Verifying Programs with Arrays and Lists

Julien Braine, Laure Gonnord, David Monniaux

Verifying Programs with Arrays and Lists

Julien BRAINE

M2's internship with Laure Gonnord and David Monniaux. February-June 2016

Abstract

Automatically verifying safety properties of programs is a tough problem that has been tackled using many different approaches: rewriting systems, abstract interpretation, SMT solving, . . .

Most techniques restrict themselves to programs operating on boolean and integer values and transposing them to infinite data structures such as arrays has not yet been satisfyingly achieved.

Recent work in Monniaux and Gonnord [2016] suggests the use of abstract interpretation to transpose programs containing arrays into Horn clauses that do not contain arrays. The major innovation of their work is that they use Horn clauses which are more general than programs, to obtain better results.

In this work, we first set the work of Monniaux and Gonnord in a more general framework that allows us to extend their abstractions, simplify the expressions they generate, and analyze the precision of their abstraction.

Finally we extend their abstractions so that we can the analyze lists and experiments show that we succeed to analyze several classical examples, including sorting algorithms.

Contents

1 Introduction

Avoiding bugs when writing programs has always been tough for many reasons: programmers must think, for example, about integer overflows, unpredicted user input and memory management. Several methods exist to avoid bugs: good coding habits and proper documentation but only automated error checking can prove their absence.

The latter requires us to define what an error is. There are errors defined by the programming language such as out of bound array accesses, and there are errors due to the semantics of the program. The former errors can be directly checked, however, the latter requires the user to add specifications. In Example 1, the user adds specifications in the form of assertions ensuring the semantics and then the bug checking tool can add automatic assertions due to well-known errors.

Example 1 (Array Initialization in C++)

```
void init(std::vector<int>& a){
  int i,j;
  for(i=0;i<a.size();i++){
    assert(i >=0 && i<a.size());
    a[i] = 0;
  }
  j = std::rand();
  if(j >=0  && j<a.size()){
    assert(j >=0  && j<a.size());
    assert(a[j] == 0);
  }
}
```

⬛ *Initial code of the function*
🟦 *User added specifications*
🟥 *Automatically added assertions*

Once specifications have been given, there are several ways to detect bugs:

- The simplest technique consists in modifying the program such that, during its execution, when an assertion fails, its saves an execution trace and files an error report so that the bug can be removed. Although the end-user experiences the bug, this method is a good solution for programs that do not have high security or quality of service requirements.
- Another technique consists in running tests on the program, hoping that they will catch most bugs. The end user does not experience any issues when a bug is caught with this technique, but there is no guarantee that all bugs are caught.
- The technique we will be focusing on this work, called static analysis, does not suffer from those flaws: we run the program symbolically, covering all possible execution traces and checking that no assertion fails. The main issue is that there exists no program that can check that no possible execution fails: the problem is undecidable. Static analysis is most suited for high security programs such as in avionics and programmers might have to help the analyzer.

During the life cycle of a given software, a combination of these techniques is usually used, weighted by time development and security requirements: these techniques serve different purposes.

Static analysis is mainly used to ensure that there are no bugs in a program whereas other techniques are used to detect bugs within programs. In this work, we focus on the design of static analyzers that over-approximate the behaviors of programs, and prove their *safety*. They will either return that there is no bug in a program, or “don’t know”.

Within the perimeter of this work are two categories of techniques to analyze programs:

- Abstract interpretation (Cousot and Cousot [1977]) consists in abstracting (possibly infinite) sets of possible values by an abstract object (for instance intervals, polyhedra, . . .) and map

operations on programs to operations on the abstract object. The analysis then returns that there is a possible error when one of the values of the abstract object makes an assertion fail. The Astree tool (Cousot et al. [2005]) has successfully applied this technique to analyze many industrial size critical programs.
- Logic based solving consists in writing (an approximation of) the semantics of a program as a logic formula that is then solved by a SAT Modulo Theory solver(SMT) such as Z3[1], Z3-Spacer[2] and Eldarica[3].

These techniques have been extensively studied on programs containing bounded data (or numeric data) but have not been able to successfully analyze programs using unbounded data structures such as arrays, lists, trees or graphs. Building on Monniaux and Gonnord [2016], we convert analyses on unbounded data structures to analyses on bounded structures.

Technique We use abstract interpretation to convert logic formulas, called Horn problems, on unbounded data structures into Horn problems on bounded data structures, that can then be solved by state-of-the-art SMT solvers.

Contribution The contributions of this work are:

- A framework for abstract interpretation on Horn clauses. This framework is used to increase the generality of current abstractions, to construct new abstractions and simplify proofs.
- An explicit abstract interpretation for linear data structures such as lists.
- An implementation that succeeds to prove the correctness of some sorting algorithms.

Organization of report In Section 2, we convert programs into logic formulas called Horn clauses. Then, Section 3 explains the conversion of these Horn clauses into a common base language. In Section 4 we give the framework for abstract interpretation on Horn clauses, then used in Section 5 to give abstractions that convert Horn clauses on unbounded data into Horn clauses over bounded data. Finally, in Section 6, we describe our implementation and give the experimental results.

2 From programs to Horn problems

A Horn problem is a special subset of first order formula that can capture the semantics of a program. A Horn problem over a theory $\mathcal{T}$ is a formula that can be written in the following form.

$$\exists P_1, \dots P_n, \bigwedge_i \forall v_{i_1}, ..., v_{i_k} C_i$$

where :

- $P_1, \dots, P_n$ are existentially quantified functions, called predicates, from types of $\mathcal{T}$ to $Bool$.
- $v_{i_1}, \dots, v_{i_k}$ are variables with types in $\mathcal{T}$
- C_i is a clause in the form $cond_1 \wedge cond_2 \wedge ... \wedge cond_j \Rightarrow result$ and each condition and the result is either a predicate applied to variables or a formula in the theory of $\mathcal{T}$.

The simplest way to visualize Horn problems is through programs. A program can be transformed into a Horn problem that captures its semantics, that is to say into a Horn problem that is unsatisfiable if and only if there is an execution of the program such that an assertion fails. A possible

[1] https://github.com/Z3Prover

[2] https://bitbucket.org/spacer/code

[3] https://github.com/uuverifiers/eldarica/

transformation consists in three steps:

1. Retrieve the control flow graph(CFG) of the program, on Example 1 this gives Example 1.1.
2. Interpret a state of the CFG as a predicate indicating the possible variable values at that state.
3. Interpret each edge e of the CFG as a relation R between the values at the source of e and the target of e, creating the clause : $\forall vars, vars', Source(vars) \wedge R(vars, vars') \Rightarrow End(vars')$.

We give the result of this transformation applied to Example 1 in Example 1.2. As such, the Horn problem in Example 1.2 can not be run on a SMT solver: SMT solvers do not deal with the theory of a programming language. In the next sections, we convert Horn problems generated by the conversion from a CFG, to Horn problems that can be run on a SMT solver.

3 To language independent Horn problems

In practice, Horn solvers do not run on theories that express all program operations. The basic theory, we call *Basic*, on which Horn solvers run simply have types $Int = \mathbb{Z}$, $Real = \mathbb{R}$, $Bool = \{0, 1\}$ and tuple types (that is to say pairs, triples, ...) with usual arithmetical operations[4].

In this section, we convert Horn problems to a language independent theory that will be converted to the *Basic* theory later on.

3.1 Conversions: definition and issues

Definition 3.1 *A Horn problem conversion from a source theory to a target theory is a function that to a Horn problem over the source theory associates a Horn problem over the target theory.*

For a conversion *conv* to be useful, we require two properties:

- Soundness: $conv(H)\ satisfiable \Rightarrow H\ satisfiable$. Ensures that when an analysis returns that there are no bugs, then there are no bugs in the original program.
- Precision : $H\ satisfiable \rightsquigarrow conv(H)\ satisfiable$. Ensures the relevance of checking for a bug when an analysis returns that there is potentially a bug.

In the rest of this section, we do not simplify the theories on which Horn problems are, we only convert them to a language independent theory without any loss of precision: we use conversions called *complete*, that is to say conversions c such that $\forall H, c(H) \Leftrightarrow H$.

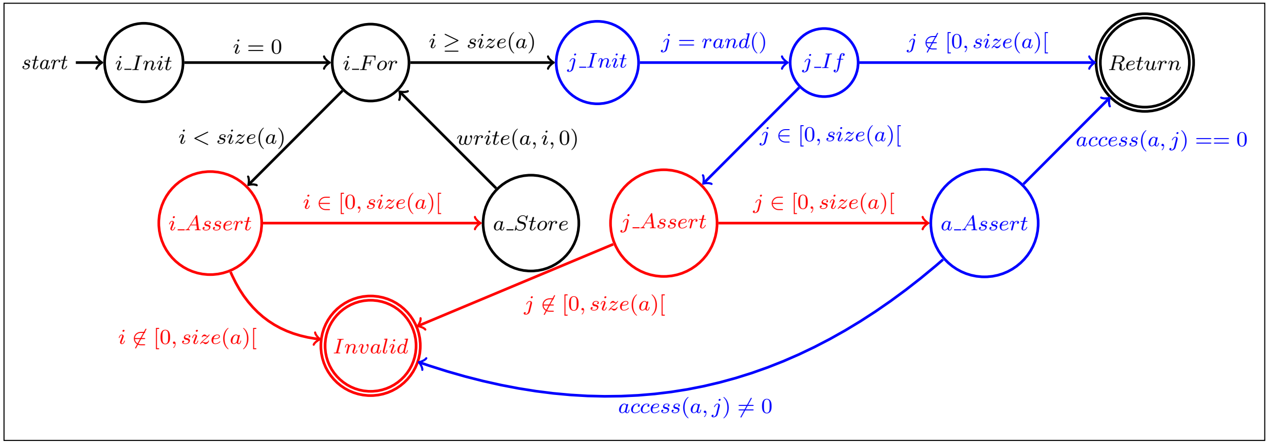
3.2 Conversions to a language independent theory

The goal of this section is to convert a Horn problem over a theory of a programming language into a Horn problem over a language independent theory we call *Fundamental*, containing the *Basic* theory and language independent unbounded data structures. However, doing so is a full research topic on its own [5] and we focus on giving the intuition of the conversions involved, the main contribution of this document starting once we have reached *Fundamental*. Example 1.3 illustrates the expected result of this section on Example 1.2.

