Static analysis of algebraic data types and arrays

Santiago Bautista

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

école normale supérieure rennes

THÈSE DE DOCTORAT DE

L'ÉCOLE NORMALE SUPÉRIEURE DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Santiago Sara BAUTISTA

Static Analysis of Algebraic Data Types and Arrays

Thèse présentée et soutenue à Rennes, le 20 décembre 2023
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Xavier RIVAL	Directeur de recherche à Inria Paris
Mihaela SIGHIREANU	Professeure des Université à l'ENS Paris-Saclay

Composition du Jury :

Présidente :	Nathalie BERTRAND	Directrice de Recherche	Inria Rennes
Examinateurs :	Matthieu LEMERRE	Ingénieur-Chercheur	CEA Paris-Saclay
	Xavier RIVAL	Directeur de Recherche	Inria Paris
	Mihaela SIGHIREANU	Professeure des Universités	ENS Paris-Saclay
Directeur de thèse :	Thomas Jensen	Directeur de recherche	Inria Rennes
Co-encadrant de thèse :	Benoît Montagu	Chargé de recherche	Inria Rennes

Table of Contents

Chapter 1

Introduction

Research in static analysis has successfully developed several techniques to ensure the safety and security of programs, by detecting bugs *before* a program actually runs. Some static analysis techniques are automatic. For example, in the context of abstract interpretation [CC77], there exists a substantial number of abstract domains that target programs with numeric or pointer-based computations and which can automatically detect frequent bugs that arise from arithmetic overflows or memory safety issues. Other static analysis techniques are not automatic. Interactive theorem proving, for example, is very expressive on the properties that it can prove, but it may require a significant amount of human effort [Mat18, Chapter 4]. Interactive theorem proving has successfully been used to verify the entire functional specifications of complex pieces of software, such as the CompCert compiler [Ler+16], verified using the Coq proof assistant, or the seL4 microkernel [Kle+14], verified using the Isabelle/HOL proof assistant.

Automatic static analysis techniques have successfully been used to decrease the amount of human effort required when using interactive theorem proving. For example, the correlations abstract domain [And+19] allowed to discharge two thirds of the proof obligations required to verify the invariants of the ProvenCore microkernel [Les15]. The long-term goal of the work done during this thesis is to be able to further apply this technique of using abstract interpretation to alleviate human effort in interactive theorem proving. This asks for abstract domains that are expressive and that can analyse languages with features such as algebraic data types (ADTs) and arrays. For example, the formal verification of seL4 done in Isabelle/HOL uses a specification in Isabelle that features ADTs and arrays, and properties that include numeric relations between parts of values of algebraic data types stored in arrays. In this manuscript we will use the term *structured values* to refer to values from algebraic types.

During this thesis, we investigated whether it is possible, for languages that feature algebraic data-types and arrays, to use techniques based in abstract interpretation to compute input-output summaries of functions, that are expressive enough to capture the

numeric relations that hold between parts of structured values; even when these structured values are stored inside arrays of an unknown size.

Two examples of the kind of programs that we want to be able to analyse are given in figures 1.1 and 1.2. Both examples are simplified versions of the kind of functions that can be found in an operating system's micro-kernel; and were inspired by the functional specification of seL4. The first example, function `do_ticks` from figure 1.1, manipulates values from an algebraic type `process` representing processes. The type `process` has three fields: a field `id` containing an identifier of the process, a field `msg` that may contain a message sent by another process, and a field `status` describing the status of the process. The status of a process might be either `Running` or `Asleep`. If the process is running, a field `count` stores how many times the process has been activated. If the process is asleep, a field `secs` stores the number of seconds the process should remain asleep, and the field `count` stores the number of times the process has been activated. The function `do_ticks` takes two parameters: a process `p` and a number of seconds `n`. The function simulates the action of `n` clock ticks on a process `p`: a clock tick leaves the process `p` unchanged if `p` is already running; or, if it is asleep, decrements the sleeping budget of `p`. If that budget is already zero, the clock tick promotes `p` into a running process.

The important properties of `do_ticks(p, n)` that we intend to infer *automatically* are the following:

1. If `p` is initially running, then it remains unchanged;
2. If `p` is initially sleeping, then it *might* wake up: in this case, its original sleeping budget was less than `n`, and `count`—its number of activations—has been incremented by one;
3. If `p` is initially sleeping, then it *might* remain sleeping: in this case, its sleeping budget decreased by `n`, and its number of activations remains the same;
4. The field `id`, of integer type, of the process `p` has not changed;
5. The field `msg`, of record type, of the process `p` has not changed either.

The main idea of our approach for analysing algebraic types is to use pairs of variable names and access paths (we call these pairs *extended variables*) to refer to different *parts* of structured values. In particular, when extended variables point to *numbers* inside structured values, then we can use numeric abstract domains that already exist to capture

```
type status = [ (* Scheduling status *)
  | Running of { count: int }
    (* Running: activation times *)
  | Asleep of { secs: int; count: int }
    (* Sleeping: remaining seconds, activation times *)
]
type msg = {              (* Messages *)
  data : int ;            (* Payload *)
  reply : [               (* Whether to reply or not *)
    | Reply of int        (* Who to reply to *)
    | DontReply of {}     (* No reply expected *)
  ]
}
type process = { id: int; msg: msg; status: status }  (* Process structure *)

def do_ticks(process p, int n) : process = {
(* Performs n clock ticks on the process p *)
  int count; int secs; int i
  assert (n > 0)
  i = 0
  while (i < n) do (* loop n times, i.e.: perform n clock ticks *)
    branch (* case where p is running *)
      count = p.status@Running.count
    or (* case where p is asleep and can sleep longer *)
      assert (p.status@Asleep.secs > 0)
      count = p.status@Asleep.count
      secs = p.status@Asleep.secs
      p = { id = p.id; msg = p.msg;
            status = Asleep { secs = secs - 1; count = count } }
    or (* case where p is asleep and has no more sleeping budget *)
      assert (p.status@Asleep.secs = 0)
      count = p.status@Asleep.count
      p = { id = p.id; msg = p.msg;
            status = Running { count = count + 1 } }
    end
    i = i + 1
  end
  return p
}
```

Figure 1.1: Example program performing clock ticks on a process's meta-data.

numeric relations between these numeric parts of structured values. For example, for the `do_ticks` function in figure 1.1, in the case where both the input and the output of the function are asleep processes, we capture the property $\texttt{p}'\texttt{.status@Asleep.secs} = \texttt{p.status@Asleep.secs} - \texttt{n}$ that states that the sleeping budget of the process has been decreased by `n`. This property can be captured for example, by the polyhedra domain, if the extended variables `p.status@Asleep.secs` and $\texttt{p}'\texttt{.status@Asleep.secs}$ are treated as if they were numeric variables. Our approach can extend any numeric domain, as long as it provides the operators and verifies the properties listed in section 2.3. We call this first construction the NPR domain, for *Numeric Path Relations* (Chapter 4).

The main challenge to this approach comes from the fact that values from a sum type can use different constructors, that are mutually exclusive. For example, the status of processes (figure 1.1) can be either `Running` or `Asleep`. This changes the fields that are defined: the sleeping budget, in our example, only exists for processes that are asleep. But it also changes the properties that need to be captured. Indeed, when a programmer decides to use a sum type, it generally models states of data that are quite different and require different treatment. Hence the analysis of programs with sum types also requires, in order to remain precise, a distinction between different cases, according to what constructors are used by values from sum types. In our example, if the input process is `Running` then the only properties that need to be captured are the fact that the message and identity fields are not modified. If the input process is `Asleep`, then additional properties need to be captured to reflect how the sleeping budget and status of the process evolve.

To tackle this challenge, we introduce a disjunctive completion (section 5.4), where different disjuncts capture information about the different possible constructors for sum types. To keep our disjunction small, we adopt two different strategies:

- We merge together disjuncts that make equivalent assumptions on the constructors used. We introduce an abstract domain that explicitly tracks the assumptions made on constructors. This abstract domain is called the *Constructor Constraints* domain (section 5.1).

- We avoid some disjunctions by capturing in a concise way the parts of structured values that are equal. For this, we introduce the *Structural Equalities* domain (section 5.2).

Our second example, function `find_max_prio` from figure 1.2, manipulates arrays. It involves thread descriptors—named *Thread Control Blocks*, or TCBs for short—that

```
type unit = {} (* Record type with no fields *)

(* Thread descriptors (Thread Control Block) *)
type tcb =
{ prio      : int; (* Priority *)
  ... (* Other fields of the TCB are elided *)
}

(* An array of TCBs. Represents a scheduler queue. *)
type queue = tcb[]

(* Options of TCBs. Serves as a return type for find_max_priority *)
type max_result = [ NoMax of unit | SomeMax of tcb ]

(* Returns the TCB with the hightest priority in the queue, if any. *)
def find_max_priority(queue q) : max_result = {
  max_result res
  unit       case
  int        i
  tcb        challenger

  i = 0
  res = NoMax{}
  while (i < |q|) do (* Iterate over the queue *)
    challenger = q[i]
    branch
      case = res@NoMax (* First iteration *)
      res  = SomeMax challenger
    or
      assert(challenger.prio > res@SomeMax.prio) (* Higher priority found *)
      res = SomeMax challenger
    or
      assert(res@SomeMax.prio >= challenger.prio) (* No change needed *)
    end
    i = i + 1
  end
  return res
}
```

Figure 1.2: Program that finds a thread descriptor with highest priority in an array.

represent information about the threads that are managed by an operating system kernel. TCBs are records of properties. To keep the example short, we only exhibit one property of TCBs—their priority—although TCBs may have more. The `find_max_priority` function, takes an array of TCBs as a parameter, and it searches in the array for a TCB whose priority is the highest. It returns an option type, such that either `NoMax{}` is returned if the array is empty, or `SomeMax d` is returned, where `d` is a TCB in the array, with the highest priority.

Ideally we would want our analysis to fully verify the `find_max_prio` function by capturing its full specification: the fact that the output is a TCB from the input array, with the highest priority. However, as we will see in Chapter 9, our domain can only capture the fact that the output TCB has a priority that is higher to the priority of any TCB inside the input array; but it does not capture the fact that the output TCB indeed *belongs* to the input array.

When analysing arrays, the main challenge is the fact that the size of the array is unknown, hence the different cells of the array cannot be analysed individually. Instead, multiple cells of arrays should be analysed together. However, analysing separately cells that behave differently allows for a better precision. In this thesis we adopt the approach of Cousot, Cousot and Logozzo [CCL11] for analysing arrays: each array variable is mapped into a *segmentation*. Segmentations have different sets of bounds that divide the array into segments, and each segment is summarized separately. Our approach has three main differences with respect to [CCL11]:

- We allow for arrays to contain values from algebraic types, whereas [CCL11] supported only arrays of scalars.
- We allow the summaries of array segments to refer to other variables of the program, hence capturing the relations between the array contents and the values stored in other variables, or the parameters of functions.
- We have a different definition for the abstract inclusion between array segmentations (section 9.4.3), to solve a problem of monotonicity that we found in [CCL11]'s concretisation for array segmentations (section 9.4.2).

For an example, we examine an intermediate result of our analysis on the function `find_max_priority` from figure 1.2. If we only consider the executions that take the loop at least once and we look at the segmentation that we get for array `q` at the head of the

loop, after the widening converges, we get

$$\{0\} \left(\begin{array}{c} 0 \leq l \leq \mathtt{i} - 1; \mathtt{i} \geq 1 \\ v.\mathtt{prio} \leq \mathtt{res@SomeMax.prio} \end{array} \right) \{i\} \top^{\mathrm{S}} \{|q|\}?$$

In this segmentation, two different segments are considered: a segment from index 0 (included) to index i (excluded), and a segment from index i (included) to the length of the array (excluded). The question mark after the sets of bounds $\{|q|\}$ indicates that the second segment might be empty. Inside segment summaries, the variables l and v are special variables: l stands for the index of array slots in the segment, while v stands for the content of array slots in the segment. The summary of the first segment properly captures the fact that the TCB stored in variable `res` has a priority that is higher than the TCBs stored in the array slots of this segment. Note that capturing this property requires to relate a variable that is outside to the array (variable `res`) with the content of the array.

The structure of this manuscript is as follows. In Chapter 2 we introduce some preliminary definitions, in which we formalize what we mean by algebraic types and values (section 2.1), we describe the language that we work with (section 2.2) and we list the hypotheses that we expect on the numeric domains that we extend (section 2.3). Part I describes how we extend numeric domains to handle algebraic types. In Part II we describe the different steps needed to turn our abstract domain into an actual program analysis, and we discuss implementation and experimental results. In Part III we further extend our approach, in order to handle arrays. The contributions of Parts I and II are implemented, but the contributions of Part III are not. Both Parts I and III start with a chapter discussing related work (Chapters 3 and 8). This manuscript includes the following contributions:

- A novel abstract domain that expresses relations between values of non-recursive ADTs (Part I). Our abstract domain can be instantiated with any numeric relational domain. This offers a choice between domains with different precision *vs* cost balances, and allows to capture numeric inequalities. This improves upon the correlation domain [And+19], that is restricted to information about equality and reachability.

- Our abstract domain uses a particular form of *disjunctive completion* (section 5.4), where we limit the number of disjuncts by *merging* some of them. Our merging strategy is guided by observing the different *cases* of algebraic values.

- We give a formal justification to the folklore assertion that *"a static analysis can*

be made relational by duplicating variables", by showing that a non input-output relational and an input-output relational analysis actually share the same *structure* (lemma 12) and by showing how any relational domain can express relations between different stores (Chapter 6).

- We formally define a relational analysis that infers relations between inputs and outputs of programs (section 7.1), and propose a modular inter-procedural extension that is based on function summaries (section 7.2). We illustrate the analyser's results on the function `do_ticks` from figure 1.1, which serves as running example for Parts I and II.

- We provide an OCaml implementation [BJM22a] of our analyser, for a `while` language with non-recursive algebraic types; together with 43 test cases, some of which are inspired from an operating system code (Tables 7.1 and 7.2 in section 7.3). We briefly discuss the complexity of our implementation (section 7.3).

- We extend our approach to encompass functional *arrays* that can contain algebraic data types (Part III). This extension of our abstract domain is based on the notion of array segmentations by Cousot, Cousot and Logozzo [CCL11].

The contributions from Parts I and II of this manuscript have already been published:

- [BJM20] contains a preliminary version of the NPR abstract domain from Chapter 4

- [BJM22b] contains all the contributions of Parts I and II. It is accompanied by a virtual machine artefact, that contains the code of our implementation [BJM22a].

The extension for functional arrays (Part III) is not published at the time of writing, but is undergoing peer-review. It was submitted as part of a journal article.

Chapter 2

Preliminaries

In this chapter, we introduce some definitions, notations and hypothesis that we will use in the rest of the manuscript. Section 2.1 introduces the definitions we use for algebraic types and values. Section 2.2 describes the programming language that we analyse. Our programming language is an extension of a classic `while` language, with algebraic data types (products and sums). Section 2.3 introduces the notations and the hypotheses for the abstract numeric domains that we use.

2.1 Algebraic Types and Values

ADTs are pervasively used in functional languages like OCaml, Haskell, Coq, or F*, and have become a central feature of more recent programming languages, such as Swift or Rust, just to name a few. We briefly recall the definitions of *algebraic types*, and of the *structured values* that inhabit them.

