiv H_{\mathfrak{C}}(\gamma(\mathscr{M}^\#)) && \text{Using item 5 of Definition 8} \\
& \equiv H_{\mathfrak{C}^\#}(\mathscr{M}^\#) && \text{Using Theorem 5} \\
& \equiv true
\end{aligned}$$

   (c) By applying the definition of *per clause relative completeness*, we obtain that for any clause $C^\# \in \mathfrak{C}^\#$, $[\![C^\#]\!]^\forall_{\alpha\circ\gamma(\mathscr{M}^\#)} \Rightarrow [\![eliminate(\{C^\#\})]\!]^\forall_{\alpha\circ\gamma(\mathscr{M}^\#)}$
   (d) But $\alpha \circ \gamma(\mathscr{M}^\#)$ also satisfies $\mathfrak{C}^\#$, thus $[\![eliminate(\mathfrak{C}^\#)]\!]^\forall_{\alpha\circ\gamma(\mathscr{M}^\#)}$
   (e) Thus, $eliminate(\mathfrak{C}^\#)$ satisfiable and $eliminate$ verifies completeness on the output of abstract
2. Let us prove that given a syntax that may express any single model, if $eliminate$ verifies completeness on the output of $abstract$ then $eliminate$ verifies *per clause relative com-*

pleteness. Reason by contradiction and assume there exists a clause $C^\#$ that can be output by $abstract$ and a model $\mathscr{M}$ such that $[\![C^\#]\!]^\forall_{\alpha(\mathscr{M})} \wedge \neg [\![eliminate(\{C^\#\})]\!]^\forall_{\alpha(\mathscr{M})}$.

(a) Let $\mathfrak{C}$ be a set of Horn clauses that are only satisfiable by $\gamma \circ \alpha(\mathscr{M})$. We use the expressiveness of the syntax assumption.

(b) Let $\mathfrak{C}^\# = abstract_{abs}(\mathfrak{C})$. $\mathfrak{C}^\#$ is satisfiable only by $\alpha(\mathscr{M})$ as $abstract$ implements the abstraction and $\alpha \circ \gamma \circ \alpha = \alpha$. Proof:

$$
\begin{aligned}
& H_{\mathfrak{C}^\#}(\mathscr{M}_2^\#) \\
& \equiv H_{\mathfrak{C}}(\gamma(\mathscr{M}_2^\#)) && \text{Using Theorem 5} \\
& \equiv \gamma(\mathscr{M}_2^\#) = \gamma \circ \alpha(\mathscr{M}) && \text{As } H_{\mathfrak{C}} \text{ only satisfiable by } \gamma \circ \alpha(\mathscr{M}) \\
& \equiv \gamma(\mathscr{M}_2^\#) \leq \gamma \circ \alpha(\mathscr{M}) \wedge \gamma(\mathscr{M}_2^\#) \geq \gamma \circ \alpha(\mathscr{M}) \\
& \equiv \mathscr{M}_2^\# \leq \alpha \circ \gamma \circ \alpha(\mathscr{M}) \wedge \mathscr{M}_2^\# \geq \alpha \circ \gamma \circ \alpha(\mathscr{M}) && \text{Using item 3 of Definition 8} \\
& \equiv \mathscr{M}_2^\# \leq \alpha(\mathscr{M}) \wedge \mathscr{M}_2^\# \geq \alpha(\mathscr{M}) && \text{Using item 5 of Definition 8} \\
& \equiv \mathscr{M}_2^\# = \alpha(\mathscr{M})
\end{aligned}
$$

(c) $\mathfrak{C}^\# \cup \{C^\#\}$ is satisfiable only by $\alpha(\mathscr{M})$.

(d) But $eliminate(\mathfrak{C}^\# \cup \{C^\#\})$ is not satisfiable by $\alpha(\mathscr{M})$ because $\neg [\![eliminate(\{C^\#\})]\!]_{\alpha(\mathscr{M})}$.

(e) $eliminate$ is not satisfiable by any other model: if it was satisfiable by a model $\mathscr{M}_2^\#$, then $\mathfrak{C}^\# \cup \{C^\#\}$ would be satisfiable by $\mathscr{M}_2^\#$ using Theorem 6 and $\mathscr{M}_2^\# = \alpha(\mathscr{M})$ by unicity of the model satisfying $\mathfrak{C}^\# \cup \{C^\#\}$.

(f) Thus $eliminate(\mathfrak{C}^\# \cup \{C^\#\})$ cannot satisfiable by any model yet $\mathfrak{C}^\# \cup \{C^\#\}$ is satisfiable. Thus $eliminate$ is not complete. Contradiction.

□

4.3.3 *Completeness of a call* to $insts$

Our goal to ensure that the data-abstraction framework algorithm implements the abstraction has been reduced to the *completeness* of $eliminate$ and then to the *per clause relative completeness* of $eliminate$. Our aim is to reduce to a condition on $insts$ and in this section, we finally do so.

Computing the right property. This step requires to unwind the definition of the $eliminate$ algorithm. In the $eliminate$ algorithm, we first transform the goal of the clause. This step preserves equivalence and causes no issues. For each clause, we then successfully instantiate the quantifiers due to the abstraction of each predicate. Our intuition is that we wish to preserve *per clause relative completeness* throughout each instantiation. In other words, we aim to prove that the transformation of $e_{res_1} \wedge \ldots e_{res_{i-1}} \wedge \ldots e_i \wedge \ldots e_n \rightarrow e'_{res}$ into $e_{res_1} \wedge \ldots e_{res_i} \wedge e_{i+1} \wedge \ldots e_n \rightarrow e'_{res}$ verifies *per clause relative completeness.* If this property is true for all i, then we can deduce that $eliminate$ verifies *per clause relative completeness.*

Let us compare the two clauses $C = e_{res_1} \wedge \ldots e_{res_{i-1}} \wedge \ldots e_i \wedge \ldots e_n \rightarrow e'_{res}$ and $C' = e_{res_1} \wedge \ldots e_{res_i} \wedge \ldots e_{i+1} \wedge \ldots e_n \rightarrow e'_{res}$, where e_i is the abstraction of a predicate (i.e. we enter step 3b). To do so, let us rewrite C and C' in a currified manner, so that the similarities may be more explicit. $C \equiv e_i \rightarrow (e_{res_1} \wedge \ldots e_{res_{i-1}} \wedge e_{i+1} \wedge \ldots e_n \rightarrow e'_{res})$ and $C' \equiv e_{res_i} \rightarrow (e_{res_1} \wedge \ldots e_{res_{i-1}} \wedge e_{i+1} \wedge \ldots e_n \rightarrow e'_{res})$. Using the definition of ctx_i from the algorithm and expanding e_i and e_{res_i}, this can simply be written as $C \equiv \forall a^\#, F_{abs(P_i)}(a^\#, a_i) \rightarrow P_i^\#(a^\#) \rightarrow ctx_i$ and $C' \equiv (\bigwedge_{(a^\#,q) \in S_i} F_{abs(P_i)}[q](a^\#, a_i) \rightarrow P_i^\#(a^\#)) \rightarrow ctx_i$.

We recall that the call to $insts$ is $insts_P^{abs}(a_i, ctx_i)$, and thus the definition of the *completeness of a call* to $insts$ is computed by unrolling the definition of *the transformation of C into C' verifies*

per clause relative completeness with the notations of the parameters of $insts$. This is formalized in Definition 24. The formal proof that this definition is correct and allows $eliminate$ to verify *per clause relative completeness* and thus, that the data-abstraction framework implements the abstraction if and only if[1] all calls to $insts$ in $eliminate$ are *complete* is given in Theorem 11. We do not provide examples on which we use Definition 24 in this chapter, however, the proof of Theorem 15 contains two interesting examples.

Definition 24 (Completeness of a call to $insts$)**.** .

We say that a call $insts_P^{abs}(a, ctx)$ *returning* $\boldsymbol{I}$ *is* complete *if and only if, for any* $\mathscr{M}$*,*

$$\begin{aligned}&\left(\forall vars, [\![\forall (a^\#, q), F_{abs(P)}[q](a^\#, a) \Rightarrow a^\# \in \alpha_{abs(P)}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right)\\ &\quad\Rightarrow \left(\forall vars, [\![\forall (a^\#, q) \in \boldsymbol{I}, F_{abs(P)}[q](a^\#, a) \Rightarrow a^\# \in \alpha_{abs(P)}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right) \quad (4.5)\end{aligned}$$

We use several other versions of Equation 4.5 in our proofs: combinations of contraposition and cases where $F_{abs(P)}$ *does not contain existential quantifiers – i.e.* $F_{abs(P)} = F_{abs(P)}[()]$ *–, in which case we introduce* I' *such that* $I = \{(a^\#, ()) \mid a^\# \in I'\}$*. The contraposition versions are used to transform universal quantifiers into existential ones and is more suited for most proofs: from vars we need to construct a* $vars'$*; and the cases where* $F_{abs(P)}$ *does not contain existential quantifiers are mainly used for cell abstractions and simplify explanations.*

$$\begin{aligned}&\left(\exists vars, [\![\forall (a^\#, q) \in \boldsymbol{I}, F_{abs(P)}[q](a^\#, a) \Rightarrow a^\# \in \alpha_{abs(P)}(\mathscr{M}(P))]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right)\\ &\quad\Rightarrow \left(\exists vars', [\![\forall (a^\#, q), F_{abs(P)}[q](a^\#, a) \Rightarrow a^\# \in \alpha_{abs(P)}(\mathscr{M}(P))]\!]^{vars'} \wedge \neg [\![ctx]\!]^{vars'}_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right) \quad (4.6)\end{aligned}$$

$$\begin{aligned}&\left(\forall vars, \sigma([\![a]\!]^{vars}) \subseteq \alpha_{abs(P)}(\mathscr{M}(P)) \Rightarrow [\![ctx]\!]_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right)\\ &\quad\Rightarrow \left(\forall vars, \sigma([\![a]\!]^{vars}) \cap [\![\boldsymbol{I'}]\!]^{vars} \subseteq \alpha_{abs(P)}(\mathscr{M}(P)) \Rightarrow [\![ctx]\!]_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right) \quad (4.7)\end{aligned}$$

$$\begin{aligned}&\left(\exists vars, \sigma([\![a]\!]^{vars}) \cap [\![\boldsymbol{I'}]\!]^{vars} \subseteq \alpha_{abs(P)}(\mathscr{M}(P)) \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right)\\ &\quad\Rightarrow \left(\exists vars', \sigma([\![a]\!]^{vars'}) \subseteq \alpha_{abs(P)}(\mathscr{M}(P)) \wedge \neg [\![ctx]\!]^{vars'}_{\alpha_{\mathscr{G}^{\sigma^{abs}}(\mathscr{M})}}\right) \quad (4.8)\end{aligned}$$

Theorem 11 (Completeness of a call to $insts$ implies completeness of $eliminate$)**.** $eliminate_{abs}$ *verifies per clause completeness relative to* $\mathscr{G}^{\sigma^{abs}}$ *if and only if all calls to* $insts$ *during its execution are complete.*

This means, using Theorems 10 and 7, that whenever all calls to $insts$ *are complete during the execution of* $eliminate$ *that* $eliminate$ *is complete, the data-abstraction framework algorithm implements the abstraction. Furthermore, given an expressive syntax for Horn clauses, all the necessary conditions of these theorems are also satisfied, thus, this property* ***exactly*** *captures what* $insts$ *should satisfy*[2]*.*

Proof of the sufficient condition. Assume all calls $insts$ are complete during the execution of $eliminate$. Assume $C^\#$ satisfiable by a model $\alpha(\mathscr{M})$, where $\alpha = \alpha_{\mathscr{G}^{\sigma^{abs}}}$, and let us show $eliminate(\{C^\#\})$ satisfiable by $\alpha(\mathscr{M})$.

[1] if the syntax is expressive enough

[2] Of course, this is in part due to the parameters of $insts$. If $insts$ had had more context than only the current clause, the per clause relative completeness property would not have been necessary.

1. Using the notations of the $eliminate$ algorithm we prove by induction on i that $e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge \ldots \wedge e_i \wedge \ldots e_n \rightarrow e'_{res}$ is satisfied by $\alpha(\mathscr{M})$.
2. The base case is simply $C^\#$ after handling the e'. Handling e' does not change satisfiability, thus, it is satisfied by $\alpha(\mathscr{M})$.
3. Let us now assume $C^\#_i \equiv e_{res_1} \wedge \ldots \wedge e_{res_{i-1}} \wedge e_i \wedge \ldots \wedge e_n \rightarrow e'_{res}$ satisfied by $\alpha(\mathscr{M})$ and let us show that $C^\#_{i+1} \equiv e_{res_1} \wedge \ldots \wedge e_{res_i} \wedge e_{i+1} \wedge \ldots \wedge e_n \rightarrow e'_{res}$ is satisfied by $\alpha(\mathscr{M})$.
4. Rewrite these two clauses as:
   $C^\#_i \equiv (\forall a^\#, F_{abs(P_i)}(a^\#, a_i) \rightarrow P^\#_i(a^\#)) \rightarrow ctx_i$
   $C^\#_{i+1} \equiv (\bigwedge_{(a^\#,q) \in insts^{abs}_{P_i}(a_i,ctx_i)} F_{abs(P_i)}[q](a^\#, a_i) \rightarrow P^\#_i(a^\#)) \rightarrow ctx_i$
5. Using the definition of completeness of the call to $insts^{abs}_{P_i}(a_i, ctx_i)$ for $\alpha(\mathscr{M})$, we obtain that $C^\#_i$ implies $C^\#_{i+1}$.
6. By induction hypothesis, we have $C^\#_i$ satisfiable by $\alpha(\mathscr{M})$, thus $C^\#_{i+1}$ is as well and our induction is finished.

□

Proof of the necessary condition. Reason by contradiction and assume there is a call to $insts$ which is not complete.

1. Use the notations of the $eliminate$ algorithm, and let i be the smallest i for which the call $insts^{abs}_{P_i}(a_i, ctx_i)$ is not complete.
2. Introduce $\mathscr{M}$ corresponding to a model for which that incompleteness occurs. Let $\alpha = \alpha_{\mathscr{G}^{\sigma^{abs}}}$.
3. Assume $C^\#$ satisfiable by $\alpha(\mathscr{M})$, let us show $eliminate(\{C^\#\})$ is not satisfiable by $\alpha(\mathscr{M})$.
4. Using the notations introduced for the proof of the sufficient condition, we have:
   (a) We do not have $C^\#_i$ implies $C^\#_{i+1}$ for $\alpha(\mathscr{M})$ by definition of $insts^{abs}_{P_i}(a_i, ctx_i)$ not complete for $\mathscr{M}$. Thus, $C^\#_{i+1}$ must not be satisfiable by $\alpha(\mathscr{M})$.
   (b) $eliminate(\{C^\#\}) \rightarrow C^\#_{i+1}$ for $\alpha(\mathscr{M})$ as we only make the clause *less satisfiable*. The proof of this is already done in the proof of Theorem 6. Thus, $eliminate(\{C^\#\})$ is not satisfiable by $\alpha(\mathscr{M})$ as we do not have $C^\#_{i+1}$ satisfiable by $\alpha(\mathscr{M})$.

□

4.3.4 Discussion of a few subtle choices

This chapter contains many subtle choices about the $eliminate$ algorithm and the $insts$ heuristic. We believe that now the reader has a full picture of the $eliminate$ algorithm and its link with $insts$, some of the subtle choices we made may be better understood.

The choice of clause locality. The choice that the context to $insts$, that is, the second parameter, is exactly the rest of the clause. We discussed this briefly in the construction of the $eliminate$ algorithm and this was mainly justified by the fact that the clause is a nice syntactical unit. In fact, this choice can be explained more thoroughly with two arguments.

First, clauses are the equivalent of transitions in programs. The $abstract$ algorithm may be viewed as a way to compute the optimal abstract transition. The $eliminate$ algorithm may be viewed as a way to simplify this abstract transition so that it fits the format of Horn clauses. Thus, together, $abstract$ and $eliminate$ are simply a way to compute the abstraction of a transition. It befits that they only depend on the transition and the knowledge of the abstraction used.

Secondly, one of the main goals is to transform the property *eliminate is complete* as properties of the calls to the instantiation heuristic $insts$. If the full Horn problem is passed as a context

parameter to $insts$, there is no way to simplify this property in a useful manner. Obviously, if no context is given, one may not achieve *completeness*. Therefore, one has to choose a context expressive enough to make completeness achievable, but yet simple enough that a simple property of completeness for the calls to the instantiation heuristic can be written. Clauses are the right choice: predicates have arguments over expressions defined on the variables of the clause. Thus, anything smaller than the clause does not reflect the constraints on these variables, and thus, on the predicate's arguments. Furthermore, taking as parameter the clause allows us to rewrite using currification the different steps of the $eliminate$ algorithm and prove that completeness is maintained throughout execution.

The choice of a sequential computation of the instantiation sets. During the construction of the $eliminate$ algorithm, we discussed that the expressions $e_1, \dots e_n$ are instantiated sequentially and the instantiation set for e_2 may depend on the e_{res_1}, and thus on the instantiation set for e_1. A natural alternative is to use *parallel instantiation*, where e_2 depends on e_1 instead of e_{res_1}. This was discussed previously and Definition 25 formalizes it.

The main reason to avoid parallel instantiation is that we do not believe the parallel counterpart to Theorem 11 stated in Conjecture 1 to hold, that is, the completeness of the calls to $insts$ would no longer imply the completeness of $eliminate$. Theorem 11 relies on the fact that in the sequential version, each predicate instantiation is handled sequentially and equivalence is preserved at each step. In the parallel version, that is, where $insts$ is given $e_1 \wedge \dots e_{i-1} \wedge e_{i+1} \dots e_n \rightarrow e'_{res}$ as context for all i, Theorem 11 would require to unify the branches. That is, for $n = 2$, one needs to prove that if e_1 is equivalent to e_{res_1} in the context $e_2 \rightarrow e'_{res}$, and e_2 is equivalent to e_{res_2} in the context $e_1 \rightarrow e'_{res}$, then $e_1 \wedge e_2$ is equivalent to $e_{res_1} \wedge e_{res_2}$ in the context e'_{res}.

Although we have not yet found a counter-example for Conjecture 1, we do not believe this unification possible for two reasons. First, the proof of Conjecture 1 requires a different approach than that of Theorem 11: in Theorem 11 we prove the completeness of $eliminate$ for model $\mathscr{M}$ by using that each call is complete for that same model $\mathscr{M}$ but in Theorem 12 we prove such an approach is not possible for parallel instantiation. Secondly, we attempted another proof for Conjecture 1 provided beneath it and a specific step fails. The failure of this step makes us believe that Conjecture 1 does not hold and that we only need to find the right counter-example. However, should Conjecture 1 be true, there still are two reasons to use sequential instantiation over parallel instantiation.

First, e_{res_i} is simpler than e_i: e_i contains additional quantifiers that were removed when transforming it to e_{res_i}. Thus it makes sense to pass e_{res_i} instead of e_i as much as possible to the instantiation heuristic. Although this may not look like much, handling quantifiers within the instantiation heuristic is complex and may lead to cases where the completeness of the call is lost, and thus, having it already simplified enables better instantiations.

Secondly, there are cases where the instantiation of e_{res_i} does not verify completeness. In those cases, passing e_i instead of e_{res_i} to the instantiation of e_{i+1} makes the information loss completely untrackable. With our scheme, if only the call to the instantiation of e_i is incomplete, the completeness of the other calls guarantees that the information loss is entirely contained in the transformation of $C_i^\#$ to $C_{i+1}^\#$ (using the notations from the proof of Theorem 11).

Definition 25 (Parallel instantiation algorithm $eliminate^{//}_{abs}$)**.** *Let $eliminate^{//}_{abs}$ be the $eliminate_{abs}$ algorithm of Algorithm 2 where $ctx_i = e_1 \wedge \dots \wedge e_{i-1} \wedge e_{i+1} \wedge \dots \wedge e_n \rightarrow e'_{res}$ instead of $e_{res_1} \wedge \dots \wedge e_{res_{i-1}} \wedge e_{i+1} \wedge \dots \wedge e_n \rightarrow e'_{res}$.*

Theorem 12 (The proof for parallel instantiation must use different models)**.**

We may not have per-clause relative completeness of $eliminate^{//}_{abs}$ *for* $\mathscr{M}$*, that is, we have* $C^{\#}, \mathscr{M}$ *such that:*

$$[\![C^{\#}]\!]^{\forall}_{\alpha(\mathscr{M})} \wedge \neg [\![eliminate^{//}_{abs}(C^{\#})]\!]^{\forall}_{\alpha(\mathscr{M})}$$

and yet have all calls to $insts$ *complete for* $\mathscr{M}$*, that is Equation 4.6 for the model* $\mathscr{M}$*.*

Proof. We give the construction and ideas of the proof and leave the unrolling of the definitions to the reader.

1. Let C be $P(a) \wedge P'(a) \wedge a[0] = 1 \rightarrow false$ and
2. Let abs such that $abs(P) = abs(P') = \sigma_{Cell}$.
3. Let $C^{\#}$ be $abstract_{abs}(C)$.
4. Let $\mathscr{M}$ such that $\mathscr{M}(P) = \mathscr{M}(P') = \{a \mid a[0] \neq 1\}$.
5. Let $insts$ such that $insts$ always returns the empty set.
6. We have $[\![C^{\#}]\!]^{\forall}_{\alpha(\mathscr{M})}$ because $\mathscr{M}(P) = \mathscr{M}(P') = \{a \mid a[0] \neq 1\}$.
7. We have $\neg [\![eliminate^{//}_{abs}(C^{\#})]\!]^{\forall}_{\alpha(\mathscr{M})}$ because after instantiation with the empty set, the clause is simply $a[0] = 1 \rightarrow false$ which does not hold for all values of a.
8. All calls to $insts$ are complete during $eliminate^{//}_{abs}$ as the context always evaluates to $true$.

□

Conjecture 1 (Parallel instantiation also works)**.** $eliminate^{//}_{abs}$ *verifies per clause completeness relative to* $\mathscr{G}^{\sigma^{abs}}$ *if and only if all calls to* $insts$ *during its execution are complete.*

Proof attempt of Conjecture 1 for the clause $abstract_{abs}(P_1(e_1) \wedge P_2(e_2) \rightarrow e')$.

We reuse the notations of Algorithm 2 with our new ctx_i. Let $\sigma_1 = abs(P_1)$ and $\sigma_2 = abs(P_2)$. Assume the calls $insts^{abs}_{P_1}(e_1, ctx_1)$ and $insts^{abs}_{P_2}(e_2, ctx_2)$ are complete. Thus we have for any $\mathscr{M}, i \in \{1,2\}$ where $\neg i$ is 1 if $i = 2$ and 2 if $i = 1$,

If $vars$ satisfies		We have $vars'$ satisfying	
$[\![\sigma_i(e_i) \cap \boldsymbol{S_i} \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_i)]\!]^{vars}$	(4.9)	$[\![\sigma_i(e_i) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_i)]\!]^{vars'}$	(4.10)
$[\![\sigma_{\neg i}(e_\ni) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_{\neg i})]\!]^{vars}$	(4.11)	$[\![\sigma_{\neg i}(e_{\neg i}) \cap \boldsymbol{S_{\neg i}} \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_{\neg i})]\!]^{vars'}$	(4.12)
$\neg [\![e'_{res}]\!]^{vars}$	(4.13)	$\neg [\![e'_{res}]\!]^{vars'}$	(4.14)

And let us show per clause relative completeness. Introduce $\mathscr{M}$ and reason by contradiction, thus assume $\mathscr{M}$ does not satisfy $eliminate_{abs}(abstract_{abs}(P_1(e_1) \wedge P_2(e_2) \rightarrow e'))$. This means:

We have $vars$ such that		We need to show we have $vars'$ such that	
$[\![\sigma_1(e_1) \cap \boldsymbol{S_1} \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_1)]\!]^{vars}$	(4.15)	$[\![\sigma_1(e_1) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_1)]\!]^{vars'}$	(4.16)
$[\![\sigma_2(e_2) \cap \boldsymbol{S_2} \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_2)]\!]^{vars}$	(4.17)	$[\![\sigma_2(e_2) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_2)]\!]^{vars'}$	(4.18)
$\neg [\![e'_{res}]\!]^{vars}$	(4.19)	$\neg [\![e'_{res}]\!]^{vars'}$	(4.20)

1. **We construct a new model due to Theorem 12.** Let $\mathscr{M}_2$ such that $\mathscr{M}_2(P_2) = \mathscr{M}(P_2) \cup \{[\![e_2]\!]^{vars}\}$ and $\mathscr{M}_2(P_1) = \mathscr{M}(P_1)$.
2. We can apply completeness of the call leading to S_1 for $\mathscr{M}_2, vars$ as we have:
   (a) $[\![\sigma_1(e_1) \cap S_1 \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}_2)(P^{\#}_1)]\!]^{vars}$ using 4.15 as $\mathscr{M}_2(P_1) = \mathscr{M}(P_1)$.
   (b) $[\![\sigma_2(e_2) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}_2)(P^{\#}_2)]\!]^{vars}$ by definition of $\mathscr{M}_2(P_2)$.
   (c) $\neg [\![e'_{res}]\!]^{vars}$ using 4.19
3. This gives us $vars_1$ satisfying properties for $\mathscr{M}_2$ that we need to link with properties for $\mathscr{M}$.
   (a) $[\![\sigma_1(e_1) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M}_{\boldsymbol{2}})(P^{\#}_1)]\!]^{vars_1}$ which is equivalent to $[\![\sigma_1(e_1) \subseteq \alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})(P^{\#}_1)]\!]^{vars_1}$

   (b) $[\![\sigma_2(e_2) \subseteq \alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M}_\mathbf{2})(P_2^\#)]\!]^{vars_1}$ **that we would want to imply** $[\![\sigma_2(e_2) \cap S_2 \subseteq \alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})(P_2^\#)]\!]^{vars_1}$. **This is not true and is the step that currently blocks the proof**. Should a *good* variation of this step be successfully proven, the proof can be continued as shown below.
   (c) $\neg[\![e'_{res}]\!]^{vars_1}$
4. We can use completeness of the call leading to S_2 for $\mathcal{M}, vars_1$ as we have
   (a) $[\![\sigma_2(e_2) \cap S_2 \subseteq \alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})(P_2^\#)]\!]^{vars_1}$
   (b) $[\![\sigma_1(e_1) \subseteq \alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})(P_1^\#)]\!]^{vars_1}$
   (c) $\neg[\![e'_{res}]\!]^{vars_1}$
5. Thus, we have $vars'$ that satisfies 4.16, 4.18 and 4.20. Our proof is finished.

□

Overview and contribution within this chapter

In this chapter, we express the data-abstraction framework algorithm as a composition of two algorithms, $abstract$ and $eliminate$, and we analyze their properties. The $abstract$ transformation implements the abstraction but fails to meet the syntax requirements and the $eliminate$ algorithm aims at fulfilling the syntax requirements while staying complete.

The principle behind the $abstract$ algorithm is well-known, that is, to consider the transition $\alpha \circ f \circ \gamma$, however, its adaptation as an algorithm is a full contribution: in most abstraction based works, the transition $\alpha \circ f \circ \gamma$ is computed by hand and not automatically. This was only possible through the use of Horn clauses and because we limit ourselves to data-abstractions.

Similarly, the principle behind the $eliminate$ algorithm, that is, quantifier instantiation is already widely used in the Sat-Modulo Theory community to handle problems with additional quantifiers. However, it usually falls into one of two categories: either there is a complete quantifier elimination procedure based on instantiation [BMS06], or there is no real framing of the heuristic used [BMR13]. Our contribution is to link this process with abstraction and correctly capture how each quantifier must be handled.

Finally, we analyze the properties of the algorithm of the data-abstraction framework, mainly whether it implements the abstraction which was the main concern of Chapter 3. This part is entirely a contribution.

We first reduce this property to the completeness of $eliminate$ on the output of $abstract$, and then aim to capture this condition of $eliminate$ as a condition on the instantiation heuristic $insts$.

Second, we reduce the global property $eliminate$ *is complete* to a property local to the clause. To do so, we first show that an intuitive version of such property, *per clause completeness* is not achievable and that by taking into account abstraction, we can recover from this defect. We obtain a property we name *per clause relative completeness*. This is a major improvement as the property is now local.

Third, we show that *per clause completeness* of $eliminate$ can be reduced to what we call the completeness of its calls to the instantiation heuristic $insts$. In Chapter 5, the main focus will be to construct $insts$ so that it verifies this property.

The property we obtain for $insts$ is not only sufficient to deduce that the algorithm of the data-abstraction framework implements the abstraction, it is also necessary if we do not consider restrictions on the theory of the Horn clauses. Therefore, this is the exact property that the calls to $insts$ must verify.

5 Data-Abstraction: instantiation heuristics

In Chapter 4 we showed that the data-abstraction algorithm requires an instantiation heuristic $insts$. In order for the data-abstraction algorithm to verify the specifications described in Chapter 3, each call to the heuristic $inst$ must verify a property called *completeness* as defined in Definition 24.

Although each call to the heuristic $insts_P^{abs}(a, ctx)$ takes as parameter the whole abstraction abs, the abstraction $abs(P)$ plays a special role: the number and types of the elements in the set returned by $insts_P^{abs}(a, ctx)$ only depend on $abs(P)$, and, when $F_{abs(P)}$ does not contain any quantifiers, instantiating using $insts_P^{abs}(a, ctx)$ consists in picking a subset of $\sigma(a)$. Thus, we define different instantiation heuristic for each $abs(P)$.

One of the goals of our data-abstraction framework is to define abstractions as combinations of basic abstractions by using combinators. Our instantiation heuristic $insts$ follows the same construction scheme, that is, we define $insts$ for the base abstractions and combinators, which can then be combined to handle any abstraction $abs(P)$.