[4] The *Basic* theory is the theory well dealt by all solvers, though many solvers have extensions.

[5] Some tools such as SeaHorn https://seahorn.github.io/ and FramaC http://frama-c.com/ provide support for developping such abstractions.

Example 1.1 (CFG of program depicted in Example 1)



Example 1.2 (Automated Basic Horn problem associated to CFG of Example 1.1)

$$Types : (((i, j, a), (i', j', a')) \in (int \times int \times Array < int >)^2$$

$$Formula : \exists i\_Init, i\_For, i\_Assert, a\_Store, j\_Init, j\_If, j\_Assert, a\_Assert, Return$$

$$\bigwedge \left\{ \begin{array}{lllll} \forall i, j, a, & true & & & \Longrightarrow i\_Init(i, j, a) \\ \forall i, i', j, j', a, a', & i\_Init(i, j, a) & \bigwedge & \left\{ \begin{array}{c} i' = 0 \\ j' = j \\ a' = a \end{array} \right\} & \Longrightarrow i\_For(i', j', a') \\ \forall i, j, a, & i\_Assert(i, j, a) & \bigwedge & (0 > i \vee i \geq size(a)) & \Longrightarrow false \\ \hdashline \forall i, i', j, j', a, a', & a\_Store(i, j, a) & \bigwedge & \left\{ \begin{array}{c} i' = i \\ j' = j \\ a' = write(a, i, 0) \end{array} \right\} & \Longrightarrow i\_For(i', j', a') \\ \hdashline \forall i, j, a, & a\_Assert(i, j, a) & \bigwedge & access(a, i) \neq 0 & \Longrightarrow false \end{array} \right.$$

Example 1.3 (Complete conversion from array to Function of Horn problem of Example 1.2)

We use simplified notation: quantifiers are assumed.

Types	$((i, j, (a_f, a_{size})), (i', j', (a'_f, a'_{size}))) \in (int \times int \times (int \times Function{<}Int, int{>}))^2$
$i\_Init \Rightarrow i\_For$	$i' = 0 \wedge j' = j \wedge (a'_f, a'_{size}) = (a_f, a_{size})$
$i\_Assert \Rightarrow false$	$0 > i \vee i \geq size(a)$
$a\_Store \Rightarrow i\_For$	$i' = i \wedge j' = j \wedge (a'_f, a'_{size}) = (store(a_f, i, 0), a_{size})$
$a\_Assert \Rightarrow false$	$select(a_f, i) \neq 0$

3.2.1 Converting non pointer types: bool, int, floats and primitive structures

bool **conversion** The conversion for *bool* is very simple as the Horn basic theory has the *Bool* type which is exactly equivalent, therefore, there is nothing to do.

int **conversion** *int* in most programming languages is bound, therefore there is no exact equivalent type in the basic Horn theory[6]: the *Int* Horn type is infinite. We convert *int* to *Int* and do each operation modulus INT_MAX.

$float$ **conversion** The Horn basic theory[7] does not have floating point numbers but contains reals. The Real type is suited for specifications that do not depend on rounding errors, but general specifications require another approach: use an exponent and a mantissa integer, and convert the operations on floating point numbers to operations on the couple (exponent, mantissa).[8]

primitive **structures** Primitive structures are equivalent to the tuple of their elements. We abstract a primitive structure into the tuple of its abstracted elements.

3.2.2 Converting unbounded structured data types: arrays, lists, trees, graphs

In most programs data types are bounded either by the available memory or some big number such as INT_MAX. However, abstracting data types bounded by such numbers by the tuple of its elements does not give an effective solution for performance reasons.

array<T> **conversion** We introduce the mathematical type *Function<Int, T>* in the *Fundamental* theory[9] and consider that an array is just a function and a size. We define two operations on *Function<Int, T>* to mimic the *access* and *write* operations on *array<T>*:

- $select(f, index)$ which returns $f(index)$.
- $store(f, index, value)$ which returns a function g identical to f except that $g(index) = value$.

Converting arrays to functions then consists in replacing :

- $a \in array{<}T{>}$ by $(size, f) \in int \times Function{<}Int, T{>}$
- $access(a, i)$ by $(size, select(f, i))$
- $write(a, i, v)$ by $(size, store(f, i, v))$
- $size(a)$ by $size$

This conversion applied to Example 1.2 gives the Horn problem in Example 1.3.

list<T> **conversion** Lists are almost semantically equivalent to arrays: they describe a linear data structure and the main difference is not in the semantics, but in the implementation which gives different operation complexity. The only operations that are semantically specific to lists are the *insert* and *erase* operations, and we need to introduce the equivalent operations on functions within the *Fundamental* theory.

Trees, DAGs, Graphs, . . . conversions Concerning trees and graphs or even other unbounded data structures such as DAGs, we keep the same idea: expand the *Fundamental* theory with the underlying infinite data structure and its operations and manage the finite aspect by adding size

[6] Some solvers have bit-vectors that can be used for bounded integers.

[7] Some solvers can deal with floating point arithmetic.

[8] This conversion is naive and causes solving time issues. Finding a good conversion is a full research topic.

[9] This has already been done in some solvers.

integers. In the case of trees for example, one would want to enlarge the theory with infinite depth and width trees and the operations $insert, erase, select$ and $store$ on those trees.

3.2.3 Pointers

We do not deal with unstructured data in this work. However, in many cases pointers are the implementation of semantically structured data dealt within this work. Some cases have solutions: argument passing as pointers (or references) can be dealt with the copy and return idiom, pointers allocated with malloc have the semantics of array, . . . Other cases, such as retrieving lists and trees represented by pointers, can be handled using shape analysis (Jones and Muchnick [1982]) and separation logic (Reynolds [2002]) for example. These concerns are not tackled in this work and can be researched upon and dealt with independently.

4 Constructing formal conversions

In the previous section we gave informal examples of conversions and although the abstractions behind those conversions are fairly intuitive, properly defining these conversions can be difficult. The goal of this section is to construct conversions from abstractions.

4.1 Abstractions

A general abstraction from a variable of type $Source$ to a type $Target$ is a Galois connection $\mathcal{G} : \mathcal{P}(Source) \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \mathcal{P}(Target)$ and a set of abstract operations (Cousot and Cousot [1977]). Intuitively $\mathcal{G}$ formalizes how states of the CFG representing possible values of $Source$ now represent possible values of $Target$. The abstract operations then correspond to defining the edges of the CFG as a relationship between values of $Target$ instead of values of $Source$.

In this work, we do not abstract sets of possible values of $Source$ by single values of $Target$ as done in polyhedral abstract interpretation Cousot and Halbwachs [1978]: simplifying sets of possible values is dealt by the SMT solver during predicate calculation. Instead, we abstract single values of complex types into possibly multiple values of simple types: we use one to many abstractions and the informal array to function conversion of Section 3.2.2 is defined in Abstraction 1.

Definition 4.1 *Let ϕ be a function from Source $\to \mathcal{P}(Target)$. ϕ induces a Galois connection $\mathcal{G} : \mathcal{P}(Source) \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \mathcal{P}(Target)$ called one to many abstraction associated to ϕ defined by:*

$$\begin{cases} \alpha(concrete \in \mathcal{P}(Source)) = \bigcup\limits_{c \,\in\, concrete} \phi(c) \\ \gamma(abstracted \in \mathcal{P}(Target)) = \{c \in Source, \phi(c) \subseteq abstracted\} \end{cases}$$

Abstraction 1 (One to many Galois connection for array to function conversion)

$\phi(a \in array\text{<}int\text{>}) = \{(\mathbf{size}, \mathbf{f}), size = size(a), \forall i \in [0, size(a)[, f(i) = a[i]\}$	
Concrete operation	*Abstract operation*
$access(a, i)$	$(\mathbf{size}, select(\mathbf{f}, i))$
$write(a, i, v)$	$(\mathbf{size}, store(\mathbf{f}, i, v))$
$size(a)$	$\mathbf{size}$

A common method replaces each operation $c_{op}(vars)$ by an operation $a_{op}(vars^{\#})$, $vars^{\#}$ representing an abstraction of $vars$, as has been done in Abstraction 1. In many cases described in Section 5.1, there is no precise enough abstract operation and most papers suggest abstracting edges of the form $P_1(vars) \wedge vars' = c_{op}(vars) \Rightarrow P_2(vars')$ instead of operations. This approach creates several unwanted consequences:

- Horn problems that have several predicates in their clauses conditions can not be transformed into clauses of the form $P_1(vars) \wedge vars' = c_{op}(vars) \Rightarrow P_2(vars')$. Although this is not an issue for Horn problems directly induced by programs, we can no longer compose conversions that take advantage of several predicates in the clause's conditions.
- The performance might be impacted when Horn problems are transformed, as edges with multiple composed operations such as $var = read(store(a, i, v), j)$ must be separated into several edges, adding temporary variables and predicates.