Definition 1 (Algebraic types and structured values)**.** Algebraic types *and* structured values *are inductively defined as follows:*

$$\begin{array}{lllllll} \tau \in \text{Types} & ::= & \text{Int} & | & \{\overline{f_i \to \tau_i}^{i\in I}\} & | & [\overline{A_i \to \tau_i}^{i\in I}] \\ v \in \text{Values} & ::= & \underline{n} & | & \{\overline{f_i = v_i}^{i\in I}\} & | & A(v) \end{array}$$

Here, Int is the type of numbers, the $(f_i)_{i\in I}$ are field names, the $(A_i)_{i\in I}$ are constructor names, and I ranges over finite sets. The compound type $\{\overline{f_i \to \tau_i}^{i\in I}\}$ is a *record type*, in which a type τ_i is associated to each field f_i. The type $[\overline{A_i \to \tau_i}^{i\in I}]$ is a *sum type* containing values formed with a head constructor that must be one of the A_i, and whose argument must be of type τ_i. $\{\overline{f_i = v_i}^{i\in I}\}$ denotes a *record value* where each field f_i has value v_i for every $i \in I$. $A(v)$ denotes a *variant value*, built by applying the constructor A to the value v. Constructors expect exactly one argument. Constructors with arities other than 1, as typically found in functional languages, are encoded by providing a (possibly empty)

record value as argument to constructors. The numeric type Int and the record type with no fields {} are the two base cases for types.

We use *projection paths* to refer to a part of a structured value (*i.e.*, to a value embedded *inside* another structured value). A path is either the empty path ε, or the path $p.f$, that first accesses the value at path p and then accesses the record field f, or the path $p@A$, that first accesses the value at path p and then accesses the argument of variant constructor A.

Definition 2 (Paths)**.** *Paths are inductively defined as follows:*

$$p \in \text{Paths} \quad ::= \quad \varepsilon \quad | \quad p.f \quad | \quad p@A$$

Because paths are simply sequences of atomic paths ($.f$ or $@A$) we allow their creation or destruction from either side, and write for example $@Ap$ to denote a path that starts with $@A$.

The *projection of the value v on the path p*, written $v \Downarrow^{\text{val}} p$, is the value that p points to inside v. It is defined as follows:

Definition 3 (Projection of a value on a path)**.**

$$v \Downarrow^{\text{val}} p \quad = \quad \begin{cases} v & \text{if } p = \varepsilon \\ v' \Downarrow^{\text{val}} p' & \text{if } p = @Ap' \text{ and } v = A(v') \\ v_i \Downarrow^{\text{val}} p' & \text{if } p = .f_i p' \text{ and } v = \{\overline{f_j = v_j}^{j \in I}\} \text{ and } i \in I \\ \text{Undef} & \text{otherwise} \end{cases}$$

Our definition returns Undef when a path does not make sense for some value.

2.2 A Language With Algebraic Data Types

Figure 2.1 presents the syntax of the language, which consists of expressions t, boolean conditions b, and commands c. Vars denotes the set of variables that may appear in commands. For any set of variables V, let $\text{Fresh}(V) = \text{Vars} \setminus V$ be the set of variables that are fresh with respect to V. Expressions include the projection of a variable $x \in \text{Vars}$ over a path $p \in \text{Paths}$, written $x.p$. The expression $t_1 \boxplus t_2$ denotes some arithmetic operations on the expressions t_1 and t_2, and $t_1 \bowtie t_2$ ranges over arithmetic comparisons.

We restrict our attention to well-typed commands (that we call *programs*), following a standard structural type system [Pie02]. For instance, well-typedness ensures that

$$
\begin{array}{lcl}
t \in \mathrm{Exp} & ::= & \underline{n} \quad | \quad A(t) \quad | \quad \{\overline{f_i = t_i}^{\,i \in I}\} \quad | \quad x.p \quad | \quad t_1 \boxplus t_2 \\
b \in \mathrm{BExp} & ::= & t_1 \bowtie t_2 \quad | \quad b_1 \wedge b_2 \quad | \quad b_1 \vee b_2 \quad | \quad \neg b \\
c \in \mathrm{Cmd} & ::= & \mathsf{skip} \quad | \quad c_1 \,;\, c_2 \quad | \quad \mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end} \quad | \\
 & & \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end} \quad | \quad \mathsf{assert}\, b \quad | \quad x := t
\end{array}
$$

Figure 2.1: Grammar of the analysed language

$$
\textsc{SeqStep}\ \frac{(c_1, s) \to (c_1', s')}{(c_1 \,;\, c_2, s) \to (c_1' \,;\, c_2, s')} \qquad
\textsc{SeqSkip}\ \frac{}{(\mathsf{skip} \,;\, c_2, s) \to (c_2, s)} \qquad
\textsc{Assign}\ \frac{v \in [\![t]\!]_s^{\mathrm{exp}}}{(x := t, s) \to (\mathsf{skip}, s(x \mapsto v))}
$$

$$
\textsc{Branch}\ \frac{1 \le i \le n}{(\mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end}, s) \to (c_i, s)} \qquad
\textsc{Assert}\ \frac{\mathsf{tt} \in [\![b]\!]_s^{\mathrm{bool}}}{(\mathsf{assert}(b), s) \to (\mathsf{skip}, s)}
$$

$$
\textsc{WhileTrue}\ \frac{\mathsf{tt} \in [\![b]\!]_s^{\mathrm{bool}}}{(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}, s) \to (c \,;\, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}, s)} \qquad
\textsc{WhileFalse}\ \frac{\mathsf{ff} \in [\![b]\!]_s^{\mathrm{bool}}}{(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}, s) \to (\mathsf{skip}, s)}
$$

Figure 2.2: Small-step semantics of commands. $\mathbf{B} = \{\mathsf{tt}, \mathsf{ff}\}$ is the set of boolean values.

arithmetic tests and operations receive arguments of integer type, and that every projection $x.p$ is consistent with the type of the variable x.

Programs operate on *stores*, denoted by s, that are finite maps from Vars to Values. We define the semantics of programs using a standard small-step semantics that specifies the effects of commands on stores (figure 2.2). The relation $(c, s) \to (c', s')$ tells that the command c transforms the store s into a store s', and that command c' is to be executed next.

The command skip performs no operation, whereas the sequence $c_1 \,;\, c_2$ executes c_1 followed by c_2. The branching command $\mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end}$ non-deterministically chooses one of the commands c_i and executes it, discarding the other branches. The command $\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}$ executes the command c as long as the condition b holds, and successfully terminates otherwise.

The command $\mathsf{assert}(b)$ tests whether the condition b holds, in which case the command succeeds, and the execution of the program continues. When b is not satisfied, $\mathsf{assert}(b)$ fails, *i.e.*, the program remains stuck. We can express the conditional construct $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ as $\mathsf{branch}\ \mathsf{assert}(b) \,;\, c_1\ \mathsf{or}\ \mathsf{assert}(\neg b) \,;\, c_2\ \mathsf{end}$.

$$
\begin{array}{rcl}
[\![x.p]\!]^{\text{exp}}_s & = & \begin{cases} \{s(x) \Downarrow^{\text{val}} p\} & \text{if } s(x) \text{ is defined and } s(x) \Downarrow^{\text{val}} p \neq \text{Undef} \\ \emptyset & \text{otherwise} \end{cases} \\
\left[\!\!\left[\{\overline{f_i = t_i}^{i \in I}\}\right]\!\!\right]^{\text{exp}}_s & = & \left\{\{\overline{f_i = v_i}^{i \in I}\} \mid \forall i \in I, v_i \in [\![t_i]\!]^{\text{exp}}_s\right\} \\
[\![A(t)]\!]^{\text{exp}}_s & = & \{A(v) \mid v \in [\![t]\!]^{\text{exp}}_s\} \\
[\![t_1 \boxplus t_2]\!]^{\text{exp}}_s & = & \{v_1 \boxplus v_2 \mid v_1 \in [\![t_1]\!]^{\text{exp}}_s \wedge v_2 \in [\![t_2]\!]^{\text{exp}}_s\}
\end{array}
$$

Figure 2.3: Denotation of expressions.

$$
\begin{array}{rcl}
[\![t_1 \bowtie t_2]\!]^{\text{bool}}_s & = & \{v_1 \bowtie v_2 \mid v_1 \in [\![t_1]\!]^{\text{exp}}_s \wedge v_2 \in [\![t_2]\!]^{\text{exp}}_s\} \\
[\![b_1 \wedge b_2]\!]^{\text{bool}}_s & = & \{\mathsf{b}_1 \wedge \mathsf{b}_2 \mid \mathsf{b}_1 \in [\![b_1]\!]^{\text{bool}}_s \wedge \mathsf{b}_2 \in [\![b_2]\!]^{\text{bool}}_s\} \\
[\![b_1 \vee b_2]\!]^{\text{bool}}_s & = & \{\mathsf{b}_1 \vee \mathsf{b}_2 \mid \mathsf{b}_1 \in [\![b_1]\!]^{\text{bool}}_s \wedge \mathsf{b}_2 \in [\![b_2]\!]^{\text{bool}}_s\} \\
[\![\neg b]\!]^{\text{bool}}_s & = & \{\neg \mathsf{b} \mid \mathsf{b} \in [\![b]\!]^{\text{bool}}_s\}
\end{array}
$$

Figure 2.4: Denotation of conditions.

Finally, the assignment command $x := t$ evaluates t to some value v and updates the variable x with v. We write $s(x \mapsto v)$ to denote the store s updated with the association from variable x to value v. If there was an entry for x in s already, then it is replaced with the value v. Otherwise, a new entry is created.

The evaluation $[\![t]\!]^{\text{exp}}_s$ of an expression t in a store s proceeds by induction on the structure of t to evaluate sub-expressions, and reads in the store s the values of variables (figure 2.3). $[\![t]\!]^{\text{exp}}_s$ is either a singleton, which denotes normal execution, or the empty set, which denotes a failure, such as an invalid projection $x.p$. For example, if $s(x) = A(v)$ then $[\![x@B]\!]^{\text{exp}}_s = \emptyset$, because the constructors A and B are different. The evaluation of booleans $[\![b]\!]^{\text{bool}}_s$ is standard (figure 2.4).

Importantly, records and variants are *immutable* in our language: it is not possible to update some field f of a record *in-place*, for example. Instead, the programmer must follow the functional idiom, and create a new record value, that contains a different value for the field f.

We recover the *pattern matching* construct $\mathsf{match}\, t\, \mathsf{with}\, A_1(x_1) \to c_1 \mid \cdots \mid A_n(x_n) \to c_n\, \mathsf{end}$ as a syntactic sugar for command $z := t\,;\mathsf{branch}\, x_1 := z@A_1\,; c_1\, \mathsf{or} \ldots \mathsf{or}\, x_n := z@A_n\,; c_n\, \mathsf{end}$ for a freshly chosen variable z.

For any expression t (respectively any boolean condition b), we call $\text{Env}\,(t)$ (resp. $\text{Env}\,(b)$) the set of all variable projections that appear in t (resp. b). For example, for boolean expression $x@A \leq y@B + 1$, we have $\text{Env}\,(x@A \leq y@B + 1) = \{x@A;\ y@B\}$.

Lemma 1 (Consistency between the semantics of expressions and the projections

being mentioned). *For any expression* $t \in \mathrm{Exp}$*, any boolean condition* $b \in \mathrm{BExp}$*, any set of variables* V *and any state* $s : V \to \mathrm{Values}$

- *If* $[\![t]\!]_s^{\mathrm{exp}} \neq \emptyset$ *then* $\forall x.p \in \mathrm{Env}\,(t)\,, s(x) \Downarrow^{\mathrm{val}} p \neq \mathrm{Undef}$.
- *If* $[\![b]\!]_s^{\mathrm{bool}} \neq \emptyset$ *then* $\forall x.p \in \mathrm{Env}\,(b)\,, s(x) \Downarrow^{\mathrm{val}} p \neq \mathrm{Undef}$.

Proof sketch. Both of this properties are proven by induction: induction on t for expressions, and induction on b for boolean conditions. The proof for expressions is done before the proof for boolean conditions, as the former is used in the latter. The base case for expressions (that is, projection), directly follows from the definition of the semantics of expressions (figure 2.3). All the recursive cases are proven in the same way:

- The non-emptiness of denotations propagates to sub-expressions, given the way $[\![\blacksquare]\!]_{\blacksquare}^{\mathrm{exp}}$ is defined (figure 2.3).
- Any projection mentioned in a compound expression is mentioned in one of its sub-expressions, which allows to use the induction hypothesis.

The base case for boolean conditions (numeric comparison between two expressions) uses the proof for expressions. The recursive cases for boolean conditions work exactly like the recursive cases for expressions. □

Lemma 1 will be used when proving the soundness of the transfer functions of the NPR domain of section 4.2.

2.3 Background: Numeric Abstract Domains

We first review the structure of traditional numeric domains [Min17] such as intervals, octagons and polyhedra. The domains are parametrised by a set of variables, and describe sets of *numeric* stores over those variables, *i.e.*, sets of maps from variables to numbers.

Given a set of variables V, we expect a numeric abstract domain $\mathrm{N}(V)$ to provide the operations listed below (which are included in the user interface of the Apron library [JM09]) in such a way that the standard soundness properties of abstract interpretation [CC77; Cou21] are met: A set of abstract values $\mathrm{N}(V)$ with a concretisation function $\gamma^{\mathrm{N}(V)} \in \mathrm{N}(V) \to \mathcal{P}(V \to \mathrm{Int})$, a pre-order on abstract values $\sqsubseteq^{\mathrm{N}(V)}$, abstract union $\sqcup^{\mathrm{N}(V)}$ and intersection $\sqcap^{\mathrm{N}(V)}$, and a widening operator $\triangledown^{\mathrm{N}(V)}$. The domain must also offer abstractions for boolean conditions $\mathrm{Cond}^{\mathrm{N}(V)} \in \mathrm{BExp} \to \mathrm{N}(V) \to \mathrm{N}(V)$ and for

assignment $\text{Assign}^{\text{N}(V)} \in V \times \text{Arith}\,(V) \to \text{N}(V) \to \text{N}(V)$ (where $\text{Arith}\,(V)$ is the set of arithmetic expressions over the variables V), satisfying the soundness properties:

$$\begin{array}{lll} \gamma^{\text{N}(V)}(\text{Assign}^{\text{N}(V)}(x := t)(d)) & \supseteq & \{s(x \mapsto v) \mid s \in \gamma^{\text{N}(V)}(d) \wedge v \in [\![t]\!]_s^{\text{exp}}\} \\ \gamma^{\text{N}(V)}(\text{Cond}^{\text{N}(V)}(b)(d)) & \supseteq & \{s \in \gamma^{\text{N}(V)}(d) \mid \mathtt{tt} \in [\![b]\!]_s^{\text{bool}}\} \end{array}$$

Additionally, we expect a predicate $\text{CanSat}^{\text{N}(V)} \in \text{N}(V) \times \text{BExp} \to \mathbf{B}$ for querying an abstract value to know whether a boolean condition can be satisfied or not. This operation must verify the property

$$\left(\exists s \in \gamma^{\text{N}(V)}(d), [\![c]\!]_s^{\text{exp}} = \{\mathtt{tt}\}\right) \Rightarrow \text{CanSat}^{\text{N}(V)}(d, c)$$

From this predicate, another predicate $\text{Satisfies}^{\text{N}(V)}$ can be deduced, by taking

$$\text{Satisfies}^{\text{N}(V)}(d, c) = \neg\, \text{CanSat}^{\text{N}(V)}(d, \neg c)$$

The predicate $\text{Satisfies}^{\text{N}(V)}$ verifies the property

$$\text{Satisfies}^{\text{N}(V)}(d, c) \Rightarrow \forall s \in \gamma^{\text{N}(V)}(d), [\![c]\!]_s^{\text{exp}} \subseteq \{\mathtt{tt}\}$$

In other words, $\text{Satisfies}^{\text{N}(V)}(d, c)$ guarantees that condition c is either true or blocking, for all states in the concretisation of d.