The abstractions and combinators defined in Chapter 3 can be classified into two categories. First, the foundation of the data-abstraction framework: the identity abstraction, other data-abstractions that we call *finite*, and the combinators $\bullet, \circ, \oplus, \otimes$. The second category contains abstractions that aim to handle a specific unbounded data-structure. Here, we only consider *cell abstraction* but we hope that future work will consider other unbounded data-structures such as trees or even graphs.

The main difficulty encountered when constructing instantiation heuristics for the foundations of the data-abstraction framework is the way the context parameter is handled. For example, with the $\bullet$ abstraction combinator, one cannot simply instantiate each element of the pair separately: by doing so, the instantiation process of one element of the pair would lose information about the rest of the clause – namely the other element of the pair – and the completeness property would be lost. This gets even worse when considering the combinators $\circ$ and $\otimes$.

The main difficulty with cell abstraction is that the abstraction of a single array is an infinite set, and yet, $insts$ must choose a finite relevant subset that preserves semantics. In this chapter, we resolve this problem for cell abstraction, and we hope the technique we used can be adapted for other abstractions of data-structures.

We divide this chapter into two independent parts: the construction of $insts$ for cell abstraction and the construction of $insts$ for the foundations of the data-abstraction framework. Both parts can be read independently and we suggest reading the cell abstraction part first as we believe it is the most interesting. The conclusion of what can be handled with this instantiation heuristic is discussed in Chapter 6.

5.1 Instantiating Cell abstraction

The goal of this section is to provide an instantiation heuristic $inst_P^{abs}$ when $abs(P) = \sigma_{Cell}$. Furthermore, the formal definition of $inst_P^{abs}$ when $abs(P) = \sigma_{Cell}$ returns a set of elements of the form $((k,v),())$ where (k,v) are the index and value pair and $()$ is the empty tuple due to the absence of prenex existential quantifiers within $F_{\sigma_{Cell}}$. From now on, we will assume $abs(P) = \sigma_{Cell}$, and we will write $insts_{Cell}(a,ctx) = \{(k,v) \mid ((k,v),()) \in inst_P^{abs}(a,ctx)\}$.

Cell abstraction is defined as $\sigma_{Cell}(a) = \{(k,v) \mid v = a[k]\}$ and using the instantiation $insts_{Cell}(a,ctx)$ means transforming a clause of the form $(\forall(k,v), v = a[k] \rightarrow P^{\#}(k,v)) \wedge e_2 \wedge \dots e_n \rightarrow e'$ into $(\forall(k,v) \in \boldsymbol{S}, v = a[k] \rightarrow P^{\#}(k,v)) \wedge e_2 \wedge \dots e_n \rightarrow e'$ where S is the set returned by $insts_{Cell}(a, e_2 \wedge \dots e_n \rightarrow e')$. The goal is to construct $insts_{Cell}(a,ctx)$ so that the transformation of one clause into the other preserves semantics, or more formally, respects *per clause relative completeness*. Even though in Chapter 4 we already isolated this as a property named *completeness of call to* $inst_P^{abs}(a,ctx)$, we will mostly leave this property to the proofs and aim to give the intuition by considering the transformation between the two clauses.

The section is divided into three parts. In the first part we strive to prepare the reader by giving an intuition of three concepts that should guide the construction of instantiation heuristics for cell abstraction and its proof of completeness. In the second part, we formalize and study the completeness of instantiation heuristics derived from transformation algorithms of the literature that study quantified properties for arrays. Finally, we give our instantiation heuristic for cell abstraction.

5.1.1 Key concepts to construct $insts_{Cell}(a,ctx)$

To understand the key concepts to construct $S = insts_{Cell}(a,ctx)$, we suggest to tackle the small example $insts_{Cell}(a, true \rightarrow a[2] = 0)$. This call to $insts_{Cell}$ comes from the call to $eliminate$ on the clause $(\forall(k,v), v = a[k] \rightarrow P^{\#}(k,v)) \rightarrow a[2] = 0$, itself created by the $abstract$ algorithm on $P(a) \rightarrow a[2] = 0$. The clause generated by $eliminate$ is equivalent to $(\forall(k,v) \in S, v = a[k] \rightarrow P^{\#}(k,v)) \rightarrow a[2] = 0$. On this example we show three main concepts that apply generally.

The quantified variable v should be chosen as $a[k]$. There is no point in choosing $v \neq a[k]$. The aim of the instantiation set is to restrict the possible infinite set of abstract elements by a finite set, more formally, the goal is to transform $\forall a^{\#} \in \boldsymbol{\sigma(a)}$ into $\forall a^{\#} \in \boldsymbol{\sigma(a) \cap S}$, where S is the instantiation set. In cell abstraction, the abstracted elements are of the form $(k, a[k])$ where a is the abstracted array, thus if $v \neq a[k]$, then $\sigma_{Cell}(a) \cap (S \cup \{(a,v)\}) = \sigma_{Cell}(a) \cap S$ and there was no point in adding (k,v) in S. Now that v has been chosen, let us call I the set of indices k to be chosen. In other words, let us simply consider $S = \{(k, a[k]) \mid k \in I\}$. The instantiated clause can now be simplified into $(\forall k \in I, P^{\#}(k, a[k])) \rightarrow a[2] = 0$.

The context is key. We are now left with the problem of choosing the right finite subset I of indices k while preserving semantics. Intuitively, this may not seem possible. However, the key is to understand the context in which that possibly infinite conjunction is used. For example, the sentence *all apples are edible* is not equivalent to the sentence *red apples are edible*; however, in the context of eating a red apple, both imply that I can eat it, which is the only thing I may care about. This is our second intuition.

The initial non-abstracted clause states that the array variable a must verify $a[2] = 0$ at program point P; and thus, the context of the call to $insts_{Cell}$ for our clause is equivalent to $a[2] = 0$. This context has the specificity of only caring about the index 2. In other words, if a' is such that $a'[2] = a[2]$, then a' and a are equivalent with respect to that context. This gives the intuition that one should pick $I = \{2\}$; and this choice would give us the clause $P^\#(2, a[2]) \rightarrow a[2] = 0$.

To understand how limiting oneself to the index 2 works, let us consider the proof that the transformation of the non-instantiated clause $(\forall k, P^\#(k, a[k])) \rightarrow a[2] = 0$ into the instantiated clause $P^\#(2, a[2]) \rightarrow a[2] = 0$ preserves per clause relative completeness. We reason by contradiction and prove that whenever the instantiated clause is not satisfiable, neither is the non-instantiated clause. The reasoning is as follows.

If the instantiated clause is false then we have $P^\#(2, v)$ for some v different from zero. Then consider any array a such that $\forall k, P^\#(k, a[k])$ and let $a' = a[2 \leftarrow v]$. Notice that the only knowledge we have about a' is that $a'[2] = v$. This array a' verifies $\forall (k, v), v = a'[k] \rightarrow P^\#(k, v))$ and not $a'[2] = 0$. Thus, the non-instantiated clause is not verified.

Considering only abstracted models is important. In the previous proof, we omitted explicitly using models to ease readability. However, the previous proof would not have worked without our use of *per clause **relative** completeness* instead of *per clause completeness*. The key argument that is missing is the existence of a such that $\forall k, P^\#(k, a[k])$.

Formulated with models, this means that we need to prove $\exists a, \forall k, (k, a[k]) \in \mathcal{M}^\#(P^\#)$. This is not verified with a model such that $\mathcal{M}^\#(P^\#) = \{(k, v) \mid k \neq 1\}$ and thus, had we considered *per clause completeness*, the proof would have failed.

However, with *per clause **relative** completeness*, we know that $\mathcal{M}^\#(P^\#) = \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$ for some model $\mathcal{M}$. As we assumed $P^\#(2, v)$, that is, $(2, v) \in \mathcal{M}^\#(P^\#)$, thus $\mathcal{M}^\#(P^\#) \neq \emptyset$, from which we deduce that $\mathcal{M}(P) \neq \emptyset$. Thus, $\exists a \in \mathcal{M}(P)$ and this a verifies $\forall k, (k, a[k]) \in \mathcal{M}^\#(P^\#)$ as $\sigma_{Cell}(a) \subseteq \mathcal{M}^\#(P^\#)$.

This idea that by considering only abstracted models we avoid degenerate models such as $\mathcal{M}^\#(P^\#) = \{(k, v) \mid k \neq 1\}$ that are non-empty but do not contain a single array is a key part in all our proofs.

5.1.2 Instantiation heuristics based on literature

5.1.2.1 The transformations from programs of [MA15; MG16]

The [MA15] transformation. The program transformation of [MA15] aims at proving properties of the form $\forall k, P(k, a[k])$ and in that sense, it is very similar to cell abstraction. The idea behind the transformation is very similar to the program transformation technique we presented for the first cell abstraction of Example 10. However, instead of creating only a variable v representing $a[0]$, the transformation creates both an index variable k and a value variable v representing $a[k]$. The key idea is that because k is unbound in this new program, any property proven for k must stand for any k and thus, one obtains a final property of the form $\forall k, P(k, a[k])$.

Formally, the transformation is implemented as follows. First, for each array a, create two new variables k_a and v_a. Second, writing to an array a through an operation `a[i] <- v;` is replaced by the operation `if(i = k_a) v_a <- v;`. This makes complete sense as the only impact a write has on the value v_a which represents $a[k_a]$ is when $k_a = i$. Finally, replace all reads instructions `v <- a[i];` by the instruction `if(i = k_a) v <- v_a; else v = rand();`. The idea behind this transformation is that we are only tracking what happens at index k_a, thus, there is no information on the other indices. This lack of information on the cells with an index different than k_a is

symbolized by a random value. Note that the technique extends to multiple cells by considering several index and value variables for each array, similarly to what we did with $Cell^n$. An example of this program transformation is given in Example 22.

Example 22 (Array initialization transformation of [MA15])**.**

Listing 5.1 – Before abstraction

```
void array_init(Array<int> a)



{
  unsigned i=0;
  while(i<a.size())
  {
    a[i] <- 0;
    i <- i+1;
  }
  //Safety property
  i=0;
  while(i<a.size())
  {
    assert(a[i]=0);

    i <-i+1;
  }
}
```

Listing 5.2 – After abstraction

```
void array_init(
    int k /*the index*/,
    int v /* representing a[k] */
    unsigned n /*the size of a*/)
{
  unsigned i=0;
  while(i<n)
  {
    if(i=k) then v <- 0;
    i <- i+1;
  }
  //Safety property
  i=0;
  while(i<n)
  {
    if(i=k) then assert(v=0);
    else {int rnd=rand();assert(rnd=0);}
    i <- i+1;
  }
}
```

Adapting the [MA15] transformation to Horn clauses. The [MA15] transformation can be adapted to Horn clauses instead of programs by considering how the Horn clauses representing the initial program are matched by the Horn clauses representing the transformed program. For the type of programs handled by the transformation, their are three types of initial clauses: those corresponding to an array read, those corresponding to an array write and those not manipulating the array. For reasons of simplicity, we only consider one clause of each type, that is, those corresponding to the instructions `v <- a[i];`, `a[i] <- v;` and `v <- 0;`.

The transformation of each of those clauses by the adaptation of the [MA15] transformation follows closely the transformation of program instructions of [MA15]: first we change the type of the predicates such that the array variable a is now represented by the two variables k_a and v_a; second, we adapt the transition relations, more specifically:

1. Clause 5.1 represents the instruction `v <- a[i];`, which by the [MA15] transformation is transformed into the instruction `if(i = k_a) v <- v_a; else v = rand();`. Thus, if we denote by v' the value of v after the instruction, we have $v' = ite(i = k_a, v_a, rnd)$ where rnd is a fresh variable and thus denotes any value. This justifies how we obtain Clause 5.4.
2. Clause 5.2 represents the instruction `a[i] <- v;`, which by the [MA15] transformation is transformed into the instruction `if(i = k_a) v_a <- v;`. Thus, if we denote by v'_a the value of v after the instruction, we have $v'_a = ite(k_a = i, v, v_a)$. This justifies how we obtain Clause 5.5.
3. Clause 5.3 represents the instruction `v <- 0;`, which is not transformed in the [MA15] transformation. Thus, if we denote by v' the value of v after the instruction, we have $v' = 0$. This justifies how we obtain Clause 5.6.

$$P_1(a,i,v) \wedge v' = a[i] \rightarrow P_2(a,i,v') \tag{5.1}$$

$$P_1(a,i,v) \wedge a' = a[i \leftarrow v] \rightarrow P_2(a',i,v) \tag{5.2}$$

$$P_1(a,i,v) \wedge v' = 0 \rightarrow P_2(a,i,v') \tag{5.3}$$

$$P_1^\#((k_a,v_a),i,v) \wedge v' = ite(i = k_a, v_a, rnd) \rightarrow P_2^\#((k_a,v_a),i,v') \tag{5.4}$$

$$P_1^\#((k_a,v_a),i,v) \wedge v'_a = ite(k_a = i, v, v_a) \rightarrow P_2^\#((k_a,v'_a),i,v) \tag{5.5}$$

$$P_1^\#((k_a,v_a),i,v) \wedge v' = 0 \rightarrow P_2^\#((k_a,v_a),i,v') \tag{5.6}$$

The Horn clauses adaptation of the [MA15] transformation as a specific instantiation heuristic. The transformation of Clauses 5.1, 5.2 and 5.3 into the Clauses 5.4, 5.5 and 5.6 is equivalent to applying the data-abstraction algorithm with input abstraction $\sigma_{Cell} \bullet \sigma_{id} \bullet \sigma_{id}$ for both P_1 and P_2 with specific results for the instantiation heuristic.

To understand this, one needs to execute the data-abstraction algorithm on the Clauses 5.1, 5.2 and 5.3. In Clauses 5.7, 5.8 and 5.9 we show the result of this execution with instantiation sets of the form $S = \{((k,a[k]),i,v) \mid k \in I\}$ followed by simple expression simplifications. This choice behind this instantiation set form is that the variable v_a should be chosen as $a[k_a]$ and that i and v are variables that are abstracted with σ_{id}. Furthermore, the detailed calls to the instantiation heuristic during this process are given in Theorem 13.

$$(\forall k_a \in I_1, P_1^\#((k_a,a[k_a]),i,v) \wedge v' = a[i]) \rightarrow P_2^\#((k'_a,a[k'_a]),i,v') \tag{5.7}$$

$$(\forall k_a \in I_2, P_1^\#((k_a,a[k_a]),i,v) \wedge a' = a[i \leftarrow v]) \rightarrow P_2^\#((k'_a,a'[k'_a]),i,v) \tag{5.8}$$

$$(\forall k_a \in I_3, P_1^\#((k_a,a[k_a]),i,v) \wedge v' = 0) \rightarrow P_2^\#((k'_a,a[k'_a]),i,v') \tag{5.9}$$

To prove that the Horn clause adaptation of the [MA15] transformation may be viewed as an instance of our data-abstraction algorithm, one needs to find the values of I_1, I_2 and I_3 so that the clauses transformed by the adaptation are equivalent to those transformed by the data-abstraction algorithm. In the [MA15] transformation, the index of focus is constant throughout the transformation: we only look at what happens to the array at cell with index k. Here, we imitate this result by picking $I_1 = I_2 = I_3 = \{k'_a\}$, thus the first parameter of all predicates is always the same and equal to k'_a. Using that instantiation set and a few simplifications, we obtain Clauses 5.10, 5.11 and 5.12.

$$(P_1^\#((k'_a,a[k'_a]),i,v) \wedge v' = a[i]) \rightarrow P_2^\#((k'_a,a[k'_a]),i,v') \tag{5.10}$$

$$(P_1^\#((k'_a, a[k'_a]), i, v) \wedge a' = a[i \leftarrow v]) \rightarrow P_2^\#((k'_a, a'[k'_a]), i, v) \tag{5.11}$$

$$(P_1^\#((k'_a, a[k'_a]), i, v) \wedge v' = 0) \rightarrow P_2^\#((k'_a, a[k'_a]), i, v') \tag{5.12}$$

Finally, we need to prove that Clauses 5.10, 5.11 and 5.12 are equivalent to Clauses 5.4, 5.5 and 5.6. The only delicate part of this proofs consists in eliminating the array variable a from the free variables of Clauses 5.10, 5.11 and 5.12. The rest of the proof is mostly basic expression simplification.

To remove the array a from the clauses, we use a common technique on arrays of the Satisfiability Modulo Theory community. The idea for Clauses 5.10 and 5.12 in which the array a is only used within read expressions is to use a process called Ackermannisation [Ack57] which is commonly used to eliminate uninterpreted functions. The idea is to create a new free variable v_i for each different expression e_i at which the array a is read and replace $a[e_i]$ by v_i. Then the constraint $\bigwedge_{i \neq j} e_i = e_j \rightarrow a[e_i] = a[e_j]$ must be added. In the case of Clause 5.12, k'_a is the only expression at which a is read, and thus, we can just replace $a[k'_a]$ by a new fresh variable and remove the variable a. For Clause 5.10, there are two indices at which a is read: i and k'_a. To obtain Clause 5.4, we replace $a[k'_a]$ by the fresh variable v_a and $a[i]$ by the fresh variable rnd.

For Clause 5.11, the problem is slightly more complicated. First, using $a' = a[i \leftarrow v]$, we replace a' by $a[i \leftarrow v]$ everywhere and remove the use of a'. Then, we simplify $a'[k'_a]$ which is now $a[i \leftarrow v][k'_a]$ by $ite(i = k'_a, v, a[k'_a])$. Finally, we are left with only array reads and we can apply Ackermannisation.

This full reasoning is formalized in Theorem 13 which shows that Clauses 5.4, 5.5 and 5.6 are equivalent to the Clauses obtained by the data-abstraction algorithm of the Clauses 5.1, 5.2 and 5.3 with input abstraction $\sigma_{Cell} \bullet \sigma_{id} \bullet \sigma_{id}$ for both P_1 and P_2 and the instantiation corresponding to $I_1 = I_2 = I_3 = \{k'_a\}$.

Theorem 13 ([MA15] as an instance of the data-abstraction algorithm)**.** *The data-abstraction algorithm (i.e. $eliminate_{abs} \circ abstract_{abs}$) applied to the Clauses 5.1, 5.2 and 5.3 (i.e. the clauses corresponding to the input of the [MA15] transformation) yield clauses equivalent to Clauses 5.4, 5.5 and 5.6 (i.e. the clauses corresponding to the output of the [MA15] transformation) when:*

1. $abs(P_1) = abs(P_2) = \sigma_{Cell} \bullet \sigma_{id} \bullet \sigma_{id}$
2. $insts^{abs}_{P_1}((a, i, v), ctx) = \{(((k'_a, a[k'_a]), i, v), ())\}$, *where:*
   (a) $insts^{abs}_{P_1}((a, i, v), ctx)$ *represents the calls to the instantiation heuristic during the $eliminate$ algorithm.*
   (b) ctx *is given in Table 5.1.*

Clause	ctx			
Clause 5.1	$v' = a[i]$	$\rightarrow ((v'_a = a[k'_a] \wedge i^\# = i \wedge v'^\# = v')$	$\rightarrow P_2^\#((k'_a, v'_a), i^\#, v'^\#))$	(5.13)
Clause 5.2	$a' = a[i \leftarrow v]$	$\rightarrow ((v'_a = a'[k'_a] \wedge i^\# = i \wedge v^\# = v)$	$\rightarrow P_2^\#((k'_a, v'_a), i^\#, v^\#))$	(5.14)
Clause 5.3	$v' = 0$	$\rightarrow ((v'_a = a[k'_a] \wedge i^\# = i \wedge v'^\# = v)$	$\rightarrow P_2^\#((k'_a, v'_a), i^\#, v'^\#))$	(5.15)

Table 5.1 – Calls to $insts$ during data-abstraction of Clauses 5.1, 5.2 and 5.3

Proof.

1. The abstraction and elimination of Clause 5.1 yields:

$$P_1(a,i,v) \wedge v' = a[i] \rightarrow P_2(a,i,v')$$

After abstraction

$$\big(\forall((k_a,val),i^\#,v^\#), val = a[k_a] \wedge i^\# = i \wedge v^\# = v \rightarrow P_1^\#((k_a,val),i^\#,v^\#)\big) \wedge v' = a[i] \rightarrow \big(\forall((k'_a,val),i^\#,v^\#), val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#)\big)$$

After first step of the instantiation: moving quantifiers in e'.

$$\big(\forall((k_a,val),i^\#,v^\#), val = a[k_a] \wedge i^\# = i \wedge v^\# = v \rightarrow P_1^\#((k_a,val),i^\#,v^\#)\big) \wedge v' = a[i] \rightarrow val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#)$$

After second step of the instantiation

$$a[k'_a] = a[k'_a] \wedge i = i \wedge v = v \rightarrow P_1^\#((k'_a,a[k'_a]),i,v) \wedge v' = a[i] \rightarrow (val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#))$$

Simplifying trivial equalities and implications

$$P_1^\#((k'_a,a[k'_a]),i,v) \wedge v' = a[i] \rightarrow (val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#))$$

Removing aliases, that is, simplify $\forall x, x = j \rightarrow expr$ by $expr$ where x is replaced by j.

$$P_1^\#((k'_a,a[k'_a]),i,v) \wedge v' = a[i] \rightarrow P_2^\#((k'_a,a[k'_a]),i,v')$$

Ackermannisation of a

$$((k'_a = i) \rightarrow v_i = v_{k'_a}) \rightarrow (P_1^\#((k'_a,v_{k'_a}),i,v) \wedge v' = v_i \rightarrow P_2^\#((k'_a,v_{k'_a}),i,v'))$$

Rewriting v_i as $v_{k'_a}$ when $k'_a = i$ and rnd otherwise

$$P_1^\#((k'_a,v_{k'_a}),i,v) \wedge v' = ite(i = k'_a, v_{k'_a}, rnd) \rightarrow P_2^\#((k'_a,v_{k'_a}),i,v')$$

Renaming k'_a to k_a and $v_{k'_a}$ to v_a

$$P_1^\#((k_a,v_a),i,v) \wedge v' = ite(i = k_a, v_a, rnd) \rightarrow P_2^\#((k_a,v_a),i,v')$$

Our desired result.

2. The abstraction and elimination of Clause 5.2 follows exactly the same steps.
3. The abstraction and elimination of Clause 5.3 follows exactly the same steps.

□

Limits of this approach. In Example 22, we already see the problem: the transformed program is not satisfiable as we enter the branch `int rnd=rand();assert(rnd=0);` which fails. As Example 22 has invariants that are expressible by cell abstraction, we deduce that the Horn clauses transformation adapted from [MA15] is incomplete.

In fact, the even the simple array initialization program of Listing 2.1 of Chapter 2 cannot be proven with this technique. In the [MA15] paper, no explicit safety property is used and the backend tool infers the abstract invariant $k < n \rightarrow v = 0$ at the end of the program. From this inferred property, the user can deduce that this indeed corresponds to the concrete invariant $\forall k < n \rightarrow a[k] = 0$.

This drawback may not seem like much: enhancing the program transformation so that assertions of the form $\forall k < n, a[k] = 0$ are abstracted into $k < n \rightarrow v = 0$ seems easily feasible. However, because the properties such as $\forall k < n, a[k] = 0$ cannot truly be encoded as such, programs that use that property as an intermediate step cannot be verified, even with the enhancement. Example

23 demonstrates this problem by storing within a variable b whether there are only occurrences of zero values in the array. In the non-abstracted case, b can be verified to be $true$, whereas in the abstracted version b can either be $true$ or $false$. Thus any following instructions and safety properties based on the assumption that $b = true$ cannot be verified anymore.

This proves that the transformation technique of [MA15] does not satisfy relative completeness and that we can construct examples where this incompleteness has consequences: the incompleteness demonstrated previously shows that modularity is not truly possible as even an inlining[1] technique will fail.

Example 23 (The transformation of [MA15] is incomplete)**.**

Listing 5.3 – Before abstraction

```
void init_check(Array<int> a)
{
  unsigned i=0;
  while(i<a.size())
  {
    a[i] <- 0;
    i <- i+1;
  }
  //Safety property
  i=0;
  bool b=true;
  while(i<a.size())
  {
    if(a[i] ≠ 0)
      b <- false;
    i <- i+1;
  }
  //The value of b is true
}
```

Listing 5.4 – After abstraction

```
void init_check(
    int k /*the index*/,
    int v /* representing a[k] */
    unsigned n /*the size of a*/)
{
  unsigned i=0;
  while(i<n)
  {
    if(i=k) then v <- 0;
    i <- i+1;
  }
  //Safety property
  i=0;
  bool b=true;
  while(i<n)
  {
    if(i=k){
      if(v ≠ 0)
        b <- false;
    }
    else {
      int rnd = rand();
      if(rnd ≠ 0)
        b <- false;
    }
    i <- i+1;
  }
  //The value of b may be false
}
```

Improving the [MA15] instantiation with [MG16]. To improve on the [MA15] transformation, it is key to understand where the lack of relative completeness lies. As expected, the transformation of Clause 5.1 representing `v <- a[i];` into the Clause 5.4 representing `if(i = k_a) v <- v_a; else v = rand();` uses a random value which breaks the *per clause relative completeness* property. In Theorem 15 we show that the loss of information is only located in the transformation of Clause 5.1, by proving that the transformations of Clauses 5.2 and 5.3 verify *per clause relative completeness* whereas the transformation of Clause 5.1 does not.

[1] This process consists in handling non-recursive function calls by copying the code of the function in the function that calls it.

One of the main reasons why the [MA15] transformation was unable to yield a good transformation for the instruction `v <- a[i];` was that it attempted to transform programs into programs. In [MG16], the authors realized that by considering a transformation from programs to Horn clauses, they can improve their results. We adapt this transformation from programs to Horn clauses to a transformation between Horn clauses by considering the same three clauses. Only the transformation of Clause 5.1 is impacted, which makes sense as it was the only one which was incomplete. Clause 5.1 is now transformed into Clause 5.16

$$P_1^\#((k_a, v_a), i, v) \wedge \boldsymbol{P_1^\#((i, v_i), i, v)} \wedge v' = ite(i = k_a, v_a, \boldsymbol{v_i}) \wedge \Rightarrow P_2^\#((k_a, v_a), i, v') \tag{5.16}$$

The main difference between the clause generated by [MA15] and the clause generated by [MG16] is that the clause generated by [MG16] contains two instances of $P_1^\#$ and uses that second instance to constrain rnd which is now called v_i. In our data-abstraction framework setting, this difference can be formalized as using a different instantiation set. This time, instead of using $I_1 = \{k'_a\}$, we use $I_1 = \{k_a, i\}$ in order to take into account what happens at index i. The proof that Clause 5.16 is equivalent to using the data-abstraction algorithm on Clause 5.1 with an heuristic returning $I_1 = \{k_a, i\}$ is formalized in Theorem 14. To prove this result, we use the same proof technique we used to prove that the adaptation of the [MA15] transformation could be seen as an instance of our data-abstraction algorithm.

Although unproven in the [MG16] paper, this change allows the transformation to satisfy relative completeness as shown in Theorem 15. Using the tools provided by Section 4.3 and the concepts introduced in Section 5.1.1, this proof does not contain any major difficulties. However, boldly attempting to prove that the clause before transformation is equivalent to the clause after transformation is incorrect as one needs to take into account the abstraction. This may be one of the reasons the authors did not prove this result.

Theorem 14 ([MG16] as an instance of the data-abstraction framework algorithm)**.** *Let* $abs(P_1) = abs(P_2) = \sigma_{Cell} \bullet \sigma_{id} \bullet \sigma_{id}$. *Applying* $eliminate_{abs} \circ abstract_{abs}$ *on Clause 5.1 yields a Clause equivalent to 5.16 whenever:*

$$insts_{P_1}^{abs}((a, i, v), ctx) = \{(((k'_a, a[k'_a]), i, v), ()), (((i, a[i]), i, v), ())\}$$

where ctx *is the context given in Equation 5.13.*

Note that because the adaptation to Horn clauses of [MA15] and [MG16] are the same for Clauses 5.2 and 5.3, this means that the full adaptation of the [MG16] transformation is an instance of the data-abstraction algorithm.

Proof. The abstraction and elimination of Clause 5.1 yields:

$$P_1(a,i,v) \wedge v' = a[i] \rightarrow P_2(a,i,v')$$

After abstraction

$$\big(\forall((k_a,val),i^\#,v^\#), val = a[k_a] \wedge i^\# = i \wedge v^\# = v \rightarrow P_1^\#((k_a,val),i^\#,v^\#)\big) \wedge v' = a[i] \rightarrow \big(\forall((k'_a,val),i^\#,v^\#), val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#)\big)$$

After first step of the instantiation: moving quantifiers in e'.

$$\big(\forall((k_a,val),i^\#,v^\#), val = a[k_a] \wedge i^\# = i \wedge v^\# = v \rightarrow P_1^\#((k_a,val),i^\#,v^\#)\big) \wedge v' = a[i] \rightarrow val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#)$$

After second step of the instantiation

$$(a[k'_a] = a[k'_a] \wedge i = i \wedge v = v \rightarrow P_1^\#((k'_a,a[k'_a]),i,v)) \wedge (a[i] = a[i] \wedge i = i \wedge v = v \rightarrow P_1^\#((i,a[i]),i,v)) \wedge v' = a[i] \rightarrow (val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#))$$

Simplifying trivial equalities and implications

$$P_1^\#((k'_a,a[k'_a]),i,v) \wedge P_1^\#((i,a[i]),i,v) \wedge v' = a[i] \rightarrow (val = a[k'_a] \wedge i^\# = i \wedge v^\# = v' \rightarrow P_2^\#((k'_a,val),i^\#,v^\#))$$

Removing aliases, that is, simplify $\forall x, x = j \rightarrow expr$ by $expr$ where x is replaced by j.

$$P_1^\#((k'_a,a[k'_a]),i,v) \wedge P_1^\#((i,a[i]),i,v) \wedge v' = a[i] \rightarrow P_2^\#((k'_a,a[k'_a]),i,v')$$

Ackermannisation of a

$$((k'_a = i) \rightarrow v_i = v_{k'_a}) \rightarrow (P_1^\#((k'_a,v_{k'_a}),i,v) \wedge (P_1^\#((i,v_i),i,v) \wedge v' = v_i \rightarrow P_2^\#((k'_a,v_{k'_a}),i,v'))$$

Rewritting v_i as $v_{k'_a}$ when $k'_a = i$ and v_i otherwise

$$P_1^\#((k'_a,v_{k'_a}),i,v) \wedge (P_1^\#((i,v_i),i,v) \wedge v' = ite(i = k'_a, v_{k'_a}, v_i) \rightarrow P_2^\#((k'_a,v_{k'_a}),i,v')$$

Renaming k'_a to k_a and $v_{k'_a}$ to v_a

$$P_1^\#((k_a,v_a),i,v) \wedge \boldsymbol{P_1^\#((i,v_i),i,v)} \wedge v' = ite(i = k_a, v_a, v_i) \wedge \Rightarrow P_2^\#((k_a,v_a),i,v')$$

Our desired result.