Furthermore, when a clause of the form $P_1(vars) \wedge vars' = c_{op}(vars) \Rightarrow P_2(vars')$, representing an edge is abstracted by a clause such as $P_1^{\#}(vars_1^{\#}) \wedge P_1^{\#}(vars_2^{\#}) \Rightarrow P_2^{\#}(vars'^{\#})$, as has been done in Monniaux and Gonnord [2016], several problem arise:

- The semantics of the abstractions are harder to grasp as an edge is abstracted into a clause that is not an edge, thus proving soundness is much harder.
- Local abstraction, operation abstraction on clause level quantified parameters, and global abstraction, the abstractions of predicates that are quantified at the top level, are mixed up. Understanding whether the precision loss is due to operation abstraction or ϕ is thus harder.
- Modularity is impacted: what if we only wish to abstract only some operations or predicates ?

The major contributions of this section are:

- An approach in which abstractions can be defined that does not suffer from the above issues.
- An explicit algorithm for conversions induced by abstractions.

4.2 New approach: a step by step conversion of Horn clauses

Our approach is defined for one to many abstractions ϕ. The key idea is to make an explicit use of ϕ, allowing us to separate the abstractions of predicates from the abstractions of operations. The major steps of the conversion are given through the pseudo-code given in Algorithm Convert[1], and will be developed later on.

Algorithm 1: Convert[1]: Conversion induced by abstraction algorithm

```
Input: HornProblem $H$, Variable $v$, Expr → Expr $AbsExpr$
foreach Clause $C \in H$ do AbsOp[2]$(C, AbsExpr, v)$;    // $v$ in each clause is limited to $\phi(v)$
foreach Predicate $P \in H$ do AddAbsPredicate[3]$(P)$;    // Abstract predicates $P^{\#}$ created
foreach Clause $C \in H$ do UseAbsPredicate[4]$(C)$;    // $P$ and $\phi$ within clause's removed
Finalize[5]() ;    // The Horn problem is over the target theory
```

We illustrate the new approach through Example 2 on which we execute each major step of the Algorithm Convert[1] with Abstraction 1 as parameter.

Example 2 (Example on which we demonstrate the new approach)

Functionality	*Increments cell i of an array a and checks that it has been incremented*
Types	$((i, j, a), (i', j', a')) \in (int \times int \times array\text{<}int\text{>})^2$
$Start \Rightarrow false$	$i \geq size(a)$ (*Clause* 1)
$Start \Rightarrow Increase$	$i < size(a) \wedge i' = i \wedge j' = read(a, i) \wedge a' = a$ (*Clause* 2)
$Increase \Rightarrow Check$	$i' = i \wedge j' = j \wedge a' = write(a, i, read(a, i) + 1)$ (*Clause* 3)
$Check \Rightarrow false$	$j + 1 \neq read(a, i)$ (*Clause* 4)

4.2.1 AbsOp[2] procedure

To improve expressiveness of operation abstraction, we abstract expressions of the form $tmp = c_{op}(args)$ instead of operations $c_{op}(args)$, giving us the added expressiveness of relations. Although the idea is similar to abstracting a whole edge $P_1(vars) \wedge vars' = c_{op}(vars) \Rightarrow P_2(vars')$, there are two main differences in our abstract expressions:

- We only abstract $tmp = c_{op}(args)$ and not an edge, that is to say that the abstract expression must only involve tmp and $args$.
- We explicitly use ϕ in the abstractions as shown in Example 2.1, allowing us to keep the concrete predicates and abstract them later on.

We do not involve predicates in the transformation and we avoid the many disadvantages described in Section 4.1 of directly abstracting edges. The expressiveness of abstracting expressions in such a way is equivalent to the expressiveness of edge abstraction[10] and the main issue is finding a way to abstract predicates and remove the uses of ϕ. The formal abstract expressions induced by Abstraction 1 are given in Example 2.1.

Example 2.1 (Abstract expressions of array to function conversion)

Concrete expression	*Abstract expression*
$var = write(a, i, v)$	$\exists (var_{size}, var_f), (a_{size}, a_f), \wedge \begin{cases} (var_{size}, var_f) \in \phi(var) \wedge (a_{size}, a_f) \in \phi(a) \\ (var_{size}, var_f) = (a_{size}, store(a_f, i, v)) \end{cases}$
$var = read(a, i)$	$\exists (a_{size}, a_f), (a_{size}, a_f) \in \phi(a) \wedge var = select(a_f, i)$
$var = size(a)$	$\exists (a_{size}, a_f), (a_{size}, a_f) \in \phi(a) \wedge var = a_{size}$

The main idea in Algorithm AbsOp[2] is to `Replace` any expression of the form $tmp = c_{op}(args)$ by its abstract expression. To do so, we first need to `Normalize` the clause to decompose expressions into several expressions of the form $tmp = c_{op}(args)$ and `Rearrange` the clause into standard form after having replaced the expressions. In fact, this last step requires that the only quantifiers within the abstract expressions are existential quantifiers placed at the beginning [11]. Furthermore, in order for Algorithm Convert[1] to succeed, we require that expressions of the form $var^{\#} \in \phi(var)$ only appear positively in abstract expressions, but we have not met cases where this is an issue.

Algorithm 2: AbsOp[2]: abstracting expressions

```
Input: Clause $C$, Expr $\rightarrow$ Expr $AbsExpr$
Normalize($C$);                      // All expressions are either predicates or $tmp = c_{op}(args)$
foreach Expression $e \in C$ do Replace($e$, $AbsExpr(e)$);   // All expressions are abstracted
Rearrange($C$);                                             // The clause is in Horn standard format
```

Each step of Algorithm AbsOp[2] applied on Clause 3 is given in Example 2.2.

Soundness and precision analysis The normalization and the rearranging steps are complete conversions, therefore the soundness and the precision of the global conversion are equivalent to the soundness and completeness of the replace conversion. The replace conversion is sound (resp. complete) if and only if the abstract expression implies (resp. is equivalent to) the concrete expression.

[10] We can generate any Horn problem that was generated through edge conversion after all the steps of the conversion.

[11] This ensures that the problem remains a Horn problem.

Example 2.2 (Steps of Algorithm AbsOp[2] on Clause 3)

Edge	$Increase \Rightarrow Check$
Input	$i' = i \wedge j' = j \wedge a' = write(a, i, read(a, i) + 1)$
Normalized	$i' = i \wedge j' = j \wedge tmp_1 = read(a, i) \wedge tmp_2 = tmp_1 + 1 \wedge a' = write(a, i, tmp_2)$
Replaced	$\wedge \begin{cases} i' = i \wedge j' = j \wedge tmp_2 = tmp_1 + 1 \\ \exists (a_{size}, a_f) \in \phi(a), tmp_1 = select(a_f, i) \\ \exists (a_{size}, a_f) \in \phi(a), (a'_{size}, a'_f) \in \phi(a'), (a'_{size}, a'_f) = (a_{size}, store(a_f, i, tmp_2)) \end{cases}$
Rearranged	$\wedge \begin{cases} (a_{size}, a_f) \in \phi(a) \wedge (a'_{size}, a'_f) \in \phi(a') \wedge i' = i \wedge tmp_2 = tmp_1 + 1 \\ j' = j \wedge tmp_1 = select(a_f, i) \wedge (a'_{size}, a'_f) = (a_{size}, store(a_f, i, tmp2)) \end{cases}$ (*Clause* 5)

4.2.2 AddAbsPredicate[3] procedure

Predicates are the existentially quantified functions expressing the set of possible values at a program state, when using Horn clauses induced by programs. When using an abstraction, we abstract a concrete predicate representing a set of possible concrete values S_1 at a program state by predicate representing a set of possible abstract values $S_2 = \alpha(S_1) = \bigcup_{v \in S_1} \phi(v)$.

For a predicate P, we CreatePredicate an abstract predicate $P^\#$ such that (1) expressing that $P^\#$ is an abstraction of P is verified.

$$\forall args, v, (v^\# \in \phi(v) \Rightarrow (P^\#(args, v^\#) \Leftrightarrow P(args, v))) \tag{1}$$

In fact, (1) can be rewritten as a conjunction of two clauses that we AddClause directly in the Horn problem giving the Algorithm AddAbsPredicate[3].

```
Algorithm 3: AddAbsPredicate[3]: adding abstract predicates
Input: Predicate $P$, HornProblem $H$, Type $AbstractType$
CreatePredicate($H, P^\#, AbstractType$);                                   // Predicate $P^\#$ created
Let $C_{P_\alpha} = \forall arg, v, v^\#, v^\# \in \phi(v) \wedge P(arg, v) \Rightarrow P^\#(arg, v^\#)$ ;  // Condition forcing $\alpha(P) \subseteq P^\#$
Let $C_{P_\gamma} = \forall arg, v, v^\#, v^\# \in \phi(v) \wedge P^\#(arg, v^\#) \Rightarrow P(arg, v)$ ;  // Condition forcing $P \supseteq \gamma(P^\#)$
AddClause($H, C_{P_\alpha}$) AddClause($H, C_{P_\gamma}$)
```

Applying AddAbsPredicate[3] on the *Increase* predicate of Example 2 gives Example 2.3.