We also assume the existence of "variable management" operators for removing, adding and renaming variables. Operator $\text{Rem}_{V'}^{\text{N}(V)}$ projects an element of $\text{N}(V)$ onto $\text{N}(V \setminus V')$. Operator $\text{Add}_{V'}^{\text{N}(V)}$ embeds an element of $\text{N}(V)$ into the domain $\text{N}(V \cup V')$. Given a bijection $r : V_1 \to V_2$, the operator $\text{Rename}_r^{\text{N}(V_1)}$ translates an element of $\text{N}(V_1)$ into $\text{N}(V_2)$. These operators satisfy the following soundness properties.

Hypothesis 1 (Soundness of variable removal)**.**

$$\gamma^{\text{N}(V \setminus V')}(\text{Rem}_{V'}^{\text{N}(V)}(d)) \supseteq \{\, s|_{(V \setminus V')} \mid s \in \gamma^{\text{N}(V)}(d)\}$$

Hypothesis 2 (Soundness of variable addition)**.**

$$\gamma^{\text{N}(V \cup V')}(\text{Add}_{V'}^{\text{N}(V)}(d)) \supseteq \{s : (V \cup V') \to \text{Int} \mid\ s|_V \in \gamma^{\text{N}(V)}(d)\}$$

Hypothesis 3 (Soundness of variable renaming)**.**

$$\gamma^{\mathrm{N}(V_2)}(\mathrm{Rename}_r^{\mathrm{N}(V_1)}(d)) \supseteq \{s \mid s \circ r \in \gamma^{\mathrm{N}(V_1)}(d)\}$$

In addition, we require three properties about the operator for removing variables of the underlying numeric domain N:

Hypothesis 4 (Monotony of variable removal with respect to abstract values)**.** *For any two sets of variables V_1 and V_2 and any two abstract values d_1 and d_2 in $\mathrm{N}(V_1)$, we require that*

$$d_1 \sqsubseteq^{\mathrm{N}(V_1)} d_2 \Rightarrow \mathrm{Rem}_{V_2}^{\mathrm{N}(V_1)}(d_1) \sqsubseteq^{\mathrm{N}(V_1 \setminus V_2)} \mathrm{Rem}_{V_2}^{\mathrm{N}(V_1)}(d_2)$$

Hypothesis 5 (Composition of variable removal)**.** *For any three sets of variables V_1, V_2 and V_3 such that $V_1 \subseteq V_2 \subseteq V_3$, and any abstract value $d \in \mathrm{N}(V_3)$, we require that*

$$\mathrm{Rem}_{V_3 \setminus V_1}^{\mathrm{N}(V_3)}(d) \sqsubseteq^{\mathrm{N}(V_1)} \mathrm{Rem}_{V_2 \setminus V_1}^{\mathrm{N}(V_2)}\left(\mathrm{Rem}_{V_3 \setminus V_2}^{\mathrm{N}(V_3)}(d)\right)$$

In other words, hypothesis 5 states that, when removing a set of variables from a numeric abstract value, it is at least as precise to remove them all at once, than to remove them in two steps.

Hypothesis 6 (Empty set removal)**.** *For any set of variables V and any abstract value $d \in \mathrm{N}(V)$, we require that*

$$\mathrm{Rem}_{\emptyset}^{\mathrm{N}(V)}(d) = d$$

Even though it is technically possible to implement in Apron a numeric domain that does not respect hypotheses 4 and 5, these hypotheses are neither surprising nor very restrictive. In particular, we believe that all the numeric abstract domains that we tested our implementation with (polyhedra, octagons and intervals) respect all of the hypotheses in this section.

Hypotheses 4 and 5 are used to prove that the pre-order of the NPR lifting is transitive (lemma 5 on page 39). Hypothesis 6 is used when proving that the widening of the NPR lifting enforces convergence in finite time (lemma 7 on page 41); and the reflexivity of the NPR pre-order (lemma 5).

Part I

Abstract Domains for Algebraic Types

Chapter 3

Related Work on Algebraic Types Analysis

In this chapter, we discuss some of the work that has been done to analyse programs that manipulate algebraic data-types:

- *Lattice tree automata* by Genet et al. [Gen+13] and the domain for trees by Journault, Miné and Ouadjaout [JMO19] that both combine tree automata and numeric abstract domains.
- The *correlation* domain by Andreescu et al. [And+19], that recursively defines abstract values that focus on equalities between parts of structured values.
- *Convoluted tree automata* by Losekoot, Genet and Jensen [LGJ23], that recognize regular sets of tuples of terms.
- Li et al. [Li+17b] who, in the context of shape analysis, introduce the notion of *silhouette* to control the number of disjuncts in a disjunctive abstract domain.
- Liu and Rival [LR15b] who focus on the particular case of optional values.
- Valnet, Monat and Miné [VMM23] who are currently working on extending the MOPSA platform ([Jou+19]) to develop a static analyser for the OCaml language.

In order to explain the different approaches, we will use the simple example programs of figure 3.1. The `alternate` function from figure 3.1a is taken from [JMO19], while the `alternate_0_1` function from figure 3.1c and `alternate_eq` from figure 3.1b are two variations of `alternate` that are easier to analyse.

The `alternate` function. The function `alternate` from figure 3.1a takes as parameters a non-negative integer n and an integer x. It returns a list of size $2n$, where the elements

```
let rec alternate x n =
  assert(n >= 0);
  match n with
  | 0 -> []
  | _ -> (x + 1) ::
         (x - 1) ::
         (alternate x (n - 1))
```

(a) An OCaml function returning a list of size $2n$ that alternates between $x+1$ and $x-1$. This example is taken from [JMO19].

```
let rec alternate_eq x n =
  assert(n >= 0);
  match n with
  | 0 -> []
  | _ -> x ::
         (x + 1) ::
         (alternate_eq x (n - 1))
```

(b) A variation on `alternate` that alternates between x and $x+1$.

```
let rec alternate_0_1 n =
  assert(n >= 0);
  match n with
  | 0 -> []
  | _ -> 0 ::
         1 ::
         (alternate_0_1 (n - 1))
```

(c) A variation on `alternate` that alternates between 0 and 1.

Figure 3.1: Three example programs

of even index are equal to $x+1$ and the elements of odd index are equal to $x-1$ (indexing starts at 0).

The `alternate_eq` function Like the `alternate` function, the `alternate_eq` function also takes as parameters a non-negative integer n and an integer x; and returns a list of size $2n$. However, instead of alternating between $x+1$ and $x-1$, the elements of the output list alternate between x and $x+1$. This function is *slightly* easier to analyse than the function `alternate`. Indeed, there is an *equality* between the input x and the values at even indices of the list; rather than a more complex numeric relation such as $+1$ or -1. As we will see, the abstract domain of *correlations* ([And+19]) specializes in equalities, and can capture more information when analyzing this function than when analyzing `alternate`.

The `alternate_0_1` function. The `alternate_0_1` function takes as a parameter a non-negative integer n and returns a list of size $2n$ that alternates between the values 0 (for even indices) and 1 (for odd indices). This function is easier to analyse than the `alternate` and `alternate_eq` functions, since the values in the output list can be described on their

own, without referring to an input x. In other words, describing the values in the output list of this function does not require to capture a *relation* between input and output.

In figures 3.2 to 3.5, we give abstract values from different related work that over-approximate either the output or the input-output relation of some of these examples. However, we adapt the notations to make these abstract values readable without having to introduce all of the formalism of the original works.

Values from algebraic types can be seen as trees. For this reason, several works analyzing programs with algebraic types and discussed in this chapter use tree automata [Com+08]. We briefly introduce tree automata by an example.

Tree automata in a nutshell. A tree automaton is a way of describing a set of terms, using a system of rewriting rules. As an example, we consider the set of lists that only contain zeros. These lists can be seen as terms built using 3 symbols:

- The symbol Nil, representing the empty list. It takes no arguments.
- The symbol Cons, representing a non-empty list. It takes two arguments: the first element of the list, and the rest of the list.
- The symbol $\mathsf{0}$, representing the integer zero.

We now define a tree automaton that recognizes the set of lists of zeros. We give this automaton two states: a state q_0 that recognizes zero, and a state q_L that recognizes lists filled with zeros. We give this automaton the following system of rewriting rules:

$$0 \to q_0 \qquad \mathsf{Nil} \to q_L \qquad \mathsf{Cons}(q_0, q_L) \to q_L$$

The meaning of the transition system is:

- The term 0 can be rewritten into q_0. We also say that state q_0 *recognizes* the term 0.
- The term Nil can be rewritten into q_L. In other words, state q_L recognizes the term Nil.
- The term $\mathsf{Cons}(q_0, q_L)$ can be rewritten into q_L. Said otherwise, if there are two terms t_1 and t_2 that are recognized by q_0 and q_L respectively, then the term $\mathsf{Cons}(t_1, t_2)$ is recognized by q_L.

We declare q_L as the final state of our automaton, which means that the terms recognized by the automaton are the ones recognized by q_L. This ends our example of tree automaton.

Symbols:	Nil	(no arguments),
	Cons	(two arguments)
Automaton states:	q_{zero}	(the integer zero)
	q_{one}	(the integer one)
	q_{even}	(lists of an even size)
	q_{odd}	(lists of an odd size)
Final state:	q_{even}	
Rewriting rules:	$[0; 0] \to q_{zero}$	
	$[1; 1] \to q_{one}$	
	$\mathsf{Nil} \to q_{even}$	
	$\mathsf{Cons}(q_{one}, q_{even}) \to q_{odd}$	
	$\mathsf{Cons}(q_{zero}, q_{odd}) \to q_{even}$	

Figure 3.2: A *lattice tree automaton* ([Gen+13]) that represents the possible outputs of function `alternate_0_1` from figure 3.1c; using intervals as numeric abstract domain.

3.1 Numbers Inside Structured Values, and Relations between them

In 2013, Genet et al. [Gen+13] introduced *lattice tree automata*, which combine tree automata and numeric abstract domains. The main idea is that some of the states of the automaton recognize numeric abstract values (representing a set of numbers). This allows to capture properties on the numbers contained in the structured values of the program being analysed. However, a limitation of this approach is that the numeric abstract values are necessarily at the *leaves* of the structured values being recognized. Thus excluding the possibility to capture *relations* between different leaves of a single structured value; or leaves of different structured values. Figure 3.2 shows the abstract value that this approach would yield to represent the possible outputs of function `alternate_0_1` from figure 3.1c. For the more complex function figure 3.1a, this approach would not yield a very precise abstraction, since it cannot capture the numeric *relation* between the input x and the list elements $x + 1$ and $x - 1$.

A later approach by Journault, Miné and Ouadjaout [JMO19] also combines tree automata and numeric abstract domains to abstract over sets of values from algebraic types. But instead of having numeric abstract values directly in the rewriting rules of the automaton, a special symbol $\square$ is used for numerical values in the rewriting rules. Constraints on numerical values are then given *alongside* the automaton, by using *regular expressions* to describe sets of access paths, and a numeric abstract value to describe

Symbols:	Nil	(no arguments),
	$\square$	(no arguments),
	Cons	(two arguments)
Automaton states:	q_{even}	(lists of an even size)
	q_{odd}	(lists of an odd size)
Final state:	q_{even}	
Rewriting rules:	$\mathsf{Nil} \to q_{even}$	
	$\mathsf{Cons}(\square, q_{even}) \to q_{odd}$	
	$\mathsf{Cons}(\square, q_{odd}) \to q_{even}$	
Regular expressions for access paths:	$r_{even} = (.1.1)^*.0$	(numeric elements at even indices)
	$r_{odd} = (.1.1)^*.1.0$	(numeric elements at odd indices)
Numeric constraints:	$t.r_{odd} = x - 1;\ t.r_{even} = t.r_{odd} + 2$	

Figure 3.3: Abstract value from [JMO19] for the `alternate` function of figure 3.1a; assuming the output is stored in a variable t.

$$\left\{ \begin{array}{lcl} \mathsf{x} & \to & \left[\begin{array}{lcl} \mathsf{Nil} & \to & \top \\ \mathsf{Cons} & \to & \left\{ \begin{array}{lcl} \mathsf{head} & \to & \mathrm{Eq} \\ \mathsf{tail} & \to & \top \end{array} \right\}^{\text{Output}} \end{array} \right]^{\text{Output}} \\ \mathsf{n} & \to & \top \end{array} \right\}^{\text{Input}}$$

Figure 3.4: *Correlation* ([And+19]) over-approximating the `alternate_eq` function from figure 3.1b.

numeric constraints on the values at those access paths. This approach manages to capture numeric *relations* between the different leaves of a structured value; as shown in figure 3.3.

Some related work focuses on capturing specific kinds of relations. It is the case for *correlations* ([And+19]) that focuses on equality, and *convoluted tree automata* ([LGJ23]) which captures some relations between the size of structured values and numbers in the program.

Andreescu et al. [And+19] define the abstract domain of *correlations* to abstract over input-output relations of functions manipulating algebraic data-types. Correlations capture *equality* relations between parts the input and parts of the output of a function. If a function has multiple inputs, correlation analysis will treat it as having a single input which is a record. Correlations are defined inductively. The base cases are $\top$ (which carries no information), $\bot$ (which corresponds to unreachable code or unreachable constructors), and Eq (which indicates an equality between the part of the input being currently considered, and the part of the output being currently considered). For the inductive cases, a superscript

indicates whether the correlation will focus on different parts of the input, or different parts of the output. An inductive case built with curly braces indicates that the value being considered is a record, and then focuses on each field of the record. An inductive case built with square brackets indicates that the value being considered is a variant, and then focuses on the different possible constructors of the variant. An example of correlation is given in figure 3.4, for the `alternate_eq` function from figure 3.1b. Correlations should be read from the outside in. For figure 3.4, the outermost curly braces with the superscript Input tell us that the input is a record. A correlation is given for each one of the two fields x and n of this record. The correlation associated to field x is built with square brackets with the superscript Output. This tells us that the output is a variant. This variant has two possible constructors Nil and Cons. The $\top$ correlation associated to constructor Nil does not tell us anything in particular. However, the correlation associated to constructor Cons indicates to us that the argument of constructor Cons is a record with two fields head and tail. Furthermore, the correlation Eq indicates that the value of the field head is equal to the input x.

To summarize, the correlation of figure 3.4 states that, if the output of function `alternate_eq` is a non-empty list (built using constructor Cons), then the head of that list is equal to the input x.

A limitation of this approach is that it can only express *equality* relations between parts of the input and parts of the output, instead of more complex numeric relations. For example, this approach would not be able to capture any interesting property for function `alternate` form figure 3.1a; since no part of the output is *equal* to any part of the input. Instead there are *numeric* relations between x and $x + 1$ and x and $x - 1$.