□

Theorem 15 (Completeness of calls to $insts$ for [MA15; MG16])**.** *The completeness of the calls to* $insts^{abs}_{P_1}((a,i,v),ctx)$ *during the data-abstraction of Clauses 5.1, 5.2 is described in Table 5.2:*

Program instruction	Clause	ctx	Instantiation	I	Complete
`v=a[i];`	Clause 5.1	Equation 5.13	[MA15]	$\{k'_a\}$	NO
`v=a[i];`	Clause 5.1	Equation 5.13	[MG16]	$\{k'_a, i\}$	YES
`a[i] <- v;`	Clause 5.2	Equation 5.14	[MA15] & [MG16]	$\{k'_a\}$	YES
`v=0;`	Clause 5.3	Equation 5.15	[MA15] & [MG16]	$\{k'_a\}$	YES

Table 5.2 – Completeness of calls to $insts$ from the adaptation to Horn clauses of [MA15] and [MG16]. I is such that $insts^{abs}_{P_1}((a,i,v),ctx) = \{(((k,a[k]),i,v),()) \mid k \in I\}$.

Proof. We tackle proving those completeness results using the definition of a complete call to $insts$.

1. Example 23 already showed that the instantiation for the read clause of [MA15] is insufficient. This proof is provided to see how we can prove this using the Equation 4.6 of a definition of a complete call to $insts_P^{abs}(a, ctx)$ where ctx is given in Equation 5.13.
   Consider the model $\mathcal{M}$ such that $\mathcal{M}(P_1) = \{(ConstArray(0), 1, 2)\}$ and $\mathcal{M}(P_2) = \{(ConstArray(0), 1, \mathbf{0})\}$. Let $vars$ such that $[\![k'_a]\!]^{vars} = 2$, $[\![i]\!]^{vars} = 1$, $[\![v]\!]^{vars} = 2$, $[\![v']\!]^{vars} = 2$, $[\![a]\!]^{vars} = ConstArray(0)[1 \leftarrow 2]$, $[\![v'_a]\!]^{vars} = 0$, $[\![i^\#]\!]^{vars} = 1$, $[\![v'^\#]\!]^{vars} = 2$.
   (a) We have $\neg[\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$ because $[\![v'^\#]\!]^{vars} = 2$ and $\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})(P_2^\#) = \{((k, 0), 1, 0)\}$.
   (b) We have $((1, 0), 1, 2) \in \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1))$.
   (c) Thus, if the instantiation was complete, we would have $vars'$ such that
   - $\neg[\![ctx]\!]^{vars'}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$
   - $\sigma_{Cell\bullet id\bullet id}([\![(a, i, v)]\!]^{vars'}) \subseteq \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1))$
   - Thus $[\![(a, i, v)]\!]^{vars'} = (ConstArray(0), 1, 2)$.
   - Not having ctx implies that $[\![v'_a]\!]^{vars'} = [\![a[k'_a]]\!]^{vars'} = 0$, $[\![i^\#]\!]^{vars'} = [\![i]\!]^{vars'} = 1$, $[\![v'^\#]\!]^{vars'} = [\![v']\!]^{vars'} = [\![a[i]]\!]^{vars'} = 0$
   - But it also implies $[\![(k'_a, v'_a), i^\#, v'^\#)]\!]^{vars'} \notin \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_2))$, which means that $(([\![k'_a]\!]^{vars'}, 0), 1, 0) \notin \{((k, 0), 1, 0) \mid k \in \mathbb{Z}\}$. Absurd !
2. For the [MG16] instantiation of the read clause. Assume $\exists vars, [\![\{((k'_a, a[k'_a]), i, v), ((i, a[i]), i, v)\}]\!]^{vars} \subset \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1)) \wedge \neg[\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$.
   (a) $\mathcal{M}(P_1) \neq \emptyset$, otherwise $\alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1)) = \emptyset$ and we do not have $[\![\{((k'_a, a[k'_a]), i, v), ((i, a[i]), i, v)\}]\!]^{vars} \subset \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1))$.
   (b) Thus, let $(a_{\mathcal{M}}, i_{\mathcal{M}}, v_{\mathcal{M}}) \in \mathcal{M}(P_1)$.
   (c) Let $vars'$ such that $[\![(i, v, v', v'_a, k'_a, i^\#, v'^\#)]\!]^{vars'} = [\![(i, v, v', v'_a, k'_a, i^\#, v'^\#)]\!]^{vars}$, and $\forall k, [\![a[k]]\!]^{vars'} = ite(k \in [\![\{i, k'_a\}]\!]^{vars}, [\![a[k]]\!]^{vars}, a_{\mathcal{M}}[k])$.
   (d) We have $\sigma([\![(a, i, v)]\!]^{vars'}) \subset \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1))$: this inclusion is true for the cells with index in $[\![\{i, k'_a\}]\!]^{vars}$ because we have $[\![\{((k'_a, a[k'_a]), i, v), ((i, a[i]), i, v)\}]\!]^{vars} \subset \alpha_{\sigma_{Cell\bullet id\bullet id}}(\mathcal{M}(P_1))$, and this inclusion is true for the other cells because we have $(a_{\mathcal{M}}, i_{\mathcal{M}}, v_{\mathcal{M}}) \in \mathcal{M}(P_1)$.
   (e) And we have not changed the value of the context: we only variable we changed is a, but not on the cells evaluated by the context. Thus, $\neg[\![ctx]\!]^{vars'}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$.
   (f) This concludes our proof.
3. The two other clauses are proven in the same manner.

□

Beyond [MG16]. The [MG16] transformation is limited to programs in a very specific form: a single array read or write per instruction. Although all usual imperative programs without function calls can be reduced to that form, we aim to improve on this result by extending the set of clauses that we handle on more than those three.

In practice, our algorithmic contribution with respect to [MG16] are:

1. proving the completeness of the [MG16] transformation
2. extending the [MG16] transformation to more than those three clauses
3. handle combinators so that we can express different abstractions

5.1.2.2 The array formulae decision procedure of [BMS06]

One of the main contributions of the [BMS06] paper is a decision procedure for a subset of first-order formulae over the theory of arrays that they call the *array property fragment*. This deci-

sion procedure is of particular interest because it is constructed of several transformations, one of which is a quantifier elimination step based on instantiation.

The array property fragment. The set of formulae handled by the decision procedure of [BMS06] is called the *array property fragment* and it is expressive enough to handle universally quantified array indices. In static analysis of programs, the decision procedure for the array property fragment is currently used [MB08] to check that invariants involving arrays are inductive: the decision procedure does not help to infer loop invariants within programs, but it allows automatic checking that those invariants are correct.

For example, in an array initialization loop `for(int i=0;i<n;i++) a[i]=0;` one may wish to prove that the invariant $\forall k, k < i \Rightarrow a[k] = 0$ is correct. In order to do so, one needs to prove $((\forall k, k < i \Rightarrow a[k] = 0) \wedge a[i] = 0) \Rightarrow (\forall k, k < i+1 \rightarrow a[k] = 0)$. This array initialization loop is only a very simple example of the type of program and invariant for which the formula *the invariant is correct* belongs in the *array property fragment*. The *array property fragment* handles much more complex examples, such as proofs of correctness of sorting algorithms, should the invariants be provided. In fact, it handles all the container algorithms of Example 14 for which our data-abstraction framework with cell abstraction is expected to succeed.

A simplified view of the *array property fragment* are formulae of the form depicted in Definition 26. In many ways, the formula of Equation 5.17 is similar to the formula of Equation 4.6 that defines that a call to the instantiation heuristic is complete. The main differences are that in Formula 5.17, there are no predicates and that the quantifiers i, j are either on the left hand side of the implication or only used within $a[i], a[j]$. These differences are better seen in the formula of Equation 5.21. In our construction of the cell abstraction instantiation heuristic, we handle these differences.

Definition 26 (Array property fragment)**.** *A simplified – yet general enough – view of the array property fragment are formulae of the form:*

$$\exists a, vars, (\forall i, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars)) \boxtimes (\forall j, \chi_1(j, vars) \rightarrow \chi_2(a[j], vars)) \boxtimes \omega(a, vars) \quad (5.17)$$

where:

1. $\phi_1(args), \phi_2(args), \chi_1(args), \chi_2(args), \omega(args)$ *are used as functions that evaluate their corresponding expressions with their variables equal to* $args$*. This allows us to explicitly state the free variables of the expressions.*
2. $vars$ *does not contain array variables.*
3. $\boxtimes$ *designates any boolean operation* $\wedge, \vee, \rightarrow, \ldots$
4. *We limited the example to two cases of formula with index quantifiers* $(\forall i, j)$ *and one case of a formula not involving quantifiers* ω*, but in practice one can use any finite number of them. One may even use several quantifiers within a same subformula to express properties such as sortedness.*
5. *There is a single array* a*, but the paper handles cases with several arrays.*
6. *Should there be several arrays, there are no array equalities in* ω*, such as* $a = b$*. However, this can be encoded as* $\forall i, a[i] = b[i]$*.*
7. *As the contribution is a decision procedure, there are limitations on the integer theory so that it is decidable.*

The decision procedure. The decision procedure for the *array property fragment* is divided into several steps. The first mainly transforms syntactical elements of the formula, such as replacing

$a = b$ by $\forall i, a[i] = b[i]$, or by applying the array axiom $a[i \leftarrow v][j] \equiv ite(i = j, v, a[j])$, or even dividing index equalities $i = j$ into $i < j+1 \wedge j < i+1$, and is not of interest to us. The second step instantiates the universal index quantifiers by finite sets. In the decision procedure of the paper, this instantiation set is computed using the syntax and requires the first step. In our presentation of this phase, we aim to present a proof that abstracts these syntactical elements. The third step removes all array variables by using Ackermannisation. This process was already described in Section 5.1.2.1 and is not of key interest for us. Finally, the last step simply decides the resulting integer problem. This step is decidable due to an adequate choice of theory used for integers.

Our interest is thus only in the second step for which we show an abstracted view of how the paper constructs an instantiation set. We only sketch out this second step on the formulae of the form described by Equation 5.17, as the understanding of the instantiation process it yields suffices for our purposes. We refer the reader to the original paper [BMS06] for a full instantiation heuristic and a formalized proof of completeness.

We present our construction in an incremental way: we consider subformulae of the formula of Equation 5.17. First we handle the simple formula of Equation 5.18, then we progress to the two formulae of Equations 5.19 and 5.20, and finally, we unite the two latter instantiation sets into an instantiation for the formula of Equation 5.17.

$$\exists a, vars, \forall i, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars) \tag{5.18}$$

$$\exists a, vars, (\forall i, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars)) \boxtimes \omega(a, vars) \tag{5.19}$$

$$\exists a, vars, (\forall i, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars)) \boxtimes (\forall j, \chi_1(j, vars) \rightarrow \chi_2(a[j], vars)) \tag{5.20}$$

Instantiation of the formula of Equation 5.18. Before we dive into the construction of the instantiation set for i, notice that in the formula of Equation 5.18, $\phi_2(a[i], vars)$ is the only formula involving a and that ϕ_2 does not depend directly on i. Now, let us build on these remarks by assuming $\exists v, vars, \phi_2(v, vars)$. In that case, the formula and any instantiation of it are satisfiable by picking a equal to the constant array equal to v.

Now, let us assume there are no such values $v, vars$. The clause can now be rewritten as $\exists a, vars, \forall i, (\phi_1(i, vars) \rightarrow false)$. This clause is satisfiable if and only if $(\phi_1(i, vars) \equiv false)$ and the instantiated clause with the set I is satisfiable if and only if $((\phi_1(i, vars) \wedge i \in I) \equiv false)$. In other words, if we have an expression e that verifies $\phi_1(i, vars) \Rightarrow \phi_1(e(vars), vars)$, then the instantiation set $\{e\}$ preserves the equivalence.

In all cases, the instantiation set $\{e\}$ where e verifies $\phi_1(i, vars) \Rightarrow \phi_1(e(vars), vars)$ preserves the semantics. The property that e must verify only depends on ϕ_1, and we say that e is a witness for ϕ_1. In the [BMS06] paper, witnesses for expressions such as ϕ_1 are computed syntactically from the syntax restrictions imposed on ϕ_1: for example, if $\phi_1(i, vars) \equiv i < n$, the value $n-1$ is a witness.

Instantiating the formula of Equation 5.19. The reasoning is similar but this time, there are two formulae involving a: $\phi_2(a[i], vars))$ and $\omega(a, vars)$. However, ω does not contain quantifiers and due to the syntax restrictions – mainly no array equalities – ω can only involve a finite number of indices of a. Let us rewrite $\omega(a, vars)$ as $\omega'(a[e_1], \ldots, a[e_n], vars)$.

This time, the array a is constrained by both $\phi_2(a[i], vars)$ and $\omega'(a[e_1], \ldots, a[e_n], vars)$, and the problem is handling the interaction between them. The idea is to separate the interaction between $\phi_2(a[i], vars)$ and $\omega'(a[e_1], \ldots, a[e_n], vars)$ by rewriting the formula as

$$\exists a, vars, (\forall i \notin \{e_1(vars), \ldots e_n(vars)\}, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars)) \wedge \bigwedge_j \phi_1(e_j, vars) \rightarrow \phi_2(a[e_j], vars) \boxtimes \omega(a, vars)$$

We handle $(\forall i \notin \{e_1(vars), \ldots e_n(vars)\}, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars))$ in a similar way as the instantiation of formula 5.18, that is, by picking a witness e for ϕ_1. Then, rewriting the formula having instantiated that part with $\{e\}$ and regrouping the terms yields the following formula, which is exactly the formula of Equation 5.19 instantiated on the set $\{e, e_1, \ldots, e_n\}$!

$$\exists a, vars, (\forall i \in \{e, e_1, \ldots, e_n\}, \phi_1(e_j, vars) \Rightarrow \phi_2(a[e_j], vars)) \boxtimes \omega(a, vars)$$

Instantiating the formula of Equation 5.20. The difficulty here is that both $\phi_2(a[i], vars)$ and $\chi_2(a[j], vars)$ involve a, and possibly the overlap of indices between $\phi_1(i, vars)$ and $\chi_1(j, vars)$ is infinite. The idea is to divide the set of indices into three disjoint parts: $\{k \mid \phi_1(k, vars) \wedge \neg\chi_1(k, vars)\}$, $\{k \mid \chi_1(k, vars) \wedge \neg\phi_1(k, vars)\}$ and $\{k \mid \chi_1(k, vars) \wedge \phi_1(k, vars)\}$.

Because these sets are disjoint, it is as if we had three different arrays. The idea is to reuse the trick for the instantiation of the formula of Equation 5.18 for each of those arrays separately. This gives us the witnesses e for $\phi_1 \wedge \neg\chi_1$, e' for $\chi_1 \wedge \neg\phi_1$ and ee for $\chi_1 \wedge \phi_2$.

Thus we need to instantiate the three subarrays with respectively $\{e\}$, $\{e'\}$ and $\{ee\}$. Instead, we instantiate all three with the instantiation $\{e, e', ee\}$ which preserves semantics as instantiating with a larger set may only *increase completeness*. This amounts to instantiating directly both quantifiers of the formula of Equation 5.20 with $\{e, e', ee\}$. In practice, this set can be refined so that i is only instantiated with $\{e, ee\}$ and j with $\{e', ee\}$.

Instantiating the formula of Equation 5.17. Finally, handling the formula of Equation 5.17 simply consists in putting the ideas of the instantiaton of the formulae of Equations 5.19 and 5.20 together: using the notations of those instantiations, we instantiate the quantifier i on the set $\{e, ee, e_1, \ldots, e_n\}$ and the quantifier j on the set $\{e', ee, e_1, \ldots, e_n\}$.

[BMS06] for cell abstraction. The decision procedure of [BMS06] for the *array property fragment* enables to automatically check invariants, but as it does not handle predicates, it does not allow to infer invariants; a drastic limitation for our purposes. Furthermore, the universal quantifiers are restricted to the left hand side of the implication, which is not the case in our formulae as we have predicates of the form $P(i, a[i])$.

Therefore, our main goal is not to use this decision procedure as such, but to reuse some of its ideas. The abstracted view we used presents these ideas so that we may reuse them to construct our instantiation heuristic, mainly by removing most of the syntactical elements and by using incremental steps instead of a full solution. Note that the construction of the instantiation heuristic in the original paper is directly given and the proof of correctness directly constructs a complicated array a for the full formulae.

5.1.2.3 The instantiation of [BMR13], Quantifiers on demand [GSV18]

Instantiation in [BMR13]. The goal of [BMR13] is to automate the verification programs containing arrays. In many ways, the approach is similar to ours: they convert programs to Horn

clauses similar to those we generate after the $abstract$ algorithm, then they eliminate the quantifiers using instantiation and finally, they solve the resulting problem using the Z3 SMT solver. Thus, the main difference in the overall program verification scheme is in the formalizing of the abstraction.

The main contribution is an instantiation heuristic that is used to determine how the infinite conjunction created by the quantifiers is transformed into a finite one. The difference compared to our instantiation heuristic is that the instantiation heuristic used in [BMR13] is based on triggers and more specifically, E-matching: they define patterns so that if the pattern appears within the clause, the quantifiers are instantiated accordingly with respect to the pattern. The pattern they use are generated from the array axioms, mainly the axiom $a[i \leftarrow v][j] \equiv ite(i = j, v, a[j])$ that generates an instantiation on j.

Limits of the [BMR13] instantiation: lack of predictability. Unlike our approach, trigger-based instantiation is difficult to predict: the instantiation of previous patterns can create new patterns which may then create new instantiations and thus new patterns, ...and this process may not terminate as demonstrated by Example 24 from the [BMR13] paper.

Furthermore, they only evaluate experimentally the choice of patterns for the triggers and the choice between *intra-procedural* and *inter-procedural* context for the triggers, that is, whether the E-matching process should be done on the full set of Horn clauses or just on the current clause. These experimental results are insufficient to determine the impact of those parameters and of how well the overall technique works for two reasons.

First, there are programs on which the back-end SMT solver timeouts. On those cases, because the approach lacks a theoretical result, it is extremely hard to determine whether the problem is with the approach of [BMR13] or if the [BMR13] instantiation yielded a satisfiable integer problem that the back-end solver was unable to solve. Note that the latter problem appears frequently in our benchmarks.

Second, the experimental results are written by hand, and it is hard to determine how resilient the successful experiments are to slight tweaks in the syntax. For example, the authors claim they have not encountered the problem of Example 24, but it is unclear whether this problem could appear with a different ordering of the triggers, or if the clauses were written in a slightly different syntax.

Example 24 (The E-matching algorithm of [BMR13] may not terminate)**.** *Consider the expression* $\forall y, P(g(y), g(f(y)))$ *and a trigger on* $g(y)$*. Consider that in the context we have* $g(a)$*.*

$g(a)$ *matches the trigger* $g(y)$*, which in turn creates the instantiation* $\{a\}$ *for the quantifier* y*, which generates* $P(g(a), g(f(a)))$ *and thus adds among other things* $g(f(a))$ *to the context, which matches the trigger and creates the instantiation* $\{f(a)\}$ *for the quantifier* y*, which adds* $g(f(f(a)))$ *to the context, which ...*

This process never terminates and successively adds the instantiation $f^n(a)$*.*

[GSV18], an approach with similar drawbacks. The work in [GSV18] is quite different but has similar drawbacks: instead of using the SMT solver as a back-end black box solver, they suggest to modify the solving process within the SMT solver in such a way that it can handle the additional quantifiers. The main idea compared to [BMR13] is that instead of computing the instantiation set for a quantifier before the clauses are fed to the back-end solver through trigger based instantiation, the quantifiers are instantiated during the solving of the Horn clauses, and the instantiation set is computed from the refutation proofs generating during solving.

This allows a form of *lazy* computation of the instantiation set but suffers from similar drawbacks as the [BMR13] technique: it is very hard to predict whether the correct instantiation set will be found, and there may be occurrences where the computation gets stuck gradually adding elements in the instantiation set. Furthermore, there are no convincing experimental results for this technique within the paper.

Conclusion of the litterature overview

The overall verification methodology proposed in this PhD and described in Figure 4.1 shares many similarities with [BMR13] and [MA15; MG16] when used with cell abstraction: we define the abstraction of [MA15; MG16], but instead of removing the quantifiers within the abstraction process, we handle quantifier elimination separately from the abstraction, as in [BMR13].

As we have proven the [MG16] instantiation complete, a simple approach would use that instantiation; however unlike in [MG16], our cell abstraction is supposed to be a building block for other abstractions. The restrictions on the form of the clauses in [MG16] does not allow that and we need to extend the [MG16] instantiation heuristic.

We do not wish to use the techniques described in [BMR13] and [GSV18], as the aim of our data-abstraction framework algorithm is to give predictable results. We thus consider instantiation techniques such as those of [BMS06] that can handle a broad class of formulae, but the form of our clauses do not quite match: we have predicates and a key difference in the restrictions of the universal quantifier: it may be used in the right hand side of the implication.

Our instantiation heuristics unifies the heuristics of [MG16] and [BMS06], in order to handle a broader class of clauses that include most of those needed for the programs of Chapter 3.

5.1.3 Our instantiation heuristic for cell abstraction

The goal is to construct an instantiation heuristic for cell abstraction that verifies completeness. This completeness property was verified in the [BMS06] and [MG16] papers, but on a narrower class of formulae than those we need to handle. We contruct the instantiation heuristic for cell abstraction by unifying the ideas of our proof of completeness for these two papers which were themselves imprecisely expressed by the concepts of Section 5.1.1.

We start by formalizing the *completeness of a call to* $insts_P^{abs}(expr, ctx)$ property of Chapter 4 specifically for cell abstraction. This formula is similar to the one of Equation 5.19 and applying the same construction technique for our formula yields two steps.

The first step, that is, the counterpart of Equation 5.18 is handled by the abstraction concept described in Section 5.1.1 and used in the proof of [MG16]; and the second step is handled by introducing the notion of *relevant cells*.

Finally, we write the $relevant$ algorithm to retrieve a set of *relevant cells* from the syntax of $expr$ and ctx, and thus, deduce the algorithm for $insts_P^{abs}(expr, ctx)$. We prove that whenever the $relevant$ algorithm succeeds, $insts_P^{abs}(expr, ctx)$ is complete.

For reasons of simplicity, the presented $relevant$ algorithm is the bare minimum to handle the programs of Chapter 3. However, this algorithm can be extended to handle a broader class of Horn clauses; and as long as it satisfies the property that it returns a relevant set of cells for $(expr, ctx)$, the call $insts_P^{abs}(expr, ctx)$ is complete. In practice we have extended our implementation of $relevant$ to handle, among other things, arrays of arrays.

5.1.3.1 Property the instantiation set must verify: *relevant Cells*.

Formalizing $insts_P^{abs}(expr, ctx)$ complete. In Chapter 3, we gave the definition of the completeness of a call to $insts_P^{abs}(expr, ctx)$. This definition can be simplified for cell abstractions: $F_{\sigma_{Cell}}$ does not contain any quantifiers and, as discussed in Section 5.1.1, instantiation sets for cell abstraction should be defined as $(k, a[k])$ for k in a set of indices I. Thus, let us introduce $I(expr, ctx)$ such that $insts_P^{abs}(expr, ctx) \equiv \{((k, a[k]), ()) \mid k \in I(expr, ctx)\}$ With this notation, the definition of completeness of $insts_P^{abs}(expr, ctx)$ in Equation 4.6 can be rewritten as *Equation 5.22 implies Equation 5.21*.

$$\exists vars, \forall k, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P)) \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{g}\sigma^{abs}}(\mathcal{M})} \tag{5.21}$$

$$\exists vars, \forall k \in [\![I(expr, ctx)]\!]^{vars}, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P)) \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{g}\sigma^{abs}}(\mathcal{M})} \tag{5.22}$$

Notice how Equation 5.21 is similar to Equation 5.19 with $\phi_1 \equiv true$, $\phi_2(vars) \equiv (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$ and $\omega(vars) \equiv \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{g}\sigma^{abs}}(\mathcal{M})}$. We suggest to construct the instantiation $I(expr, ctx)$ such that 5.22 implies Equation 5.21 similarly to how we constructed an instantiation set that preserves the semantics for the quantifier $\forall i$ in Equation 5.19. Thus, we divide Equation 5.21 into two parts: $\forall k, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$ and $\neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{g}\sigma^{abs}}(\mathcal{M})}$.

First step of the instantiation: abstraction. It may be tempting to find a semantics preserving instantiation set for $\exists vars, (\forall k, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$ in a similar fashion as for $\exists a, vars, \forall i, \phi_1(i, vars) \rightarrow \phi_2(a[i], vars)$. However, this will not work: what made the proof of [BMS06] work was that ϕ_2 was not directly dependent on i, and therefore, a constant array could be used for part of the proof. Here, k is part of $(k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$.

Let us find another approach. First let us realize that, for this step, we need to use the abstraction as in the proof of [MG16]: if we did not have abstracted models, that is, if the formula had been $\exists vars, (\forall k, (k, [\![expr]\!]^{vars}[k]) \in \mathcal{M}^{\#}(P^{\#}))$, there would have been no instantiation set I such the semantics is preserved. The proof of this fact has already been discussed in Section 5.1.1 and relies on considering a model $\mathcal{M}^{\#}$ which represents no arrays and yet is non-empty: consider $i \notin I$ and $\mathcal{M}^{\#}(P^{\#}) = \{(k, v) \mid k \neq i\}$. The non-instantiated formula is satisfiable by any choice of $vars$ whereas the instantiated clause is unsatisfiable. Thus the semantics is not preserved. This example explains why it was so key to use *per clause relative completeness* instead of *per clause completeness* and why we should use the knowledge that we have an abstracted model $\alpha_{\sigma_{Cell}}(\mathcal{M}(P))$ and not just any $\mathcal{M}^{\#}$.

Now, let us use the abstraction concept in a similar way as in the proof of [MG16]: if the instantiation set I for k is chosen non-empty, then, if the instantiated formula is non-satisfiable, then $\exists (i, expr[i]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$, thus $\exists a \in \mathcal{M}(P)$ as $\alpha_{\sigma_{Cell}}(\emptyset) = \emptyset$, and thus, $\sigma_{Cell}(a) \subseteq \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$, that is, $\forall k, (k, a[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$. Thus, if we have $\exists vars, [\![expr]\!]^{vars} = a$, then any non-empty instantiation set I for the quantifier k preserves the semantics.

The requirement $\exists vars, [\![expr]\!]^{vars} = a$ is satisfied when $expr$ is in fact a variable a_{var} by just picking $a_{var} = a$. In general, we might not have $\exists vars, [\![expr]\!]^{vars} = a$ and thus there might be a non-empty instantiation heuristic that does not preserve the semantics. For example, if $expr = a_{var}[2 \leftarrow 0]$ where a_{var} is a variable of $vars$, $\{2\} \notin I$ and $\mathcal{M}(P) = \{a \mid a[2] \neq 0\}$, then the instantiated

formula is satisfiable but not the non-instantiated formula: $(2, a_{var}[2])$ is what makes the non-instantiated formula fail but is not considered in the instantiated clause.

For now, we work around this problem by considering that we introduce a new variable a_{var} and that we add $a_{var} = expr$ within the context. The formula of Equation 5.21 that we need to instantiate now becomes the formula of Equation 5.23. For the first part of this formula, a non-empty instantiation set suffices.

$$\exists a, vars, (\forall k, (k, a[k]) \in \alpha_{\sigma_{Cell}}(\mathscr{M}(P))) \wedge ([\![a_{var}]\!]^a = [\![expr]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}) \tag{5.23}$$

Second step of the instantiation: *relevant cells*. Now that we have handled the part $\exists a, (\forall k, (k, a[k]) \in \alpha_{\sigma_{Cell}}(\mathscr{M}(P)))$ of the formula of Equation 5.23, we need to combine it with the part $[\![a_{var}]\!]^a = [\![expr]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$. This part does not contain any quantifiers and can be viewed as $\omega(a, vars)$ in our description of the [BMS06] instantiation with $\omega(a, vars) \equiv a = [\![expr]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$.

In our presentation of the [BMS06] instantiation with $\omega(a, vars)$, we handled $\omega(a, vars)$ by using the assumption that $\omega(a, vars)$ could be rewritten as $\omega'(a[e_1(vars)], \ldots, a[e_n(vars)], vars)$. The basis of this assumption is that ω can only involve a finite number of cells of a, those with index $e_1, \ldots, e_n$, as ω does not contain any quantifiers. The intuition is that $\{e_1, \ldots, e_n\}$ are the index expressions of the *relevant cells*.

However, if ω contains array equalities, this approach fails: array equalities involve an unbounded number of cells. This is troublesome as in our definition of ω, we have $a = [\![expr]\!]^{vars}$. First, we show that [BMS06] copes with the problem of array equalities using syntax and explain why this approach does not work in our setting. Second, we show how we handle the problem in our case using the same syntax requirements. Finally, we adapt the syntactic requirement into the semantic property of *relevant cells*. The use of a semantic property enables future extentions of the theory in which the formulae are written without any changes in the definition of *relevant cells*.