Example 2.3 (Added clauses by AddAbsPredicate[3] on *Increase* of Example 2)

Clause $C_{Increase_\alpha}$:	$(a_{size}, a_f) \in \phi(a) \wedge Increase(i, j, a) \Rightarrow Increase^\#(i, j, a_{size}, a_f)$
Clause $C_{Increase_\gamma}$:	$(a_{size}, a_f) \in \phi(a) \wedge Increase^\#(i, j, a_{size}, a_f) \Rightarrow Increase(i, j, a)$

Soundness and precision analysis The conversion is sound as the original problem is contained within the new problem. The conversion is complete if and only if $\gamma \circ \alpha = Id$. In practice, we only lose precision if P is a solution to the original problem and $\gamma(\alpha(P))$ is not.

4.2.3 UseAbsPredicate[4] procedure

Once abstract predicates have been added, the goal is to replace concrete predicates by abstract predicates, first in the result of each clause, then in the conditions of each clause. The first step consists in using (1) to define the complete conversion: Replace, in the result of a clause, $P(args, v)$ by $P^\#(args, v^\#)$ when $v^\# \in \phi(v)$ is in the conditions.

The second step follows the same idea and the first attempt for the second step conversion gives: replace, in the condition of a clause, $P(args, v) \wedge v^{\#} \in \phi(v)$ by $P^{\#}(args, v^{\#})$. We extend the idea for any number of predicates and uses of ϕ. For n predicates and k uses of ϕ, the conversion consists in replacing (2) by (3).

$$\left(\bigwedge_{i \in [0,n]} P_i(args_i, v) \right) \wedge \left(\bigwedge_{j \in [0,k]} v_k^{\#} \in \phi(v) \right) \tag{2}$$

$$\bigwedge_{i \in [1,n], j \in [0,k]} P_i^{\#}(args_i, v_j^{\#}) \tag{3}$$

Although this conversion is sound, it is not complete: we do not have (2) $\Leftrightarrow$ (3). In fact, we have (2) $\Leftrightarrow$ ((3) $\wedge$ $AreAbstractions_k(v_1^{\#}, \ldots, v_k^{\#})$) with $AreAbstractions_k(v_1^{\#}, \ldots, v_k^{\#})$) equivalent to (4)

$$\exists v, \forall j \in [0, k], v_j^{\#} \in \phi(v) \tag{4}$$

Therefore, the second step conversion consists in replacing (2) by (3)$\wedge AreAbstractions_k(v_1^{\#}, \ldots, v_k^{\#})$). We combine the first and second step, we get Algorithm `UseAbsPredicate[4]`.

Algorithm 4: `UseAbsPredicate[4]`: replacing concrete predicates by abstract predicates

```
Input: Clause $C$, $Int(k) \rightarrow (Variable^k) \rightarrow Expr$ $AreAbstractions$
Let $\forall params, conditions(params) \wedge v^{\#} \in \phi(v) \Rightarrow P(args, v) = C$ ;          // Decomposition of $C$
Replace($C, \forall params, conditions(params) \wedge v^{\#} \in \phi(v) \Rightarrow P^{\#}(args, v^{\#})$) ;     // First step done
foreach Predicate $P_i$ such that $P(args_i, v) \in C$ do
    foreach Variable $v_j^{\#}$ such that $(v_j^{\#} \in \phi(v)) \in C$ do AddCondition($C, P_i^{\#}(args_i, v_j^{\#})$) ;
    RemoveCondition($C, P_i(args_i, v)$) ;                     // $P_i(args_i, v)$ replaced by $\wedge_j P_i^{\#}(args_i, v_j^{\#})$
end                                                            // All $P_i$ are abstracted
Replace($v_1^{\#} \in \phi(v) \wedge \ldots \wedge v_k^{\#} \in \phi(v), AreAbstraction(k)(v_1^{\#}, \ldots, v_k^{\#})$) ;          // $\phi$'s removed
```

However, if $AreAbstractions$ is written as in (4), we introduce a concrete variable as well as many uses of ϕ. We wish to write $AreAbstractions$ without using any concrete variables. In all the abstractions we have seen so far, we have been able to define $AreAbstractions_k$ for any needed k without using any other variables than $v_1^{\#}, \ldots, v_k^{\#}$. To give some insight of how simple $AreAbstractions$ can be, we give $AreAbstractions$ in Example 2.4 for Abstraction 1.

Example 2.4 ($AreAbstractions$ for array to function conversion)

$AreAbstractions_1(size, f)$	$size \geq 0$
$AreAbstractions_k((size_1, f_1), \ldots, (size_k, f_k))$	$\bigwedge_{j \in [1,k]} size_j \geq 0 \wedge \forall i, f_1(i) = f_2(i) = \cdots = f_k(i)$
Remark: for $k > 1$, $AreAbstractions_k$ is not a valid Horn condition. Fortunately, there is no need for $k > 1$ with the abstract expressions defined in Example 2.1 as they use at most one existentially quantified abstraction of a given concrete variable.	

We apply `UseAbsPredicate[4]` on Clause 5 of Example 2.2 to get Example 2.5.

Soundness and precision analysis The conversion is sound (respectively complete) if and only if (1) is implied by the Horn problem and $AreAbstraction_k(v_1^{\#}, \ldots, v_k^{\#})$ is implied by (respectively equivalent to) (4).

Example 2.5 (Result of `UseAbsPredicate[4]` on Clause 5)

$$\boxed{Increase^{\#}(i,j,a_{size},a_f) \wedge \left\{\begin{array}{l} a_{size} \geq 0 \wedge a'_{size} \geq 0 \wedge tmp_2 = tmp_1 + 1 \\ i' = i \wedge j' = j \wedge tmp_1 = select(a_f, i) \\ (a'_{size}, a'_f) = (a_{size}, store(a_f, i, tmp_2)) \end{array}\right\} \Rightarrow Check^{\#}(i',j',a'_{size},a'_f)}$$

4.2.4 `Finalize[5]` procedure

At this point, there should be no uses of concrete variables or any concrete predicates, but in clauses equivalent to (1). Therefore the Algorithm `Finalize[5]` simply consists in, for all concrete predicate P, `RemoveClause` clauses equivalent to (1) and then `RemoveUnusedPredicates` and `RemoveUnusedVariables` from the quantifier lists, thus removing all concrete predicates and variables.

Algorithm 5: `Finalize[5]`: Wipe concrete variables and predicates out

```
Input: HornProblem $H$
foreach Predicate $P \in H$ do
    Let $C_{P_\alpha} = \forall arg, v, v^{\#}, v^{\#} \in \phi(v) \wedge P(arg, v) \Rightarrow P^{\#}(arg, v^{\#})$;
    Let $C_{P_\gamma} = \forall arg, v, v^{\#}, v^{\#} \in \phi(v) \wedge P^{\#}(arg, v^{\#}) \Rightarrow P(arg, v)$;
    RemoveClause($H, C_{P_\alpha}$) RemoveClause($H, C_{P_\gamma}$) ;          // (1) removed from $H$
end                                                      // No more concrete predicates used
RemoveUnusedPredicates($H$) ;                                 // Concrete predicates removed
foreach Clause $C \in H$ do RemoveUnusedVariables($C$);         // Concrete variables removed
```

Algorithm `Finalize[5]` is the last step of the `Convert[1]` and we the final result of Example 2, is given in Example 2.6.

Soundness and precision analysis The conversion is sound if P is only used in clauses implied by (1): assume the result of the conversion is satisfiable, then the initial problem is satisfiable by taking $P = \gamma(P^{\#})$. The conversion is complete as we do not lose precision by removing clauses.

4.2.5 Conclusion

Although there are many steps when using this abstraction technique, the final result is usually close to what one would expect from abstract interpretation techniques. In Example 2.6, we give the result of Algorithm `Convert[1]` applied on Example 2.