There is an approach by Losekoot, Genet and Jensen [LGJ23] that does not use abstract numeric domains, but relies entirely on tree automata. Their approach translates both the program to be analysed and the property to be verified into constrained Horn clauses, and then uses a learner-teacher implication counter-example procedure to either find a model that proves the property or a counter-example that disproves it. The procedure may also loop, or fail to answer. An advantage of [LGJ23] compared with the other works cited in this chapter is that they can relate the *size* of structured values to other numeric quantities. For example, they can prove that the length of a reversed list is the same than the one of the original list. However, the expressiveness of their approach is limited by the models that they produce: convoluted tree automata. Only some kinds of relations can be recognized by such automata. For example, their approach cannot relate the size of the

output of function `alternate_0_1` with the input `n`, since the multiplication by two of an integer cannot be encoded inside a convoluted tree automata.

3.2 Disjunctive Abstract Domains

As pointed out by Kim, Rival and Ryu [KRR18], different static analyses use either disjunctions or conjunctions of implications to achieve an improved precision. It then becomes important to limit the number of disjunctions, so as to keep the cost of the analysis under control. As we will see in Chapter 5, we use a form of disjunction when analyzing algebraic types. We separate disjuncts according to which constructors are used to build variants. We limit the size of our disjunction by merging together disjuncts that consider the same sets of constructors.

Our strategy for limiting the size of our disjunction is similar to the one used by Li et al. [Li+17b] in the context of shape analysis. Indeed, in both works, disjuncts are merged together if they are *equivalent*, for an equivalence relation defined by the analysis. In the case of [Li+17b], two disjuncts are equivalent if they have the same *silhouette*, which is an abstraction of which symbolic variables point to each other in memory. In our case, two disjuncts are equivalent if they consider the same sets of constructors for variants.

In the particular case of optional values (that are either None or Some of a number), Liu and Rival ([LR15b]) don't use a disjunction nor a conjunction of implications. Instead, they use multiple *avatars* for the variables that have an option type. For each optional variable, the two avatars represent respectively a lower-bound and an upper-bound on the number of the Some case. If the two avatars hold contradictory constraints, then the optional variable is in the None case.

3.3 Recursive Algebraic Types

The approach that we will present in this thesis for analyzing algebraic types does not handle recursive algebraic types. By contrast, the techniques based on tree automata [Gen+13; LGJ23; JMO19] (presented in section 3.1) handle recursive algebraic types. Correlations [And+19] (presented in section 3.1) handle recursive types in a limited way. A correlation can capture information for the first few stages of recursion, but the rest will sooner or later be over-approximated with $\top$. In particular, for function `alternate_eq`, this domain cannot capture the fact that one of every two elements of the output is equal

Possible constructors at top-level:	$\{\mathsf{Cons}, \mathsf{Nil}\}$
Symbolic variable for the head of the list:	r
Possible constructors for nested values:	$\{\mathsf{Cons}, \mathsf{Nil}\}$
Numeric constraints on symbolic variables:	$0 \leq r \leq 1$

Figure 3.5: Value from the abstract domain of [VMM23] over-approximating the possible outputs of function `alternate_0_1`.

to the input `x`.

Valnet, Monat and Miné [VMM23] have developed an abstract domain to analyse languages with algebraic data types. Their approach is based on using symbolic variables to summarise the different values that a field of a structured value might take at different depths of recursion; and indicating the list of constructors that are possible for recursive cases. Figure 3.5 shows an example of an abstract value from their domain, that over-approximates the possible outputs of the `alternate_0_1` function from figure 3.1c. Compared to our approach, they have the advantage of handling *recursive* algebraic data types. When it comes to which relations can be captured between the different fields of structured values or the precision of input-output summaries for functions, it is difficult to compare the precision of their approach to ours, since these aspects are not yet part of their implementation.

Chapter 4

NPR: An Abstract Domain for Algebraic Types

The goal of this chapter and the next is to define an abstract domain that will allow to analyse programs that manipulate algebraic types. The syntax of the programs was defined in Chapter 2. In this chapter we will use as a running example the `do_ticks` function from the introduction (figure 1.1 on page 7), that is presented again in figure 4.2. The different steps in the construction of our abstract domain are summarised in figure 4.1. Our domain is parametric with respect to a numeric abstract domain N, so that we can instantiate it on different precision versus cost trade-offs. We expect the numeric domain N to provide the operations described in section 2.3, which are a subset of the API offered by Apron [JM09]. Numeric abstract domains are tailored to abstract sets of stores where *variables* hold *numbers*. Instead, in a program that manipulates algebraic types, variables hold structured values. The main idea of our approach is to use projection paths to artificially build stores in which *pairs* of *variables and projection paths* are associated to *numbers*. We call these variable-path pairs *extended variables*. Section 4.1 defines extended variables and some operations on them. Using extended variables, we define in section 4.2 a first way to lift numeric domains to languages with algebraic types: the *Numeric Path Relations* lifting, or *NPR lifting* for short. It can express, for example, that a call to `do_ticks` can only decrease the value in the field `secs` of processes (that denotes the number of seconds for which a process should remain asleep), thanks to the constraint on extended variables $\texttt{p.status@Asleep.secs} \geq \texttt{p}'\texttt{.status@Asleep.secs}$.

In next chapter, we improve the precision of the *NPR lifting* by combining it with two other domains (sections 5.1 and 5.2) in a product domain (section 5.3). A first domain of *constructor constraints* tracks which constructors are used for values of sum types (section 5.1). Constructor constraints allow us to distinguish between different cases, by stating which extended variables are valid in each case. For the `do_ticks` program, a possible case is when the input process `p` is sleeping—*i.e.*, `p.status@Asleep` is valid—and

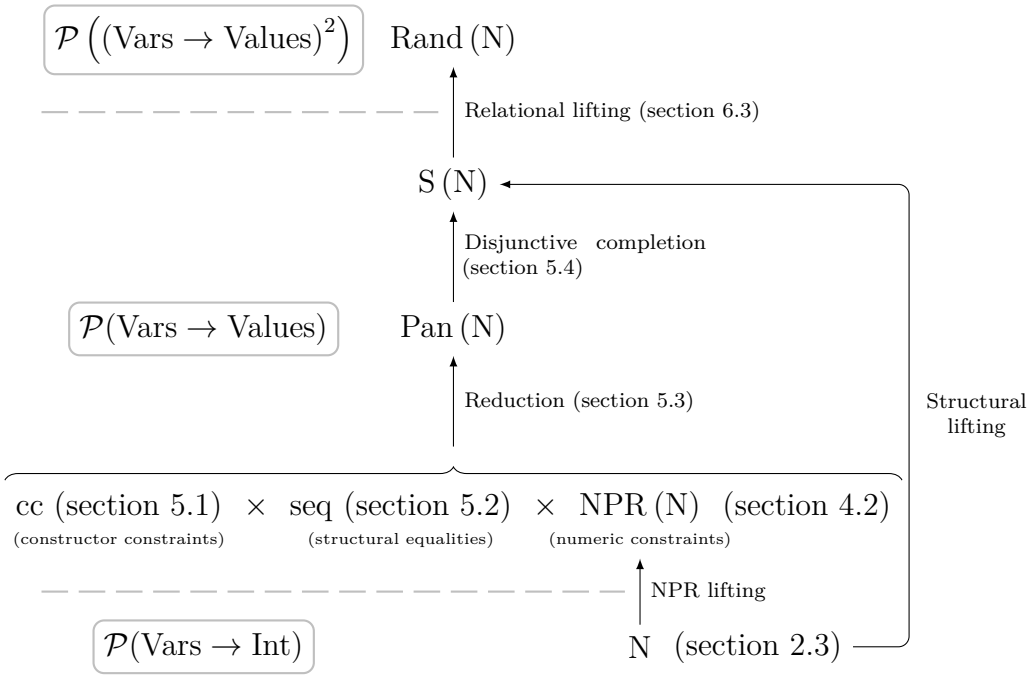


Figure 4.1: The construction of the RAND abstract domain. The frame-enclosed sets are the sets the abstract domains concretize to.

the output process `p'` is running—*i.e.*, `p'.status@Running` is valid. Another domain, called *structural equalities* (section 5.2), uses equality constraints between extended variables to express equalities that *must* hold between arbitrary parts—of any type—of structured values. With this domain, we can tell for the `do_ticks` program that the `msg` field of processes cannot change, by saying that the extended variables `p.msg` and `p'.msg` are equal. Finally, in order to obtain additional precision when analysing pattern-matching, we use a *disjunctive completion* of the product of these domains (section 5.4): we obtain the *structural lifting* of the numeric abstract domain. Each value of the structural lifting can contain multiple cases, and each case has three components: one that expresses constructor constraints, one that expresses structural equalities, and one that expresses numeric constraints. Sections 4.2, 5.1 and 5.2 also define abstractions for assignments and conditionals, that are needed in Chapter 7 to define the analysis of our language.

Running Example Figure 4.2 recalls the function `do_ticks` from figure 1.1 in the introduction, for which we would like to infer precise input-output properties. This function features algebraic data types that represent the meta-data of a process, as usually found in operating system implementations. Here, a process is a record composed of an identifier,

```
type status = [ (* Scheduling status *)
  | Running of { count: int }
    (* Running: activation times *)
  | Asleep of { secs: int; count: int }
    (* Sleeping: remaining seconds, activation times *)
]
type msg = {              (* Messages *)
  data : int ;            (* Payload *)
  reply : [               (* Whether to reply or not *)
    | Reply of int        (* Who to reply to *)
    | DontReply of {}     (* No reply expected *)
  ]
}
type process = { id: int; msg: msg; status: status }  (* Process structure *)

def do_ticks(process p, int n) : process = {
(* Performs n clock ticks on the process p *)
  int count; int secs; int i
  assert (n > 0)
  i = 0
  while (i < n) do (* loop n times, i.e.: perform n clock ticks *)
    branch (* case where p is running *)
      count = p.status@Running.count
    or (* case where p is asleep and can sleep longer *)
      assert (p.status@Asleep.secs > 0)
      count = p.status@Asleep.count
      secs = p.status@Asleep.secs
      p = { id = p.id; msg = p.msg;
            status = Asleep { secs = secs - 1; count = count } }
    or (* case where p is asleep and has no more sleeping budget *)
      assert (p.status@Asleep.secs = 0)
      count = p.status@Asleep.count
      p = { id = p.id; msg = p.msg;
            status = Running { count = count + 1 } }
    end
    i = i + 1
  end
  return p
}
```

Figure 4.2: Example program performing clock ticks on a process' meta-data.

some incoming message that was sent by another process and finally a piece of data that describes the status of the process. The message is a record that contains some payload and whether it needs a reply (and to whom). The process status is either running, in which case it records how many times the process has been activated, or it is asleep, in which case it also records how many seconds the process should remain asleep before waking up again. The function `do_ticks(p, n)` simulates the action of `n` clock ticks on a process `p`: a clock tick leaves the process `p` unchanged if `p` is already running, or, if it is asleep, decrements the sleeping budget of `p`. If that budget is already zero, the clock tick promotes `p` into a running process.

The important properties of `do_ticks(p, n)` that we intend to infer *automatically* are the following:

1. If `p` is initially running, then it remains unchanged;
2. If `p` is initially sleeping, then it *might* wake up: in this case, its original sleeping budget was less than `n`, and `count`—its number of activations—has been incremented by one;
3. If `p` is initially sleeping, then it *might* remain sleeping: in this case, its sleeping budget is decreased by `n`, and its number of activations remains the same;
4. The field `id`, of integer type, of the process `p` has not changed;
5. The field `msg`, of record type, of the process `p` has not changed either.

Chapters 4 and 5 explain in detail how we express and capture these properties by presenting the structure of the RAND abstract domain. The correlation abstract domain [And+19] was also designed to handle programs that manipulate algebraic data types, but cannot express, on numbers, properties other than binary equalities. Using the correlation domain, we could infer all the properties listed above, *except* the ones that involve arithmetics: properties 2 and 3.

4.1 Extended Variables: Variable-Path Pairs

Paths were defined in section 2.1, on page 14. We call *extended variable* the pair of a variable and a path. Extended variables designate some values that are located *inside* a structured value. We only consider paths that make sense for the given variables, *i.e.* paths whose projections on a variable's type are valid in the following sense:

Definition 4 (Projection of a type on a path)**.** *The judgement* $\tau \Downarrow^{\text{typ}} p$ *defines when a path* p *is consistent with a type* τ*, and is inductively defined by:*

$$\frac{}{\tau \Downarrow^{\text{typ}} \varepsilon} \qquad \frac{\tau_i \Downarrow^{\text{typ}} p \qquad i \in I}{\{\overline{f_j \to \tau_j}^{j \in I}\} \Downarrow^{\text{typ}} .f_i p} \qquad \frac{\tau_i \Downarrow^{\text{typ}} p \qquad i \in I}{[\overline{A_j \to \tau_j}^{j \in I}] \Downarrow^{\text{typ}} @A_i p}$$

For example, for the type `status` from figure 4.2, both the judgement status $\Downarrow^{\text{typ}}$ @Running.count and the judgement status $\Downarrow^{\text{typ}}$ @Asleep.secs hold. However, the judgment status $\Downarrow^{\text{typ}}$ @Cons.head does not hold, since constructor Cons does not belong to the sum type status.

Typing contexts, written Γ, are mappings from variables to types. We write $\mathcal{E}(\Gamma) = \{x.p \mid x \in \operatorname{dom}\Gamma \wedge \Gamma(x) \Downarrow^{\text{typ}} p\}$ for the set of extended variables $x.p$ such that p is consistent with the type of x in Γ. For example, if a typing context Γ contains a single binding $\Gamma = [x \mapsto \mathsf{status}]$, then the set $\mathcal{E}(\Gamma)$ is

$$\mathcal{E}(\Gamma) = \{x, x@\mathsf{Running}, x@\mathsf{Asleep}, x@\mathsf{Running.count}, x@\mathsf{Asleep.secs}, x@\mathsf{Asleep.count}\}$$

We say that two extended variables $x.p_1$ and $x.p_2$ are *incompatible* — written $x.p_1 <> x.p_2$ — if they would force a value (or part of a value) to be in two different variants of a sum type. Definition 5 formalises this notion of incompatibility, using the prefix order $\preccurlyeq$ on extended variables ($x.p \preccurlyeq y.q$ iff $x = y$ and p is a prefix of q).

Definition 5 (Incompatibility and inconsistency)**.** *Two extended variables* $x_1.p_1$ *and* $x_2.p_2$ *are* incompatible*, written* $x_1.p_1 <> x_2.p_2$*, if and only if* $x_1 = x_2$ *and there is a path* p *and two* distinct *constructors* A_1 *and* A_2*, such that* $x_1.p@A_1 \preccurlyeq x_1.p_1$ *and* $x_2.p@A_2 \preccurlyeq x_2.p_2$*. A set of extended variables* E *is* inconsistent *if it contains two or more incompatible extended variables. Two sets of extended variables are incompatible, written* $E_1 <> E_2$*, if their union is inconsistent.*

For example, if the typing context is $\Gamma = [x \mapsto \mathsf{status}]$, then the two extended variables x@Running.count and x@Asleep.count are incompatible; hence the set of extended variables $\{x@\mathsf{Running.count}, x@\mathsf{Asleep.count}, x@\mathsf{Asleep.secs}\}$ is inconsistent. This implies that the two sets of extended variables $\{x@\mathsf{Running.count}\}$ and $\{x@\mathsf{Asleep.count}, x@\mathsf{Asleep.secs}\}$ are incompatible.

In section 5.1, we use the fact that inconsistent sets of extended variables denote empty sets of stores. Such inconsistent sets correspond to unreachable program points, and can

be safely removed from the disjunctive completion of section 5.4.