`The [BMS06] handling of array equalities.` First, they rewrite $a = expr$ as $\forall i, a[i] = expr[i]$. However in [BMS06], all array expressions $expr$ are assumed to have been constructed from successive writes into an array variable: for example, if $expr$ is $b[l \leftarrow val][l' \leftarrow val']$, then $expr$ is constructed by two array writes into the array variable b. This explicit mention of b enables them to rewrite $\forall i, a[i] = [\![expr]\!]^{vars}[i]$ into $\forall i, a[i] = expr'(b[i], vars')$, where $expr'(b[i], vars')$ is the evaluation of $expr$ on $vars = \{b, vars'\}$ when $[\![b]\!]^b[i] = b[i]$. This is because the value of an array after successive writes at a given index i only depends on the initial value of that array at index i and on the other variables.

The $\forall i, a[i] = expr'(b[i], vars')$ can be rewritten as $\phi_1(i, vars') \rightarrow \phi_2(a[i], b[i], vars')$ where $\phi_1(i, vars') \equiv true$ and $\phi_2(a[i], b[i], vars') \equiv a[i] = expr'(b[i], vars')$. These types of expressions are handled in [BMS06] by using witnesses[2]; however, in our setting, we do not use witnesses but instead use the abstraction to handle similar expressions. This approach is described in our first step and fails on such expressions.

`A restricted definition of` *`relevant cells`*. Let us first attempt to solve the case where all array expressions $expr$ are assumed to have been constructed from successive writes into an array variable b. Thus, $vars$ can be decomposed into b and $vars'$ such that $[\![expr]\!]^{vars}[i]$

[2] See our description of [BMS06].

can be written as $expr'(b[i], vars')$. There is a major difference between the formula of Equation 5.19 of our explanation of [BMS06] and the formula of Equation 5.23: in Equation 5.19, ϕ_1, ϕ_2 not only depend on a, but also on $vars$, whereas in our setting, $\forall k, (k, a[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$ only depends on the array a. Thus, the quantifier b of $vars$ may be moved and Equation 5.23 can be rewritten as: $\exists a, vars', (\forall k, (k, a[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))) \wedge \omega_2(a, vars')$, where $\omega_2(a, vars') \equiv \exists b, \omega(a, b, vars')$.

The goal is now to compute the *relevant cells* of ω_2 with our initial approach, that is, to consider whether $\omega_2(a, vars')$ can be rewritten into $\omega'_2(a[e_1(vars')], \ldots, a[e_n(vars')], vars')$. This time, we might be able to handle the array equality in ω_2 because, for some expression E, $\exists b, \forall i, a[i] = expr'(b[i], vars') \wedge E(a, b, vars')$ does not constrain the array a in the same manner as $\forall i, a[i] = expr'(b[i], vars')$.

The idea is that, as b can be picked to have the adequate value where it is not constrained by E, the equality between a and b only adds restrictions on cells where b is constrained by E. Thus, if $\omega(a, b, vars')$ can be rewritten as $\omega'(a[e_1(vars')], \ldots, a[e_n(vars')], b[e'_1(vars')], \ldots, b[e'_m(vars')], vars')$ then $\omega_2(a, vars')$ can be rewritten as $\omega'_2(a[e_1(vars')], \ldots, a[e_n(vars')], a[e'_1(vars')], \ldots, a[e'_m(vars')], vars')$, and the index expressions of the *relevant cells* are simply $\{e_1, \ldots, e_n, e'_1, \ldots e'_m\}$.

`The definition of relevant cells.` In our previous approach, we assumed that all array expressions $expr$ are constructed from successive writes into an array variable. This means that if we extend the theory on which we work with constant arrays or functions to handle expressions of the form $f(x)$ or $ConstArray(2)$, where $f(x)$ is an array, the definition of *relevant cells* will need to be updated. We wish to construct a definition of *relevant cells* that does not need to be updated, even though the algorithm that returns such a set will have to handle the additional expressions.

The key idea in our restricted approach was to move the quantified variable b. Here, we do not have the array b, therefore, we move all the quantified variables $vars$, and our goal is to define what a set of relevant cells for $\omega_2(a) = \exists vars, \omega(a, vars)$ is. Replicating our previous approach consists in considering whether $\omega_2(a)$ can be rewritten into $\omega'_2(a[e_1()], \ldots, a[e_n()], vars')$. The problem of this approach is that $e_1, \ldots, e_n$ cannot depend on $vars$ anymore, thus for the simple expression $a[i] = 2$, one can not say that $\{i\}$ is a relevant set of index expressions...

This problem is solved by a combination of two ideas. First, instead of rewriting ω with an expression that does not directly depend on a but only on specific cells of a, we say that $\omega_2(a)$ should be equivalent to $\omega_2(a')$ whenever the arrays a and a' match on the relevant cells. Second, the expressions on which the relevant cells should match are $[\![e_1]\!]^{vars}, \ldots, [\![e_n]\!]^{vars}$ where $e_1, \ldots, e_n$ are the indices of the relevant cells. This is because our goal is to prove that the instantiated version of Equation 5.23 implies its non-instantiated version. Thus, instead of equivalence – which is guaranteed through soundness – the instantiation only needs to prove an implication and thus create a $a', vars'$ from $a, vars$. The preservation of semantics must thus happen on $[\![e_1]\!]^{vars}, \ldots, [\![e_n]\!]^{vars}$. Thus, $R = \{e_1, \ldots, e_n\}$ is a set of index expressions of the relevant cells of ω if and only if:

$$\forall a, a', vars, (a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}] \wedge \omega(a, vars)) \Rightarrow (\exists vars', \omega(a', vars'))$$

where $a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}]$ is a shorthand for $\forall i, a[[\![e_i]\!]^{vars}] = a'[[\![e_i]\!]^{vars}]$.

In practice ω depends on the model $\mathcal{M}$ as ω contains the evaluation of the context. Thus, this property should be required for all models. This leads to the definition of *relevant cells* of Definition 27. Notice that this definition is completely independent of the syntax in which the expressions are written.

We formalize the results of the two steps of our reasoning in Therorem 16 which states that the call $insts_P^{abs}(expr, ctx)$ is complete whenever the instantiation set returns a non-empty set of relevant cells for $\omega(a, vars) \equiv a = [\![expr]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$.

In order to claim that one should instantiate on a non-empty set of *relevant cells*, one should prove that this condition is not only sufficient for a complete instantiation but also necessary. In Theorem 16 we also prove that whenever the context ctx does not use the predicate P – or that the instantiation heuristic does not take advantage of the P parameter – a non-empty set of relevant cells are necessary for a complete instantiation. This only happens if P appears twice in a clause[3]. This restriction on the necessary property allows us to avoid degenerate cases, such as the one given in Example 25.

We do not believe this restriction to be important enough to question whether *a non-empty set of relevant cells* is the correct property: one still needs to handle the class of sets of Horn clauses where predicates do not appear twice in a clause, which is as expressive as those that allow it as one can always transform P into a new predicate P' by adding the clause $P(args) \rightarrow P'(args)$. However, perhaps future work can make a slight improvement by formalizing Definition 27 and Therorem 16 in such a way that this restriction can be removed and degenerate cases such as Example 25 are naturally avoided.

Definition 27 (Relevant cells of a function $\omega(a, vars)$)**.** *A set of expressions $R = \{e_1, \ldots, e_n\}$ is said to be relevant for ω if and only if*

$$\forall a, a', vars, (a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}] \wedge \omega(a, vars)) \Rightarrow (\exists vars', \omega(a', vars')) \tag{5.24}$$

where $a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}]$ means that $\forall e \in R, a[[\![e]\!]^{vars}] = a'[[\![e]\!]^{vars}]$

Theorem 16 (A non-empty relevant set yields a complete instantiation.)**.** *Let $R = \{e \mid ((e, expr[e]), ()) \in insts_P^{abs}(expr, ctx)\} = \{e_1, \ldots, e_n\}$. If for all models $\mathscr{M}$, R is a non-empty set of expressions relevant for $\omega_{\mathscr{M}}$ defined in Equation 5.25, then $insts_P^{abs}(expr, ctx)$ is complete.*

$$\omega_{\mathscr{M}}(a, vars) \equiv a = [\![expr]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})} \tag{5.25}$$

Furthermore, if $P \notin ctx$, this condition is also necessary: if $insts_P^{abs}(expr, ctx)$ is complete and $P \notin ctx$, then for all $\mathscr{M}$, R is a non-empty set of expressions relevant for $\omega_{\mathscr{M}}$

Proof of the sufficient condition. Assume R relevant.

1. Introduce $expr, ctx, \mathscr{M}, abs$ such that $abs(P) = \sigma_{Cell}$. Let $R = \{e_1, \ldots, e_n\} = \{e \mid ((e, expr[e]), ()) \in insts_P^{abs}(expr, ctx)\}$.
2. We need to prove that Equation 5.22 implies Equation 5.21 with $I(expr, ctx) = R$.
3. Assume Equation 5.22 and introduce *vars*. We have $\omega_{\mathscr{M}}([\![expr]\!]^{vars}, vars)$.
4. As R is non-empty, $I(expr, ctx)$ is non-empty.
5. As $I(expr, ctx) \neq \emptyset$ and $\forall k \in [\![R]\!]^{vars}, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathscr{M}(P))$, we have $\alpha_{\sigma_{Cell}}(\mathscr{M}(P)) \neq \emptyset$ and thus $\mathscr{M}(P) \neq \emptyset$. Let $a_{\mathscr{M}} \in \mathscr{M}(P)$.
6. Let a' such that $\forall i, a'[i] = ite(i \in [\![R]\!]^{vars}, [\![expr]\!]^{vars}[i], a_{\mathscr{M}}[i])$
7. Thus, using that R is relevant with $a = [\![expr]\!]^{vars}, a' = a', vars = vars$, we have $\exists vars', \omega_{\mathscr{M}}(a', vars')$. Introduce $vars'$.
8. $\omega_{\mathscr{M}}(a', vars')$ means that we have $\neg [\![ctx]\!]^{vars'}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$ and $\forall k, [\![expr]\!]^{vars'}[k] = a'[k]$.
9. We have $\forall k, (k, [\![expr]\!]^{vars'}[k]) \in \alpha_{\sigma_{Cell}}(\mathscr{M}(P))$.

[3] And when combinators are not used as they create new predicate names.

   (a) if $k \notin [\![R]\!]^{vars}$, then $[\![expr]\!]^{vars'}[k] = a'[k] = a_{\mathcal{M}}[k]$, and $\forall k, (k, a_{\mathcal{M}}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$.
   (b) if $k \in [\![R]\!]^{vars}$, we have $[\![expr]\!]^{vars'}[k] = a'[k] = [\![expr]\!]^{vars}[k]$, and $k \in [\![I(expr, ctx)]\!]^{vars}$ and $\forall k \in [\![I(expr, ctx)]\!]^{vars}, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P))$.
10. Thus, we have $\forall k, (k, [\![expr]\!]^{vars'}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}(P)) \wedge \neg [\![ctx]\!]^{vars'}_{\alpha_{\gamma_{\sigma^{abs}}}(\mathcal{M})}$ and our proof is finished.

□

Proof of the necessary condition. Assume $insts^{abs}_P(expr, ctx)$ complete and $P \notin ctx$.

1. Introduce $\mathcal{M}, a, a', vars$. Assume $a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}]$ and $\omega_{\mathcal{M}}(a, vars)$.
2. Let $\mathcal{M}'$ such that $\forall P', \mathcal{M}'(P') = ite(P' = P, \{a'\}, \mathcal{M}(P'))$.
3. As P is not used in ctx, $\forall vars', \omega_{\mathcal{M}}(a, vars') = \omega_{\mathcal{M}'}(a, vars')$
4. Let us prove we have $\forall k \in [\![I(expr, ctx)]\!]^{vars}, (k, [\![expr]\!]^{vars}[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}'(P))$, that is, the first part of Equation 5.22 for $\mathcal{M}'$ with $vars$.
   (a) $[\![expr]\!]^{vars} = a$ as $\omega_{\mathcal{M}}(a, vars)$
   (b) Let $k \in [\![I(expr, ctx)]\!]^{vars} = [\![R]\!]^{vars}$
   (c) $[\![expr]\!]^{vars}[k] = a[k] = a'[k]$ using the assumption $a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}]$
   (d) As $a' \in \mathcal{M}'(P)$, we have $\forall k, (k, a'[k]) \in \alpha_{\sigma_{Cell}}(\mathcal{M}'(P))$
5. As we have $\omega_{\mathcal{M}}(a, vars)$, and thus $\omega_{\mathcal{M}'}(a, vars)$, we have $\neg [\![ctx]\!]^{vars}_{\mathcal{M}'}$, that is, the second part of Equation 5.22 for $\mathcal{M}'$ with $vars$.
6. We thus have Equation 5.22 for $\mathcal{M}'$ with $vars$.
7. Using completeness of $insts^{abs}_P(expr, ctx)$, we have Equation 5.21 for $\mathcal{M}'$ with some $vars'$.
8. The first part of Equation 5.21 is thus satisfiable for $\mathcal{M}'$ with $vars'$.
9. But using the definition of $\mathcal{M}'(P)$, this means that $[\![expr]\!]^{vars'} = a'$.
10. The second part of Equation 5.21 for $\mathcal{M}'$ with $vars'$ gives us $\neg [\![ctx]\!]^{vars'}_{\mathcal{M}'}$
11. Thus, we have $\omega_{\mathcal{M}'}(a', vars')$
12. And thus, $\omega_{\mathcal{M}}(a', vars')$, our goal.

□

Example 25 (A degenerate case where $P \in ctx$.)**.** *Let $abs(P) = \sigma_{Cell}$. Consider the following instantiation $insts^{abs}_P(ConstArray(0), ((\forall i, v, (v = ConstArray(0)[i]) \rightarrow P^{\#}(i, v)) \rightarrow false))$, which comes from the data-abstraction algorithm on $P(ConstArray(0)) \wedge P(ConstArray(0)) \rightarrow false$. Remark this clause is degenerate: we repeated $P(ConstArray(0))$ for no reason.*

Let us name $expr \equiv ConstArray(0)$ and $ctx \equiv (\forall i, v, (v = ConstArray(0)[i]) \rightarrow P^{\#}(i, v)) \rightarrow false$ and $R = \{e \mid ((e, expr[e]), ()) \in insts^{abs}_P(expr, ctx)\}$.

Let us construct R such that $insts^{abs}_P(expr, ctx)$ is complete but there exists $\mathcal{M}$ such that R is not a relevant set for $\omega_{\mathcal{M}}$, thus contradicting the necessary property for Theorem 16 when $P \in ctx$.

1. *For any non-empty R, the call $insts^{abs}_P(expr, ctx)$ is complete, thus, this condition does not restrict our choice of R. Proof:*
   (a) if Equation 5.22 is satisfied for some $\mathcal{M}$ with $vars$, then the second part is with $\mathcal{M}$ with $vars$, and thus $\mathcal{M}(P) = \emptyset \vee \mathcal{M}(P) = \{ConstArray(0)\}$. But if the first part of Equation 5.22 is satisfied, then $\mathcal{M}(P) = \{ConstArray(0)\}$, as $R \neq \emptyset$.
   (b) But then Equation 5.21 is satisfiable for $\mathcal{M}$ with $vars$ as $expr = ConstArray(0)$ and $\mathcal{M}(P) = \{ConstArray(0)\}$.
2. *For any R, R is not a relevant set for $\omega_{\mathcal{M}}$: we need $\forall a', \omega_{\mathcal{M}}(a', vars')$ for some $vars'$. Yet, $\omega_{\mathcal{M}}(a', vars')$ implies $a' = [\![expr]\!]^{vars'}$. But $expr = ConstArray(0)$, and thus, we would need $\forall a', a' = ConstArray(0)$, which is evidently wrong.*
3. *Thus, any choice of a non-empty R achieves our goal, for example $R = \{0\}$.*

The key part in creating this example is to construct ctx using P such that the completeness of the call assumption does not help. Thus, to improve Definition 27 and Theorem 16, one would have to take in account the loss of information on R that the completeness of the call assumption yields when $P \in ctx$.

5.1.3.2 The $relevant$ and $insts_P^{abs}(expr, ctx)$ algorithms

In the previous section we showed that if we had a non-empty set of relevant cells R, then defining $insts_P^{abs}(expr, ctx) = \{((e, expr[e]), ()) \mid e \in R\}$ yields a complete instantiation. The goal is thus reduced to computing a non-empty set of relevant cells R.

To do so, we compute a possible empty set of relevant cells R and make it non-empty: if $R = \emptyset$, we add an element $\bot$ to R, where $\bot$ is any value of the index type. Because $R = \emptyset$ is a set of relevant cells, so is $R \cup \{\bot\} = \{\bot\}$. We thus use the set $\{\bot\}$ and this yields the instantiation $\{((\bot, expr[\bot]), ())\}$.

To compute a set of relevant cells R, we need to define R such that for any model $\mathscr{M}$ $\omega_{\mathscr{M}}(a, vars)$ of Equation 5.25 verifies Equation 5.24. $\omega_{\mathscr{M}}(a, vars)$ of Equation 5.25 can be defined as $[\![a_{var} = expr \wedge \neg ctx]\!]^{a,vars}_{\alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})}$, where a_{var} is a new variable, and the context $a, vars$ interprets the variable a_{var} with a and the other variables with $vars$.

In practice, the $relevant$ algorithm may not always succeed in returning a relevant set of cells. In that case, $relevant$ still returns a set of indices on which to instantiate which it believes is good, but adds an element $\top$ indicating that the returned set may not satisfy the *relevant* requirement. This enables us to track such cases.

Given such a $relevant$ algorithm, the computation of $insts_P^{abs}(expr, ctx)$ simply consists in constructing a_{var} and $\omega \equiv a_{var} = expr \wedge \neg ctx$, calling $relevant$ on those parameters and removing $\top$ from it and returning $\{((\bot, expr[\bot]), ())\}$ if it is empty and $\{((e, expr[e]), ()) \mid e \in relevant\}$ otherwise. The code written in Ocaml style for the algorithm $insts_P^{abs}(expr, ctx)$ is given in Listing 5.5. Note that it does not use the parameters P and abs.

We need to define $relevant(a_{var}, \omega)$ such that Equation 5.24 is verified with $R = relevant(a_{var}, \omega)$ and $\omega(a, vars) = [\![\omega]\!]^{a,vars}_{\alpha_{\mathscr{G}^{\sigma^{abs}}}(\mathscr{M})}$ or contains an element $\top$. As discussed during our construction of the *relevant* property, this amounts to finding the indices of the cells of a_{var} that are of importance in ω. In other words, changing cells that are not in R should not impact the value of ω.

We adapt the concept of *important* cells on which ω depends into the algorithm $relevant(a_{var}, \omega)$ using the following main ideas :

1. If $a_{var}[i]$ appears in ω then i belongs to the indices of the relevant cells: if the value of cell i of a_{var} is changed, then the expression $a_{var}[i]$ changes.
2. If $a_{var}[i_1 \leftarrow v_1] \ldots [i_n \leftarrow v_n][j]$ appears in ω, then j belongs to the indices of the relevant cells. This is because $v = a_{var}[i_1 \leftarrow v_1] \ldots [i_n \leftarrow v_n][j] \equiv v = ite(j = i_n, v_n, \ldots ite(j = i_1, v_1, a_{var}[j]))$, thus, it only depends on index j.
3. If $a_{var} = b_{var}$ then we need to find the relevant indices for b_{var}. Thus, $relevant(b_{var}, \omega)$ belongs to $relevant(a_{var}, \omega)$.
4. If $a_{var}[i \leftarrow v] = b_{var}$ then $relevant(b_{var}, \omega) - \{i\}$ belongs to $relevant(a_{var}, \omega)$. This is because $a_{var}[i \leftarrow v] = b_{var} \Rightarrow \forall j \neq \boldsymbol{i}, a_{var}[j] = b_{var}[j]$.
5. If a_{var} is used in one of the ways we do not know how to handle, $\top$ belongs to $relevant(a_{var}, \omega)$, indicating that we do not know how to handle that case.

In practice, we combine these ideas and the main leftover problem is the possibly infinite recursion to $relevant$: $relevant(a_{var}, \omega)$ may depend on $relevant(b_{var}, \omega)$ and vice versa. We break this infinite recursion by keeping track of variables that have already been visited. The full

algorithm is given in Listing 5.6. The correctness of $insts_P^{abs}(expr, ctx)$ and $relevant$ is stated in Theorem 17.

Theorem 17 (Algorithms $insts_P^{abs}(expr, ctx)$, $relevant(a_{var}, \omega)$ are correct.)**.** *If* `relevant` a_{var} $(a_{var} = expr \wedge \neg ctx)$ *does not contain* $\top$ *then the call to* $insts_P^{abs}(expr, ctx)$ *as defined in Algorithm 3 is complete.*

Proof. Introduce $expr, ctx$. Introduce a_{var} the fresh variable created by the algorithm of Algorithm 3. Define ω as the expression $a_{var} = expr \wedge \neg ctx$. Let $R =$ `relevant` a_{var} ω and assume R does not contain $\top$. We use Theorem 16 to show that $insts_P^{abs}(expr, ctx)$ as defined in Algorithm 3 is complete. Thus, introduce $\mathcal{M}$, we need to show that R is relevant for $\omega_{\mathcal{M}}$ defined as:

$$\omega_{\mathcal{M}}(a, vars) \equiv a = [\![expr]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

Using the evaluation context $a_{var} \leftarrow a, vars$ for the free variables of ω which evaluates the variable a_{var} with a and the other variables with $vars$, this can be rewritten into

$$\omega_{\mathcal{M}}(a, vars) \equiv [\![\omega]\!]^{a_{var} \leftarrow a, vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

Let us prove that R is relevant for $\omega_{\mathcal{M}}$. Introduce $a, a', vars$, assume $a[[\![R]\!]^{vars}] = a'[[\![R]\!]^{vars}]$ and $[\![\omega]\!]^{a_{var} \leftarrow a, vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$ and let us show there exists $vars'$ such that $[\![\omega]\!]^{a_{var} \leftarrow a', vars'}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$. Because we will be using the two evaluations context $[\![e]\!]^{a_{var} \leftarrow a, vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$ and $[\![e]\!]^{a_{var} \leftarrow a', vars'}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$. Let us simply write in this proof these contexts respectively $[\![e]\!]^{c_a}$ and $[\![e]\!]^{c'_a}$.

Our proof is constructive and we construct $vars'$ is the following way:

1. If v is a non-array variable, then $[\![v]\!]^{vars'} = [\![v]\!]^{vars}$
2. If v is a array variable not linked to a_{var}, that is, such that, during the execution of `relevant` a_{var} ω, `relevant_impl` ω `visited` v ω is not called, for some list $visited$, then $[\![v]\!]^{vars'} = [\![v]\!]^{vars}$
3. If v is a array variable linked to a_{var}, that is, such that, during the execution of `relevant` a_{var} ω, `relevant_impl` ω `visited` v ω is called, for some list $visited$, then: let $I_v = [\![$ `relevant` v ω $]\!]^{c_a}$. Note that we use `relevant v` ω even though it was possibly not called during the execution of `relevant a_var` ω.
   We define $vars'$ such that $\forall i, [\![v[i]]\!]^{c'_a} =$
   (a) $[\![v[i]]\!]^{c_a}$ if $i \in I_v$: we do not change the values of indices in the relevant set.
   (b) $a'[i]$ if $i \notin I_v$ and $[\![v[i]]\!]^{c_a} = a[i]$: we change the values previously equal to $a[i]$ to $a'[i]$.
   (c) $a[i]$ if $i \notin I_v$ and $[\![v[i]]\!]^{c_a} = a'[i]$: we change the values previously equal to $a'[i]$ to $a[i]$.
   (d) $[\![v[i]]\!]^{c_a}$ otherwise.

Let us show we have $[\![\omega]\!]^{c'_a}$. To show this, we show that for all subexpressions ω_I of ω, $[\![\omega_I]\!]^{c_a} = [\![\omega_I]\!]^{c'_a}$. Note that this is not necessarily boolean equality. We show this by induction. Note that we will regularly use that $relevant$ is increasing, that is: $\forall v, e_1, e_2$, e_1 is a subexpression of $e_2 \Rightarrow$ `relevant` v e_1 $\subseteq$ `relevant` v e_2. This property is a consequence that the algorithm only adds elements to the retuned set. Let us now continue with our induction:

1. Case ω_I is $a_1[i_1 \leftarrow v_1]\ldots[i_n \leftarrow v_n][j]$: By induction hypothesis $[\![(j, i_1, v_1, \ldots i_n, v_n)]\!]^{c_a} = [\![(j, i_1, v_1, \ldots i_n, v_n)]\!]^{c'_a}$. But $j \in$ `relevant` a_1 ω_I. Thus, as $relevant$ is increasing, $j \in$ `relevant` a_1 ω. And by construction of $vars'$, $[\![a_1[j]]\!]^{c_a} = [\![a_1[j]]\!]^{c'_a}$.

   Finally, as $a_1[i_1 \leftarrow v_1]\ldots[i_n \leftarrow v_n][j]$ is just an if-then-else combination of $(a[j], j, i_1, v_1, \ldots i_n, v_n)$, and these are equal in both evaluation contexts, we have $[\![\omega_I]\!]^{c_a} = [\![\omega_I]\!]^{c'_a}$.

Algorithm 3 $(insts_P^{abs}(expr, ctx), relevant(a_{var}, \omega))$**.**

Listing 5.5 – Instantiation heuristic for cell abstraction.

```
let insts_P^abs expr ctx =
  let a_var = new_variable () in (* We need a_var to be unused in expr and ctx*)
  let R = relevant a_var (a_var = expr ∧ ¬ctx) in
  let R' = filter (fun e -> e ≠ ⊤) R in (*We remove ⊤*)
  if R' = [] then [((⊥, expr[⊥]), ())] (*We need R' non-empty*)
  else map (fun e -> ((e, expr[e]), ())) R'
```

Listing 5.6 – Computation of a relevant set of cells for basic array theory.

```
let relevant a_var ω =
  (*We add two parameters that we set in first position
    to keep track of full context and visited variables*)
  relevant_impl ω [] a_var ω

let rec relevant_impl ω visited a_var expr=
  if a_var ∈ visited then [] (*We ignore visited variables*)
  else
  (
    match expr with
    (*the read after possibly multiple writes. If n=0, this is simply a_var[j]*)
    | a_var[i_1 ← v_1]...[i_n ← v_n][j] ->
        let relevant_from_args = concat_map (relevant_impl ω visited a_var)
                                            [i_1, v_1, ..., i_n, v_n, j] in
        j::relevant_from_args
    (*The array equality case twice on the variable a_var.*)
    |a_var[i_1 ← v_1]...[i_n ← v_n] = a_var[i'_1 ← v'_1]...[i'_m ← v'_m] ->
         let relevant_from_args = concat_map (relevant_impl ω visited a_var)
                                            [i_1, v_1, ..., i_n, v_n, i'_1, v'_1, ..., i'_m, v'_m] in
         [i_1, ..., i_n, i'_1, ..., i'_m] @ relevant_from_args
    (*The array equality case. b_var must be a variable, not any expression*)
    (*Note that we consider the match with = to be modulo commutativity*)
    | a_var[i_1 ← v_1]...[i_n ← v_n] = b_var[i'_1 ← v'_1]...[i'_m ← v'_m] ->
         let relevant_b = relevant_impl ω a_var::visited b_var ω in
         let rm_useless_rel_b = filter (fun e -> e ∉ {i_1,...i_n}) relevant_b in
         let relevant_from_args = concat_map (relevant_impl ω visited a_var)
                                            [i_1, v_1, ..., i_n, v_n, i'_1, v'_1, ..., i'_m, v'_m] in
         [i'_1, ..., i'_m] @ rm_useless_rel_b @ relevant_from_args
    (*Quantifiers*)
    | ∀q, e | ∃q, e ->
         let relevant_e = relevant_impl ω visited a_var e in
         (*We replace the indices that use q by ⊤ *)
         map (fun e -> if q ∈ e then ⊤ else e) relevant_e
    (*A non-array theory constructor or an unknown constructor or predicate*)
    | C(e_1,...,e_n) -> (*We just apply recursively*)
        concat_map (relevant_impl ω visited a_var) [e_1,...,e_n]
    (*The case where a_var was not within a handled expression*)
    | a_var -> [⊤]
    (*A variable different from a_var*)
    | _ -> []
  )
```

2. Case ω_I is $a_1[i_1 \leftarrow v_1]\dots[i_n \leftarrow v_n] = a_1[i'_1 \leftarrow v'_1]\dots[i'_m \leftarrow v'_m]$: By induction hypothesis $[\![(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots, i'_m, v'_m)]\!]^{c_a} = [\![(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots, i'_m, v'_m)]\!]^{c'_a}$. But $a_1[i_1 \leftarrow v_1]\dots[i_n \leftarrow v_n] = a_1[i'_1 \leftarrow v'_1]\dots[i'_m \leftarrow v'_m]$ is just an if-then-else combination of $(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots, i'_m, v'_m)$, thus we have $[\![\omega]\!]^{c_a} = [\![\omega]\!]^{c'_a}$.

3. Case ω_I is $a_1[i_1 \leftarrow v_1]\dots[i_n \leftarrow v_n] = a_2[i'_1 \leftarrow v'_1]\dots[i'_m \leftarrow v'_m]$ and a_1 is linked to a_{var} (which implies that a_2 is as well).

   Let $E_1 \equiv a_1[i_1 \leftarrow v_1]\dots[i_n \leftarrow v_n]$ and $E_2 \equiv a_2[i'_1 \leftarrow v'_1]\dots[i'_m \leftarrow v'_m]$

   We need to prove that $[\![E_1 = E_2]\!]^{c_a} \equiv [\![E_1 = E_2]\!]^{c'_a}$. We prove the stronger property $\forall k, [\![E_1[k] = E_2[k]]\!]^{c_a} \equiv [\![E_1[k] = E_2[k]]\!]^{c'_a}$. This is what we believe will allow the proof of the future extension of Listing 6.1.