Example 2.6 (Result of array abstraction on Example 2 using Algorithm `Convert[1]`)

$$\begin{array}{|llr|}
\hline
\textit{Types:} & \begin{array}{l} ((i,j,\mathbf{a_{size}},\mathbf{a_f}),(i',j',\mathbf{a'_{size}},\mathbf{a'_f})) \in (int \times int \times int \times Function{<}Int, int{>})^2 \\ (tmp, tmp_1, tmp_2) \in int \times int \times int \end{array} & \\
\hline
Start^{\#}(i,j,\mathbf{a_{size}},\mathbf{a_f}) & \wedge \mathbf{a_{size}} \geq 0 \wedge tmp = \mathbf{a_{size}} \wedge i \geq tmp & \Rightarrow false \\
Start^{\#}(i,j,\mathbf{a_{size}},\mathbf{a_f}) & \wedge \left\{\begin{array}{l} \mathbf{a_{size}} \geq 0 \wedge \mathbf{a'_{size}} \geq 0 \\ tmp = \mathbf{a_{size}} \wedge i < tmp \wedge i' = i \\ j' = select(\mathbf{a_f}, i) \wedge (\mathbf{a'_{size}}, \mathbf{a'_f}) = (\mathbf{a_{size}}, \mathbf{a_f}) \end{array}\right\} & \Rightarrow Increase^{\#}(i',j',\mathbf{a'_{size}},\mathbf{a'_f}) \\
Increase^{\#}(i,j,\mathbf{a_{size}},\mathbf{a_f}) & \wedge \left\{\begin{array}{l} \mathbf{a_{size}} \geq 0 \wedge \mathbf{a'_{size}} \geq 0 \wedge i' = i \wedge j' = j \\ tmp_1 = select(\mathbf{a_f}, i) \wedge tmp_2 = tmp_1 + 1 \\ (\mathbf{a'_{size}}, \mathbf{a'_f}) = (\mathbf{a_{size}}, store(\mathbf{a_f}, i, tmp_2)) \end{array}\right\} & \Rightarrow Check^{\#}(i',j',\mathbf{a'_{size}},\mathbf{a'_f}) \\
Check^{\#}(i,j,\mathbf{a_{size}},\mathbf{a_f}) & \wedge \mathbf{a_{size}} \geq 0 \wedge tmp = select(\mathbf{a_f}, i) \wedge j + 1 \neq tmp & \Rightarrow false \\
\hline
\end{array}$$

Although one of the advantages of this abstraction technique is that each conversion is independent, allowing more flexibility when needed, most uses of our framework only require defining three elements to abstract $Source$ by $Target$:

1. A function ϕ from $Source$ to $\mathcal{P}(Target)$.
2. Abstract expressions $abs_{expr}(c_{op}, tmp, args)$ for expressions of the form $tmp = c_{op}(args)$, using ϕ such that $tmp = c_{op}(args) \Rightarrow abs_{expr}(c_{op}, tmp, args)$.
3. Expressions $AreAbstractions_k$ such that (4) $\Rightarrow AreAbstraction_k(v_1^{\#}, ..., v_k^{\#})$.

The precision analysis our framework entails is divided into three parts:

1. The precision of the Galois connection: how close is $\gamma \circ \alpha$ from Id.
2. The precision of abstract expressions: how close is $abs_{expr}(c_{op}, tmp, args)$ from $tmp = c_{op}(args)$.
3. The precision with which we express the image of ϕ: how close is $AreAbstraction$ from (4).

Inducing one of the main advantage of this framework, a three step reflection:

1. Can I have a more expressive target type ? In the case of infinite data structures to finite data structure abstractions, we will never have $\gamma \circ \alpha = Id$. However, one can usually improve precision by making the finite data structure bigger[12].
2. Can I improve the precision of my abstract expressions ? In most cases we encountered so far, abstract expressions can be complete.
3. Can I improve the precision of $AreAbstractions$? In all cases we encountered so far, we can define a complete $AreAbstractions_k$ for any needed k.

In the next Section, we define abstractions to convert problems over unbounded data into problems over bounded data, and we will keep in mind this three step reflection when constructing the conversions.

5 To the theory of Horn solvers

Section 3.2 has allowed us to transform a Horn problem over a theory of a programming language into a Horn problem over the $Fundamental$ theory that contains the $Basic$ theory, functions and possibly many other infinite mathematical structures.

In this section, we focus on abstracting types that belong to the $Fundamental$ theory into types of the $Basic$ theory. These abstractions are incomplete as we abstract infinite data by finite data and we use the technique in Section 4.2 to give precise abstractions.

5.1 Related work: array abstraction

There are several defined abstractions to prove specifications on structured unbounded data and the main focus has been on arrays. In Section 3.2 we transformed arrays to functions with *store* and *select* operations, and in this section, we adapt previous work to our context: we abstract functions and we formalize abstractions within the framework presented in Section 4.2. Doing so increases the generality of previous work as they will work on all Horn problems, and gives us a better precision analysis.

[12] This is done repeatedly in Section 5.1, either by using more slices or more distinguished cells

5.1.1 Cell coalescing

Cell coalescing consists in viewing a function as the set of its values. Intuitively, the store operation just adds the stored value to the set and the select operation just returns the set, giving the Abstraction 2.

Abstraction 2 (Cell coalescing abstraction of functions)

<table>
<tr><td colspan="2">$\phi(f \in Function\text{<}Ind, Val\text{>}) = \{y \in Val \vert \exists x, y = f(x)\}$</td></tr>
<tr><td><i>Concrete expression</i></td><td><i>Abstract expression</i></td></tr>
<tr><td>$g = store(f, i, v)$</td><td>$\exists y_g \in \phi(g), (y_g = v \vee y_g \in \phi(f))$</td></tr>
<tr><td>$val = select(f, i)$</td><td>$val \in \phi(f)$</td></tr>
<tr><td colspan="2">$AreAbstractions_k(y_1, \dots, y_k) = true$</td></tr>
</table>

Cell coalescing is a very imprecise abstraction: the abstract domain does not depend on any indexes whereas the operations do. In practice cell coalescing is very limited as one can not prove cell initialization: $g = store(f, i, v); assert(select(g, i) == v)$; fails.

5.1.2 Slices

There are several definitions of slice analysis for functions, but the basic idea is to split the index domain of the function into several segments and apply cell coalescing within each of those sets.

The method to determine the number of sets used and the bounds of each segment vary on the analysis, but the goal on loops is usually to separate indexes that have already gone through the loop and indexes that have not yet. Taking an example where the domain is split into in three segments $]-\infty, j[, \{j\},]j, \infty[$, with j a variable, gives Abstraction 3.

Abstraction 3 (Slice abstraction of functions)

<table>
<tr><td colspan="2">$\phi(f \in Function\text{<}Int, Val\text{>}) = \{(y_1, y_2, y_3) \vert y_1 \in f(]-\infty, j[) \wedge y_2 = f(j) \wedge y_3 \in f(]j, \infty[)\}$</td></tr>
<tr><td><i>Concrete expression</i></td><td><i>Abstract expression</i></td></tr>
<tr><td>$g = store(f, i, v)$</td><td>$\exists (y_1^f, y_2^f, y_3^f) \in \phi(f), (y_1^g, y_2^g, y_3^g) \in \phi(g)$
$\wedge \left\{ \begin{array}{l} i < j \Rightarrow ((y_1^g, y_2^g, y_3^g) = (y_1^f, y_2^f, y_3^f) \vee (y_1^g, y_2^g, y_3^g) = (v, y_2^f, y_3^f)) \\ i = j \Rightarrow (y_1^g, y_2^g, y_3^g) = (v, y_2^f, y_3^f) \\ i > j \Rightarrow ((y_1^g, y_2^g, y_3^g) = (y_1^f, y_2^f, y_3^f) \vee (y_1^g, y_2^g, y_3^g) = (y_1^f, y_2^f, v)) \end{array} \right.$</td></tr>
<tr><td>$val = select(f, i)$</td><td>$\exists (y_1^f, y_2^f, y_3^f) \in \phi(f)$
$\wedge \left\{ \begin{array}{l} i < j \Rightarrow val = y_1^f \\ i = j \Rightarrow val = y_2^f \\ i > j \Rightarrow val = y_3^f \end{array} \right.$</td></tr>
<tr><td colspan="2">$AreAbstractions_k((y_1^1, y_2^1, y_3^1), \dots, (y_1^k, y_2^k, y_3^k)) = y_2^1 = ... = y_2^k$</td></tr>
</table>

Slice abstraction is extremely dependent on how the domain is split into segments: if the current variable read or written to is not is a singleton segment, then cell coalescing is applied and we have the same problems. In our framework, we get that the select operation and the store operation are only complete when $i = j$. Therefore, our framework naturally entails that dividing the slices should be done according to the variable being read or written on.

Furthermore, even if some variants of slice abstraction have relations between slices, a sorting invariant would require as many slices as there are cells in the array, which is not a viable option.

5.1.3 Cell abstraction

Monniaux and Alberti [2015] made a program to program transformation abstracting a function f to its set of cells $(x, f(x))$. Intuitively, the $store(f, i, v)$ operation replaces the couple $(i, f(i))$ by (i, v) and $select(f, i)$ returns a value val such that the couple (i, val) is in the cells of f, giving Abstraction 4 with abstract *select* Expression 1.

However, the *select* abstract expression is incomplete and the transformation fails to prove simple programs such as array initialization with loop check[13] does not contain any bugs. In Monniaux and Gonnord [2016], using the added expressiveness of Horn clauses, the *select* abstract expression is replaced by Expression 2, making the *select* expression complete.