Assignment decomposition To easily define the abstract transfer functions for assignment in section 4.2, it is useful to decompose an assignment command $x := t$—where t can be a compound expression—into an equivalent set of *parallel* assignments of the form $x.p := t'$, where t' is either an expression of numeric type or an extended variable. The idea is to model the effect of the assignment as a set of parallel assignments on the paths of variable x.

Definition 6. *The decomposition of the assignment $x := t$ is defined by:*

$$\text{Decomp}\,(x.p := t) = \begin{cases} \bigcup_{i\in I} \text{Decomp}\,(x.p.f_i := t_i) & \text{if } t = \{\overline{f_i = t_i}^{i\in I}\} \\ \text{Decomp}\,(x.p@A := t') & \text{if } t = A(t') \\ \{x.p := t\} & \text{if } \Gamma \vdash t : \text{Int} \vee t \in \mathcal{E} \end{cases}$$

We write $\text{Decomp}\,(x := t)$ as a shorthand for $\text{Decomp}\,(x.\varepsilon := t)$.

For example, in a typing context $\Gamma = [\mathsf{st} \mapsto \mathsf{status}; \mathsf{secs} \mapsto \text{Int}; \mathsf{count} \mapsto \text{Int}]$ the assignment `st = Asleep {secs = secs - 1; count = count}` would be decomposed as follows

$$\begin{aligned} &\text{Decomp}\,(\mathsf{st} := \mathsf{Asleep}\{\mathsf{secs} = \mathsf{secs} - 1; \mathsf{count} = \mathsf{count}\}) = \\ &\quad \{\mathsf{st@Asleep.secs} = \mathsf{secs} - 1;\ \mathsf{st@Asleep.count} = \mathsf{count}\} \end{aligned}$$

The fact that assignment decomposition properly reflects what happens for extended variables after an assignment is stated by the following lemma.

Lemma 2 (Soundness of assignment decomposition). *For any set of variables V, for any variable $x \in V$, for any expression $t \in \text{Exp}$, any value $v \in \text{Values}$, and any store $s : V \to \text{Values}$; if t evaluates to v in s* — i.e. $v \in [\![t]\!]_s^{\text{exp}}$ —*, then*

$$\forall (x.p := u) \in \text{Decomp}\,(x := t)\,, v \Downarrow^{\text{val}} p \in [\![u]\!]_s^{\text{exp}}$$

Proof sketch. The main idea of the proof is to do an induction on expression t. However, the property of this lemma is not strong enough to carry through the induction, since $\text{Decomp}\,(x := t)$ is necessarily the beginning of the decomposition, with an empty path. Hence, the property proven by induction is a more general one: for any path p,

$$\forall (x.pp' := u) \in \text{Decomp}\,(x.p := t)\,, v \Downarrow^{\text{val}} p' \in [\![u]\!]_s^{\text{exp}}$$

The base cases (when t is either an extended variable or a numeric expression) are very simple: we then have $p' = \varepsilon$ and $u = t$.

The inductive cases (when t is either a variant or a record) go through because the definition of value projection $\Downarrow^{\text{val}}$ (definition 3 on page 14), the definition of the denotational semantics of expressions $[\![\bullet]\!]^{\text{exp}}$ (figure 2.3 on page 16) and the definition of assignment decomposition work well together. Indeed, when t is a variant of the form $A(t')$ (respectively, a record of the form $\left\{\overline{f_i = t_i}^{i \in I}\right\}$), then we know that p' starts with the corresponding constructor access $@A$ (resp. with a field access $.f_i$) and the rest of p' is a certain path p''. Then, the induction hypothesis can be applied with $p@A$ (resp. $p.f_i$) as the new p, p'' as the new p' and t' (resp. t_i) as the new t. By the definition of $[\![t]\!]_s^{\text{exp}}$, there is an argument v' such that v is the variant $v = A(v')$ (resp. a field v_i such that v is the record $v = \left\{\overline{f_i = v_i}^{i \in I}\right\}$). By definition of $\Downarrow^{\text{val}}$, we can go from $v' \Downarrow^{\text{val}} p'' \in [\![u]\!]_s^{\text{exp}}$ (resp. $v_i \Downarrow^{\text{val}} p'' \in [\![u]\!]_s^{\text{exp}}$) to the conclusion $t \Downarrow^{\text{val}} p' \in [\![u]\!]_s^{\text{exp}}$.

□

This will be used when proving the soundness of the transfer function for assignment for the NPR domain of section 4.2.

For the different objects defined in this paper, we write $\text{Env}(\bullet)$ for the set of extended variables that appear in them.

4.2 Numeric Domain Over Extended Variables: the NPR Domain

In this section, we define the *Numeric Path Relations lifting* NPR as a generic way to lift a domain N that is numeric—*i.e.*, that denotes sets of stores that map variables to numbers—to a domain that denotes sets of stores that map variables to *structured values* (definition 1 in section 2.1). The main idea is to use *extended variables* as the variables of the underlying numeric domain.

For a typing context Γ, the abstract values of $\text{NPR}(\text{N})(\Gamma)$ are pairs of a set E of extended variables that are valid in Γ, and a numeric abstract value from $\text{N}(E)$—*i.e.*, whose variables are the extended variables in E.

Definition 7 (Abstract values of the NPR domain)**.** *For any numeric domain satisfying the hypothesis of section 2.3 and any typing context* Γ*, the set of abstract values of the*

NPR domain is given by:

$$\mathrm{NPR}\,(\mathrm{N})(\Gamma) = \{(d, E) \mid E \in \mathcal{P}(\mathcal{E}(\Gamma)) \wedge d \in \mathrm{N}(E)\}$$

In this definition, an abstract numeric value d can refer to any extended variable $x.p$ declared in E, and does not need to reason on whether $x.p$ is a valid projection. In practice, though, the complete domain of section 5.4 will only consider sets E that are consistent. When writing examples in the rest of the paper, we will write any numeric abstract value d as a set of constraints, and we may omit the set E when it can be deduced from context, for example when E is exactly the set of extended variables used in d. For example, if we have as typing context $\Gamma = [\mathsf{st} \mapsto \mathsf{status}]$ — where status is the type defined in figure 4.2—, then we have $\{\mathsf{st@Asleep.secs} \geq 0;\ \mathsf{st@Asleep.count} \geq 0\} \in \mathrm{NPR}\,(\mathrm{N})(\Gamma)$.

Intuitively, an abstract value (d, E) denotes a set of stores that map regular variables to structured values, such that the paths listed in E point to integer values, and such that those integers are related by the numeric abstract value d. Using the projection function for values $\Downarrow^{\mathrm{val}}$ (definition 3 in section 2.1), it is easy to transform a store whose indices are variables into a store whose indices are *extended variables*:

Definition 8 (Projection of a store)**.** *The projection of a store* $s \in \mathrm{Vars} \to \mathrm{Values}$ *on a set of extended variables* $E \in \mathcal{P}(\mathcal{E})$ *is a store in* $E \to (\mathrm{Values} \cup \{\mathrm{Undef}\})$*, written* $s \Downarrow^{\mathrm{sto}} E$*, and is defined by:* $(s \Downarrow^{\mathrm{sto}} E)(x.p) = s(x) \Downarrow^{\mathrm{val}} p$.

As an example, we consider a store $s = [\mathsf{st} \mapsto \mathsf{Asleep}\{\mathsf{secs} = 42; \mathsf{count} = 7\}]$ — where a variable st has type status and holds a value built using constructor Asleep, for which the field secs has value 42 and the field count has value 7 — and a set of extended variables $E = \{\mathsf{st@Asleep.secs};\ \mathsf{st@Asleep.count}\}$. The projection of store s on the set of extended variables E is the function $[\mathsf{st@Asleep.secs} \mapsto 42;\ \mathsf{st@Asleep.count} \mapsto 7]$.

The concretisation of an element $(d, E) \in \mathrm{NPR}\,(\mathrm{N})(\Gamma)$ easily follows: it is the set of well-typed stores whose projections on E satisfy the numeric constraints d. The typing judgement $\Gamma \vdash s$ means that $s(x)$ has type $\Gamma(x)$ for every x.

Definition 9 (Concretisation for the NPR domain)**.** *The concretisation* $\gamma^{\mathrm{NPR(N)}(\Gamma)}(d, E)$ *of an abstract value* (d, E) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$ *is given by:*

$$\gamma^{\mathrm{NPR(N)}(\Gamma)}(d, E) = \left\{ s \mid \Gamma \vdash s \ \wedge\ s \Downarrow^{\mathrm{sto}} E \in \gamma^{\mathrm{N}(E)}(d) \right\}$$

For example, if the typing context is $\Gamma = [\mathsf{st} \mapsto \mathsf{status}]$, then the concretization of the

abstract value $n = \{\mathsf{st@Asleep.secs} \geq 0;\ \mathsf{st@Asleep.count} \geq 0\}$ contains any store in which variable st is mapped to a value of type status built using constructor Asleep and where both fields secs and count are non-negative. In particular,

$$[\mathsf{st@Asleep.secs} \mapsto 42;\ \mathsf{st@Asleep.count} \mapsto 7] \in \gamma^{\mathrm{NPR(N)(\Gamma)}}(n)$$

When projecting a store on a set of extended variables, some extended variables may map to Undef, if a variant does not use the constructor assumed by the path of the extended variable. However, in the stores of the concretization of a numeric abstract value, all variables map to a number. Hence, if a store s belongs to $\gamma^{\mathrm{NPR(N)(\Gamma)}}(d, E)$ for some set of extended variables E, this means that for any extended variable $x.p \in E$, the projection $s(x) \Downarrow^{\mathrm{val}} p$ is well defined. A consequence of this is that if a set of extended variables E is inconsistent (in the sense of definition 5), then the concretisation $\gamma^{\mathrm{NPR(N)(\Gamma)}}(d, E)$ will be empty for any numeric abstract value $d \in \mathrm{N}(E)$.

Lemma 3 (Concretisation is empty if the set of extended variables is inconsistent). *Let Γ be a typing context. Let $E \in \mathcal{E}(\Gamma)$ be an* inconsistent *set of extended variables. Let $d \in \mathrm{N}(E)$ be a numeric abstract value. We have $\gamma^{\mathrm{NPR(N)(\Gamma)}}(d, E) = \emptyset$.*

Proof sketch. Lemma 3 is proven by contradiction. Assume there exists a store $s \in \gamma^{\mathrm{NPR(N)(\Gamma)}}(d, E)$. Then, by definition of *inconsistent*, there would be two different constructors A and B, a variable x and paths p, q_1 and q_2 such that both $x.p@Aq_1$ and $x.p@Bq_2$ belong to E. Since $s \in \gamma^{\mathrm{NPR(N)(\Gamma)}}(d, E)$, it can be shown that both projections $s(x) \Downarrow^{\mathrm{val}} p@A$ and $s(x) \Downarrow^{\mathrm{val}} p@B$ must be well defined. This implies that the value $s(x) \Downarrow^{\mathrm{val}} p$ is both of the form $A(v)$ and $B(v')$ for some values v and v', which is impossible. □

Remark 1 (Extended variables as constraints)**.** *The stores in $\gamma^{\mathrm{NPR(N)(\Gamma)}}(d, E)$ are such that the paths in E are valid. In particular, a larger E means a more* constrained *—a more* precise*— abstract value. In a way, E acts as a set of constraints.*

For an example, we consider the typing context $\Gamma = [\mathsf{st} \mapsto \mathsf{status}]$ and two stores $s_1 = [\mathsf{st} \mapsto \mathsf{Asleep}\{\mathsf{secs} = 42; \mathsf{count} = 7\}]$ and $s_2 = [\mathsf{st} \mapsto \mathsf{Running}\{\mathsf{count} = 8\}]$. The NPR abstract value $n = (\emptyset, \top^{\mathrm{N}})$ — with no extended variables and no numeric constraints — has as concretisation all the stores that are well-typed with respect to Γ. In particular, $\{s_1, s_2\} \subseteq \gamma^{\mathrm{NPR(N)(\Gamma)}}(n)$. However, the NPR abstract value $n' = \left(\{\mathsf{st@Running.count}\}, \top^{\mathrm{N}}\right)$ — with extended variable st@Running.count and no numeric constraint — requires that

variable st is built using constructor Running. In particular $s_1 \notin \gamma^{\mathrm{NPR(N)(\Gamma)}}(n')$ and $s_2 \in \gamma^{\mathrm{NPR(N)(\Gamma)}}(n')$.

Definition 10 (Pre-order for the NPR domain)**.** *For any typing context* Γ *and any two NPR abstract values* (d_1, E_1) *and* (d_2, E_2) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$*, the relation* $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$ *is defined by*

$$(d_1, E_1) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2) \text{ iff } E_2 \subseteq E_1 \wedge \mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1) \sqsubseteq^{\mathrm{N}(E_2)} d_2$$

As seen in remark 1, extended variables behave like constraints. This explains why the definition of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$ requires that $E_2 \subseteq E_1$. In other words, the more precise abstract value (d_1, E_1) has a larger set of extended variables, than the less precise abstract value (d_2, E_2). The numeric abstract values d_1 and d_2 have different sets of extended variables. In order to compare them using the underlying numeric relation $\sqsubseteq^{\mathrm{N}}$, we first need to transform them so that they have the same set of extended variables. We could either add variables to d_2 to embed it in $\mathrm{N}(E_1)$, or remove variables from d_1 to embed it in $\mathrm{N}(E_2)$. In the definition of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$, we made the choice of removing variables from d_1 to embed it in $\mathrm{N}(E_2)$. This choice makes the proofs involving widening easier. Indeed, widening needs to relax constraints to be sound, and extended variables behave like constrains. Hence, widening needs to use variable removal; and the proofs are easier if widening and pre-order use the same operator.

Lemma 4 ($\gamma^{\mathrm{NPR(N)(\Gamma)}}$ is monotonic with respect to $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$). *For any typing context* Γ *and any two NPR abstract values* (d_1, E_1) *and* (d_2, E_2) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$*, we have*

$$\Big((d_1, E_1) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2)\Big) \Rightarrow \Big(\gamma^{\mathrm{NPR(N)(\Gamma)}}(d_1, E_1) \subseteq \gamma^{\mathrm{NPR(N)(\Gamma)}}(d_2, E_2)\Big)$$

Proof. We assume $(d_1, E_1) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2)$ and we need to prove $\gamma^{\mathrm{NPR(N)(\Gamma)}}(d_1, E_1) \subseteq \gamma^{\mathrm{NPR(N)(\Gamma)}}(d_2, E_2)$.

Let s be a store in $\gamma^{\mathrm{NPR(N)(\Gamma)}}(d_1, E_1)$. By the definition of $\gamma^{\mathrm{NPR(N)(\Gamma)}}$ we have

$$s \Downarrow^{\mathrm{sto}} E_1 \in \gamma^{\mathrm{N}(E_1)}(d_1)$$

Hence, by the soundness property of variable removal (hypothesis 1 on page 18), we have

$$s \Downarrow^{\mathrm{sto}} E_2 \in \gamma^{\mathrm{N}(E_2)}\Big(\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1)\Big)$$

Besides, by the definition of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$ we have

$$\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1) \sqsubseteq^{\mathrm{N}(E_2)} d_2$$

Therefore, the monotony of $\gamma^{\mathrm{N}(E_2)}$ with respect to $\sqsubseteq^{\mathrm{N}(E_2)}$ allows to conclude that

$$s \Downarrow^{\mathrm{sto}} E_2 \in \gamma^{\mathrm{N}(E_2)}(d_2)$$

In other words,

$$s \in \gamma^{\mathrm{NPR(N)(\Gamma)}}(d_2, E_2)$$

□

Lemma 5 ($\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$ is a pre-order). *For any typing context Γ, the relation $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$ from definition 10 is a pre-order.*

Proof. We will first prove that $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$ is reflexive, then that it is transitive.