   Let us prove the equality by dividing the indices into four portions:

   (a) The case $k \in [\![\{i_1, \dots, i_n\}]\!]^{c_a}$. Let l such that $k = [\![i_l]\!]^{c_a}$. $E_1[i_l] = E_2[i_l]$ can be rewritten into a combination of if-then-else using $a_2[i_l], i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots i'_m, v'_m$. But by induction hypothesis $[\![(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots i'_m, v'_m)]\!]^{c_a} = [\![(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots i'_m, v'_m)]\!]^{c'_a}$. Thus, if $[\![a_2[i_l]]\!]^{c_a} = [\![a_2[i_l]]\!]^{c'_a}$, we have our result.
   $i_l \in$ `relevant` $a_2\ \omega_I$ and thus, $i_l \in$ `relevant` $a_2\ \omega$. But by construction, $[\![a_2[i_l]]\!]^{c'_a} = [\![a_2[i_l]]\!]^{c_a}$ as $i_l \in$ `relevant` $a_2\ \omega$. Which gives our result.

   (b) The case $k \in [\![\{i'_1, \dots, i'_m\}]\!]^{c_a}$ is handled symmetrically.

   (c) The case $k \in [\![$ `relevant` $a_1\ \omega \cap$ `relevant` $a_2\ \omega\ ]\!]^{c_a}$: We have $[\![a_1[k]]\!]^{c_a} = [\![a_1[k]]\!]^{c'_a}$ and $[\![a_2[k]]\!]^{c_a} = [\![a_2[k]]\!]^{c'_a}$ by construction. Then by induction hypothesis, $[\![(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots i'_m, v'_m)]\!]^{c_a} = [\![(i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots i'_m, v'_m)]\!]^{c'_a}$. Thus, as $E_1[k] = E_2[k]$ can be rewritten as an if-then-else combination of $a_1[k], a_2[k], i_1, v_1, \dots, i_n, v_n, i'_1, v'_1, \dots i'_m, v'_m$ and all these elements are equal in both contexts, we have $[\![E_1[k] = E_2[k]]\!]^{c_a} \equiv [\![E_1[k] = E_2[k]]\!]^{c'_a}$.

   (d) The case $k \notin [\![$ `relevant` $a_1\ \omega \cup$ `relevant` $a_2\ \omega\ ]\!]^{c_a}$. In that case $E_1[k] = E_2[k] \equiv a_1[k] = a_2[k]$. We prove that $[\![a_1[k] = a_2[k]]\!]^{c_a} \equiv [\![a_1[k] = a_2[k]]\!]^{c'_a}$ by dividing into subcases. Each subcase is the consequence of our construction of *vars'* for non relevant indices: a swap between $a[k]$ and $a'[k]$. We use values x_1 and x_2 as general names for the values of $[\![a_1[k]]\!]^{c_a}$ and $[\![a_2[k]]\!]^{c_a}$.

$[\![a_1[k]]\!]^{c_a}$	$[\![a_2[k]]\!]^{c_a}$	$[\![a_1[k]]\!]^{c'_a}$	$[\![a_2[k]]\!]^{c'_a}$	$[\![a_1[k] = a_2[k]]\!]^{c_a}$	$[\![a_1[k] = a_2[k]]\!]^{c'_a}$
$a[k]$	$a[k]$	$a'[k]$	$a'[k]$	true	true
$a[k]$	$a'[k]$	$a'[k]$	$a[k]$	$a'[k] = a[k]$	$a'[k] = a[k]$
$a[k]$	$x_2 \notin \{a[k], a'[k]\}$	$a'[k]$	x_2	false	false
$a'[k]$	$a[k]$	$a[k]$	$a'[k]$	$a'[k] = a[k]$	$a'[k] = a[k]$
$a'[k]$	$a'[k]$	$a[k]$	$a[k]$	true	true
$a'[k]$	$x_2 \notin \{a[k], a'[k]\}$	$a[k]$	x_2	false	false
$x_1 \notin \{a[k], a'[k]\}$	$a[k]$	x_1	$a'[k]$	false	false
$x_1 \notin \{a[k], a'[k]\}$	$a'[k]$	x_1	$a[k]$	false	false
$x_1 \notin \{a[k], a'[k]\}$	$x_2 \notin \{a[k], a'[k]\}$	x_1	x_2	$x_1 = x_2$	$x_1 = x_2$

   To show that these four portions cover all indices, we need to prove that (`relevant` $a_1\ \omega - \{i'_1, \dots i'_m\}) \subseteq$ `relevant` $a_2\ \omega$, and, symmetrically, (`relevant` $a_2\ \omega - \{i_1, \dots i_n\}) \subseteq$ `relevant` $a_1\ \omega$.

This is because

$\texttt{relevant}\ a_2\ \omega$

$= (\texttt{relevant}\ a_2\ \omega) \cup (\texttt{relevant}\ a_2\ \omega)$

$\supseteq (\texttt{relevant_impl}\ \omega\ []\ a_2\ \omega) \cup (\texttt{relevant_impl}\ \omega\ []\ a_2\ \omega)$

As `relevant_impl` is increasing in its last argument

$\supseteq (\texttt{relevant_impl}\ \omega\ []\ a_2\ \omega) \cup (\texttt{relevant_impl}\ \omega\ []\ a_2\ \boldsymbol{\omega_I})$

As `relevant_impl` is decreasing in the $visited$ argument

$\supseteq (\texttt{relevant_impl}\ \omega\ \boldsymbol{[a_1]}\ a_2\ \omega) \cup (\texttt{relevant_impl}\ \omega\ []\ a_2\ \omega_I)$

By unrolling `relevant_impl` on ω_I

$\supseteq (\texttt{relevant_impl}\ \omega\ [a_1]\ a_2\ \omega) \cup ((\texttt{relevant_impl}\ \omega\ [a_2]\ a_1\ \omega) - \{i'_1, \dots i'_m\})$

$\supseteq ((\texttt{relevant_impl}\ \omega\ [a_1]\ a_2\ \omega) \cup (\texttt{relevant_impl}\ \omega\ [a_2]\ a_1\ \omega)) - \{i'_1, \dots i'_m\}$

As the difference between $\texttt{relevant_impl}\ \omega\ []\ a_1\ \omega$ and $\texttt{relevant_impl}\ \omega\ [a_2]\ a_1\ \omega$ is the call to $\texttt{relevant_impl}\ \omega\ [a_1]\ a_2\ \omega$.

$\subseteq (\texttt{relevant_impl}\ \omega\ []\ a_1\ \omega) - \{i'_1, \dots i'_m\}$

$= (\texttt{relevant}\ a_1\ \omega) - \{i'_1, \dots i'_m\}$

The same reasoning applies for the symmetric case.

4. The other cases are straightforward by induction and mainly require to use that R does not contain $\top$.

□

Conclusion of the instantiation for cell abstraction. The completeness of a call to the instantiation heuristic for cell abstraction highly depends on the context parameter, and more precisely, on its *relevant* cells. The intuition is that we may have a complete call whenever the set of cells of the abstracted array the context depends on is finite. In practice, this is safely approximated in our instantiation heuristic by cases where $\top$ is not returned by the `relevant` algorithm. The discussion on the impact on program verification is left to Chapter 6 as the result also depends on the instantiation of combinators.

5.2 Instantiation for the foundation of the data-abstraction framework

In this section we show how to construct the instantiation heuristic for the foundation of the data-abstraction framework, that is, the abstractions and combinators that allow us to combine and expand theory specific abstractions. For example, in the case of the theory of arrays, using these combinators, the instantiation heuristic for cell abstraction can be expanded into instantiation heuristics for several interesting abstractions: abstractions of the form $\sigma_{Cell} \bullet \sigma_{Cell} \bullet \sigma_{id}$ to handle multiple array variables and non array variables; abstractions of the form $\sigma^n_{Cell} \bullet \sigma_{Cell} \bullet \sigma_{id}$ to handle multiple cells and express invariants such as sortedness; but also abstractions such as smashing and slicing using the composition combinator; and even more complex abstractions such as those of Example 17 using the $\oplus$ combinator.

Creating abstraction heuristics for the combinators requires to use the instantiation heuristics for the abstractions they combine. These abstraction heuristics take several parameters: P, the

predicate being abstracted; abs, the abstraction that is used; $expr$, the abstracted value; and ctx, the context. When calling the instantiation heuristic, we may need to create a new predicate to be abstracted, adapt the abstraction abs to handle that new predicate, pass the subpart of $expr$ that corresponds to that abstraction, and mostly change ctx.

The main concern is preserving the completeness of the call to the instantiation heuristic: we aim to provide instantiation heuristics for combinators such that, whenever the call to the instantiation heuristics of the abstractions they combine are complete, then the call to the combinator is also complete. The main way to achieve this is by using the correct ctx parameter in the calls of the abstraction we combine: ctx is the parameter stating what must be *kept* during the transformation of a possibly infinite conjunction into a finite conjunction. This may not be evident in Equation 4.5 of the definition of a complete call to an instantiation heuristic which is not contraposed, but is made clear using the version of Equation 4.6 which is contraposed: the instantiation heuristic may change $vars$ into $vars'$, but that change must preserve $\neg ctx$. This is particularly evident in our heuristic for Cell abstraction of Section 5.1.

In practice, to correctly combine instantiations for two abstraction σ_1 and σ_2, we usually need to pick an order in which we instantiate: for example, for the pair abstraction $\sigma_1 \bullet \sigma_2$, we can retrieve the instantiation set for the first element of the pair using σ_1 and then for the instantiation set for the second element of the pair using σ_2, or vice-versa. After picking an order, let us say σ_1 before σ_2, we usually need, in addition to ctx, to preserve values due to σ_2 when instantiating σ_1 and vice-versa: instantiation may change $vars$ to $vars'$, and by doing so, the value of the second element of the pair may be changed during the instantiation of the first.

The technique to preserve additional values is the following. If the value v to preserve is boolean, we simply pass $v \rightarrow ctx$ as t he new context: thus using Equation 4.6 which is contraposed, we have $v \rightarrow ctx$ which must stay false during the transformation from $vars$ to $vars'$, and as $\neg(v \rightarrow ctx)$ is equivalent to $v \wedge \neg ctx$, the valuation of v must stay true during the transformation from $vars$ to $vars'$. If the value v that needs to be preserved is not boolean, we introduce a fresh predicate P_{any} and state that the boolean value $P_{any}(v)$ must be kept. Thus we use as context $P_{any}(v) \rightarrow ctx$. As this must be true for all models, thus for any valuation of P_{any}, this ensures that the value v is preserved.

This section is organized rather straightforwardly: we go through the abstractions and combinators of the foundation of the data-abstraction framework and give the corresponding instantiation heuristic and its proof of completeness. We remind the reader that the abstraction corresponding to the current instantiation is given by $abs(P)$ and that $F_\sigma[q]$ is used to handle existential quantifiers in F_σ. As existential quantifiers within F_σ do not change the reasoning for our instantiations heuristics, they are not discussed informally and are restricted to algorithms and proofs.

5.2.1 Instantiating σ_{id} and other finite abstractions

In this section, we give the instantiation heuristic for when $abs(P)$ is a finite abstraction, that is, whenever $abs(P)$ is an abstraction σ such that for all a, $\sigma(a)$ is finite. Among the finite abstractions, we mainly have the identity abstraction, but also many helper abstractions such as those of Example 16.

Definition 28 (Finite abstractions)**.** *A data abstraction σ is said to be finite whenever for all a, $\{(a^\#, q) \mid F_\sigma[q](a^\#, a)\}$ is finite.*

Instantiation amounts to transforming $\sigma(a)$ into $\sigma(a) \cap I$ where I is the instantiation set, as shown by Equations 4.7 and 4.8. Thus if $\sigma(a)$ is finite, picking I such that $\sigma(a) \subseteq I$ will enable complete instantiation. All we need is to return a set of expression $E(a)$ that returns such an I. This idea is straightforward and one can use $vars' = vars$ of Equation 4.8. Thus, the instantiation does not depend on the ctx parameter.

Algorithm 4 (Instantiation for σ_{id} and other finite abstractions)**.** *Let $E(a)$ be a set of expressions such that $\forall vars, [\![\{(a^\#, q) \mid F_\sigma[q](a^\#, a)\} \subseteq E(a)]\!]^{vars}$. In the case of the identity abstraction, $E(a) = \{(a, ())\}$.*

```
(* If abs(P) is finite *)
insts_P^abs a ctx = E(a)
(* The specific case where abs(P) = σ_id *)
insts_P^abs a ctx = {(a, ())}
```

Theorem 18 (Instantiation for finite abstraction)**.** *The instantiation heuristic of Algorithm 4 is complete.*

Proof. Introduce a, ctx. Assume $\forall vars, [\![\{(a^\#, q) \mid F_\sigma[q](a^\#, a)\} \subseteq E(a)]\!]^{vars}$. Let us use Equation 4.6 to prove completeness. Assume the left hand side of the implication where $I = E(a)$ and let us show the right hand side with $vars' = vars$. for all $(a^\#, q)$ we have, $[\![(a^\#, q) \in I \wedge F_{abs(P)}[q](a^\#, a)]\!]^{vars} \equiv [\![F_{abs(P)}[q](a^\#, a)]\!]^{vars}$ as $\forall vars, [\![\{(a^\#, q) \mid F_\sigma[q](a^\#, a)\} \subseteq I]\!]^{vars}$ As this is the only difference between the left hand side and the right hand side, we have the right hand side and the proof is finished. □

5.2.2 Instantiating $\sigma_1 \bullet \sigma_2$

In this section, we give the instantiation heuristic for when $abs(P)$ is the abstraction of pairs, that is, $\sigma_1 \bullet \sigma_2$. The main idea to create the instantiation of $insts_P^{abs}((a_1, a_2), ctx)$ is to use the instantiation heuristic for a_1 to retrieve an instantiation set I_1 and then the instantiation heuristic for a_2 to retrieve a set I_2 and simply use $I_1 \times I_2$ as instantiation set for $insts_P^{abs}((a_1, a_2), ctx)$. The goal is that when both of those calls are complete, the call to $insts_P^{abs}((a_1, a_2), ctx)$ is as well.

To call the instantiation heuristic for a_1 – and similarly for a_2– we need to figure out the parameters. We would like to use P as the predicate parameter, but this is not possible: P is not of the right type. We thus create the new predicate P_1 whose type is predicate over the type of a_1. The abstraction parameter must now account for P_1 and we use abs_1 which is the same as abs except that $abs_1(P_1) = \sigma_1$, as the abstraction for a_1 is σ_1. The last parameter to figure out is the context.

Let us use Equation 4.8 to understand the problem if we use ctx as the context parameter for the calls to instantiation heuristic for a_1 and a_2: using the completeness of the call for a_1, we will obtain $vars_1$ such that a_1 is now instantiated by I_1 and we have $\neg ctx$; and using the completeness

of the call for a_2, we will obtain $vars_2$ such that a_2 is now instantiated by I_2 and we have $\neg ctx$. The problem is that we need to unite $vars_1$ and $vars_2$ into to create $vars'$. This is not possible.

Therefore, the idea is to order our instantiations, for example, by first instantiating a_1 which gives us $vars_1$, and then using $vars_1$ to instantiate a_2 yielding $vars_2$. We basically need to track three values: ctx which must stay false, a_1 which starts at $\sigma(a_1)$ due to abstraction and is transformed into $\sigma(a_1) \cap I_1$ after instantiation, and a_2 which starts at $\sigma(a_2)$ due to abstraction and is transformed into $\sigma(a_2) \cap I_2$ after instantiation. The following table attempts to summarize this.

	Initial values	After instantiation of a_1	After instantiation of a_2
Evaluation context	$vars$	$vars_1$	$vars_2$
a_1	$\sigma(a_1)$	$\sigma(a_1) \cap I_1$	$\sigma(a_1) \cap I_1$
a_2	$\sigma(a_2)$	$\sigma(a_2)$	$\sigma(a_2) \cap I_2$
$\neg ctx$	$\neg ctx$	$\neg ctx$	$\neg ctx$

The important thing is that each instantiation must not change the values of the other elements. Thus, in addition to ctx, σ_2 must not change during the instantiation of a_1 and $\sigma(a_1) \cap I_1$ must not change during the instantiation of a_2. The asymmetry is due to the ordering of the instantiations. To keep those values intact, we preserve the value of a_2 during the instantiation of a_1 and the values a_1, I_1 during the instantiation of a_2. Note that this is not optimal but has the advantage of being extremely simple and enough for all cases we have encountered. A better version would for example use $\text{ite}(F_\sigma(e_1, a_1), e_1, \bot), \ldots, \text{ite}(F_\sigma(e_n, a_1), e_n, \bot)$ as values to preserve for $\sigma(a_1) \cap I_1$, where $\bot$ is any value and $\{e_1, \ldots e_n\} = I_1$. The exact property for what is needed is given in the proof of Theorem 5.

We now use the trick of modifying the context as described at the beginning of Section 5.2 to ensure that those values are preserved, and thus we introduce two new predicates P^1_{any} and P^2_{any} and modify abs_1 and abs_2 to abstract them with the identity abstraction. We formalize these ideas in Algorithm 5 with both orders for the instantiations.

Algorithm 5 (Instantiation algorithm for the $\bullet$ combinator)**.** *Computation for* $insts^{abs}_P(a, ctx)$ *when* $abs(P) = \sigma_1 \bullet \sigma_2$.

We only give the left to right instantiation, but the right to left one is symmetric.

1. *Create fresh predicates* $P_1, P_2, P^1_{any}, P^2_{any}$ *of the right type*
2. *Retrieve* I_1 *using:*
   (a) *Create* abs_1 *such that* $abs_1(P') = \text{ite}(P' = P_1, \sigma_1, \text{ite}(P' = P^1_{any}, \sigma_{id}, abs(P')))$
   (b) *Create* $preserve_1 = a_2$
   (c) *Compute* $I_1 = insts^{abs_1}_{P_1}(a_1, (P^1_{any}(preserve_1) \rightarrow ctx))$
3. *Retrieve* I_2 *using:*
   (a) *Create* abs_2 *such that* $abs_1(P') = \text{ite}(P' = P_2, \sigma_2, \text{ite}(P' = P^2_{any}, \sigma_{id}, abs(P')))$
   (b) *Create* $preserve_2 = (a_1, e_1, \ldots, e_n)$ *where* $\{e_1, \ldots, e_n\} = I_1$
   (c) *Compute* $I_2 = insts^{abs_2}_{P_2}(a_2, (P^2_{any}(preserve_2) \rightarrow ctx))$
4. *Return* $\{((a^\#_1, a^\#_2), (q_1, q_2)) \mid (a^\#_1, q_1) \in I_1 \wedge (a^\#_2, q_2) \in I_2\}$

Theorem 19 (Completeness of the instantiation for the $\bullet$ combinator.)**.** *Let* $abs(P) = \sigma_1 \bullet \sigma_2$*, the call to* $insts^{abs}_P((a_1, a_2), ctx)$ *is complete whenever the calls to* $insts^{abs_1}_{P_1}$ *and* $insts^{abs_2}_{P_2}$ *are.*

Proof. We use Equation 4.6. As this Equation has been created using contraposition, we first apply the completeness of I_2 and then of I_1.

1. Introduce the left hand side of the implication and $vars$ within it, and let us find $vars'$

satisfying the right hand side. We have

$$\big(\forall((a_1^\#,a_2^\#),(q_1,q_2)) \in [\![insts_P^{abs}((a_1,a_2),ctx)]\!]^{vars},$$
$$[\![F_{\sigma_1\bullet\sigma_2}[(q_1,q_2)]((a_1^\#,a_2^\#),(a_1,a_2))]\!]^{vars} \Rightarrow (a_1^\#,a_2^\#) \in \alpha_{\sigma_1\bullet\sigma_2}(\mathcal{M}(P))\big)$$
$$\wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

2. By reordering, expanding definitions and using notations of the algorithm, we get:

$$\big(\forall(a_2^\#,q_2) \in [\![I_2]\!]^{vars}, F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars})$$
$$\Rightarrow (\forall(a_1^\#,q_1) \in [\![I_1]\!]^{vars}, F_{\sigma_1}[q_1](a_1^\#,[\![a_1]\!]^{vars}) \Rightarrow (a_1^\#,a_2^\#) \in \alpha_{\sigma_1\bullet\sigma_2}(\mathcal{M}(P)))\big)$$
$$\wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

3. We use the completeness of I_2 to remove the instantiation and obtain $vars_2$ such that:

$$\big(\forall(a_2^\#,q_2), F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars_2})$$
$$\Rightarrow (\forall(a_1^\#,q_1) \in [\![I_1]\!]^{vars_2}, F_{\sigma_1}[q_1](a_1^\#,[\![a_1]\!]^{vars_2}) \Rightarrow (a_1^\#,a_2^\#) \in \alpha_{\sigma_1\bullet\sigma_2}(\mathcal{M}(P)))\big)$$
$$\wedge \neg [\![ctx]\!]^{vars_2}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

Proof:

(a) The Formula of Step 2 of this proof can be rewritten as:

$$\big(\forall(a_2^\#,q_2) \in [\![I_2]\!]^{vars}, F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars})$$
$$\Rightarrow a_2^\# \in \{x \mid \forall(a_1^\#,q_1) \in [\![I_1]\!]^{vars}, F_{\sigma_1}[q_1](a_1^\#,[\![a_1]\!]^{vars}) \Rightarrow (a_1^\#,x) \in \alpha_{\sigma_1\bullet\sigma_2}(\mathcal{M}(P))\}\big)$$
$$\wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

(b) Which can be rewritten as

$$\big(\forall(a_2^\#,q_2) \in [\![I_2]\!]^{vars}, F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars})$$
$$\Rightarrow a_2^\# \in \alpha_{\sigma_2}(\{x \mid \forall(a_1^\#,q_1) \in [\![I_1]\!]^{vars}, F_{\sigma_1}[q_1](a_1^\#,[\![a_1]\!]^{vars}) \Rightarrow (a_1^\#,x) \in \alpha_{\sigma_1\bullet\sigma_{id}}(\mathcal{M}(P))\})\big)$$
$$\wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs}}(\mathcal{M})}$$

(c) Let $\mathcal{M}_2$ such that:

i. $\mathcal{M}_2(P_2) = \{x \mid \forall(a_1^\#,q_1) \in [\![I_1]\!]^{vars}, F_{\sigma_1}[q_1](a_1^\#,[\![a_1]\!]^{vars}) \Rightarrow (a_1^\#,x) \in \alpha_{\sigma_1\bullet\sigma_{id}}(\mathcal{M}(P))\}$
ii. $\mathcal{M}_2(P^2_{any}) = \{[\![preserve_2]\!]^{vars}\}$
iii. Otherwise $\mathcal{M}_2(X) = \mathcal{M}(X)$

(d) Rewriting using $\mathcal{M}_2$ yields (as P_2 and P^2_{any} do not appear in ctx):

$$\big(\forall(a_2^\#,q_2) \in [\![I_2]\!]^{vars}, F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars}) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\mathcal{M}_2(P_2))\big) \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs_2}}(\mathcal{M}_2)}$$

(e) Using that we have $[\![P^2_{any}(preserve_2)]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs_2}}(\mathcal{M}_2)}$ by definition of $\mathcal{M}_2(P^2_{any})$.

$$\big(\forall(a_2^\#,q_2) \in [\![I_2]\!]^{vars}, F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars}) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\mathcal{M}_2(P_2))\big)$$
$$\wedge \neg [\![P^2_{any}(preserve_2) \rightarrow ctx]\!]^{vars}_{\alpha_{\mathcal{G}\sigma^{abs_2}}(\mathcal{M}_2)}$$

(f) We can now apply the completeness of the call to I_2 for $\mathcal{M}_2$ which yields $vars_2$ such that

$$\big(\forall(a_2^\#,q_2), F_{\sigma_2}[q_2](a_2^\#,[\![a_2]\!]^{vars_2}) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\mathcal{M}_2(P_2))\big)$$
$$\wedge \neg [\![P^2_{any}(preserve_2) \rightarrow ctx]\!]^{vars_2}_{\alpha_{\mathcal{G}\sigma^{abs_2}}(\mathcal{M}_2)}$$

(g) Unrolling $\neg[\![P^2_{any}(preserve_2) \rightarrow ctx]\!]$ gives us

$$\big(\forall(a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, [\![a_2]\!]^{vars_2}) \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\mathscr{M}_2(P_2))\big) \\ \wedge [\![P^2_{any}(preserve_2)]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)} \wedge \neg[\![ctx]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)}$$

(h) Expanding $\mathscr{M}_2(P_2)$ gives us:

$$\big(\forall(a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, [\![a_2]\!]^{vars_2}) \\ \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\{x \mid \forall(a_1^\#, q_1) \in [\![I_1]\!]^{vars}, F_{\sigma_1}[q_1](a_1^\#, [\![a_1]\!]^{vars}) \Rightarrow (a_1^\#, x) \in \alpha_{\sigma_1 \bullet \sigma_{id}}(\mathscr{M}(P))\})\big) \\ \wedge [\![P^2_{any}(preserve_2)]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)} \wedge \neg[\![ctx]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)}$$

(i) But $[\![P^2_{any}(preserve_2)]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)}$ means that $[\![preserve_2]\!]^{vars_2} = [\![preserve_2]\!]^{vars}$ by construction of $\mathscr{M}_2(P^2_{any})$.
Thus, $\forall(a_1^\#, q_1) \in [\![I_1]\!]^{vars}, F_{\sigma_1}[q_1](a_1^\#, [\![a_1]\!]^{vars})$ is identical to $\forall(a_1^\#, q_1) \in [\![I_1]\!]^{vars_2}, F_{\sigma_1}[q_1](a_1^\#, [\![a_1]\!]^{vars_2})$ by definition of $preserve_2$.

(j) Using that fact and removing $[\![P^2_{any}(preserve_2)]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)}$ means that we have:

$$\big(\forall(a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, [\![a_2]\!]^{vars_2}) \\ \Rightarrow a_2^\# \in \alpha_{\sigma_2}(\{x \mid \forall(a_1^\#, q_1) \in [\![I_1]\!]^{vars_2}, F_{\sigma_1}[q_1](a_1^\#, [\![a_1]\!]^{vars_2}) \Rightarrow (a_1^\#, x) \in \alpha_{\sigma_1 \bullet \sigma_{id}}(\mathscr{M}(P))\})\big) \\ \wedge \neg[\![ctx]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs_2}}(\mathscr{M}_2)}$$

(k) Reversing Step 3b gives us:

$$\big(\forall(a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, [\![a_2]\!]^{vars_2}) \\ \Rightarrow \forall(a_1^\#, q_1) \in [\![I_1]\!]^{vars_2}, F_{\sigma_1}[q_1](a_1^\#, [\![a_1]\!]^{vars_2}) \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \bullet \sigma_2}(\mathscr{M}(P))\big) \\ \wedge \neg[\![ctx]\!]^{vars_2}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

Which is the desired result!

4. Similarly, we use the completeness of I_1 to remove the instantiation and obtain $vars_1$ such that:

$$\big(\forall(a_2^\#, q_2), F_{\sigma_2}[q_2](a_2^\#, [\![a_2]\!]^{vars_1}) \\ \Rightarrow (\forall(a_1^\#, q_1), F_{\sigma_1}[q_1](a_1^\#, [\![a_1]\!]^{vars_1}) \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \bullet \sigma_2}(\mathscr{M}(P)))\big) \\ \wedge \neg[\![ctx]\!]^{vars_1}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

To prove this, follow the proof of Step 3 with the roles reversed.

5. Renaming $vars_1$ into $vars'$ and reversing Step 2, we obtain the desired result, that is:

$$\big(\forall((a_1^\#, a_2^\#), (q_1, q_2)), \\ [\![F_{\sigma_1 \bullet \sigma_2}[(q_1, q_2)]((a_1^\#, a_2^\#), (a_1, a_2))]\!]^{vars'} \Rightarrow (a_1^\#, a_2^\#) \in \alpha_{\sigma_1 \bullet \sigma_2}(\mathscr{M}(P))\big) \\ \wedge \neg[\![ctx]\!]^{vars'}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

□

5.2.3 Instantiating the ⊙ combinator

In this section, we give the instantiation heuristic for when $abs(P)$ is a composition of two abstractions, that is, $\sigma_2 \odot \sigma_1$. The simplest way to construct the result of $insts^{abs}_P(a, ctx)$ is simply

by running the data-abstraction algorithm with σ_1 on the clause $P(a) \rightarrow ctx$, yielding clauses C_1, then rerunning the data-abstraction algorithm with σ_2 and C_2 and finally rewriting C_2 so that it corresponds to a single run of the data-abstraction algorithm with the right instantiation set for $\sigma_2 \odot \sigma_1$.

In this approach, we first call $insts_P^{abs_1}(a, ctx)$ where abs_1 is the modification of abs such that $abs_1(P) = \sigma_1$. This yields an instantiation set for I_1. After the call to $eliminate$, the output clause has multiple instances of the predicate P, one for each $i^\# \in I_1$. These instances are ordered and, as we do not use parallel instantiation, this order matters for the second run of the $eliminate$ algorithm: all orders work but yield different calls to the instantiation heuristic. The context of each call is now huge as all other instances need to be passed into context, with the previous instances being instantiated and the others not. The generated context for the calls is fairly horrible and has been hidden for readability from Algorithm 6 by creating a variable INV whose value is given in the proof of Theorem 20.

The main difficulty is that the $eliminate$ algorithm assumes the clauses to be normalized which is not the case after the first run of the data-abstraction algorithm. Therefore, one needs to do as if we had normalized the clause into several clauses, instantiated each one of them and then managed to group back the clauses. This process is mainly syntactical and is not of great interest.

We prove the completeness result in Theorem 20. One of the major difficulties is correctly managing the abstractions and in this version, we managed to remove all hypotheses, unlike the result of our previous paper [BGM21]. In that version, we skipped Step 3.4 of the proof and required an additional assumption.