Abstraction 4 ("One distinguished cell abstraction" of functions)

$\phi(f \in Function{<}Int, Val{>}) = \{(x,y) \mid y = f(x)\}$	
Concrete expression	*Abstract expression*
$g = store(f, i, v)$	$\exists (x_g, y_g) \in \phi(g) \wedge \begin{cases} i = x_g \Rightarrow y_g = v \\ i \neq x_g \Rightarrow (x_g, y_g) \in \phi(f) \end{cases}$
$val = select(f, i)$	$\exists (x_f, y_f) \in \phi(f) \wedge \begin{cases} i = x_f \Rightarrow val = y_f \\ i \neq x_f \Rightarrow y = \top \end{cases}$ $(Expression\ 1)$ $(i, val) \in \phi(f)$ $(Expression\ 2)$
$AreAbstractions_k((x_1, y_1), \ldots, (x_k, y_k)) = \bigwedge_{i,j} x_i = x_j \Rightarrow y_i = y_j$	

The abstraction given in Monniaux and Gonnord [2016] has complete abstract expressions and complete *AreAbstractions*. Therefore, precision loss is only due to ϕ, that is to say that abstract predicates must only depend on one couple $(x, f(x))$, instead of f which represents all couples $(x, f(x))$. This allows us to express complex states such as $f(0) = 0 \wedge \forall x \neq 0, f(x) = x^2 + 1$ by setting $P : (x, y) \rightarrow (x = 0 \wedge y = 0) \vee (x \neq 0) \wedge y = x^2 + 1$. However, the dependence of predicates on only one couple $(x, f(x))$ does not allow the expressiveness of states such as f is increasing (sortedness of arrays).

Monniaux and Gonnord [2016] use the principle of many distinguished cells to solve the problem. The idea is fairly simple: enlarge the abstract domain to several couples $(x, f(x))$ so that predicates can depend on relations between cells. For example, two distinguished cells can express that f is increasing: $P : ((x^1, y^1), (x^2, y^2)) \rightarrow x^1 < x^2 \Rightarrow y^1 \leq y^2$. A small optimization that Monniaux and Gonnord [2016] suggest is to order the cells to avoid repetition. Abstraction 5 describes the technique for two distinguished cells.

The main expressiveness limit of distinguished cell abstraction is program states that depend on an unbounded number of cells. For example, the program state "The sum of all the cells of the function is equal to 42" depends on all the cells of the function and the abstract predicate will always be false. This seems to be a general limit of Horn clauses on bounded data and we believe there is no abstraction with better expressiveness.

[13] This is Example 1 with a loop within the user specifications instead of a rand().

Abstraction 5 ("Two distinguished cell abstraction" of functions)

<table>
<tr><td colspan="2">$\phi(f \in \mathit{Function}{<}\mathit{Int}, \mathit{Val}{>}) = \{((x_1, y_1), (x_2, y_2)), x_1 < x_2 \wedge y_1 = f(x_1) \wedge y_2 = f(x_2)\}$</td></tr>
<tr><td><i>Concrete expression</i></td><td><i>Abstract expression</i></td></tr>
<tr><td>$g = \mathit{store}(f, i, v)$</td><td>$\exists((x_g^1, y_g^1), (x_g^2, y_g^2)) \in \phi(g) \wedge \begin{cases} i > x_g^2 \Rightarrow ((x_g^1, y_g^1), (x_g^2, y_g^2)) \in \phi(f) \\ i = x_g^2 \Rightarrow \exists x_f, y_f, ((x_g^1, y_g^1), (x_f, y_f)) \in \phi(f) \wedge y_g^2 = v \\ x_g^2 > i > x_g^1 \Rightarrow ((x_g^1, y_g^1), (x_g^2, y_g^2)) \in \phi(f) \\ i = x_g^1 \Rightarrow \exists x_f, y_f, ((x_g^2, y_g^2), (x_f, y_f)) \in \phi(f) \wedge y_g^1 = v \\ x_g^1 > i \Rightarrow ((x_g^1, y_g^1), (x_g^2, y_g^2)) \in \phi(f) \end{cases}$</td></tr>
<tr><td>$\mathit{val} = \mathit{select}(f, i)$</td><td>$\exists(x_f, y_f),\ ((i, \mathit{val}), (x_f, y_f)) \in \phi(f)$</td></tr>
<tr><td colspan="2">$\mathit{AreAbstractions}_k(((x_1^1, y_1^1), (x_1^2, y_1^2)), \ldots, ((x_k^1, y_k^1), (x_k^2, y_k^2))) = \bigwedge_i x_i^1 < x_i^2 \bigwedge_{i,j,b_1,b_2} x_i^{b_1} = x_j^{b_2} \Rightarrow y_i^{b_1} = y_j^{b_2}$</td></tr>
</table>

The main contributions of the work in Section 4.2 to these abstractions are:

- Simplified abstract expressions.
- A generalization to all Horn clauses.
- A proof that with the given ϕ, there can be no more improvement.

5.2 Contributions: new abstractions

Monniaux and Gonnord [2016] gave us what seems like the best abstractions for arrays, that is to say functions with operations *store* and *select* once in the *Fundamental* theory. Thus, our focus has been on abstracting other structures from the *Fundamental* theory into the *Basic* theory.

One of the main advantage of the *Fundamental* theory is that we only need to abstract infinite structures whose skeleton does not change: inserting and removing data does not change the skeleton of an infinite list or of an infinite tree: we only need to change the index to which the values correspond. This remark radically changes the point of view of the abstractions we will define: instead of expressing an insertion or a removal of data, we only update the indexes.

5.2.1 Abstracting infinite lists

In fact, within the *Fundamental* theory, $\mathit{list}{<}\mathit{Val}{>}$ has become a pair $(\mathit{int}, \mathit{Function}{<}\mathit{Int}, \mathit{Val}{>})$ representing the size and the functionality of the infinite list. The operations on lists can be mapped to the operations *store*, *select*, *insert* and *erase* on $\mathit{Function}{<}\mathit{Int}, \mathit{Val}{>}$ with:

- $\mathit{insert}(f, i, v)$ returns a function g such that $\forall k, \begin{cases} k < i \Rightarrow g(k) = f(k) \\ k = i \Rightarrow g(k) = v \\ k > i \Rightarrow g(k) = f(k-1) \end{cases}$
- $\mathit{erase}(f, i)$ returns a functions g such that $\forall k, \begin{cases} k < i \Rightarrow g(k) = f(k) \\ k \geq i \Rightarrow g(k) = f(k+1) \end{cases}$

We use cell abstraction defined in Section 5.1.3 to abstract $\mathit{Function}{<}\mathit{Int}, \mathit{Val}{>}$, and we add abstract expressions for *insert* and *erase*, giving Abstraction 6. The main idea is that adding or removing a value in the middle of an infinite function just consist in shifting the indexes of the rest of the data. The added abstract expressions are complete and the scheme can be extended to any number of distinguished cells.

Abstraction 6 (*insert* and *erase* expressions for one distinguished cell abstraction)

<table>
<tr><td colspan="2">$\phi(f \in \mathit{Function}{<}\mathit{Int}, \mathit{Val}{>}) = \{(x, y) | y = f(x)\}$</td></tr>
<tr><td><i>Concrete expression</i></td><td><i>Abstract expression</i></td></tr>
<tr><td>$g = \mathit{insert}(f, i, v)$</td><td>$\exists (x_g, y_g) \in \phi(g) \wedge \begin{cases} x_g < i \Rightarrow (x_g, y_g) \in \phi(f) \\ x_g = i \Rightarrow y_g = v \\ x_g > i \Rightarrow (x_g - 1, y_g) \in \phi(f) \end{cases}$</td></tr>
<tr><td>$g = \mathit{erase}(f, i)$</td><td>$\exists (x_g, y_g) \in \phi(g) \wedge \begin{cases} x_g < i \Rightarrow (x_g, y_g) \in \phi(f) \\ x_g \geq i \Rightarrow (x_g + 1, y_g) \in \phi(f) \end{cases}$</td></tr>
<tr><td colspan="2">$\mathit{AreAbstractions}_k((x_1, y_1), \ldots, (x_k, y_k)) = \bigwedge_{i,j} x_i = x_j \Rightarrow y_i = y_j$</td></tr>
</table>

This is a major contribution as $\mathit{Function}{<}\mathit{Ind}, \mathit{Val}{>}$ with the *store*, *select*, *insert* and *erase* operations, enables us to abstract C++ containers: vectors, lists, deque, maps, multimaps, sets, multisets... We can even abstract compound structures such as $\mathit{list}{<}\mathit{array}{<}\mathit{int}{>}{>}$ by applying this abstraction recursively.

Furthermore, the precision limit of the abstraction is not a big problem on such structures as most algorithms on such structures do not require an unbounded number of distinguished cells: most invariants only depend on the relation index to value and not so much on relations between many cells. As with arrays, we have enough precision to prove many sorting algorithms, searching algorithms, . . . The sortedness of *insertion sort* can be proved through Example 3, which represents a simplified output of Algorithm 1 with two distinguished cell abstraction.

We have given a general technique to abstract linear data structures. In the next section, we give research directions to abstract non linear data structures.

5.2.2 Research directions: trees, graphs, . . .

Our whole technique relies on Abstraction 4. Intuitively, the *select* and *store* operations allow us to abstract arbitrary containers that do not change structure. In Abstraction 6, we extended that scheme to linear containers by masking the change of structure through an index shift.

We generalize this scheme and define abstractions on containers of type Val as $\mathit{Function}{<}\mathcal{L}, \mathit{Val}{>}$ where $\mathcal{L}$ is a labeling of the structure we abstract. In the case of arrays and lists, $\mathcal{L}$ is Int.