Reflexivity of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$**.** Let $(d, E) \in \mathrm{NPR}\,(\mathrm{N})(\Gamma)$ be an abstract value. To prove that $(d, E) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d, E)$, we need to prove both $E \subseteq E$ and $\mathrm{Rem}^{\mathrm{N}(E)}_{E \setminus E}(d) \sqsubseteq^{\mathrm{N}(E)} d$. The former is immediate, since set inclusion is reflexive. The latter is easily proven thanks to hypothesis 6 on page 19 — that states that removing an empty set of variables does not have any effect on an abstract value — and the reflexivity of $\sqsubseteq^{\mathrm{N}}$.

Transitivity of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$**.** Let (d_1, E_1), (d_2, E_2) and (d_3, E_3) be three abstract values from $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$. In order to prove the transitivity of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$, we assume both $(d_1, E_1) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2)$ and $(d_2, E_2) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_3, E_3)$, and we need to prove that $(d_1, E_1) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_3, E_3)$. By unfolding the definition of $\sqsubseteq^{\mathrm{NPR(N)(\Gamma)}}$, the two things that we need to prove are $E_1 \subseteq E_3$ and $\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_3}(d_1) \sqsubseteq^{\mathrm{N}(E_3)} d_3$. The former follows from the transitivity of set inclusion. We will prove the latter using the monotony and composition assumptions of variable removal (hypotheses 4 and 5 on page 19). Indeed, the assumption $(d_1, E_1) \sqsubseteq^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2)$ gives us

$$\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1) \sqsubseteq^{\mathrm{N}(E_2)} d_2$$

By applying the monotony of variable removal (hypothesis 4) to the previous inequality, in the case of the removal of extended variables $E_2 \setminus E_3$, we get

$$\mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2 \setminus E_3}\left(\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1)\right) \sqsubseteq^{\mathrm{N}(E_3)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2 \setminus E_3}(d_2)$$

Besides, by applying the property about composition of variable removals (hypothesis 5) we get

$$\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_3}(d_1) \sqsubseteq^{\mathrm{N}(E_3)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_3}\left(\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_2}(d_1)\right)$$

Hence, by transitivity of $\sqsubseteq^{\mathrm{N}(E_3)}$, we have

$$\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_3}(d_1) \sqsubseteq^{\mathrm{N}(E_3)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_3}(d_2)$$

The assumption $(d_2, E_2) \sqsubseteq^{\mathrm{NPR(N)}(\Gamma)} (d_3, E_3)$ gives us $\mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_3}(d_2) \sqsubseteq^{\mathrm{N}(E_3)} d_3$ which, by using again the transitivity of $\sqsubseteq^{\mathrm{N}(E_3)}$, allows to conclude. □

The binary operators of the NPR domain (abstract intersection, abstract union and widening) are all defined following a common structure:

- they use either variable addition or variable removal to bring all numeric abstract values to a common set of extended variables,
- then they use the corresponding operator of the underlying numeric domain.

As explained in remark 1, the extended variables behave like constraints. In order to be sound, abstract union and widening can only keep the constraints that are *common* to the two NPR abstract values (d_1, E_1) and (d_2, E_2) being considered. Hence abstract union and widening have the *intersection* $E_1 \cap E_2$ as set of extended variables, and use variable removal to project the numeric abstract values d_1 and d_2 into $\mathrm{N}(E_1 \cap E_2)$.

Definition 11 (Abstract union for the NPR domain)**.** *For any typing context* Γ *and any two abstract values* (d_1, E_1) *and* (d_2, E_2) *from* $\mathrm{NPR}(\mathrm{N})(\Gamma)$*, the abstract union operator* $\sqcup^{\mathrm{NPR(N)}(\Gamma)}$ *is defined by*

$$(d_1, E_1) \sqcup^{\mathrm{NPR(N)}(\Gamma)} (d_2, E_2) = \left(\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_2}(d_1) \sqcup^{\mathrm{N}(E_1\cap E_2)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_1}(d_2), E_1 \cap E_2\right)$$

Lemma 6 (Soundness of abstract union for the NPR domain). *For any typing context* Γ *and any two abstract values* (d_1, E_1) *and* (d_2, E_2) *from* $\mathrm{NPR}(\mathrm{N})(\Gamma)$*, we have*

$$\gamma^{\mathrm{NPR(N)}(\Gamma)}(d_1, E_1) \cup \gamma^{\mathrm{NPR(N)}(\Gamma)}(d_2, E_2) \subseteq \gamma^{\mathrm{NPR(N)}(\Gamma)}\left((d_1, E_1) \sqcup^{\mathrm{NPR(N)}(\Gamma)} (d_2, E_2)\right)$$

Proof. Let s be a store in $\gamma^{\mathrm{NPR(N)}(\Gamma)}(d_1, E_1) \cup \gamma^{\mathrm{NPR(N)}(\Gamma)}(d_2, E_2)$. There exists $i \in \{1, 2\}$

such that $s \in \gamma^{\mathrm{NPR(N)(\Gamma)}}(d_i, E_i)$. By the definition of the NPR concretisation, we have

$$s \Downarrow^{\mathrm{sto}} E_i \in \gamma^{\mathrm{N}(E_i)}(d_i)$$

Let $j \in (\{1,2\} \setminus \{i\})$ be the index of the other abstract value. The abstract union uses the variable removal $\mathrm{Rem}^{\mathrm{N}(E_i)}_{E_i \setminus E_j}(d_i)$ to project d_i into $\mathrm{N}(E_1 \cap E_2)$. Indeed, we have

$$E_1 \setminus (E_1 \setminus E_2) = E_2 \setminus (E_2 \setminus E_1) = E_i \setminus (E_j \setminus E_i) = E_1 \cap E_2$$

and

$$\left(s \Downarrow^{\mathrm{sto}} E_i\right)\Big|_{E_i \setminus (E_j \setminus E_i)} = \left(s \Downarrow^{\mathrm{sto}} E_i\right)\Big|_{E_1 \cap E_2} = s \Downarrow^{\mathrm{sto}} (E_1 \cap E_2)$$

From the soundness property of variable removal (hypothesis 1 on page 18), we deduce

$$s \Downarrow^{\mathrm{sto}} (E_1 \cap E_2) \in \gamma^{\mathrm{N}(E_1 \cap E_2)} \left(\mathrm{Rem}^{\mathrm{N}(E_i)}_{E_i \setminus E_j}(d_i)\right)$$

Then, the soundness of $\sqcup^{\mathrm{N}(E_1 \cap E_2)}$ allows to conclude. □

Definition 12 (Widening operator for the NPR domain)**.** *For any typing context* Γ *and any two abstract values* (d_1, E_1) *and* (d_2, E_2) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$*, a widening operator* $\nabla^{\mathrm{NPR(N)(\Gamma)}}$ *is defined for the NPR domain by:*

$$(d_1, E_1) \nabla^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2) = \left(\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1) \nabla^{\mathrm{N}(E_1 \cap E_2)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2 \setminus E_1}(d_2), E_1 \cap E_2\right)$$

Lemma 7 (Soundness of the widening for the NPR domain). *For any typing context* Γ *the widening operator* $\nabla^{\mathrm{NPR(N)(\Gamma)}}$ *is sound* — i.e. *it computes upper-bounds and enforces convergence.*

Proof. We will first prove that the widening computes upper bounds; and then we will prove that it enforces convergence.

The widening computes upper bounds. Let (d_1, E_1) and (d_2, E_2) be two abstract values from $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$. We want to prove that

$$(d_1, E_1) \sqsubseteq^{\mathrm{NPR}} (d_1, E_1) \nabla^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2) \wedge (d_2, E_2) \sqsubseteq^{\mathrm{NPR}} (d_1, E_1) \nabla^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2)$$

Notice that both abstract values $\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1 \setminus E_2}(d_1)$ and $\mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2 \setminus E_1}(d_2)$ belong to $\mathrm{N}(E_1 \cap E_2)$ because $E_1 \setminus (E_1 \setminus E_2) = E_2 \setminus (E_2 \setminus E_1) = E_1 \cap E_2$. Given that the widening of the

underlying numeric domain computes upper-bounds, we have both

$$\mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_2}(d_1) \sqsubseteq^{\mathrm{N}(E_1\cap E_2)} \mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_2}(d_1) \triangledown^{\mathrm{N}(E_1\cap E_2)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_1}(d_2)$$

$$\mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_1}(d_2) \sqsubseteq^{\mathrm{N}(E_1\cap E_2)} \mathrm{Rem}^{\mathrm{N}(E_1)}_{E_1\setminus E_2}(d_1) \triangledown^{\mathrm{N}(E_1\cap E_2)} \mathrm{Rem}^{\mathrm{N}(E_2)}_{E_2\setminus E_1}(d_2)$$

Since we also have

$$E_1 \cap E_2 \subseteq E_1 \qquad E_1 \cap E_2 \subseteq E_2 \qquad E_1 \setminus E_2 = E_1 \setminus (E_1 \cap E_2) \qquad E_2 \setminus E_1 = E_2 \setminus (E_1 \cap E_2)$$

The definition of $\sqsubseteq^{\mathrm{NPR}}$ allows to conclude:

$$(d_1, E_1) \sqsubseteq^{\mathrm{NPR}} (d_1, E_1) \triangledown^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2) \qquad (d_2, E_2) \sqsubseteq^{\mathrm{NPR}} (d_1, E_1) \triangledown^{\mathrm{NPR(N)(\Gamma)}} (d_2, E_2)$$

Widening enforces convergence. We consider a sequence of abstract values $(d_n, E_n)_{n\in\mathbf{N}}$ from $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$. Let the sequence $(d'_n, E'_n)_{n\in\mathbf{N}}$ be the sequence defined by $(d'_0, E'_0) = (d_0, E_0)$ and $\forall n \in \mathbf{N}^*, (d'_n, E'_n) = (d'_{n-1}, E'_{n-1}) \triangledown^{\mathrm{NPR(N)(\Gamma)}} (d_n, E_n)$. Our goal is to prove that the sequence $(d'_n, E'_n)_{n\in\mathbf{N}}$ stabilizes in finite time.

We start by proving that $(E_n)_{n\in\mathbf{N}}$ stabilizes in finite time. The set E'_0 is finite, and each E'_n is a subset of the previous set of the sequence. Indeed:

$$\forall n \in \mathbf{N}^*, E'_n = E'_{n-1} \cap E_n \subseteq E'_{n-1}$$

Since there cannot be an infinitely decreasing sequence of finite sets, there exists an index $n_0 \in \mathbf{N}$ such that the sequence stabilizes at n_0:

$$\forall n \geq n_0, E'_n = E'_{n_0}$$

We now consider the sequence $(d'_n)_{n>n_0}$. Given the assumption on the removal of the empty set of variables (hypothesis 6), the definition of d'_n for $n > n_0$ can be simplified

$$\begin{array}{lrcl} \text{from} & d'_n & = & \mathrm{Rem}^{\mathrm{N}(E'_{n-1})}_{E'_{n-1}\setminus E_n}(d'_{n-1}) \;\; \triangledown^{\mathrm{N}(E'_{n_0})} \;\; \mathrm{Rem}^{\mathrm{N}(E_n)}_{E_n\setminus E'_{n-1}}(d_n) \\ \text{to} & d'_n & = & d'_{n-1} \qquad\qquad\qquad\qquad\quad \triangledown^{\mathrm{N}(E'_{n_0})} \;\; \mathrm{Rem}^{\mathrm{N}(E_n)}_{E_n\setminus E'_{n-1}}(d_n) \end{array}$$

Since the numeric widening $\triangledown^{\mathrm{N}(E'_{n_0})}$ enforces convergence, we know that the sequence $(d'_n)_{n>n_0}$ stabilizes in finite time. Hence the sequence $(d'_n, E'_n)_{n\in\mathbf{N}}$ stabilizes in finite time.

□

By contrast with abstract union and widening, the abstract intersection can keep any constraint present in either (d_1, E_1) or (d_2, E_2). Hence the set of extended variables of the abstract intersection is $E_1 \cup E_2$, and variable addition injects the numeric abstract values into $\mathrm{N}(E_1 \cup E_2)$.

Definition 13 (Abstract intersection for the NPR domain)**.** *For any typing context* Γ *and any two abstract values* (d_1, E_1) *and* (d_2, E_2) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$*, the abstract intersection operator* $\sqcap^{\mathrm{NPR(N)}(\Gamma)}$ *is defined by:*

$$(d_1, E_1) \sqcap^{\mathrm{NPR(N)}(\Gamma)} (d_2, E_2) = \left(\mathrm{Add}_{E_2}^{\mathrm{N}(E_1)}(d_1) \sqcap^{\mathrm{N}(E_1 \cup E_2)} \mathrm{Add}_{E_1}^{\mathrm{N}(E_2)}(d_2), E_1 \cup E_2\right)$$

Lemma 8 (Soundness of abstract intersection for the NPR domain). *For any typing context* Γ *and any two NPR abstract values* (d_1, E_1) *and* (d_2, E_2)*, we have*

$$\gamma^{\mathrm{NPR(N)}(\Gamma)}(d_1, E_1) \cap \gamma^{\mathrm{NPR(N)}(\Gamma)}(d_2, E_2) \subseteq \gamma^{\mathrm{NPR(N)}(\Gamma)} \left((d_1, E_1) \sqcap^{\mathrm{NPR(N)}(\Gamma)} (d_2, E_2)\right)$$

Proof. Let s be a store in $\gamma^{\mathrm{NPR(N)}(\Gamma)}(d_1, E_1) \cap \gamma^{\mathrm{NPR(N)}(\Gamma)}(d_2, E_2)$. Both projections $s \Downarrow^{\mathrm{sto}} E_1$ and $s \Downarrow^{\mathrm{sto}} E_2$ are well defined, in the sense that they do not map any extended variable into Undef. Hence, the projection $s \Downarrow^{\mathrm{sto}} (E_1 \cup E_2)$ is well-defined as well. Hence the soundness property of variable addition (hypothesis 2 on page 18) allows us to go from

$$s \Downarrow^{\mathrm{sto}} E_1 \in \gamma^{\mathrm{N}(E_1)}(d_1) \wedge s \Downarrow^{\mathrm{sto}} E_2 \in \gamma^{\mathrm{N}(E_2)}(d_2)$$

to

$$s \Downarrow^{\mathrm{sto}} (E_1 \cup E_2) \in \gamma^{\mathrm{N}(E_1 \cup E_2)} \left(\mathrm{Add}_{E_2}^{\mathrm{N}(E_1)}(d_1)\right) \wedge s \Downarrow^{\mathrm{sto}} (E_1 \cup E_2) \in \gamma^{\mathrm{N}(E_1 \cup E_2)} \left(\mathrm{Add}_{E_1}^{\mathrm{N}(E_2)}(d_2)\right)$$

Hence, by soundness of $\sqcap^{\mathrm{N}(E_1 \cup E_2)}$, we have

$$s \Downarrow^{\mathrm{sto}} (E_1 \cup E_2) \in \gamma^{\mathrm{N}(E_1 \cup E_2)} \left(\mathrm{Add}_{E_2}^{\mathrm{N}(E_1)}(d_1) \sqcap^{\mathrm{N}(E_1 \cup E_2)} \mathrm{Add}_{E_1}^{\mathrm{N}(E_2)}(d_2)\right)$$

In order words,

$$s \in \gamma^{\mathrm{NPR(N)}(\Gamma)} \left(\mathrm{Add}_{E_2}^{\mathrm{N}(E_1)}(d_1) \sqcap^{\mathrm{N}(E_1 \cup E_2)} \mathrm{Add}_{E_1}^{\mathrm{N}(E_2)}(d_2), E_1 \cup E_2\right)$$

which allows to conclude. □

Transfer Functions The transfer function for assignment $x := t$ works by temporarily introducing a new variable x' (that represents the value of x *after* assignment). First, it applies the transfer function for assignment on every numeric assignment in the decomposition of $x' := t$ (definition 6). Then, it removes the references to the paths of x, and finally renames x' into x. The auxiliary variable x' is introduced to avoid clashes between the paths that are valid for x *before* the assignment and those that are valid *after* the assignment.