Algorithm 6 (Instantiation algorithm for the $\odot$ combinator)**.** *Computation for* $insts_P^{abs}(a, ctx)$ *when* $abs(P) = \sigma_2 \odot \sigma_1$.

1. *Retrieve* I_1*, the result of instantiation with* σ_1
   (a) *Create* abs_1 *such that* $\forall P', abs_1(P') = \text{ite}(P' = P, \sigma_1, abs(P'))$
   (b) *Compute* $I_1 = insts_P^{abs_1}(a, ctx)$
2. *Let* $I_1^l = I_1$.
3. *Order* I_1^l
4. *Create a new predicate* P'
5. ***While*** $I_1^l \neq \emptyset$
   (a) *Let* $(i, q) = min\ I_1^l$
   (b) $I_1^l -= (i, q)$
   (c) *Let* INV_{prog} *be* INV *of the proof of Theorem 20 where* $a^\# \in \alpha_2 \circ \alpha_1(\mathscr{M}(P))$ *is replaced by* $P'(a^\#)$.
   (d) *Let* $abs_2(X) = \text{ite}(X = P', \sigma_2, abs(X))$
   (e) $I_2^i = insts_{P'}^{abs_2}(i, INV_{prog} \rightarrow ctx)$
6. *Return* $\{(a^\#, (i^\#, q_1, q_2)) \mid (i^\#, q_1) \in I_1 \wedge (a^\#, q_2) \in I_2^{i^\#}\}$

Theorem 20 (Completeness of the instantiation for the $\odot$ combinator.)**.** *Let* $abs(P) = \sigma_2 \odot \sigma_1$*, the call to* $insts_P^{abs}(a, ctx)$ *of Algorithm 6 is complete whenever all its calls to the instantiation heuristic are.*

Proof. We use the definition of completeness without contraposition of Equation 4.5 for the main steps of the proof which we decompose in the following steps. Steps 3 and 4 are proven as separate parts and use the definition of completeness of Equation 4.6.

1. Introduce $\mathscr{M}$ and assume

$$\forall vars, [\![\forall (a^\#, (i^\#, q_1, q_2)), F_{\sigma_2 \circ \sigma_1}(a^\#, a) \Rightarrow a^\# \in \alpha_{\sigma_2 \odot \sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\wp\sigma^{abs}}(\mathscr{M})}$$

2. Reorder and expand to obtain:
$$\forall vars, [\![\forall (i^\#, q_1), \forall (a^\#, q_2), (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

3. Using the completeness of the call yielding I_1, we have:
$$\forall vars, [\![\forall (i^\#, q_1) \in I_1, \forall (a^\#, q_2), (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

4. Using the completeness of the calls yielding I_2^i we show the following while loop invariant:
$$\forall vars, \big([\![\forall (i^\#, q_1) \in I_1^l, \forall (a^\#, q_2), (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \wedge [\![\forall (i^\#, q_1) \in I_1 - I_1^l, \forall (a^\#, q_2) \in I_2^{i^\#}, (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars}\big) \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

5. Thus, as $I_1^l = \emptyset$ at loop exit, we have:
$$\forall vars, [\![\forall (i^\#, q_1) \in I_1, \forall (a^\#, q_2) \in I_2^{i^\#}, (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

6. Which is the desired result:
$$\forall vars, [\![\forall (a^\#, (i^\#, q_1, q_2)) \in insts_P^{abs}(a, ctx), F_{\sigma_2 \odot \sigma_1}(a^\#, a) \Rightarrow a^\# \in \alpha_{\sigma_2 \odot \sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

Proof of Step 3

3.1 From Step 2, we have
$$\forall vars, [\![\forall (i^\#, q_1), \forall (a^\#, q_2), (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

3.2 Which we can rewrite into
$$\forall vars, [\![\forall (i^\#, q_1), F_{\sigma_1}[q_1](i^\#, a) \Rightarrow i^\# \in \{x \mid \forall (a^\#, q_2), F_{\sigma_2}[q_2](a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]\}^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

3.3 But $\{x \mid \forall (a^\#, q_2), F_{\sigma_2}[q_2](a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))\}$ is simply $\gamma_{\sigma_2} \circ \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))$. Proof:
$$\begin{aligned}
&\{x \mid \forall (a^\#, q_2), F_{\sigma_2}[q_2](a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))\} \\
&= \{x \mid \forall a^\#, F_{\sigma_2}(a^\#, x) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))\} \\
&= \{x \mid \sigma_2(x) \subseteq \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))\} \\
&= \{x \mid x \in \gamma_{\sigma_2} \circ \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))\} \\
&= \gamma_{\sigma_2} \circ \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))
\end{aligned}$$

3.4 But $\gamma_{\sigma_2} \circ \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P)) = \alpha_1(\mathscr{M}(P))$. Proof:
$$\begin{aligned}
&\gamma_{\sigma_2} \circ \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P)) = \alpha_{\sigma_1}(\mathscr{M}(P)) \\
&\equiv \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P)) = \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P)) && \text{Using item 3 of Definition 8} \\
&\equiv true
\end{aligned}$$

3.5 Thus we have:

$$\forall vars, [\![\forall (i^\#, q_1), F_{\sigma_1}[q_1](i^\#, a) \Rightarrow i^\# \in \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

3.6 Applying the completeness of the call to I_1 yields:

$$\forall vars, [\![\forall (i^\#, q_1) \in I_1, F_{\sigma_1}[q_1](i^\#, a) \Rightarrow i^\# \in \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

3.7 Doing the rewrite of Steps 5 and 2 in reverse yields the desired result:

$$\forall vars, [\![\forall (i^\#, q_1) \in I_1, \forall (a^\#, q_2), (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

Proof of Step 4

4.1 Let INV be the following, where $(i,q) \notin I_1^l$

$$\forall (i^\#, q_1) \in I_1^l, \forall (a^\#, q_2), (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))$$
$$\wedge \forall (i^\#, q_1) \in I_1 - I_1^l - \boldsymbol{(i,q)}, \forall (a^\#, q_2) \in I_2^{i^\#}, (F_{\sigma_1}[q_1](i^\#, a) \wedge F_{\sigma_2}[q_2](a^\#, i^\#)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))$$

4.2 Proving the invariant amounts to assuming

$$\forall vars, [\![INV \wedge \forall (a^\#, q_2), (F_{\sigma_1}[q](i,a) \wedge F_{\sigma_2}[q_2](a^\#, i)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

and proving

$$\forall vars, [\![INV \wedge \forall (a^\#, q_2) \in \boldsymbol{I_2^i}, (F_{\sigma_1}[q](i,a) \wedge F_{\sigma_2}[q_2](a^\#, i)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \Rightarrow [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

4.3 Reason by contraposition, and assume that there exists $vars$ such that

$$[\![INV \wedge \forall (a^\#, q_2) \in I_2^i, (F_{\sigma_1}[q](i,a) \wedge F_{\sigma_2}[q_2](a^\#, i)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

4.4 If $[\![F_{\sigma_1}[q](i,a)]\!]^{vars} \equiv false$, we directly obtain the result of Step 4.10 with $vars' = vars$. Let us continue with $[\![F_{\sigma_1}[q](i,a)]\!]^{vars} \equiv true$

4.5 We thus have

$$[\![\forall (a^\#, q_2) \in I_2^i, F_{\sigma_2}[q_2](a^\#, i) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars} \wedge \neg [\![ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

4.6 Let $\mathscr{M}'$ be the model such that:

(a) $\mathscr{M}'(P') = \{\alpha_{\sigma_1}(\mathscr{M}(P))\}$

(b) Otherwise $\mathscr{M}'(X) = \mathscr{M}(X)$

4.7 Using the definition of $\mathscr{M}'(P')$ and the fact we have $[\![INV]\!]^{vars}$ from Step 4.3, we have $[\![INV_{prog}]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs_2^i}}(\mathscr{M}')}$. Thus, we have

$$[\![\forall (a^\#, q_2) \in I_2^i, F_{\sigma_2}[q_2](a^\#, i) \Rightarrow a^\# \in \alpha_{\sigma_2}(\mathscr{M}'(P'))]\!]^{vars} \wedge \neg [\![INV_{prog} \rightarrow ctx]\!]^{vars}_{\alpha_{\mathscr{G}\sigma^{abs_2^i}}(\mathscr{M}')}$$

4.8 Using the completeness of the call yielding I_2^i, we have $vars'$ such that:

$$[\![\forall (a^\#, q_2), F_{\sigma_2}[q_2](a^\#, i) \Rightarrow a^\# \in \alpha_{\sigma_2}(\mathscr{M}'(P'))]\!]^{vars'} \wedge \neg [\![INV_{prog} \rightarrow ctx]\!]^{vars'}_{\alpha_{\mathscr{G}\sigma^{abs_2^i}}(\mathscr{M}')}$$

4.9 We thus have, $[\![INV_{prog}]\!]^{vars'}_{\alpha_{\mathscr{G}\sigma^{abs}_2}(\mathscr{M}')}$, Thus, we have $[\![INV]\!]^{vars'}$.

4.10 We thus have, unwinding the definition of $\mathscr{M}'$

$$[\![INV \wedge \forall (a^\#, q_2), (F_{\sigma_1}[q](i,a) \wedge F_{\sigma_2}[q_2](a^\#, i)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars'} \wedge \neg [\![ctx]\!]^{vars'}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

4.11 Reversing the initial steps (mainly contraposition), we obtain the desired result:

$$\forall vars', [\![INV \wedge \forall (a^\#, q_2) \in I^i_2, (F_{\sigma_1}[q](i,a) \wedge F_{\sigma_2}[q_2](a^\#, i)) \Rightarrow a^\# \in \alpha_{\sigma_2} \circ \alpha_{\sigma_1}(\mathscr{M}(P))]\!]^{vars'} \Rightarrow [\![ctx]\!]^{vars'}_{\alpha_{\mathscr{G}\sigma^{abs}}(\mathscr{M})}$$

□

5.2.4 Instantiating the ⊗ combinator

In this section, we give the instantiation heuristic for when $abs(P)$ is a product of two abstractions, that is, $\sigma_1 \otimes \sigma_2$. The simplest way to construct the result of $insts^{abs}_P(a, ctx)$ is to view it as $insts^{abs'}_P((a,a), ctx)$, where $abs'(P) = \sigma_1 \bullet \sigma_2$: the product combinator abstracts a single element by a pair and keeps links between them. This is similar to abstracting a pair of two identical elements. This is how our algorithm is defined.

Expanding this definition means that we use the instantiation set $I_1 \times I_2$ where I_1 is the instantiation set for a with abstraction σ_1 and I_1 is the instantiation set for a with abstraction σ_2. The computation of the values that need to be preserved are $\sigma(a)$ during the first instantiation and $\sigma(a) \cap I_1$ during the second.

This approach can be refined in the case of the power combinator used to construct $\sigma^n_{Cell} \equiv \sigma_{Cell} \otimes \ldots \otimes \sigma_{Cell}$, n times. We thus refine the instantiation set for $\sigma_1 \otimes \sigma_2$ in the specific case of $\sigma_1 = \sigma_2$, that we simply call σ. If we name I the instantiation of a with σ, we simply return $I \times I$ without requiring any values to be preserved.

Algorithm 7 (Instantiation algorithm for the ⊗ combinator)**.**

Listing 5.7 – The general case where $abs(P) = \sigma_1 \otimes \sigma_2$

```
let insts^abs_P a ctx =
  let P' = new_predicate () in
  let abs' = fun X → if X = P' then σ1•σ2 else abs(X) in
  insts^abs'_P'((a,a), ctx)
```

Listing 5.8 – The specific case where $abs(P) = \sigma \otimes \sigma$

```
let insts^abs_P a ctx =
  let abs' = fun X → if X = P then σ else abs(X) in
  let I = insts^abs'_P(a,ctx)
  (* We basically return I×I *)
  concat_map (fun (a#1, q_1) → map (fun (a#2, q_2) → (((a#1,a#2), (q_1, q_2))) I) I
```

Theorem 21 (Completeness of the instantiation for the ⊗ combinator.)**.** *Let $abs(P) = \sigma_1 \otimes \sigma_2$, the call to $insts^{abs}_P((a,ctx)$ of Algorithm 7 is complete whenever all its calls to the instantiation heuristic are.*

Proof of the general case. This proof is not of any interest and a simple unrolling of the definitions of both completeness call yields the desired result. □

Proof of the specific case. For simplicity, consider Equation 4.7. The proof for the case with quantifiers with F_σ relies on the same result.

We have:

1. $\sigma \otimes \sigma(a) \equiv \sigma(a) \times \sigma(a)$
2. $\alpha_{\sigma\otimes\sigma}(\mathcal{M}(P)) = \alpha_\sigma(\mathcal{M}(P)) \times \alpha_\sigma(\mathcal{M}(P))$
3. But $\sigma(a) \times \sigma(a) \subseteq \alpha_\sigma(\mathcal{M}(P)) \times \alpha_\sigma(\mathcal{M}(P))$ is equivalent to: $\sigma(a) \subseteq \alpha_\sigma(\mathcal{M}(P))$
4. Thus, $\sigma \otimes \sigma(a) \subseteq \alpha_{\sigma\otimes\sigma}(\mathcal{M}(P))$ is equivalent to $\sigma(a) \subseteq \alpha_\sigma(\mathcal{M}(P))$
5. Using the same principle for $\sigma(a) \cap I'$, where $I \times I$ is the instantiation set returned by $insts_P^{abs}$ and $I' \times I'$ is its equivalent with the notations of Equation 4.7, we have: $((\sigma \otimes \sigma(a)) \cap (I' \times I')) \subseteq \alpha_{\sigma\otimes\sigma}(\mathcal{M}(P))$ is equivalent to $(\sigma(a) \cap I') \subseteq \alpha_\sigma(\mathcal{M}(P))$
6. We have thus transformed both parts of Equation 4.7 for abs into their counterpart for abs', which is assumed complete. The proof is finished.

□

5.2.5 Instantiating the ⊕ combinator

In this section, we give the instantiation heuristic for when $abs(P)$ is a sum of two abstractions, that is, $\sigma_1 \oplus \sigma_2$. The simplest way to construct the result of $insts_P^{abs}(a, ctx)$ is to view $\sigma_1 \oplus \sigma_2(a)$ as sometimes $\sigma_1(a)$ and sometimes $\sigma_2(a)$. Thus, we simply instantiate with the instantiation set for σ_1 when is should be viewed as $\sigma_1(a)$ and with the instantiation set for σ_2 when it should be viewed as σ_2.

Algorithm 8 (Instantiation algorithm for the ⊕ combinator)**.**

```
let insts_P^abs a ctx =
  let (P_1, P_2) = new_predicates () in
  let abs_1 = fun X → if X = P_1 then σ_1 else abs(X) in
  let abs_2 = fun X → if X = P_2 then σ_2 else abs(X) in
  let I_1 = insts_P1^abs1(a,ctx) in
  let I_2 = insts_P2^abs2(a,ctx) in
  (* We basically return {T_1(x) | x∈I_1}∪{T_2(x) | x∈I_2} *)
  (map (fun (a_1^#,q_1) → (T_1(a_1^#),(T'_1(q_1)))) I_1) @
  (map (fun (a_2^#,q_2) → (T_2(a_2^#),(T'_2(q_2)))) I_2)
```

Theorem 22 (Completeness of the instantiation for the ⊕ combinator.)**.** *Let $abs(P) = \sigma_1 \oplus \sigma_2$, the call to $insts_P^{abs}((a, ctx)$ of Algorithm 7 is complete whenever all its calls to the instantiation heuristic are.*

Proof. This proof is not of any interest and a simple unrolling of the definitions of all completeness call yields the desired result. □

Overview and contribution within this chapter

In this chapter we constructed instantiation heuristics for cell abstraction and for the combinators of the data-abstraction framework. Combining them gives us a full instantiation heuristic that we can use for the programs of Chapter 2 and Chapter 3.

The main concern is whether this definition of the instantiation heuristic allows its calls to be complete. In the case of the combinators, it only depends on whether the abstractions they combine are complete for a specific context whereas for cell abstraction, it all depends on the concept of what we call relevant cells. Thus in practice not all instantiations are complete and the impact on programs will be discussed in Chapter 6.

The main contributions of this chapter are:

1. Showing that existing techniques can be expressed as specific instances of an instantiation heuristic.
2. Proving the completeness of the [MG16] heuristic.
3. Showing that the completeness of the instantiation heuristic for cell abstraction depends on a concept called *relevant* cells.
4. Constructing the instantiation heuristic for cell abstraction using this concept. A big improvement compared to existing techniques [BMR13; MA15; MG16] and previous iterations of this framework is the handling of array equalities [BG20; BGM21]. This is non-trivial as array equalities involve an unbounded number of cells.
5. The construction of the instantiation heuristic for combinators such that completeness of calls is preserved. Capturing the correct context and proving that completeness is preserved is non-trivial and requires precise proof technique: the previous iteration [BGM21] had additional hypothesis for the $\odot$ combinator due to a missing idea within the proof.

6 Theoretical contribution: impact on program verification

Our scheme to verify safety properties of programs consists in first transforming programs into Horn clauses, then simplifying the unbounded data-structures of our Horn clauses using the data-abstraction framework, and finally solving them using a Horn clause solver. The theoretical contribution of this manuscript is the simplification process which is parameterized by a user-chosen abstraction. One of the main focuses of this manuscript is to ensure that the data-abstraction framework algorithm does not over-approximate the simplification induced by the abstraction but exactly implements it.

In Chapter 3 we showed that if the data-abstraction framework algorithm *implements* the abstraction, then the Horn clauses constructed from a broad class of container algorithms could be automatically simplified. Thus, the main issue is writing the data-abstraction framework algorithm so that it implements the abstraction for the clauses encoding the programs we consider.

For a given abstraction, the *implements the abstraction* property of Chapter 3 classifies programs into two categories. The first category is programs that we say are *properly handled*, that is, programs for which the data-abstraction framework algorithm *implements the abstraction* on the set of Horn clauses that encode its verification problem, as described in Chapter 2. We hope to prove that the container algorithms of Chapter 3 belong to that category of programs. The second category is programs that we say are *improperly handled*, that is, programs for which the data-abstraction framework algorithm may fail to *implement the abstraction* on the set of Horn clauses that encode its verification problem.

Although the data-abstraction framework does not implement the abstraction on *improperly handled* programs, they may still be simplified by the data-abstraction algorithm and used for static verification purposes: *soundness* is still ensured and the only drawback is that the framework may over-simplify as the *relative completeness* property may be loss. In many ways, for *improperly handled* programs, the data-abstraction framework behaves like many other techniques: there is no theoretical guarantee of how well it performs and one should use an experimental evaluation to conclude.

However, *properly handled* programs have the additional property that if our verification scheme fails to certify them, then the issue is not within the simplification due to the data-abstraction algorithm. It is either that the program itself is *buggy*, or that the user-chosen abstraction was not adapted, or that the back-end Horn clause solver was not powerful enough. Thus, if we prove that the container algorithms of Chapter 3 belong to *properly handled* programs, and we assume we have a perfect Horn solver for Horn clauses without data-structure invariants, then these algorithms can be automatically proven.

As the data-abstraction algorithm operates on each clause independently, the characterization

of *properly* and *improperly* handled programs depends on the individual Horn clauses that encode them. We thus extend the notion of *properly* and *improperly* handled to Horn clauses (respectively program instructions), in such a way that programs that only contain *properly handled* Horn clauses in their encoding instructions (respectively program instructions) are *properly* handled.

Furthermore, the notion of *properly* or *improperly* handled depends on the abstraction. In this manuscript, we focused on arrays and mainly abstractions that can be constructed by σ_{Cell} abstraction. In this chapter, we aim to classify program instructions on whether they are *properly* or *improperly* handled for such abstractions.

In Section 6.1, we attempt to characterize the Horn clauses that are currently *properly* and *improperly* handled. In Section 6.2, we use the characterization of Section 6.1 to help characterize program instructions that are currently *properly* and *improperly* handled. We show that this characterization is not sufficient and improve on it. Then, in Section 6.3, we give possible extensions of our current verification scheme such that additional program instructions may be *properly* handled. Finally, in Section 6.4, we summarize for each type of program instruction the possible extensions that we believe are required to *properly* handle them.

6.1 Classifying the current handling of Horn clauses

In Section 4.3 we showed that Horn clauses that are *properly handled* correspond to clauses for which all calls to $insts$ are *complete*. The *completeness* of a call to $insts$ depends on whether the `relevant` algorithm of Listing 5.6 on page 102 returns a set containing $\top$. Thus, the set of Horn clauses that are *properly handled*, can just be defined as the set of clauses for which no calls to `relevant` returns a set containing the element $\top$.

Although this definition of the clauses we properly handle – i.e. Horn clauses for which no call to `relevant` returns a set containing the element $\top$ – is not practical to work with, our attempts to characterize those Horn clauses differently lead to a huge non-readable definition. Thus, in this section we opt to give an intuition of that set by using examples.

In this section, we limit our discussion to abstractions that abstract tuples with the $\bullet$ combinator where each individual element is abstracted by σ_{id} if the element is not of array type and σ^n_{Cell} if the element is of array type. These abstractions represent the main current use of our framework and the discussion for these abstractions should develop enough intuition for the other abstractions.

Furthermore, this discussion can be simplified by only considering σ^1_{Cell} instead of σ^n_{Cell}: this is because the only difference between $insts$ for σ^1_{Cell} and σ^n_{Cell}, as witnessed in Algorithm 7, is the abs parameter which is unused in the `relevant` algorithm.

Let us now classify the Horn clauses depending on whether they generate a call to `relevant` that returns a set containing $\top$.

1. For Start Horn clauses, that is, Horn clauses with no predicates in their premises, there are no calls to $insts$, thus, there are no calls to `relevant` that return a set containing $\top$. An example of such a clause is $sorted(a) \rightarrow P(a)$.
2. For Horn clauses that use, in a manner that is linked to an array expression used in a predicate within the premises, quantifiers or array theory constructors that are not reads, writes or equalities, `relevant` may return a set containing $\top$: currently the `relevant` algorithm is not made to handle such cases.
   Examples of such clauses are $P(a) \wedge sorted(a) \rightarrow P'(a)$, $P(a) \wedge (\forall i, b[i] = a[i+1]) \rightarrow P'(b)$, $P(a,n) \wedge BoundEq(a,b,0,n) \rightarrow P'(b)$, ... However, this is not the case for

$P(a) \wedge BoundEq(b, ConstArray(0), 0, n) \rightarrow P'(b)$ as in this case the use of $BoundEq$ and $ConstArray$ are not linked to a.

3. For Horn clauses that constrain an infinite number of cells of an array in two different ways in the premises, `relevant` may return a set containing $\top$: there is no finite *relevant* set of cells. The main cases where this occurs either by using quantifiers or complex array theory operations, or more simply, when two similar array expressions are used as parameters to predicates in the premises.
   Examples of such clauses are $P(a,b) \wedge a = b \wedge a[i] = 0 \rightarrow P'(a)$, $P(a) \wedge F(a,b) \rightarrow P'(b)$, $P(a, a[i \leftarrow 3]) \wedge a[i] = 0 \rightarrow P'(a)$, $P(a) \wedge sorted(a) \rightarrow P'(a)$, ... but not $P(a) \wedge a = b \wedge a[i] = 0 \rightarrow P'(a)$, $P(a) \wedge F(b) \rightarrow P'(b)$ or $P(a,i,v) \wedge a' = a[i \leftarrow v] \wedge a[i] = 0 \rightarrow P'(a')$.
   In the latter case, the array equality $a' = a[i \leftarrow v]$ does not constrain the cells of a as the cells of a' are not constrained elsewhere in the premises; it is only used as a way to define a'.

6.2 Classifying the current handling of program instructions

In this section, the goal is to classify programs that are *properly* handled and programs that are not.

A naive approach to this problem consists in using the discussion of Section 2.3 which links categories of program instructions to types of Horn clauses and combine it with the results of Section 6.1. This yields Table 6.1 that reproduces Table 2.3 and provides, for each category of instructions, our first understanding of whether these types of program instructions are *properly* handled.

However, we believe the results of Table 6.1 can be refined: the classification of Horn clauses used in that table does not correctly capture the specificity of the clauses generated from programs. Thus, in this section, we revisit some of these results. The final results are available in Table 6.2.

Table 6.1 – Naive understanding of the current handling of program operations

Category	Example	Type	Handled[#]
Simple prg transitions	Loop block of `merge_sorted`	linear, basic	No[1]
Complex prg transitions	`b = sub_array(a, 0, middle)`	linear, complex	No
Program point verification	`assert(sorted(res));`	linear, assert, complex	No[2]
Function verification	Verifying `merge_sorted`	linear, start, complex	Yes[*]
		linear, assert, complex	No
Inline function call	`res= merge_sorted(b, c);`	linear, trivial	No[1]
		linear, trivial	No[1]
Modular function call	`res= merge_sorted(b, c);`	linear, trivial	No[1]
		non-linear, trivial	No
Summary function call	`res= merge_sorted(b, c);`	linear, assert, complex	No[2]
		linear, complex	No

Yes stands for *properly* handled and No stands for *improperly* handled.
[1,2] These results are revised in the discussion of this section.
[*] A small additional discussion is provided in this section for *start, complex* Horn clauses.

Revisiting the No[1] results of Table 6.1. In Table 6.1, we claim not to handle clauses that are *linear, basic* or even *linear, trivial*, and thus, even simple program transitions cannot be *properly* handled. While it is true that some *linear, trivial* clauses, such as $P(a,a) \rightarrow P'(a)$, are not *properly* handled, we argue they do not occur in clauses generated by simple program transitions.

The clause $P(a,a) \rightarrow P'(a)$ in fact represents the program instruction `if(a==b) ...`: in the transformation of programs into Horn clauses, each parameter of P represents a different program variable. Thus, $P(a,a)$ states that the second variable, here assumed to be named b, is equal to a.

As *linear, basic* Horn clauses that are not *properly* handled in Section 6.1 all share the property that a is somehow linked to two different parameters of P, we believe these cases should not be considered in the case of program instructions. This is also the case for *linear, trivial* clauses generated by both inline and modular function calls, and thus, all No[1] results of Table 6.1 should be updated to Yes.

Revisiting the No[2] results of Table 6.1. In Table 6.1, we claim not to handle clauses that are *linear, assert, complex*, and thus, the verification of program points and functions cannot be *properly* handled. While it is true that some *linear, assert, complex* clauses, such as $P(a,a) \rightarrow e'$ and $P(a) \rightarrow (\exists i, a[i] = 2)$, are not *properly* handled, we argue that they are not relevant to our programs.

First, the problem of clauses such as $P(a,a) \rightarrow e'$ has already been discussed in our revisiting of the No[1] results.

Secondly, assertion Horn clauses such as $P(a) \rightarrow (\exists i, a[i] = 2)$ express the safety property $\exists i, a[i] = 2$. However such a property is not expressible by cell abstraction: as discussed in Section 3.3.2, properties expressible by cell abstractions are of the form $\forall k, P(k, a[k])$. Furthermore this safety property is also not expressible by abstractions created from cell-abstraction using our combinators, that is, by abstractions discussed in this manuscript.

Thus, as this safety property is not expressible by the abstractions we consider, the transformation of this clause after the $abstract$ algorithm is unsatisfiable[1]. Thus whether or not $eliminate$ is complete for such clauses is not relevant to our current discussion.

Let us now discuss cases that correspond to safety properties expressible by the abstractions we consider. That is, clauses of the form $P(a,\ldots) \rightarrow (\forall k_1,\ldots,k_n, expr)$ where $expr$ uses only basic operations. Such Horn clauses can be rewritten, while preserving semantics, into $P(a,\ldots) \rightarrow expr$, by transforming the universal quantifiers into free variables. Such Horn clauses are *properly* handled, and thus, all No[2] results of Table 6.1 should be updated to Yes.

An additional discussion about *start, complex* clauses. *Complex* Horn clauses, as defined in Chapter 2, use either additional quantifiers over non-predicate expressions or a broader theory to manipulate arrays. During the execution of data-abstraction framework algorithm, more precisely after the $abstract$ algorithm, we introduce additional quantifiers. These quantifiers are then removed during the $eliminate$ algorithm.

However, we do not remove quantifiers that were already present in the input clauses, as may be the case for *complex* Horn clauses. These quantifiers may still be an issue for solvers although this is not the fault of the data-abstraction framework. We explain that in most cases, after abstraction, *start, complex* Horn clauses can be transformed into *start, basic* Horn clauses.

Similarly to our discussion for *linear, assert, complex* Horn clauses, the *start, complex* Horn clauses that appear when using cell abstraction are expected to be of the form $(\forall k_1,\ldots,k_n, expr) \rightarrow P(a,\ldots)$. As such, these Horn clauses can not be simplified. However, after

[1] This is not entirely true if we account for degenerate nodels such as the empty model. However, the next sentence still holds.

cell abstraction, they are equivalent to $(\forall k_1,\ldots,k_n, expr) \rightarrow P^{\#}((a[k_a],k_a),\ldots)$ and the quantifiers in these clauses may be removed. This is because they belong in the *array property fragment*[2] for which a simplification algorithm already exists [BMS06]. Note that we gave the example for σ^1_{Cell} but this also holds for σ^n_{Cell}.

Note that this removal of quantifiers is only possible after abstraction: in $(\forall k_1,\ldots,k_n, expr) \rightarrow P(a,\ldots)$, P is not abstracted and all properties about the array a can be expressed, thus there is no complete instantiation for the quantifiers $\forall k_1,\ldots,k_n$, and the clause cannot be simplified.

Therefore, although the data-abstraction framework *properly* handles *start, complex* Horn clauses, we believe that one may need to apply the method of [BMS06] for current solvers to have a real chance at solving them.