We must then abstract operations that modify the structure of $\mathcal{L}$ by operations on $\mathcal{L}$. For example, in lists we abstracted the *erase* operation by a shift in the indexes, that is to say as a shift in the labeling. For example, binary trees have several classical labelings Rastello [2012]:

- The labeling for which we already have abstractions is $\mathit{list}{<}\mathit{Bool}{>}$: each node position is described by a list of left/right indications from the top node. Insertions and deletions within the tree are mapped to operations on labels: insertions and deletions on $\mathit{list}{<}\mathit{Bool}{>}$ correspond to insertions and deletions of a tree using $\mathit{list}{<}\mathit{Bool}{>}$ as label type. However, this abstraction fails: The domain is $\mathit{Function}{<}\mathit{Function}{<}\mathit{Int}, \mathit{Bool}{>}, \mathit{Val}{>}$ and after abstraction, the abstract value of a tree becomes several triples $(\mathit{depth}, \mathit{right}/\mathit{left}, \mathit{val})$ depending on the number of distinguished cells used. We can not express the parent/child relationships between two such triples, which is necessary for many algorithm on trees.
- Labelings from depth or breadth first search algorithms seem adequate: insertion and deletion correspond to index shifts and the precision loss is small: we can express parent/child relations with two distinguished cells and thus prove many algorithms. The main issue with this abstraction is that a given labeling corresponds to many trees, creating a loss of precision.

Example 3 (Insertion sort)

(a) C++ code with Section 3.2 conversion in comments

```
std::list<int> insertion_sort(const std::list<int>& l){         // $\ell_{size} \geq 0$
  std::list<int> res;                                          // $res_{size} \geq 0$
  std::list<int>::const_iterator lit = l.begin();              // $lit' = 0$
  while(lit != l.end()){                                       // $lit \neq \ell_{size}$
    int val = *lit;                                            // $val' = select(\ell_f, lit)$
    std::list<int>::iterator resit = res.begin();              // $resit' = 0$
    while(resit != res.end()){                                 // $resit \neq res_{size}$
      int read = *resit;                                       // $read' = select(res_f, resit)$
      if(read>val) break;                                      // $read > val \Rightarrow res\_Insert$
      resit++;                                                 // $resit' = resit + 1$
    }                                                          // $resit = res_{size}$
    res.insert(resit, val);        // $res'_{size} = res_{size} + 1 \wedge res'_f = insert(res_f, resit, val)$
    lit++;                                                     // $lit' = lit + 1$
  }                                                            // $lit = \ell_{size}$
  return res;                      // $\mathbf{x < y \wedge res_f(x) > res_f(y) \Rightarrow false}$
}
```

(b) Final result of function abstraction with two distinguished cells

$\ell_{size} \geq 0 \wedge \ell_x^1 < \ell_x^2$	$\Rightarrow Start(\ell_{size}, \ell_x^1, \ell_y^1, \ell_x^2, \ell_y^2)$
$Start(\ell_{size}, \ell_x^1, \ell_y^1, \ell_x^2, \ell_y^2) \wedge res_{size} \geq 0 \wedge res_x^1 < res_x^2 \wedge \ldots$	$\Rightarrow lit\_Init(\ldots, res_{size}, res_x^1, res_y^1, res_x^2, res_y^2)$
$lit\_Init(\ldots) \wedge lit' = 0 \wedge \ldots$	$\Rightarrow l\_While(\ldots)$
$l\_While(\ldots) \wedge lit \neq \ell_{size} \wedge \ldots$	$\Rightarrow val\_Init(\ldots)$
$val\_Init(\ldots, \ell_x^1, \ell_y^1, \ell_x^2, \ell_y^2, \ldots) \wedge lit \neq \ell_x^1 \wedge val\_Init(\ldots, lit, val', \ell_x^2, \ell_y^2, \ldots) \wedge \ldots$ $val\_Init(\ldots, \ell_x^1, \ell_y^1, \ell_x^2, \ell_y^2, \ldots) \wedge lit = \ell_x^1 \wedge val' = \ell_y^1 \wedge \ldots$	$\Rightarrow resit\_Init(\ldots)$ $\Rightarrow resit\_Init(\ldots)$
$resit\_Init(\ldots) \wedge resit' = 0 \wedge \ldots$	$\Rightarrow res\_While(\ldots)$
$res\_While(\ldots) \wedge resit \neq res_{size} \wedge \ldots$	$\Rightarrow read\_Init(\ldots)$
$read\_Init(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit \neq res_x^1 \wedge read\_Init(\ldots, resit, read', res_x^2, res_y^2, \ldots) \wedge \ldots$ $read\_Init(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit = res_x^1 \wedge read' = res_y^1 \wedge \ldots$	$\Rightarrow If(\ldots)$ $\Rightarrow If(\ldots)$
$If(\ldots) \wedge read > val \wedge \ldots$	$\Rightarrow res\_Insert(\ldots)$
$If(\ldots) \wedge read \leq val \wedge \ldots$	$\Rightarrow res\_Incr(\ldots)$
$res\_Incr(\ldots) \wedge resit' = resit + 1 \wedge \ldots$	$\Rightarrow res\_While(\ldots)$
$res\_While(\ldots) \wedge resit = res_{size} \wedge \ldots$	$\Rightarrow res\_Insert(\ldots)$
$res\_Insert(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit > res_x^2 \wedge res'_{size} = \ldots$ $res\_Insert(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit = res_x^2 \wedge res_y^{'2} = val \wedge \ldots$ $res\_Insert(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit = res_x^2 \wedge res_x^{'1} = res_x^2 \wedge res_y^{'1} = val \wedge res_x^{'2} = res_x^2 + 1 \wedge \ldots$ $res\_Insert(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit < res_x^2 \wedge res_x^{'2} = res_x^2 + 1 \wedge \ldots$ $res\_Insert(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit = res_1^2 \wedge res_x^{'2} = res_x^2 + 1 \wedge res_y^{'1} = val \wedge \ldots$ $res\_Insert(\ldots, res_x^1, res_y^1, res_x^2, res_y^2, \ldots) \wedge resit < res_1^2 \wedge res_x^{'2} = res_x^2 + 1 \wedge res_x^{'1} = res_x^1 + 1 \wedge \ldots$	$\Rightarrow lit\_Incr(\ldots)$ $\Rightarrow lit\_Incr(\ldots)$ $\Rightarrow lit\_Incr(\ldots)$ $\Rightarrow lit\_Incr(\ldots)$ $\Rightarrow lit\_Incr(\ldots)$ $\Rightarrow lit\_Incr(\ldots)$
$lit\_Incr(\ldots) \wedge lit' = lit + 1 \wedge \ldots$	$\Rightarrow l\_While(\ldots)$
$l\_While(\ldots) \wedge lit = \ell_{size} \wedge \ldots$	$\Rightarrow Return(\ldots)$
$\mathbf{Return}(\ldots) \wedge \mathbf{0 \leq res_x^1 < res_x^2 < res_{size} \wedge res_y^1 > res_y^2 \Rightarrow false}$	$/ * Sortedness * /$

We use $\ell_x^1, \ell_y^1, \ell_x^2, \ell_y^2$ as an abstraction of ℓ_f and $res_x^1, res_y^1, res_x^2, res_y^2$ for res_f. To increase readability, disjunctions within expressions are separated into several clauses, equalities have been propagated, and the final condition has been simplified.

We have not yet implemented any of these abstractions and we leave the proper analysis, implementation and tests of these abstractions for future work.

6 Implementation and experiments

Implementation We separated the implementation in two parts:

- The front end that takes as input a mini-Java program (a variation of While with array, lists, and assertions), and outputs a Smtlib2 file[14] describing a Horn problem in a theory close to *Fundamental*.
- The back-end that takes as input a Smtlib2 file describing a Horn problem in a theory close to *Fundamental* and outputs a Smtlib2 file describing a Horn problem in the *Basic* theory.

The front end is an adaptation of the Vaphor tool (Monniaux and Gonnord [2016]) to which we added lists, multidimensional arrays and composed operations[15]. We entirely developed the back-end from scratch following the technique described in Section 4.2 in 1.5k lines of Ocaml. There are slight differences as the theory around *AreAbstractions* had not been developed yet, but the overall result for function abstraction is equivalent.

Furthermore, the back-end is modular and independent of the front end, and can be used to test many other abstractions. The main purpose of the implementation is to serve as proof of concept for abstractions to come so that abstractions can then be implemented within Horn solvers or code analysis tools such as SeaHorn.

Various tools can solve systems of Horn clauses in the *Basic* theory. In this work, we tried Z3[16] with the PDR fixed point solver [Hoder and Bjørner, 2012], Z3 with the Spacer solver [Komuravelli et al., 2013, 2014],[17] and Eldarica[Rümmer et al., 2013].[18] Since program verification is undecidable, such tools, in general, may fail to terminate, or may return “unknown”.

Experiments We tested our analyzer on several examples from the literature:

- array benchmarks from the literature, [Dillig et al., 2010], [Bjørner et al., 2013], are in Table 1.
- classical algorithms on arrays and lists, including *bubble sort* and *insertion sort* in Table 2.

There has been no real tests on Horn problems that were not induced by programs as we have not been able to find examples of such Horn problems. The output is equivalent to the experiments of Monniaux and Gonnord [2016], the abstraction being the same.

Conversion tool limitations The front end suffers from limitations due to the Vaphor tool as the code was not completely rewritten:

- It does not deal with types other than integers, booleans, arrays and lists.
- Although it deals with multidimensional arrays, it does not deal with compound structures such as lists of arrays.