Definition 14 (Transfer function for assignment in the NPR domain)**.** *For any typing context* Γ*, any assignment* $x := t$*, and any abstract value* (d, E) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$*, the* transfer function for assignment *in the NPR domain is given by*

$$\mathrm{Assign}^{\mathrm{NPR(N)}(\Gamma)}(x := t)(d, E) \;=\; \mathrm{Rename}^{\mathrm{NPR(N)}(\Gamma_2)}_{[x' \mapsto x]}\left(\mathrm{Rem}^{\mathrm{NPR(N)}(\Gamma_1)}_{\{x\}}(d_1, E_1)\right)$$

$$\textit{where} \begin{cases} x' \in \mathrm{Fresh}\,(\mathrm{dom}(\Gamma)) \\ d_1 = \displaystyle\prod^{\mathrm{N}(E_1)}_{x'.p:=u \in \mathrm{Decomp}(x':=t), \Gamma \vdash u:\mathrm{Int}} \mathrm{Assign}^{\mathrm{N}(E_1)}(x'.p := u)(\mathrm{Add}^{\mathrm{N}(E)}_{E_0}(d)) \\ E_1 = E \cup E_0 \\ E_0 = \{y.p \in \mathrm{Env}\,(\mathrm{Decomp}\,(x' := t)) \mid \Gamma \vdash y.p : \mathrm{Int}\} \\ \Gamma_1 = \Gamma(x' \mapsto \Gamma(x)) \\ \Gamma_2 = \Gamma_1|_{(\mathrm{dom}(\Gamma_1) \setminus \{x\})} \end{cases}$$

Lemma 9 (Soundness of the transfer function for assignment in the NPR domain). *For any assignment* $x := t$ *and any abstract value* (d, E) *from* $\mathrm{NPR}\,(\mathrm{N})(\Gamma)$*, we have*

$$\left\{s(x \mapsto v) \mid s \in \gamma^{\mathrm{NPR(N)}(\Gamma)}(d, E) \wedge v \in [\![t]\!]^{\mathrm{exp}}_s\right\} \subseteq \mathrm{Assign}^{\mathrm{NPR(N)}(\Gamma)}(x := t)(d, E)$$

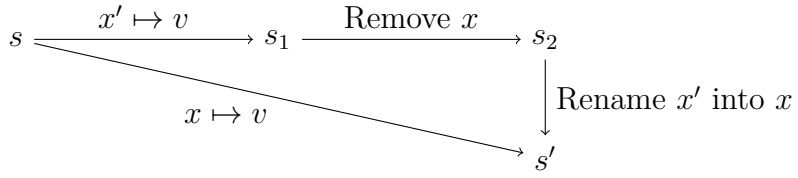
Proof. Let s' be a store in

$$\left\{s(x \mapsto v) \mid s \in \gamma^{\mathrm{NPR(N)}(\Gamma)}(d, E) \wedge v \in [\![t]\!]^{\mathrm{exp}}_s\right\}$$

By definition, we know that there exists some store $s \in \gamma^{\mathrm{NPR(N)}(\Gamma)}(d, E)$ such that expression t evaluates to some value $v \in [\![t]\!]^{\mathrm{exp}}_s$ and $s' = s(x \mapsto v)$.

The definition of the transfer function for assignment $\mathrm{Assign}^{\mathrm{NPR(N)}(\Gamma)}(x := t)(\centerdot)$ works by introducing a fresh variable x', analysing the assignment as if it was $x' := t$, forgetting about variable x, and then renaming x' into x. In this proof, we introduce intermediate

stores that follow this same logic. Let $s_1 = s(x' \mapsto v)$ be the store obtained from s by mapping x' into v (the value t evaluates to). Let $s_2 = s_1|_{\mathrm{dom}(s_1)\setminus\{x\}}$ the store obtained from s_1 by removing the binding of variable x. From s_2, if we rename x' into x, we get back the store s' previously defined, as summarized in the following diagram:



As in the definition of $\mathrm{Assign}^{\mathrm{NPR(N)(\Gamma)}}(\centerdot)(\centerdot)$, we call

$$E_0 = \{y.p \in \mathrm{Env}\,(\mathrm{Decomp}\,(x' := t)) \mid \Gamma \vdash y.p : \mathrm{Int}\}$$

the set of extended variables with a numeric type that appear (either on the left-hand side or the right-hand side) in the decomposition of assignment $x' := t$. We also call $E_1 = E \cup E_0$ the set of all the extended variables that should be considered before the removal of x. Given that s and s_1 only differ on x' and that x' does not appear in E, we have $s_1 \Downarrow^{\mathrm{sto}} E = s \Downarrow^{\mathrm{sto}} E$, hence $s_1 \Downarrow^{\mathrm{sto}} E \in \gamma^{\mathrm{N}(E)}(d)$. Together with the fact that $(s_1 \Downarrow^{\mathrm{sto}} E_1)|_E = s_1 \Downarrow^{\mathrm{sto}} E$, this allows to use the soundness of variable addition (hypothesis 2 on page 18) to deduce that

$$s_1 \Downarrow^{\mathrm{sto}} E_1 \in \gamma^{\mathrm{N}(E_1)}\left(\mathrm{Add}_{E_0}^{\mathrm{N}(E)}(d)\right)$$

Let's prove that the store s_1 that we have defined belongs to the concretisation of the abstract value d_1. We recall the definition of d_1:

$$d_1 = \prod_{x'.p:=u \in \mathrm{Decomp}(x':=t),\Gamma\vdash u:\mathrm{Int}}^{\mathrm{N}(E_1)} \mathrm{Assign}^{\mathrm{N}(E_1)}(x'.p := u)(\mathrm{Add}_{E_0}^{\mathrm{N}(E)}(d))$$

Let $x'.p := u$ be an element of the assignment decomposition $\mathrm{Decomp}\,(x' := t)$. If we evaluate $s_1 \Downarrow^{\mathrm{sto}} E_1$ on $x'.p$, that yields

$$\left(s_1 \Downarrow^{\mathrm{sto}} E_1\right)(x'.p) = s_1(x') \Downarrow^{\mathrm{val}} p = v \Downarrow^{\mathrm{val}} p$$

By the soundness of assignment decomposition (lemma 2 on page 34), we know that $v \Downarrow^{\mathrm{val}} p \in [\![u]\!]^{\mathrm{exp}}_{s'}$. Given that s' and s only differ on x', and x' does not appear in u, we

have $v \Downarrow^{\text{val}} p \in [\![u]\!]^{\text{exp}}_{s_1}$. Using the soundness of the transfer function for assignment of the underlying numeric domain, we deduce that

$$s_1 \Downarrow^{\text{sto}} E_1 \in \gamma^{\text{N}(E_1)}\left(\text{Assign}^{\text{N}(E_1)}(x'.p := u)(\text{Add}^{\text{N}(E)}_{E_0}(d))\right)$$

This being true for all $x'.p := u \in \text{Decomp}\,(x' := t)$, we deduce, by soundness of the numeric abstract intersection, that $s_1 \Downarrow^{\text{sto}} E_1 \in \gamma^{\text{N}(E_1)}(d_1)$; therefore $s_1 \in \gamma^{\text{NPR(N)}(\Gamma_1)}(d_1, E_1)$. By the soundness of variable removal,

$$s_2 \in \gamma^{\text{NPR(N)}(\Gamma_2)}\left(\text{Rem}^{\text{NPR(N)}(\Gamma_1)}_{\{x\}}(d_1, E_1)\right)$$

We have $s'(x) = s_2(x')$ and s' and s_2 are identical on every variable other than x and x'. By the soundness of variable renaming, $s \in \text{Assign}^{\text{NPR(N)}(\Gamma)}(x := t)(d, E)$. □

The transfer function for conditionals is simpler: it suffices to add to the extended variables those that occur in the test, and to call the transfer function of domain N for conditionals.

Definition 15 (Transfer function for conditions in the NPR domain)**.** *For any typing context* Γ*, any boolean condition* $b \in \text{BExp}$ *and any NPR abstract value* $(d, E) \in \text{NPR}\,(\text{N})(\Gamma)$*; the transfer function for conditions of the NPR domain is defined by*

$$\text{Cond}^{\text{NPR(N)}(\Gamma)}(b)(d, E) = \left(\text{Cond}^{\text{N}(E \cup \text{Env}(b))}(b)(\text{Add}^{\text{N}(E)}_{\text{Env}(b)}(d)), E \cup \text{Env}\,(b)\right)$$

Lemma 10 (Soundness of the transfer function for conditions). *For any boolean condition* $b \in \text{BExp}$ *and any abstract value* (d, E) *from* $\text{NPR}\,(\text{N})(\Gamma)$*, we have*

$$\left\{s \in \gamma^{\text{NPR(N)}(\Gamma)}(d, E) \mid \mathtt{tt} \in [\![b]\!]^{\text{bool}}_s\right\} \subseteq \gamma^{\text{NPR(N)}(\Gamma)}\left(\text{Cond}^{\text{NPR(N)}(\Gamma)}(d, E)(b)\right)$$

Proof. Let s be a store in $\gamma^{\text{NPR(N)}(\Gamma)}(d, E)$ for which the boolean condition b evaluates to true — *i.e.* $\mathtt{tt} \in [\![b]\!]^{\text{bool}}_s$. Since $s \in \gamma^{\text{NPR(N)}(\Gamma)}(d, E)$, we have

$$s \Downarrow^{\text{sto}} E \in \gamma^{\text{N}(E)}(d)$$

In addition, according to lemma 1 on page 16, all the extended variables in $\text{Env}\,(b)$ are

valid for store s, in the sense that

$$\forall x.p \in \mathrm{Env}\,(b)\,, s(x) \Downarrow^{\mathrm{val}} p \neq \mathrm{Undef}$$

Hence, by the soundness of variable addition (hypothesis 2 on page 18), we have

$$s \Downarrow^{\mathrm{sto}} (E \cup \mathrm{Env}\,(b)) \in \gamma^{\mathrm{N}(E \cup \mathrm{Env}(b))} \left(\mathrm{Add}^{\mathrm{N}(E)}_{\mathrm{Env}(b)}(d)\right)$$

Then, by the soundness of abstract conditions for the underlying numeric domain $\mathrm{N}(E \cup \mathrm{Env}\,(b))$, we have

$$s \Downarrow^{\mathrm{sto}} (E \cup \mathrm{Env}\,(b)) \in \gamma^{\mathrm{N}(E \cup \mathrm{Env}(b))} \left(\mathrm{Cond}^{\mathrm{N}(E \cup \mathrm{Env}(b))}(b) \left(\mathrm{Add}^{\mathrm{N}(E)}_{\mathrm{Env}(b)}(d)\right)\right)$$

which allows to conclude. □

We could use the NPR abstract domain to analyse programs that manipulate algebraic types. This however, would not yield precise results, because it would loose any information about values that can use different constructors. For example, in figure 4.2 on page 31, the status of the process manipulated by function `do_ticks` can be either `Running` or `Asleep`, hence the NPR domain alone cannot capture any information about this status. To have a more precise analysis, we introduce in the next chapter a disjunctive completion, where the different disjuncts can talk about different constructors for the same variable. In order to control the size of the disjunction and improve scalability, we merge disjuncts that that talk about equivalent extended variables, and we introduce a domain for talking about equality of two structured values.

Chapter 5

Improving Precision and Scalability for Algebraic Types Analysis

This chapter improves on the precision of Chapter 4 by introducing a disjunctive completion. This disjunctive completion allows to store in different disjuncts the information about different constructors of variant values. It poses, however, the challenge to keep the number of disjuncts small, in order to maintain scalability. Two different things are done to help keep the number of disjuncts small:

- Merge together the disjuncts that impose equivalent constraints on which constructors are used.
- Introduce a domain for equalities between structured values. As we will see, this domain avoids some disjunctions.

Section 5.1 introduces the domain of *Constructor Constraints*, which fully formalizes remark 1 on page 37: extended variables behave as constraints, as they impose the use of certain constructors for the values from sum types. The Constructor Constraints domain from section 5.1 will be used to determine when to merge disjuncts in the disjunctive completion. Section 5.2 introduces the domain of *Structural Equalities*, that can express equalities between structured values. This domain of structural equalities helps to keep the number of disjuncts in the disjunction small. Section 5.3 describes the reduced product between the Constructor Constraints domain of section 5.1, the Structural Equalities domain of section 5.2 and the NPR lifting of section 4.2. This product domain is called Pan for *Product domain for Algebraic types and Numbers*. Section 5.4 presents how we take a disjunctive completion of the Pan domain. Disjuncts are merged if they have equivalent constructor constraints.

5.1 Constructor Constraints

We introduce the abstract domain of *constructor constraints*, that intuitively describes in which *cases* the values of a store might be, *i.e.*, which are the allowed variant constructors of the values of a store. We write $\mathrm{cc}(\Gamma)$ for the set of constructor constraints for a typing environment Γ. An element $c \in \mathrm{cc}(\Gamma)$ is a set of extended variables, that restricts the possible sets of stores to those that are compatible with *every* path in c. In other words, if a path in c mentions some constructor, then the corresponding value in any store of the concretisation must be built using that constructor. Constructor constraints are a key ingredient of the disjunctive completion of section 5.4, as they serve as hints for which disjuncts need to be kept separate, and which should be merged.

An element $c \in \mathrm{cc}(\Gamma)$ is either the bottom value $\bot^{\mathrm{cc}(\Gamma)}$, or must be a set of extended variables that is both *consistent* and *closed under the prefix order* $\preccurlyeq$.

Definition 16 (Constructor constraints)**.** *The domain of constructor constraints is defined by* $\mathrm{cc}(\Gamma) = \{c \subseteq \mathcal{E}(\Gamma) \mid c \text{ is } \preccurlyeq\text{-closed and consistent }\} \cup \{\bot^{\mathrm{cc}(\Gamma)}\}$ *and is equipped with the ordering* $\sqsubseteq^{\mathrm{cc}(\Gamma)}$ *defined as* $c_1 \sqsubseteq^{\mathrm{cc}(\Gamma)} c_2$ *iff* $c_1 = \bot^{\mathrm{cc}(\Gamma)}$ *or* $c_1 \supseteq c_2$.

For any set of extended variables E, we write $\mathrm{clos}^{\mathrm{cc}(\Gamma)}(E)$ to denote the prefix-closure of E, *i.e.*, the smallest $\preccurlyeq$-closed set that contains E. For example, if we consider the type status from figure 4.2 defined by

```
type status = [Running of { count: int } | Asleep of { secs: int; count: int }]
```

and the typing context $\Gamma = [x \mapsto \mathsf{status}]$ where variable x has type status, then the prefix-closure of the singleton $E = \{x@\mathsf{Asleep.secs}\}$ is the set $\{x@\mathsf{Asleep.secs}, x@\mathsf{Asleep}, x\}$. For a given Γ, the domain $\mathrm{cc}(\Gamma)$ is *finite*: because our types are not recursive, the valid paths necessarily have finite lengths.