Summary of the current handling of program instructions. In Table 6.2 we provide an updated version of Table 6.1 that takes into account our discussion. The results are much more promising and in fact, this shows that for programs for which the function calls are inlined, only use basic array operations, and with safety properties that are expressed with universal quantifiers and basic array operations, the simplification scheme of the data-abstraction framework **successfully implements the abstraction**.

With respect to Example 14 on page 49 giving examples of container algorithms, this means the simplification of the data-abstraction framework successfully implements the abstraction on all of them, except *fold* operations and algorithms that are written recursively, as may be the case for *merge-sort*, *quick-sort* or *binary-search*. However, non-recursive versions of these algorithms can still be handled, such as the non-recursive merge sort of Listing 2.10 or the binary search program of Listing 1.2.

However, we are still unable to *properly* handle programs with complex transitions, or functions calls that are handled using either summaries or the modular approach. Thus, in the current verification scheme, the user needs to make a choice between two solutions.

1. Using only simple program transitions and inlining function calls at the cost of simplicity and scalability, but with the insurance that the simplification induced by the data-abstraction framework will implement the abstraction.
2. Using possibly complex program transitions and using scalable techniques to handle function calls at the cost of losing all relative completeness guarantees about what the data-abstraction framework may do. As discussed at the beginning of this chapter, many verification techniques do not provide any relative completeness guarantees and fall within this case. Therefore, this should not be seen as back-breaking.

In the next section, we aim to discuss extensions that would enable the simplification induced by the data-abstraction framework to implement the abstraction, even on programs containing complex program transitions and using scalable techniques to handle function calls. The extensions we discuss are mainly work in progress and should be viewed as such.

6.3 Extending the current verification scheme

The current verification scheme does not *properly* handle complex program transitions, function calls using summaries, and function calls using the modular approach. We have also mainly limited our discussion to abstractions that abstract non-array variables by the identity and array vari-

[2] A simplified definition of that fragment is given in Definition 26 on page 90.

Table 6.2 – Current handling of program operations after discussion

Category	Example	Type	Handled[#]
Simple prg transitions	Loop block of `merge_sorted`	linear, basic	Yes
Complex prg transitions	`b = sub_array(a, 0, middle)`	linear, complex	No
Program point verification	`assert(sorted(res));`	linear, assert, complex	Yes
Function verification	Verifying `merge_sorted`	linear, start, complex	Yes[*]
		linear, assert, complex	No
Inline function call	`res= merge_sorted(b, c);`	linear, trivial	Yes
		linear, trivial	Yes
Modular function call	`res= merge_sorted(b, c);`	linear, trivial	Yes
		non-linear, trivial	No
Summary function call	`res= merge_sorted(b, c);`	linear, assert, complex	Yes
		linear, complex	No

Yes stands for *properly* handled and No stands for *improperly* handled.
[*] One should probably apply the array property fragment algorithm on the clause output by the data-abstraction algorithm if we want current solvers to handle them.

ables with σ^n_{Cell}. In this section, we discuss how the current verification scheme may be improved to *properly* handle such program instructions and how other abstractions for arrays may be handled.

Currently our limitation to *properly* handle some Horn clauses mainly comes from the requirement that `relevant a ω` of Listing 5.6 on page 102 must return a *relevant* set of cells for a in ω, as defined in Definition 27 of page 98. In the current verification scheme, this property is not only sufficient, but also necessary[3]. This is a consequence of the combination of Theorem 11 on page 73 that states that all calls to $insts$ must be complete and of Theorem 16 on page 98 that states that $insts$ for cell abstraction is only complete if it uses a *relevant* set of indices.

Thus, we only see two ways to improve on this limitation. The first consists in improving the `relevant` algorithm, and the second consists in changing some of the choices of the current verification scheme.

6.3.1 Improving the *relevant* algorithm

The goal of `relevant a ω` algorithm is to return a finite set of indices for the array a that is said *relevant* for the expression ω, as in Definition 27 of page 98. To improve `relevant a ω`, we first need for such a finite set to exist. Let us start by showing that such finite sets may not always exist by considering two clauses: $P(a) \wedge cond = (\forall i, a[i] = 0) \rightarrow P'(cond)$, encoding the instruction `bool cond = is_zero_initialized(a)`, and $P(a) \wedge F(a,b) \rightarrow P'(b)$, encoding the instruction `b = f(a);`. But, before we do so, let us give an intuition of what are *relevant sets*. Note that the full formalization of this idea is the focus of Section 5.1. The intuition is that for an array and a clause, saying that the call to `relevant` due to that clause returns a relevant set for that array simply means that it returns all the indices of that array that are constrained in multiple ways in the premises of the clause.

Thus, in the clause $P(a) \wedge cond = (\forall i, a[i] = 0) \rightarrow P'(cond)$, there is no finite *relevant* set: all cells

[3] Formally, it is only almost necessary as shown by the additional hypothesis of Theorems 11 and 16, but we do not believe the *almost* part to leave room for improvements that have significant impact.

of a may be constrained by P, but all cells of a are also constrained by $(\forall i, a[i] = 0)$. Thus, a *relevant* set would have to contain all indices and cannot be finite. In the clause $P(a) \wedge F(a,b) \rightarrow P'(b)$, a similar problem happens: all cells of a may be constrained by P, but all cells of a may also be constrained by $F(a,b)$: $F(a,b)$ may imply $(\forall i, a[i] = 0)$ depending on the $\mathscr{M}$. As the set returned by `relevant` must be *relevant* for all models, we deduce that a *relevant* set would have to contain all indices and cannot be finite.

These examples aim to show that *proper* handling of modular function calls is not within reach of improvements to the `relevant` algorithm, nor are some *summary handling of function calls*. However, we will show that some *summary handling of function calls*, even some involving an unbounded number of cells of an array, are within reach. Therefore, in this section, our goal tackles simultaneously two problems: determining the types of clauses for which a finite relevant set exists and improving the `relevant` algorithm so that, whenever such a set exists, it finds it.

This goal takes very different forms depending on what we aim to improve. In Section 6.3.1.1 we aim to improve the `relevant` algorithm on the abstraction that abstracts non-array variables with the identity abstraction and array variables with σ^n_{Cell}; whereas in Section 6.3.1.2 we show how improvements to the `relevant` algorithm may enable us to handle arrays of arrays, or even the multiset abstraction of Example 17, so that we may verify that merge sort does not alter array contents. Finally, we discuss the consequences of such extensions in Section 6.3.1.3.

6.3.1.1 Improvements targeting cell abstraction

Currently, we believe the handling of unquantified array reads, array writes and array equalities in the `relevant a ω` algorithm of Listing 5.6 on page 102 is well tackled: our intuition is that whenever ω only contains such operations on arrays, then the existence of a *relevant* set implies that `relevant` returns a *relevant* set. Formally, such a theorem cannot be stated, mainly due to degenerate cases such as $P(a,b) \wedge (false \rightarrow a = b) \rightarrow P'(a)$. However, we believe this intuition holds.

Thus, our goal is to improve the `relevant` algorithm when other types of operations are used in these clauses, mainly when these operations involve an unbounded number of cells of the array. We start by discussing how new constructors for the theory of arrays, such as $BoundEq, Insert, Erase$, may be handled. Then, as these constructors may be expressed with quantified array reads, we attempt to generalize these ideas to quantified array reads. Throughout this process, a distinction appears between program instructions that may be *properly* handled through such improvements and program instructions that cannot be *properly* handled through such improvements.

The ideas of this section are mostly generalization of the hard work we have done in the `relevant` algorithm for array equalities: array equalities already are a form of operation that involve an unbounded number of cells!

Handling of $BoundEq$. Our idea to handle $BoundEq(a,b,lower,upper)$ expressions within ω in the `relevant` algorithm mainly comes from array equalities: $BoundEq$ represents bounded array equalities. Our idea is that the set returned by `relevant` for expressions of the form $BoundEq(a,b,lower,upper)$ should be the same as for expressions of the form $a = b$, except that we restrict ourselves to elements within bounds.

However, we cannot statically determine whether an expression e verifies $lower \leq e < upper$: this depends on the evaluation context for the variables of the clause. Thus, our idea to discard e when e is not within bounds is to transform e into the expression $\text{ite}(lower \leq e < upper, e, \perp)$, where $\perp$ may be any value of the same type as e, but semantically represents a value that should

not be used.

To explain how this changes the code of the `relevant` algorithm, let us explain what happens for the simple equality $a = b$ when a and b are both variables. The generalization to when a and b are expression containing array writes follows the same process in both cases. The current code in the `relevant` algorithm to handle the array equality $a = b$ can be described as: (1) retrieve `relevant` for b (2) retrieve the relevant cells of the other parameters of the expression – here nothing (3) return the union of both The handling we suggest in the `relevant` algorithm for array equality $BoundEq(a,b,lower,upper)$ can be described as: (1) retrieve `relevant` for b (2) for each expression e of that set, transform it into ite$(lower \leq e < upper, e, \bot)$ (3) retrieve the relevant cells of the other parameters to the expression – here $lower$ and $upper$ (4) return the union of both

This technique allows one to *properly* handle clauses such as Clause 2.2 from the merge sort example, and thus program instructions such as `Array b=sub_array(a, lower, upper);` can now be *properly* handled . However, it does not allow one to handle clauses such as $P(a,b,n) \wedge BoundEq(a,b,0,n) \wedge a[i] = 0 \rightarrow P'(a)$ for the same reason that we could not handle $P(a,b) \wedge a = b \wedge a[i] = 0 \rightarrow P'(a)$: the array a is constrained on a possibly infinite set of cells in two different ways, first as the first argument of P, second as the second argument of P for the cells between 0 and n.

In many ways, array instructions such as array equalities or bounded array equalities can be *properly* handled in program statements that create a new variable, such as `Array b=sub_array(a, lower, upper);` , but not when they are used within a complex branching statements such as `if(a==sub_array(b, 0, n)) ...` , which is the bounded equality counterpart to `if(a==b) ...` . We already see a distinction of the program instructions that may be *properly* handled through improvements to the `relevant` algorithm and program instructions that cannot be *properly* handled through such improvements.

Handling $Insert, Erase, \ldots$ We introduced $BoundEq$ in the theory of arrays to transform program statements such as `Array b=sub_array(a, lower, upper);` without needing quantifiers. One may introduce $Insert, Erase$ to the theory of arrays to similar transform statements such as `a.insert(i, v);` . $Insert(a,i,v)$ is defined as the array a' such that $\forall k, a'[k] = \text{ite}(k < i, a[k], \text{ite}(k = i, v, a[k-1]))$. One may do the same for $Erase$.

To handle the expressions such as $a' = Insert(a,i,v)$ in the `relevant` algorithm, the main idea is to view them as $a'[i] = v$ and $a' = a$ with a shift in some indices. This leads to the following additional case in the `relevant` algorithm for the expression $a' = Insert(a,i,v)$, when a' is the variable in which we are interested: First retrieve `relevant` for a. Secondly for each expression e of that set, transform it into ite$(e < i, e, e+1)$. Then retrieve the relevant cells of the other parameters to the expression – here i and v. Finally return the i and the union of both: i is added as it is as if we had $a'[i] = v$. The case where a is the variable in which we are interested is similar except that the shift of indices is reversed and that i does not need to be added.

Again this allows us to *properly* handle clauses such as $P(a,i,v) \wedge a' = Insert(a,i,v) \rightarrow P'(a',i,v)$ and $P(a,i,v) \wedge b = Insert(a,i,v) \rightarrow P'(a,b,i,v)$ which respectively represent the instructions that modifies an array using `insert` , and the instruction that creates a new array b from the insertion of the element v in a. However, branching statements such as `if(a==insert(b, 0, n)) ...` cannot be *properly* handled for the reasons as `if(a==sub_array(b, 0, n)) ...` or `if(a==b) ...` cannot.

Generalizing these ideas with quantifiers. Previously, we discussed cases where program instructions such as `a.insert(i, v);` or `Array b=sub_array(a, lower, upper);` , were transformed

using new theory elements such as $BoundEq$ and $Insert$. Another approach it to transform these instructions by using only reads and writes but within additional universal quantifiers. Thus, the translation of `a.insert(i, v);` would use $\forall k, a'[k] = \text{ite}(k < i, a[k], \text{ite}(k = i, v, a[k-1]))$ instead of $a' = Insert(a, i, v)$.

The approach using quantifiers has a major advantage and a major drawback. Without quantifiers, new theory constructors are created for each specific program construction, and thus require to add specific subcases in the `relevant` algorithm. However, each specific subcase does not seem to be very hard to write. When using quantifiers, the `relevant` algorithm does not need to be modified for each specific program construction; however, it needs to handle a general set of quantified expressions, and this seems much harder than adding specific cases. Here, we explain how general is the set of quantified expressions that may be tackled.

Array equality can be expressed with quantifiers as $\forall k, a[k] = b[k]$ and is currently already handled in the `relevant` algorithm. To handle it, the `relevant` algorithm for a calls `relevant` for b. We now discuss how our previous ideas to extend it can be generalized.

The expression $BoundEq(a, b, lower, upper)$ can be expressed with quantifiers as $\forall k, (lower \leq k < upper) \rightarrow a[k] = b[k]$. To handle it, our extension to the `relevant` algorithm mainly transforms each expression e in the set returned by `relevant` for b by $\text{ite}(lower \leq e < upper, e, \bot)$.

We generalize this idea to quantified expressions of the form $\forall k, cond \rightarrow a[k] = b[k]$ by transforming each expression e in the set returned by `relevant` for b by $\text{ite}(cond[k \leftarrow e], e, \bot)$, where $cond[k \leftarrow e]$ is $cond$ where all instances of k are replaced by e.

The expression $a' = Insert(a, i, v)$ can be expressed with quantifiers as $a'[i] = v \wedge (\forall k \neq i, a'[k] = a[\text{ite}(k < i, k, k-1])])$. The handling of the quantified part, that is, $\forall k \neq i, a'[k] = a[\text{ite}(k < i, k, k-1])]$ was handled by transforming each expression e in the set returned by `relevant` for a by $\text{ite}(e < i, e, e+1)$. The important thing to notice is that the function that to e associates $\text{ite}(e < i, e, e+1)$ is exactly the inverse function of the function that to k associates $\text{ite}(k < i, k, k-1])$.

We thus generalize this idea to quantified expressions of the form $\forall k, cond \rightarrow a[k] = b[\rho]$, where ρ is an expression which represents an invertible function with respect to k. For such expression, we first transform each expression e in the set returned by `relevant` for b by the inverse expression of ρ before handling $cond$ as previously.

Furthermore, we also wish to handle expressions with quantifiers such as $\forall k, a[k] = b[k] + 1$. For this expression, the `relevant` algorithm should return the same set of indices as for $\forall k, a[k] = b[k]$: the difference only affects values, not indices.

We thus generalize this idea to quantified expressions of the form $\forall k, cond \rightarrow a[k] = \phi(b[\rho])$, where $\phi(b[\rho])$ simply denotes an expression for which the only use of k is in $b[\rho]$.

For such expressions, the `relevant` algorithm should be the same as for expressions of the form $\forall k, cond \rightarrow a[k] = b[\rho]$: first retrieve `relevant` for b. Secondly for each expression e of that set, transform it with the inverse of ρ. Thirdly,for each element of that new set, transform it with $\text{ite}(cond[k \leftarrow e], e, \bot)$ Then retrieve the relevant cells of the other parameters to the expression – here the expressions $cond$ and ρ. Finally return the union of both sets.

An important restriction which may not seem obvious is that we require ϕ to be invertible. This is because $\forall k, a[k] = b[k] + 1$ returns the result of `relevant` for b whereas $\forall k, a[k] = 0$ constrains all indices of the array a and thus `relevant` must return a set containing $\top$. Ensuring that ϕ is invertible is key to avoiding such problems.

Moreover, to handle symmetric cases, we also handle quantified expressions of the form

$\forall k, cond \rightarrow \phi'(a[\rho']) = \phi(b[\rho])$. We formalize these ideas with an additional case in the `relevant` algorithm of Listing 5.6 on page 102 described in Listing 6.1, where we limited ourselves to the case where ϕ and ϕ' are the identity: this does not change the algorithm and avoids complicating Listing 6.1.

We believe that this extension of the `relevant` algorithm still satisfies Theorem 17 and that the proof of it should be similar: this extension mainly uses the same ideas as array equalities with the following differences:

1. Bounds are added by using the condition $cond$. We believe this should not affect the proof much.
2. Index permutations are possible by using the invertible function ρ. We believe this does not impact the proof much as in Step 3 of the proof of Theorem 17, we show that forall values of k, the value of $a[k] = b[k]$ is preserved. Thus, we believe that we should now state that forall values of k, the value $a[\rho'] = b[\rho]$ is preserved.
3. Mapping the value of $b[\rho]$ is possible using ϕ and ϕ'. We believe this should not affect the proof much.

Listing 6.1 – Improvement of the handling of quantifiers in Listing 5.6.

```
let rec relevant_impl ω visited a_var expr=
  if a_var ∈ visited then [] (*We ignore visited variables*)
  else
  (
    match expr with
    | ...
    | ∀k, cond → a_var[e1] = b_var[e2]
        when (is_invertible k e1) and (is_invertible k e2) ->
        let relevant_b = relevant_impl ω a_var::visited b_var ω in

        let inv_2 = compute_inverse_expr k e_2 in
        let relevant_k = map inv_2 relevant_b in
        let relevant_cond_k =
          map (fun e -> ite((replace cond k e) e ⊥))
                                              relevant_k in

        let inv_1 = compute_inverse_expr k e_1 in
        let relevant_a = map inv_1 relevant_cond_k in

        let relevant_from_args = concat_map (relevant_impl ω visited a_var)
                                            [cond,e1,e2] in
        relevant_a @ relevant_from_args
    | ...
  )
```

6.3.1.2 Improvements targeting other abstractions

Handling of arrays of arrays. To abstract arrays of arrays, one probably wishes to use the abstraction $(\sigma_{id} \bullet \sigma_{Cell}) \odot \sigma_{Cell}$, or a several cell variant of it.

Calls to the `relevant` algorithm for such abstractions yield expressions containing array equalities such as $a = b[i]$, where a is an array. Currently, the `relevant` algorithm adds $\top$ to the returned set for such expressions.

To handle this, one would want to use the technique we used for array equalities and simply call `relevant` for $b[i]$. However, this is not possible: $b[i]$ is not a variable and the array it represents

depends on the value of i. The idea is to take the union of the relevant set of all arrays $b[expr]$ that appear in the expression for some $expr$.

Thus, the `relevant_impl` algorithm is expanded to take as parameter not only the array variable `a_var`, but also a depth component. This variant of the algorithm is already implemented and we redirect the reader towards the implementation which is described in Chapter 7.

This improvement to the `relevant` algorithm enables one to use abstractions such as $(\sigma_{id} \bullet \sigma_{Cell}) \odot \sigma_{Cell}$ to abstract arrays of arrays and its n cell variant.

Handling complex primitives such as multiset. In Example 17 of page 54 we discussed combining cell abstractions with abstractions that use the multiset of the elements of an array so that properties such as *sorting algorithms do not change the multiset of elements* expressible by the abstraction. The problem with such abstractions, is that the F_σ formula needs to be written in a given theory and the constructors of that theory must be handled by `relevant`. We suggest adding the constructor $Multiset$ such that $a = Multiset(b)$ states that $\forall k, a[k] = card\{i \mid b[i] = k\}$.

Thus, we expand the `relevant` algorithm to handle the relevant set for a in specific cases where expressions of the form $a' = Multiset(b)$ are used. The main idea is that the expression $a = Multiset(b) \wedge b' = b[i \leftarrow v] \wedge a' = Multiset(b')$ can be rewritten as $a' = \text{ite}(b[i] = v, a, a[v \leftarrow a[v]+1][b[i] \leftarrow a[v]-1])$ and thus we only need to retrieve the relevant set for a, which now also contains v.

We have not yet implemented this extension, but we believe that the expression passed to `relevant` that arise from the abstraction of Example 17 should be within reach of variations of this technique.

6.3.1.3 Discussion of the extensions to the *relevant* algorithm

Should the improved `relevant` algorithm and proofs of the extensions we suggest work as expected, these extensions should improve our handling of *linear, complex* Horn clauses and open up possibilities for more complex abstractions, such as abstracting arrays of arrays or abstractions using constructors such as $Multiset$.

The main discovery during these attempts to improve the `relevant` algorithm is that not all *linear, complex* Horn clauses are the same.

For the first category of *linear, complex* Horn clauses, such as those generated using function summaries by the program instructions `Array b=sub_array(a, i, n);` or `a.insert(i, v);`, there exists a finite *relevant* set of cells and our improvements to the `relevant` aim to compute them. We call such Horn clauses *linear, complex,* ***bound-perm-map*** as our understanding is that they mainly bound conditions, permute indices, and map array contents, as shown by our extension for quantifiers. We believe such Horn clauses correspond to function calls using summaries – or program instructions – that create or modify existing arrays. However, unlike the *basic* operations that are limited to reads and writes, they may do so on an unbounded number of cells at once: in fact, we believe they correspond to function or instructions whose return value may be expressed through combinations of permutations, and the functional `map` and `filter` operations, though this remains to be proven.

For the second category of *linear, complex* Horn clauses, such as those generated using function summaries by the program instructions `if(a==sub_array(b, 0, n)) then ...` or `if(a==insert(b, 0, n)) then ...`, but also simply `bool b=(a==sub_array(b, 0, n));`, no finite *relevant* set of cells exists and no possible improvements to the `relevant` algorithm may help *properly* handle them. We call such clauses *linear, complex,* ***collapsing***: our understanding is that these

clauses, instead of transferring properties to another array, collapse an unbounded number of cells into a single boolean or possibly a single integer. We believe such Horn clauses mainly correspond to function calls using summaries – or program instructions – that use function calls that return non-array values and yet involve an unbounded number of cells: in the cases previously mentionned, `a==sub_array(b, 0, n)` should in fact be viewed as `is_sub_array(a, b, 0, n)` which is a function returning a boolean value.

However, we are as yet unsure how complete this categorization of Horn clauses and program instructions is. This is still work in progress and we hope future work will improve the understanding of what may and may not be *properly* handled through improvements to the `relevant` algorithm. One of the main remaining questions is to what extent, if we consider only clauses with reads, writes and quantifiers, the improvement of Listing 6.1 is from exactly computing whether a finite *relevant* set exists.

6.3.2 Modifying the verification scheme

In the previous section we discussed that non-linear clauses and clauses we call *linear, complex, **collapsing*** cannot be *properly* handled through improvements of the `relevant` algorithm. In this section, we discuss two ways the our verification scheme may be improved for such clauses.

6.3.2.1 Passing additional information to the context

In the $eliminate$ algorithm, we chose that the only information the instantiation heuristic $insts$ should have is the current clause, expressed through $P, expr$ and ctx parameters, and the current abstraction abs. These choices seemed to be the only sensible choice considering what we knew at the time as other options seemed to be passing the full set of Horn clauses, or passing no context information or, ... However, the understanding of the type of clauses we may *properly* handle, and the limits that we encounter for a complete instantiation scheme may lead to specific improvements in this regard.

For example, in non-linear function calls such as $P(a) \wedge F(a,b) \rightarrow P'(b)$, the main problem for a complete instantiation is that when considering only this clause, we have no information about $F(a,b)$. Should we know that $F(a,b)$ should in fact be $a = b$, then this clause could be replaced by $P(a) \wedge F(a,b) \rightarrow P'(b)$ and we could *properly* handle it. Our idea to pass such information to the instantiation heuristic is by constraining the models on which the definition of the completeness of a call to $insts$ must hold. For example, the fact that $F(a,b)$ is $a = b$ can be expressed by only constraining the models to those that verify $\mathcal{M}(F) = \{(a,a)\}$.

In fact, we have already used this technique to limit ourselves to abstract models of the form $\alpha_{\mathcal{G}^{\sigma^{abs}}}(\mathcal{M})$, and this enabled us to construct a complete instantiation heuristic for cell abstraction on *linear, basic* clauses: as discussed in Section 5.1.1, without this restriction, we would never have been able to avoid degenerate models.

Now that we have described what we believe is the correct way to pass on additional information to help solving complex Horn clauses, the problem of finding out what is the important information to pass on and how to generate it still remains to be explained. Finding out the type of information that will significantly help our instantiation heuristic $insts$ to be complete, without requiring full information as in providing $\mathcal{M}(F) = \{(a,a)\}$ is an interesting path of research we wish to explore. As to how to compute the information that may be passed, we have two ideas.

First, predicates encoding functions calls have a specific property of the models that satisfy them: for any input, either there exists an output or the problem is already unsatisfiable without

analyzing the function call (i.e. a precondition is not satisfied). In practice, this means we can discard models such as those that imply $\forall a, a[i] = 0$ for $F(a,b)$. We do not yet know the importance of such additional information, but we have two reasons to believe this may have an impact. First, one of the main problems we have encountered is the degenerate non-empty model discussed in Section 5.1.1. Second, one of the main differences between *bound-perm-map* and *collapsing* Horn clauses discussed in Section 6.3.1.3 is whether functions collapse the array information into a boolean or integer. However, we do not believe this idea alone to be sufficient.

Secondly, one may ask the user or use another automated tool to provide additional information about the models we may need to consider. For example, there may be cases where providing even a very under-specified summary of the function may be sufficient to construct a complete call for the heuristic $insts$.

6.3.2.2 Pre-transforming the Horn clauses

Perhaps the most effective option to *properly* handle more complex Horn clauses may be to simply transform, before we use the data-abstraction framework, a set of Horn clauses that contains them into another simpler set of Horn clauses while preserving semantics.

We have already encountered this with *inlining*: the *inlining* technique we described to handle function calls in programs has its counterpart on Horn clauses. This inlining technique for Horn clauses transforms non-linear Horn clauses into linear Horn clauses and has the same drawbacks: it does not work on all sets of Horn clauses, mainly those encoding recursive functions; and the number of Horn clauses may drastically increase.

We suggest another transformation technique to *properly* handle *linear, complex* Horn clauses, especially those that are *linear, complex, collapsing* that cannot be *properly* handled with improvements to the `relevant` algorithm. We give an example of how to transform the *linear, complex, collapsing* clause $P(a,b,n) \wedge (\forall i, 0 \leq i < n \rightarrow a[i] = b[i]) \rightarrow P'(a,b,n)$ encoding the instruction `if(a==sub_array(b, 0, n)) then ...` into a set of *linear, basic* Horn clauses.

The main idea is to encode the quantifier by simulating a loop. This yields the following set of Horn clauses:

$$\begin{aligned} P(a,b,n) &\rightarrow P_1(a,b,n,0) \\ P_1(a,b,n,i) \wedge i < n \wedge a[i] = b[i] &\rightarrow P_1(a,b,n,i+1) \\ P_1(a,b,n,i) \wedge i = n &\rightarrow P'(a,b,n) \end{aligned}$$

We believe such an approach is possible for all kinds of *linear, complex* Horn clauses, and thus enables us to fully *properly* handle function summaries. However, we feel this approach is less satisfying than improvements within the data-abstraction framework; though we believe it is still much better than inlining: the number of additional Horn clauses is much smaller as we do not embed the full function. In practice, we expect at most to generate three additonal clauses for each function call.

This transformation still needs to be automated, even though we believe that, when limited to quantifiers, this should not be much of an issue.

6.4 Summary of the impact on the verification of programs

In this section, we summarize the cases for which we wrote in Table 6.2 that our current verification scheme cannot handle, that is, for which the simplification it induces may not implement the

abstraction.

1. For Horn clauses that are *linear, assert, complex*, we separated them into two categories
   (a) Those that are *bound-perm-map* such as those encoded by `b = sub_array(a, 0, middle);` or `a.insert(i, v);` that we expect to *properly* handle with the improvements to the `relevant` algorithm of Section 6.3.1. Thus complex program instruction or summary handling of function calls that are used to create a new array or modify a current array are expected to be handled by this technique.
   (b) Those that are potentially *collapsing* such as those encoded by `if(a==sub_array(b, 0, n)) then ...`, `if(a==insert(b, 0, n)) then ...`, or `bool b=(a==sub_array(b, 0, n));`. We cannot *properly* handle these Horn clauses through improvements of the `relevant` algorithm. Thus, we mainly suggest to use a pre-transformation of Section 6.3.2.2 that transforms such complex Horn clauses into basic clauses. Thus complex program instruction or summary handling of function calls are expected to be *properly* handled by this technique.
2. For Horn clauses that are *non-linear, trivial*, such as $P(a) \wedge F(a,b) \rightarrow P'(b)$, they are generated by modular handling of function calls. We currently suggest to avoid them by using inlining or function summaries; perhaps future work improving our discussion of Section 6.3.2.1 might show how passing additional information to the $insts$ heuristic enables to *properly* handle them.

Therefore, we currently already *properly* handle the non-recursive version of the merge-sort algorithm using inlining and we expect to *properly* handle the recursive version of merge sort in the near future by using function summaries. Furthermore, we also discussed in Section 6.3.1.2 how the `relevant` algorithm may be improved to *properly* handle other abstractions, mainly arrays of arrays and the multiset abstraction of Example 17. We believe that these improvements should already enable us in theory[4] to verify that the non-recursive merge-sort function using inlining not only returns a sorted array, but also keeps the contents of the array intact; however, this still needs to be implemented and validated.

[4] That is, if we have a perfect Horn solver for integer problems.

7 Data-Abstraction: experiments

In Chapters 3, 4 and 5, we presented the theoretical goals, algorithms and results of the data-abstraction framework. Our focus has been arrays, and more specifically, properties that can be proven using cell abstraction. The full discussion on what cell abstraction with the data-abstraction combinators may prove can be found in Chapter 3.

One of the main concerns was whether our algorithm would allow to prove all these properties, that is, whether it would implement the abstraction. We showed this concern was highly dependent on the instantiation heuristic, that we constructed in Chapter 5, and in Chapter 6 we justified that the instantiation heuristic handles most of the cases discussed in Chapter 3, especially the container algorithms of Example 14.