Horn solver limitations The Horn problems we retrieve from our technique are very close to those of Monniaux and Gonnord [2016] and suffer from the same Horn solver limitation: unreliable solving times as Horn solvers rely on backtracking and a simple change in the variable declaration

[14] http://smtlib.cs.uiowa.edu/

[15] The VapHor tool used edge abstraction and could not deal with composed operations.

[16] https://github.com/Z3Prover hash 7f6ef0b6c0813f2e9e8f993d45722c0e5b99e152; due to various problems we preferred not to use results from later versions.

[17] https://bitbucket.org/spacer/code hash 7e1f9af01b796750d9097b331bb66b752ea0ee3c

[18] https://github.com/uuverifiers/eldarica/releases/tag/v1.1-rc

Table 1: Comparison on the array benchmarks of [Dillig et al., 2010].

(Average) timing are in seconds, CPU time. Abstraction with $N = 1$. "sat" means the property was proved, "unsat" that it could not be proved. "hints" means that some invariants had to be manually supplied to the solver (e.g. even/odd conditions). A star means that we used another version of the solver. Timeout was 5 mn unless otherwise noted. The machine has 32 i3-3110M cores, 64 GiB RAM, C/C++ solvers were compiled with gcc 4.8.4, the JVM is OpenJDK 1.7.0-85.

Benchmark	Z3/PDR		Z3/Spacer		Eldarica		Comment
	Res	Time	Res	Time	Res	Time	
Correct problems, "sat" expected							
array copy	sat	0.42	sat	0.23	timeout(300s)		
array init2d	sat	1.12	sat	0.44	timeout(300s)		
array init2i	sat	0.56	sat	0.22	timeout(300s)		
array initcte	sat	0.15	sat	0.08	timeout(300s)		
array partialcopy	sat	0.46	sat	0.60	timeout(300s)		
array reverse	sat	82.32	sat	0.26	sat	36.61	
array strcpy	sat*	4.31	sat	0.30	sat	18.21	
array strlen	sat	0.20	sat	0.11	sat	27.39	
array swapncopy	sat	1.73	sat	0.86	timeout(300s)		
arrayappend	sat	61.40	sat	0.78	sat	17.17	
arrayfind	sat	0.20	sat	0.11	sat	10.76	
arrayfindnonnull	sat	0.28	sat	0.25	sat	14.10	
memcpy	sat	0.44	sat	0.23	timeout(300s)		
array initeven	timeout(300s)		timeout(300s)		timeout(300s)		
array initeven hinted	sat	0.03	sat	0.02	sat	4.74	Hinted
array swapncopy twice	timeout(300s)		sat	15.46	timeout(300s)		
array swapncopy twice hinted	sat	0.13	sat	0.08	sat	12.64	Hinted
mergeinterleave	sat	8.49	sat	207.02	sat	77.95	
mergeinterleave hinted	sat	0.19	sat	0.05	sat	10.54	Hinted
Incorrect problems, "unsat" expected							
array copyodd buggy	unsat	0.05	unsat	0.02	unsat	7.29	
array initeven buggy	unsat	0.05	unsat	0.03	unsat	5.50	
array reverse buggy	unsat	0.52	unsat	0.71	unsat	57.69	
array swapncopy buggy	unsat	0.93	unsat	0.19	unsat	32.08	
mergeinterleave buggy	unsat	0.54	unsat	0.22	unsat	26.77	

Table 2: Other array-manipulating programs, including various sorting algorithms.

A star means that we used a previous version of the solver.

Benchmark	Z3/PDR		Z3/Spacer		Eldarica		Distinct number
	Res	Time	Res	Time	Res	Time	
bin search	sat	0.38	timeout(300s)		Exception		N=1
find mini	sat	2.99	sat	1.16	sat	58.97	N=1
bubble sort	sat	4.42	sat	3.17	sat	68.86	N=2
insert sort	sat*	146.42	timeout(300s)		timeout(300s)		N=2

order, or in the pseudo-random number generator can change execution time by over a factor 100. Guiding the solver by asserting simple invariants (e.g. $0 \leq k < i$ for a loop from k to $i - 1$) greatly improves the reliability of the solvers.

Like Monniaux and Gonnord [2016], we believe that solving times should not be regarded too closely: research on Horn solvers is recent and the solving times might change drastically in the years to come through the combination of abstract interpretation and SMT solving. The purpose of our experimental evaluation is not to benchmark solvers relative to each other, but to show that our abstraction, even though it is incomplete, is powerful enough to lead to fully automated proofs of correctness of nontrivial manipulations of linear data structures, including sorting algorithms.

7 Conclusion and perspectives

We proposed a generic approach to verify specifications on programs containing unbounded data structures. This approach has been applied to arrays and lists: arrays have been fully tested on algorithms requiring non trivial invariants, such as sorting algorithms, and lists have been implemented but not thoroughly tested. We expect to have solid test results on lists in the very near future.

The limitations of the approach can be divided into three categories: limitations due to program conversions and mainly pointers as described in Section 3.2, limitations due to unbounded data abstraction as explained in Section 5 and limitations due to Horn solvers.

Converting programs in real world programming languages to Horn clauses over a theory that does not involve memory is a tough task and the work in Section 3.2 does not cover it. In fact, we expect this step to be at least as hard as converting a program into a functional language as it requires the static analyzer to abstract pointers. There is active research on the topic: alias analyses, shape analyses, separation logic, . . . and tools such as SeaHorn and FramaC, and we do not expect this limitation to be the main issue.

The second limitation has been the main focus of our work. We have succeeded in defining what are probably the best possible abstractions on Horn clauses for lists and arrays and gave a framework as well as research directions to abstract general unbounded data structures. Furthermore, we gave an implementation and experiments that show that our abstractions succeed in proving non trivial algorithms.

However, our technique can not prove specifications such as "the sum of the array is equal to 42" as this requires a number of distinguished cells equal to the size of the array. We believe this is a limit of abstract interpretation and that such specifications can not be proven through the use of abstract interpretation only, or at least without major drawbacks. We intend to investigate another technique which consists in expressing local changes instead of global changes: we may not be able to express that the sum of an array is equal to 42, but we can express that the store operation adds $stored\_val - old\_val$ to the sum of the array. By introducing a variable representing the sum of the array and modifying it at each *store* operation, we can prove that the sum of the array is equal to 42. A similar technique has been applied in Monniaux and Gonnord [2016] to prove that the multiset of values is unchanged by a sorting algorithm.

Another issue is extending the abstract interpretation of unbounded data structures into bounded data structure. We succeeded for lists and arrays and we seem close to succeeding for trees. However, finding abstract interpretation on graphs that could prove shortest path algorithms remains an untouched research topic. In the near future, we intend to analyze, implement and test the tree abstraction using depth/breadth first search labeling described in Section 5.2.2.

Finally, a major issue of our approach is that Horn solvers still have unreliable solving times. The main reason for this issue is that Horn solvers calculate predicates by backtracking and refining predicates when the current solution is unsatisfiable. This technique is subject to huge variance as choosing the value of predicates when backtracking occurs is done through heuristics.

There has been major improvement on Horn solvers in the past years and we expect improvements in the years to come. Mainly, we expect that invariants that can be calculated through abstract interpretation, for example in the polyhedron model, will not require backtracking and predicate refinement.

References

N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *Static Analysis Symposium (SAS)*, pages 105–125, 2013. doi: 10.1145/2695664.2695784.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977. doi: 10.1145/512950.512973.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96, 1978. doi: 10.1145/512760.512770.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *European symposium on programming (ESOP)*, number 3444 in Lecture Notes in Computer Science, pages 21–30, 2005. ISBN 978-3-540-25435-5. doi: 10.1007/978-3-540-31987-0_3.

I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Conference on Programming Languages and Systems (ESOP)*, pages 246–266, 2010. doi: 10.1007/978-3-642-11957-6_14.

K. Hoder and N. Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012. ISBN 978-3-642-31611-1. doi: 10.1007/978-3-642-31612-8_13.

N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 66–74, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: 10.1145/582153.582161. URL http://doi.acm.org/10.1145/582153.582161.

A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *Computer-aided verification (SAS)*, volume 8044, pages 846–862. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_59.

A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In A. Biere and R. Bloem, editors, *Computer-aided verification (SAS)*, volume 8559 of *LNCS*, pages 17–34. Springer, 2014. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_2.

D. Monniaux and F. Alberti. A simple abstraction of arrays and maps by program translation. In *Static analysis (SAS)*, volume 9291 of *Lecture Notes in Computer Science*, pages 217–234. Springer Verlag, 2015. ISBN 978-3-662-48287-2. doi: 10.1007/978-3-662-48288-9. URL https://hal.archives-ouvertes.fr/hal-01162795.

D. Monniaux and L. Gonnord. Cell morphing: from array programs to array-free Horn clauses. Working paper, Apr. 2016. URL https://hal.archives-ouvertes.fr/hal-01206882.

F. Rastello. On Sparse Intermediate Representations: Some Structural Properties and Applications to Just-In-Time Compilation. University works, Dec. 2012. URL https://hal.inria.fr/hal-00761555. Habilitation à diriger des recherches, École normale supérieure de Lyon.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL http://dl.acm.org/citation.cfm?id=645683.664578.

P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In N. Sharygina and H. Veith, editors, *Computer-aided verification (CAV)*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_24.