Therefore, one expects that in practice the data-abstraction algorithm should enable us to automate the proof of these container algorithms. To confirm our expectations, we implemented the data-abstraction algorithm in 2k lines of code in Ocaml and tested it on array algorithms written in a toy java language. The code of the data-abstraction algorithm is publicly available at `https://github.com/vaphor/DataAbstraction/` and the array example at `https://github.com/vaphor/array-benchmarks`. The full toolchain is as described in the Figure 1.1 of the introduction. We submitted the full toolchain in a docker container as a artefact for SAS21 and it has been awarded the extensible badge. The docker container is available at `https://hub.docker.com/r/jbraine/data_abstraction_benchmarks`.

However, the results were not as expected on these array benchmarks. Therefore, we did not extend the toolchain to handle more complex examples such as those discussed in Example 17 as most of our effort has been on improving the results of these simpler examples.

In this chapter, we first explain our choice of front-end and back-end tools of our toolchain; then we give the experimental results and argue that the problems are not created by our data-abstraction tool; finally, we discuss how the results can be improved.

7.1 Toolchain

The toolchain is composed of three parts: a front-end that transforms programs to Horn clauses, the data-abstraction tool and a back-end Horn clauses solver.

Front-end. Horn clauses have become quite a popular format in the last decade and we had several possible front-end tools. Among them, we can mostly name SeaHorn, JayHorn and Vaphor [Gur+15; Kah+16; MG16].

The SeaHorn tool transforms the LLVM representation of a program into a set of Horn clauses. The main drawback of SeaHorn is the use of the LLVM representation: it introduces problems that would not arise from a natural translation from programs. For example, cells for an integer array are indexed at the byte level in the LLVM representation. Thus, as integer span over several bytes, any invariant about the cells of that array requires handling modulo arithmetic. This greatly complexifies invariants and we feared the back-end solver might find the modulo arithmetic troublesome. Other problems of using the LLVM representation are optimizations that discard undefined behaviors, ... Furthermore, SeaHorn is perhaps the most mature of the three tools and modifying it or handling the options correctly for our purposes felt complicated. However, SeaHorn demonstrated that it could handle the parsing and transformation of most benchmarks of the Static Verification Competition and thus was of great interest.

The JayHorn tool was in its infancy when I started my PhD and at the time it did not yet feel reliable enough. Furthermore, it did not handle a set of already written interesting benchmarks and thus did not present much benefit.

The Vaphor tool implements the [MG16] transformation discussed in Section 5.1.2.1 from pseudo java programs to Horn clauses without arrays using a version of cell abstraction. It handles a set of array benchmarks very relevant to cell-abstraction as the aim of the [MG16] paper was similar. The main advantage of this tool is that the code is small and clear enough that it can be modified to suit our purposes, and we were familiar with the implementation.

The Vaphor tool from [MG16] seemed almost perfect as a good testing ground for our purposes and I started modifying the implementation so that it would transform programs into Horn clauses without the cell abstraction of [MG16]. The aim was to use this tool to figure out what was really necessary, and perhaps later move to SeaHorn in order to have access to the SV-comp benchmarks. In practice this latter stage was never reached as the issues we encountered made SV-comp benchmarks unnecessary. The modification of the Vaphor tool implements basic program transformation and Listing 7.1 taken from our paper [BGM21] should demonstrate the syntax it handles. The modified Vaphor tool is available at `https://github.com/vaphor/hornconverter`.

Listing 7.1 – Example program from our SAS21 paper

```
class array_odd{

    static int i,N, v;
    static int a[] ;
    static void main() {
      N=Support.random();//size of array
      //Initialize array to only even values
      i=0;
      while(i<N)
      {

        v = Support.random();
        a[i] = v*2;
        i=i+1;
      }
      //Increase each element by 1
      i=0;
      while(i<N)
      {
        a[i] = a[i]+1;
        i=i+1;
```

```
      }
      //Assert that all elements are odd
      i=0;
      while(i<N)
      {
        assert(a[i]%2==1);
        i=i+1;
      }
    }
}
```

The data-abstraction tool. We implemented the data-abstraction framework in 2k lines of Ocaml. The code is publicly available at https://github.com/vaphor/DataAbstraction/. The code is structured in similar fashion to how the data-abstraction framework has been described in this PhD. The code is fully extensible with other abstractions and the way to do so is described in the Readme of the git repository.

Back-end solver. The solver we use as back-end is Z3. Initially, Z3 was an SMT solver and was not equipped to handle Horn clauses. However, in the last decade, Z3 has been improved to handle many kinds of problems, including Horn clauses. One of the technique Horn solvers are based on is counterexample-guided abstraction refinement by using Craig interpolants. For Horn clauses the technique amounts to

- Unroll a finite execution[1] of the Horn clauses.
- Solve using the a SMT solver whether there is no path leading to a failed assertion on that finite execution. If this is not the case, we have found a counter-example for that finite execution and thus, the set of Horn clauses is not satisfiable. Otherwise, continue.
- From the proof, attempt to generalize the reason why such a path does not exists. Formally, this is called an interpolant but more simply put, this is an invariant for that finite execution.
- Check whether that invariant is inductive, that is, works for all executions: we know it works for that finite execution, but we need to prove it works for any execution.
- If it is, the proof is done. Otherwise, add another finite execution and hope that the interpolant generated from the proof of all checked finite execution is inductive.

This technique does not necessarily finish as we may never find an inductive interpolant. The key part when using this technique is thus the generalization step. We have also considered other solvers such as Eldarica based on possibly other techniques, but the results have been much worse.

7.2 Experimental results

We have tested our toolchain on benchmarks mostly adapted from [MG16]. However, as discussed in Chapter 2, one of the major differences is that we have rewritten the safety properties by using while loops instead of a random index check. Thus, unlike the [MG16] experimentation, our benchmarks require real quantified invariants, which explains why our results are drastically different. These benchmarks are available at https://github.com/vaphor/array-benchmarks, and the full executable toolchain with reproducible results is available as a Docker container at

[1] This is similar to unrolling loops in programs. In Horn clauses these are chain of Horn clauses that go from a given predicate to that same predicate.

https://hub.docker.com/r/jbraine/data_abstraction_benchmarks. Documentation on how to use and modify these tools is available on their respective readme files.

The abstraction used on these benchmarks are σ_{id}, σ^1_{Cell} for all arrays and σ^2_{Cell} for all arrays. The purpose of id is to show the results given by the back-end solver without any meaningful transformation, and they perfectly match with what the solver outputs on the Horn clauses without any transformation. The results of the solver may be *Buggy*, stating that the Horn clauses are unsatisfiable; *Certified*, stating that the Horn clauses are satisfiable; *Timeout*, stating that the time limit has been reached without an answer; *Unknown*, stating that the Horn solver believes it can not solve the problem. In practice, the *Unknown* answer seems to occur when the solver's attempts to find an inductive invariant loops on the same non-inductive invariant; whereas the *Timeout* answer seems to occur when the solver's attempts to find an inductive invariant successively improves the invariant but without ever making it inductive. The results have been run with several random seeds for the back-end solver on a desktop computer with a 10 minute time limit and the results are summarized in Table 7.1. We do not believe the hardware or the time limit to be of importance.

Table 7.1 – Experimental results

Example type	Abstraction	Result
Buggy examples	all	*Buggy* in $< 2s$
Expressible by $Cell^1$ or $Cell^2$	id	mostly *Unknown*, sometimes *Timeout*
Expressible by $Cell^1$	$Cell^1$ or $Cell^2$	mostly *Timeout*, sometimes *Unknown*
Expressible only by $Cell^2$	$Cell^1$	*Buggy* in $< 2s$
Expressible by $Cell^2$	$Cell^2$	mostly *Timeout*, sometimes *Unknown*

Our results are extremely disappointing: there is no *Certified* benchmark! However, we still have the following. First, the solver returns *Buggy* on exactly the examples on which it should, that is, programs that are initially bugged and programs for which we used an abstraction that cannot express its invariants. Secondly, using the data-abstraction tool has not made results worse: the solver already only returned *Unknown* or *Timeout* on the non-buggy examples. Thirdly, we may see what can be considered a slight improvement: we have cases for which the solver returns *Unknown* with the id abstraction that become *Timeout* with Cell abstraction. Fourthly, we were concerned about how our toolchain was so much worse than the one of [MG16], but launching the [MG16] tool on our modified benchmarks yielded the same result. Lastly, we do not have any *wrong* answer, but only answers for which the solver returns *Unknown* or *Timeout* instead of *Certified*.

Our main concern was whether the data-abstraction tool was working properly. We address this concern by checking two properties: first, that the predicates of the Horn clauses we generate are of the appropriate type, mainly that they do not use arrays; secondly, that the generated Horn clauses are satisfiable for the cases on which the solver returned *Unknown* or *Timeout*. If both these properties are satisfied, this indicates that our data-abstraction tool is functioning properly and that the problem is that the Horn clauses, even after abstraction, are still too complex for the back-end solver.

We checked the first property by hand, and for the second property our goal was to help the solver find a model that satisfies the generated Horn clauses, thus proving them satisfiable. Our idea consisted in adding additional assertion clauses to the Horn problem, thus constraining the Horn problem so that the interpolants generated by the Horn solver have a better chance of being inductive. Note that if the Horn problem with an additional clause is satisfiable by a model, so is

the Horn problem without the additional clause.

However, instead of writing these additional assertion clauses by hand in the Horn problem after transformation by the data-abstraction tool, we opted to automatically generate them from *hints* that could be added in programs. We modified our front-end to handle such hints and in Listing 7.2, we show how hints are written in programs and the additional clauses they generate. These generated additional assertion clauses are added to the set of Horn clauses given to the data-abstraction tool and are transformed by it, thus yielding *abstract hints*. By using hints and playing with the random seed, we managed to prove all *Unknown* and *Timeout* answers into *Certified* ones; thus, our data-abstraction tool achieves its purpose, but this is not enough for current state of the art solvers.

Listing 7.2 – Hints on the example program from our paper [BGM21]

```
class array_odd{

    static int i,N, v;
    static int a[] ;
    static void main() {
      N=Support.random();//size of array
      //Initialize array to only even values
      i=0;
      while(i<N)
      {
        //hint forall k, (0<=k && k<i) -> (a[k] %2 == 0);
        /* If we call this program point P1, this generates the clause:
                                  P1(a,i,N,v) ∧ (0 ≤ k ∧ k < i) → a[k]%2 = 0                    */

        v = Support.random();
        a[i] = v*2;
        i=i+1;
      }
      //Increase each element by 1
      i=0;
      while(i<N)
      {
        //hint forall k, (0<=k && k<i) -> (a[k] %2 == 1);
        /* If we call this program point P2, this generates the clause:
                                  P2(a,i,N,v) ∧ (0 ≤ k ∧ k < i) → a[k]%2 = 1                    */

        a[i] = a[i]+1;
        i=i+1;
      }
      //Assert that all elements are odd
      i=0;
      while(i<N)
      {
        //hint forall k, (0<=k && k<N) -> (a[k] %2 == 1);

        /* If we call this program point P3, this generates the clause:
                                  P3(a,i,N,v) ∧ (0 ≤ k ∧ k < N) → a[k]%2 = 1                    */

        assert(a[i]%2==1);
        i=i+1;
      }
    }
}
```

7.3 Improving the results

Even though we showed that the data-abstraction tool experimentally achieves its purpose, this does not enable the automatic verification of even simple array programs. The cause is the back-end solver, and in our implementation of the data-abstraction tool, we have attempted to improve the generated Horn clauses so that the solver may succeed in the following ways:

Fully removing arrays. In our generated Horn clauses, predicates do not have any array arguments as those have been abstracted. However, clauses still use arrays to express the abstract transition relation. In theory, such arrays should not impact the Horn solver much: the steps used by Horn solvers based on counterexample-guided abstraction refinement using Craig interpolants can be reproduced with these arrays as the SMT solver handles the array theory.

However, we were unsure whether the generalization step currently handles such arrays correctly and we decided to fully eliminate arrays from our Horn clauses by using the Ackermannisation process described in Section 5.1.2.1. This process is only used to ensure that this is not the cause of struggle for the Horn solver, otherwise it should be avoided: Ackermannisation introduces a quadratic blowup in the number of array reads that an SMT avoids in most cases by only lazily considering the relevant cases[2].

Trivial simplification. The Horn clauses we generate from the data-abstraction tool are huge, mainly because our scheme is general and introduces trivial expressions. For example, the data-abstraction tool used with the identity abstraction introduces the expression $\forall i^\#, i^\# = i \rightarrow P^\#(i^\#)$ during the $abstract$ algorithm, which is then transformed into $i = i \rightarrow P^\#(i)$ by the $eliminate$ algorithm. To address this problem and transform that expression into $P^\#(i)$, we wrote a rewritting system that simplifies trivial expressions.

Reducing the size of the instantiation set. In the case of σ^n_{Cell}, for $n > 1$, the abstraction and instantiation is not optimal: consider the trivial clause $P(a) \rightarrow P'(a)$. This clause after σ^2_{Cell} abstraction and slight simplification becomes $(\forall k_1, k_2, P^\#(k_1, a[k_1], k_2, a[k_2])) \rightarrow P'^\#(k'_1, a[k'_1], k'_2, a[k'_2])$. Because the semantics of that clause is *pass to P' the information I have on P*, we would expect the clause after elimination to be: $P^\#(k'_1, a[k'_1], k'_2, a[k'_2]) \rightarrow P'^\#(k'_1, a[k'_1], k'_2, a[k'_2])$. However, our current scheme instantiates k_1 and k_2 by the relevant set $\{k'_1, k'_2\}$, thus yielding the clause $(P^\#(k'_1, a[k'_1], k'_1, a[k'_1]) \wedge P^\#(k'_2, a[k'_2], k'_1, a[k'_1]) \wedge P^\#(k'_1, a[k'_1], k'_2, a[k'_2]) \wedge P^\#(k'_2, a[k'_2], k'_2, a[k'_2]))$ $\rightarrow P'^\#(k'_1, a[k'_1], k'_2, a[k'_2])$, which is much more complex.

Our solution to this problem is to define $Cell^2$ not as $Cell^1 \otimes Cell^1$, but to introduce an ordering. We do this by introducing a new combinator $\sigma^{<n}$ which is the generalization of $\sigma^{<2}$ defined as $\sigma^{<2}(a) = \{(e_1, e_2) \mid e_1 \in \sigma(a) \wedge e_2 \in \sigma(a) \wedge e_1 < e_2\}$. This new combinator is just as expressive as σ^n and its goal is to reduce the size of the instantiation set. The instantiation set for that new combinator is computed by taking the order of the instantiation set for the $\sigma \otimes \sigma$ combinator: if we call S the instantiation set for σ, the instantiation set for $\sigma \otimes \sigma$ is $S \times S$, whereas for $\sigma^{<2}$, the instantiation set is $\{(e_1, e_2) \mid e_1 \in S \wedge e_2 \in S \wedge e_1 < e_2\}$. Of course, one cannot compute the ordering $e_1 < e_2$ during the instantiation process, and thus the ordering must be embedded as expressions: for $S = \{k'_1, k'_2\}$, this yields $\{(ite(k'_1 < k'_2, k'_1, k'_2), ite(k'_1 < k'_2, k'_2, k'_1))\}$ which contains only a single element.

[2] The goal of an SMT solver compared to a SAT solver is to remove the need for expansive preprocessing by lazily handling the theory.

Applying $Cell^{<2}$ abstraction on our previous example yields the clause $(P^{\#}(ite(k'_1 < k'_2, k'_1, k'_2), a[ite(k'_1 < k'_2, k'_1, k'_2)], ite(k'_1 < k'_2, k'_2, k'_1), a[ite(k'_1 < k'_2, k'_2, k'_1)]) \wedge k'_1 < k'_2) \rightarrow P'^{\#}(k'_1, a[k'_1], k'_2, a[k'_2])$ which with the simplifier can be simplified into $(P^{\#}(k'_1, a[k'_1], k'_2, a[k'_2]) \wedge k'_1 < k'_2) \rightarrow P'^{\#}(k'_1, a[k'_1], k'_2, a[k'_2])$.

Using $Cell^{<n}$ instead of $Cell^n$ reduces an instantiation set from size $|S|^n$ to the size $\binom{|S|}{n}$, where $|S|$ is the size of the relevant set – i.e. the size for $Cell^1$. However, it introduces more complex expressions which must then either be handled by a theory of ordering in the solver or through greedy preprocessing. In our case, we do the latter, but most of these expressions simplify themselves as in our example.

Results of improvement & discussion The results of these improvements are still disappointing: without hints, we are still unable to certify any of the benchmarks. However, results with these simplifications for hinted problems are slightly improved: the proportion of random seeds for which we certify the problem has slightly increased. This is not a great improvement and we do not suggest attempting to further simplify the generated Horn problems as this demonstrates that Z3 currently has drastic limitations that first need to be addressed.

Note that, during this process, we also noted that changing the names of our predicates in the Horn clauses impacts the solver. This is probably due to sorting on the predicate names at some point within the Horn solver which impacts the proofs that the calls to the SMT solver return, and thus the generalization step. These problems demonstrate how dependent on syntax Z3 is and how difficult it is to work with current Horn solvers and why having a theoretical guarantee on our technique is so important.

Techniques based on interpolation highly depend on heuristics to generate inductive invariants for Horn problems. We believe that the heuristics within Z3 have been optimized for Horn clauses coming from direct transformation from programs and perhaps a few other types of problems, but not for the type of Horn problems we generate after the data-abstraction tool. We have not studied the exact technique Z3 implements and it is extremely hard to determine whether adapting the current heuristic for our problems has not been done because it is hard or simply because there were no such problems. In either case, our work may lead to improvements within Horn solvers by providing examples on which they struggle.

Perhaps another interesting path for future work is to solve our Horn clauses by using abstract interpretation instead of interpolants. The problem with that approach is that the invariants we need are disjunctive, for example $k_a < i \rightarrow v_a = 0$, and to our knowledge there are no good disjunctive abstract domain for integers. However, in our examples, the variables used in each part of the conjunction may barely overlap – usually, the left-hand side of the implication is about indices whereas the right-hand side is about values– and thus, using such a specificity, perhaps a good abstract domain can be constructed. Such an abstract domain would have to handle relations, and thus, perhaps what we need is a form of disjunctive octogonal abstract domain, or something even more expressive [BM18]. This would not only enable to more reliably – especially if the widening is *perfect* – solve our Horn problems, but also generate summaries.

8 Conclusion

In this manuscript we tackled the problem of verifying programs with unbounded data-structures by using a transformation on Horn clauses. The goal of that transformation is to simplify using abstraction the data-structures within those Horn clauses so that another back-end solver can be used on the simplified problem. Our focus has been on the predictability of the transformation's results and this has led us to investigate a property called *relative completeness*, which is a measure of the information loss during the simplification. This is the main difference compared to most existing approaches [BMR13; GSV18; Ish+20].

Furthermore, instead of directly defining transformations for specific abstractions, we created a framework to create such transformations by combining basic abstractions together. We believe the advantages of such a framework have been demonstrated in several manners: in Chapter 3 we show that it enables more flexibility in the abstractions that are used and to discuss previous approaches [GRS05; CCL11] as combinations of simple abstractions; in Chapter 4 the framework enables us to give a general structure of the algorithms for any abstraction of data and also enables us to frame what *relative completeness* means for each abstraction; Chapter 5 demonstrates that the separation of the abstraction in several blocks enables us to focus on exactly what each abstraction must handle and by doing so, we prove results on previous abstraction-based techniques [MA15; MG16]; obviously, all chapters show that this approach enables future abstractions to be written and implemented much more straightforwardly.

The data-abstraction algorithm is parameterized by the chosen abstraction. In practice, this means that two parameters need to be provided: a formula F_σ encoding by what a concrete value should be abstracted and a heuristic $insts$ that helps the data-abstraction algorithm instantiate quantifiers for that abstraction. Any definition of $insts$ leads to a sound data-abstraction algorithm that may be used for static verification purposes; however, in order to have the predictability property of *relative completeness*, $insts$ needs to be well tailored as described in Chapter 4. Properly defining $insts$ such that it enables *relative completeness* is the main issue for using the data-abstraction framework. In this manuscript, our work focused on arrays and, more specifically, on abstractions that can be defined by combining the cell abstraction [MG16] of Chapter 3 with other basic abstractions of our framework.

Such abstractions are of interest for many container algorithms, including efficient sorting algorithms such as the merge sort algorithm of Chapter 2. In Chapter 5 we give what we believe to be a good definition of $insts$ for such array abstractions. However, this definition has limitations on the program instructions it may handle. In Chapter 6, we show that these limitations do not impact programs where functions calls are handled with inlining and written in a language where the basic array operations are read and write. Thus, on this class of programs, our simplification

satisfies *relative completeness*; and on other programs, the simplification only verifies soundness, that is, the property that is usually used in the field [BMR13; GSV18; Ish+20].

Furthermore, in Chapter 6, we discussed possible extensions so that *relative completeness* is achieved for a broader class of programs. We mainly expect these extensions to drastically improve the handling of complex array instructions and the handling of function calls using summaries. However, the handling of modular function calls, as defined in Chapter 2, seems much harder and may require more work. We believe that if the formalization of these extensions demonstrates that function calls using summaries are indeed well-handled, then not only do we have a framework for verifying container algorithms, but our verification scheme may scale and be applied to large code-bases.

Finally, in Chapter 7, we discussed our implementation of this framework and the results on container algorithms using a back-end solver. The implementation we propose for the data-abstraction simplification algorithm on Horn clauses is available, extensible, and can use a custom abstraction. We experimentally confirm using container algorithms that the simplification may indeed fully eliminate arrays from the Horn clauses and that it satisfies *relative completeness* while doing so. However, even if the resulting Horn clauses are only over booleans and integers, it seems that, Z3, the back-end solver we use, is unable to automatically find the required integer invariants. Thus, currently, this framework cannot be used impactfully to automatically verify programs; however, the issue is not within the framework, but within the capabilities of current state of the art integer Horn solvers.

We believe future work should go in multiple directions. First, in order for the data-abstraction framework to be of any practical use, improvements on integer Horn solvers for the type of clauses we generate is required. Secondly, we limited ourselves to abstractions that are defined mainly by using cell-abstraction. Although that abstraction is quite expressive, it is ill-suited for some array manipulating programs such as fold operations. We believe other abstractions for arrays need to be considered and hope that our work for cell-abstraction can help construct them. Thirdly, we discussed extensions of the framework for abstractions based on cell-abstraction. We explained most of these extensions in Chapter 6; however, they still need to be formalized and the *relative completeness* results still need to be proven. However, doing so has been made much easier through the general results proven for the data-abstraction framework. Furthermore, there are two unsolved questions in this manuscript: completeness of parallel instantiation of Conjecture 1, and whether the relevant algorithm is optimal, for the definition of optimality discussed in Section 6.3.1.1. Lastly, in this manuscript, we mainly focused on arrays. However, the goal of this framework is to be general enough to handle many kinds of unbounded data-structures, and we hope in the future to define abstractions for trees and perhaps even graphs.

Bibliography

[Ack57] W. Ackermann. “Solvable cases of the decision problem”. In: *Journal of Symbolic Logic* (1957).

[Bau+08] Patrick Baudin et al. “ACSL: Ansi c specification language”. In: *CEA-LIST, Saclay, France, Tech. Rep. v1* 2 (2008).

[Bey+09] Dirk Beyer et al. “Software model checking via large-block encoding”. In: *2009 Formal Methods in Computer-Aided Design*. IEEE. 2009, pp. 25–32.

[Bey12] Dirk Beyer. “Competition on software verification”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2012, pp. 504–524.

[BG20] Julien Braine and Laure Gonnord. “Proving array properties using data abstraction”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains*. 2020, pp. 17–21.

[BGM21] Julien Braine, Laure Gonnord, and David Monniaux. “Data Abstraction: A General Framework to Handle Program Verification of Data Structures”. In: *International Static Analysis Symposium*. Springer. 2021, pp. 215–235.

[Bjø+15] Nikolaj Bjørner et al. “Horn clause solvers for program verification”. In: *Fields of Logic and Computation II*. Springer, 2015, pp. 24–51.

[BM18] Alexey Bakhirkin and David Monniaux. “Extending Constraint-Only Representation of Polyhedra with Boolean Constraints”. In: *Static Analysis*. Ed. by Andreas Podelski. Cham: Springer International Publishing, 2018, pp. 127–145. ISBN: 978-3-319-99725-4.

[BMR13] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. “On solving universally quantified horn clauses”. In: *International Static Analysis Symposium*. Springer. 2013, pp. 105–125.

[BMS06] Aaron R Bradley, Zohar Manna, and Henny B Sipma. “What’s decidable about arrays?” In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2006, pp. 427–442.

[CC76] Patrick Cousot and Radhia Cousot. “Static determination of dynamic properties of programs”. In: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod. 1976, pp. 106–130.

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[CC79] Patrick Cousot and Radhia Cousot. “Systematic design of program analysis frameworks”. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1979, pp. 269–282.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. “A parametric segmentation functor for fully automatic and scalable array content analysis”. In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 105–118.

[CH78] Patrick Cousot and Nicolas Halbwachs. “Automatic discovery of linear restraints among variables of a program”. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1978, pp. 84–96.

[Cou+05] Patrick Cousot et al. “The ASTRÉE analyzer”. In: *European Symposium on Programming*. Springer. 2005, pp. 21–30.

[Cou90] Bruno Courcelle. “The monadic second-order logic of graphs. I. Recognizable sets of finite graphs”. In: *Information and computation* 85.1 (1990), pp. 12–75.

[Cuo+12] Pascal Cuoq et al. “Frama-C”. In: *Software Engineering and Formal Methods*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–247. ISBN: 978-3-642-33826-7.

[Cyt+91] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[Fij+19] Nathanaël Fijalkow et al. “On the Monniaux problem in abstract interpretation”. In: *International Static Analysis Symposium*. Springer. 2019, pp. 162–180.

[Fil+97] Jean-Christophe Filliâtre et al. *The Coq Proof Assistant - Reference Manual Version 6.1*. Tech. rep. 1997.

[FL11] Manuel Fähndrich and Francesco Logozzo. “Static Contract Checking with Abstract Interpretation”. In: *Formal Verification of Object-Oriented Software: International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*. Ed. by Bernhard Beckert and Claude Marché. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 10–30. ISBN: 978-3-642-18070-5. DOI: 10.1007/978-3-642-18070-5_2. URL: http://dx.doi.org/10.1007/978-3-642-18070-5_2.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3—where programs meet provers”. In: *European symposium on programming*. Springer. 2013, pp. 125–128.

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. “A framework for numeric analysis of array operations”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2005, pp. 338–350.

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

[Gur+15] Arie Gurfinkel et al. “The SeaHorn verification framework”. In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 343–361.

[Ish+20] Oren Ish-Shalom et al. “Putting the Squeeze on Array Programs: Loop Verification via Inductive Rank Reduction”. In: *VMCAI*. 2020.

[ISO17] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. Dec. 2017, p. 1605. URL: https://www.iso.org/standard/68564.html.

[Jou16] Jacques-Henri Jourdan. “Verasco: a Formally Verified C Static Analyzer”. Theses. Universite Paris Diderot-Paris VII, May 2016. URL: https://hal.archives-ouvertes.fr/tel-01327023.

[Kah+16] Temesghen Kahsai et al. “JayHorn: A framework for verifying Java programs”. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 352–358.

[LA04] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[Len83] Hendrik W Lenstra Jr. “Integer programming with a fixed number of variables”. In: *Mathematics of operations research* 8.4 (1983), pp. 538–548.

[Ler+21] Xavier Leroy et al. *The OCaml system release 4.13: Documentation and user's manual*. Intern report. HTML version available at: https://ocaml.org/manual/polymorphism.html. Inria, Sept. 2021. Chap. 6: Polymorphism and its limitations, pp. 87–97. URL: https://hal.inria.fr/hal-00930213.

[Ler20] Xavier Leroy. “The CompCert C verified compiler: Documentation and user's manual”. PhD thesis. Inria, 2020.

[MA15] David Monniaux and Francesco Alberti. “A simple abstraction of arrays and maps by program translation”. In: *International Static Analysis Symposium*. Springer. 2015, pp. 217–234.

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[MG16] David Monniaux and Laure Gonnord. “Cell morphing: From array programs to array-free horn clauses”. In: *International Static Analysis Symposium*. Springer. 2016, pp. 361–382.

[Mon19] David Monniaux. “On the decidability of the existence of polyhedral invariants in transition systems”. In: *Acta Informatica* 56.4 (2019), pp. 385–389.

[MS04] Markus Müller-Olm and Helmut Seidl. “A note on Karr's algorithm”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2004, pp. 1016–1028.

[NWP02] “5. The Rules of the Game”. In: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Ed. by Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 67–104. ISBN: 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9_5. URL: https://doi.org/10.1007/3-540-45949-9_5.

[OG86] Alex Orailoglu and Daniel D Gajski. “Flow graph representation”. In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. 1986, pp. 503–509.

[Rey02] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.

[Ric53] H. Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366.

[RU71] Nicholas Rescher and Alasdair Urquhart. “The Background of Temporal Logic”. In: *Temporal Logic*. Vienna: Springer Vienna, 1971, pp. 1–12. ISBN: 978-3-7091-7664-1. DOI: 10.1007/978-3-7091-7664-1_1. URL: https://doi.org/10.1007/978-3-7091-7664-1_1.

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

[Sch78] Thomas J. Schaefer. “The Complexity of Satisfiability Problems”. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC ’78. San Diego, California, USA: Association for Computing Machinery, 1978, pp. 216–226. ISBN: 9781450374378. DOI: 10.1145/800133.804350. URL: https://doi.org/10.1145/800133.804350.

[Tse83] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: https://doi.org/10.1007/978-3-642-81955-1_28.

[VW86] Moshe Y Vardi and Pierre Wolper. “An Automata-Theoretic Approach to Automatic Program Verification”. English. In: *Proceedings of the First Symposium on Logic in Computer Science*. Cambridge, United States - Massachusetts, 1986.