The concretisation $\gamma^{\mathrm{cc}(\Gamma)}$ defines the stores denoted by constructor constraints.

Definition 17 (Concretisation for constructor constraints)**.**

$$\gamma^{\mathrm{cc}(\Gamma)}(\bot^{\mathrm{cc}(\Gamma)}) = \emptyset \qquad \gamma^{\mathrm{cc}(\Gamma)}(c) = \{s \mid \Gamma \vdash s \wedge \forall x.p \in c, s(x) \Downarrow^{\mathrm{val}} p \neq \mathrm{Undef}\}$$

The concretisation of a set c produces a set of well-typed stores such that the values in the stores can be projected along the paths in c. For example, the concretization of the abstract value $c = \{x@\mathsf{Asleep.secs}, x@\mathsf{Asleep}, x\}$ contains all the stores in which variable x

holds a value built with constructor Asleep. In particular, the store

$$s_1 = [\mathsf{st} \mapsto \mathsf{Asleep}\{\mathsf{secs} = 42; \mathsf{count} = 7\}]$$

belongs to $\gamma^{\mathrm{cc}(\Gamma)}(c)$; whereas the store $s_2 = [\mathsf{st} \mapsto \mathsf{Running}\{\mathsf{count} = 8\}]$ does not.

The abstract union and intersection for the $\mathrm{cc}(\Gamma)$ domain are easily obtained:

$$\bot^{\mathrm{cc}(\Gamma)} \sqcup^{\mathrm{cc}(\Gamma)} c = c \sqcup^{\mathrm{cc}(\Gamma)} \bot^{\mathrm{cc}(\Gamma)} = c \qquad c_1 \sqcup^{\mathrm{cc}(\Gamma)} c_2 = c_1 \cap c_2 \text{ otherwise}$$

$$c_1 \sqcap^{\mathrm{cc}(\Gamma)} c_2 = \begin{cases} \bot^{\mathrm{cc}(\Gamma)} & \text{if } c_1 = \bot^{\mathrm{cc}(\Gamma)} \text{ or } c_2 = \bot^{\mathrm{cc}(\Gamma)} \text{ or } c_1 <> c_2 \\ c_1 \cup c_2 & \text{otherwise} \end{cases}$$

Because the domain is finite, there is no issue with infinite ascending chains, and we can simply define the widening as the abstract union.

Transfer Functions We express the abstract transfer function for assignment in the $\mathrm{cc}(\Gamma)$ domain in a standard *"kill-gen"* form as follows:

$$\begin{aligned} &\mathrm{Assign}^{\mathrm{cc}(\Gamma)}(x := t)(c) \;=\; (c \setminus \mathrm{Kill}^{\mathrm{cc}(\Gamma)}(x)(c)) \sqcap^{\mathrm{cc}(\Gamma)} \mathrm{Gen}^{\mathrm{cc}(\Gamma)}(x := t)(c) \\ &\text{where } \mathrm{Kill}^{\mathrm{cc}(\Gamma)}(x)(c) \;=\; \{y.p \in c \mid y = x\} \\ &\text{and } \mathrm{Gen}^{\mathrm{cc}(\Gamma)}(x := t)(c) \;= \\ &\qquad \mathrm{clos}^{\mathrm{cc}(\Gamma)}(\{y.q \in \mathrm{Env}\,(t) \mid y \neq x\} \cup \{x.p \mid \exists t', x.p := t' \in \mathrm{Decomp}\,(x := t)\}) \end{aligned}$$

The extended variables that must be removed are those that have x as root, since the value stored in variable x changes, and the old projection paths are not necessarily valid for the new value. Two kind of extended variables are added:

- The extended variables that appear in the expression t, as long as the variable is different from x. Indeed, the assignment $x := t$ only succeeds for the stores in which all the extended variables mentioned in t are valid before the assignment. As x is the only variable that changes, all the extended variables in $\{y.q \in \mathrm{Env}\,(t) \mid y \neq x\}$ are valid after the assignment.

- The extended variables that are valid for the new value of variable x. In particular, all the extended variables in $\{x.p \mid \exists t', x.p := t' \in \mathrm{Decomp}\,(x := t)\}$ —*i.e.* the extended variables appearing on the left-hand side of the assignment decomposition— are valid after the assignment.

We ensure that the added variables remain prefix-closed thanks to a call to $\text{clos}^{\text{cc}(\Gamma)}$.

The transfer function for conditionals is straightforward: it adds the extended variables of the boolean condition:

$$\text{Cond}^{\text{cc}(\Gamma)}(b)(c) = c \sqcap^{\text{cc}(\Gamma)} \text{clos}^{\text{cc}(\Gamma)}(\text{Env}\,(b))$$

The domain for constructor constraints introduced in this section will prove useful in the disjunctive completion of section 5.4, as we use it to determine when to merge disjuncts.

5.2 Structural Equalities

The NPR lifting of section 4.2 can only express relations between the *numeric* parts of values. It can't record whether some non-numeric part of a value has not changed. In our example of figure 4.2 on page 31, this is the case of the `msg` field of processes, that is not modified, and is of record type. We introduce in this section the domain $\text{seq}(\Gamma)$, that tracks *structural equalities*. The domain $\text{seq}(\Gamma)$ tells which parts of the values of a store *must* be identical.

One could argue that any equality between structured values could be replaced with a conjunction of equalities between the integer fields of those values, and, consequently, that the $\text{seq}(\Gamma)$ domain is hardly useful. Such a decomposition could lead, however, to more verbose abstract values, and could also introduce extra disjunctions when dealing with values of sum types. Thus, our choice of handling equality constraints between structured values is beneficial, as it helps keep our abstract values small in size.

We give here a simplified definition of the domain, where the abstract values of $\text{seq}(\Gamma)$ are either the bottom element $\bot^{\text{seq}(\Gamma)}$ —denoting the empty set of stores— or a finite set e of pairs of extended variables $(x.p, y.q)$ —denoting a set of stores s in which the value at path p in $s(x)$ is equal to the one at path q in $s(y)$. In practice, our implementation uses a map from extended variables to equivalence class indices, to ensure we remain closed by reflexivity, symmetry and transitivity.

Definition 18 (Domain of structural equalities)**.** *The domain of structural equalities* $\text{seq}(\Gamma) = \mathcal{P}(\mathcal{E}(\Gamma) \times \mathcal{E}(\Gamma)) \cup \{\bot^{\text{seq}(\Gamma)}\}$ *is equipped with the concretisation function* $\gamma^{\text{seq}(\Gamma)} \in$

$\text{seq}(\Gamma) \to \mathcal{P}(\text{Vars} \to \text{Values})$ *that is defined as follows:*

$$\gamma^{\text{seq}(\Gamma)}(\bot^{\text{seq}(\Gamma)}) = \emptyset$$

$$\gamma^{\text{seq}(\Gamma)}(e) = \{s \mid \Gamma \vdash s \wedge \forall (x.p, y.q) \in e,\ s(x) \Downarrow^{\text{val}} p = s(y) \Downarrow^{\text{val}} q \neq \text{Undef}\}$$

Abstract values in this domain might carry some implicit information. For example, if x and y have type $\{f \to \text{Int}; g \to \text{Int}\}$, the abstract value $\{(x, y)\}$ also *implicitly* implies that $x.f = y.f$ and $x.g = y.g$. To avoid losing precision, it is sometimes necessary to *saturate* an abstract value by congruence, so that it contains all the valid equalities that mention *a given set of extended variables*. For this purpose, we define the following closure operator.

Definition 19 (Closure of structural equalities)**.** *The* closure *of a set of structural equalities e with respect to a set of extended variables E, written* $\text{clos}_E^{\text{seq}(\Gamma)}(e)$*, is the smallest set that is larger than e, that mentions the variables in E, is closed under symmetry, reflexivity and transitivity, and satisfies the following congruence property:*

$$\left.\begin{array}{l}(x.p, y.q) \in \text{clos}_E^{\text{seq}(\Gamma)}(e) \\ (x.pr) \in \text{Env}\left(\text{clos}_E^{\text{seq}(\Gamma)}(e)\right)\end{array}\right\} \Rightarrow (x.pr, y.qr) \in \text{clos}_E^{\text{seq}(\Gamma)}(e)$$

The need for a closure operator is not surprising, as it occurs in other relational domains, like octagons [Min06]. We use this closure operator to gain precision in the transfer function for assignment, and in the reduction operator of the product domain of section 5.3.

Transfer Functions The transfer function for assignment $x := t$ for the $\text{seq}(\Gamma)$ domain exploits the decomposition of assignments from definition 6. It considers only the assignments of the form $x.p := y.q$, where the right-hand side is an extended variable. We express the transfer function in a *"kill-gen"* form, where we kill every equality that involves x, and add the new equalities $x.p = y.q$ where we are careful to avoid any use of x that refers to the value *before* the assignment.

$$\begin{aligned}
&\mathrm{Assign}^{\mathrm{seq}(\Gamma)}(x := t)(e) = \Big(e \setminus \mathrm{Kill}^{\mathrm{seq}(\Gamma)}(x)(e)\Big) \cup \mathrm{Gen}^{\mathrm{seq}(\Gamma)}(x := t)(e)\\
&\text{where } \mathrm{Kill}^{\mathrm{seq}(\Gamma)}(x)(e) = \{(y.p, z.q) \in e \mid y = x \vee z = x\}\\
&\text{and } \mathrm{Gen}^{\mathrm{seq}(\Gamma)}(x := t)(e) =\\
&\qquad \bigcup_{x.p:=y.q \in \mathrm{Decomp}(x:=t)} \{(x.p, z.r) \mid z \neq x \wedge (y.q, z.r) \in \mathrm{clos}^{\mathrm{seq}(\Gamma)}_{\{y.q\}}(e)\}
\end{aligned}$$

The transfer functions for conditionals can only exploit equality tests between extended variables: $\mathrm{Cond}^{\mathrm{seq}(\Gamma)}(b)(e) = e \sqcap^{\mathrm{seq}(\Gamma)} \{(x.p, y.q)\}$ if b is $x.p = y.q$.

5.3 Reduced Product

We call $\mathrm{Pan}\,(\mathrm{N}) = \mathrm{cc} \times \mathrm{seq} \times \mathrm{NPR}\,(\mathrm{N})$ the product of the constructor constraints domain, the structural equalities domain and the NPR lifting. Pan stands for *Product domain for Algebraic types and Numbers*. We equip Pan with a reduction operator ρ, that enables information transfer between the different domains of the product.

Definition 20 (Reduction operator)**.** *The reduction operator ρ for the product of constructor constraints, structural equalities and the NPR lifting is defined as follows:*

$$\begin{aligned}
&\rho(c, e, n) = \begin{pmatrix} c \sqcap^{\mathrm{cc}(\Gamma)} \mathrm{clos}^{\mathrm{cc}(\Gamma)}(\mathrm{Env}\,(e')),\\ e',\\ \mathrm{Cond}^{\mathrm{NPR}(\mathrm{N})(\Gamma)}(\bigwedge_{(x.p,y.q)\in e' \wedge \Gamma \vdash x.p:\mathrm{Int}} x.p = y.q)(n) \end{pmatrix}\\
&\textit{where } e' = \mathrm{clos}^{\mathrm{seq}(\Gamma)}_{\mathrm{clos}^{\mathrm{cc}(\Gamma)}(\mathrm{Env}((c,e,n)))}(e)
\end{aligned}$$

The reduction operator ρ transfers the following pieces of information between the three components of the product:

- Structural equalities are completed with additional constraints, so that all the extended variables that are used in the constructor constraints and the numeric constraints are mentioned (this is the role of e').
- If some equalities between integers are deduced from the structural equalities, then they are added to the numeric constraints.
- The extended variables from the structural equalities and the numeric constraints are added to the constructor constraints, which may reveal some inconsistent cases.

Union, intersection and widening for the reduced product domain add variables to the structural equalities component, use component-wise operations and use the reduction operator. For widening, reduction is only applied to the right-hand side argument to avoid interfering with convergence. We invite the reader to look at [BJM22c] for further details. The transfer functions for assignment and conditionals use the transfer functions of each component.

5.4 Disjunctive completion

Pattern matching performs a case analysis on the different constructors a value may start with: these cases are pairwise incompatible. To analyse pattern matching with precision, we add disjunctions to our abstract domain by means of a disjunctive completion, so that each pattern matching case has a distinct disjunct. Hence, for any numeric domain N, we take the disjunctive completion [Cou21] of the reduced product of constructor constraints, structural equalities and the NPR lifting of N. We call this the *structural lifting* of N, written S, and defined as $\mathrm{S}(\mathrm{N}) = \mathcal{P}(\mathrm{Pan}(\mathrm{N}))$. To control the number of disjuncts, however, we *merge* some cases together: merging is performed when the constructor constraints of two abstract values concretise to the same sets of stores—*i.e.*, when they impose the same constraints on the constructors used for variant values. Definition 21 defines what it means for two sets of constructor constraints to be equivalent. This definition is used in definition 22 to describe how, for the elements of the disjunctive completion, cases with equivalent constructor constraints are merged together, using abstract union.

Definition 21 (Equivalence of constructor constraints)**.** *Two sets of constructor constraints c_1 and c_2 are said to be* equivalent*, written $c_1 \equiv c_2$, if any path* ending with a constructor *present in one of them is also present in the other. Formally:*

$$c_1 \equiv c_2 \quad \textit{iff} \quad \forall x.(p@A) \in c_1, x.(p@A) \in c_2 \wedge \forall y.(q@B) \in c_2, y.(q@B) \in c_1$$

$\equiv$ is an equivalence relation for constructor constraints.

We can extend this into an equivalence relation for elements of the structural lifting, by:

$$(c_1, e_1, n1) \equiv_{\mathrm{E}}^{\mathrm{Pan}} (c_2, e_2, n2) \ \textit{iff}\ c_1 \equiv c_2$$

This equivalence only considers the extended variables, and does not correspond to an equivalence with respect to $\sqsubseteq^{\mathrm{Pan}}$ nor to a semantic equivalence. The sub-script in the

notation $\equiv_{\mathrm{E}}^{\mathrm{Pan}}$ *reminds of that.*

Given an element d *of* S*, we write* $d/\equiv_{\mathrm{E}}^{\mathrm{Pan}}$ *the set of equivalence classes of* d *with respect to* $\equiv_{\mathrm{E}}^{\mathrm{Pan}}$*.*

This notion of equivalence allows us to define an operator that collapses together equivalent triplets in an element of $\mathcal{P}(\mathrm{Pan})$.

Definition 22 (Collapse operator for the disjunctive completion)**.** *We define an operator* $\mathrm{Collapse}^{\mathrm{S}}$ *that takes a set of elements of* Pan *and merges together (by taking the abstract union), the elements that are equivalent with respect to* $\equiv_{\mathrm{E}}^{\mathrm{Pan}}$*. Formally,*

$$\mathrm{Collapse}^{\mathrm{S}}(O) = \left\{ \bigsqcup_{a \in \overline{a}}^{\mathrm{Pan}} a \mid \overline{a} \in O/\equiv_{\mathrm{E}}^{\mathrm{Pan}} \right\}$$

We use this collapse operator to provide an abstract union and intersection for the structural lifting

$$d_1 \sqcup^{\mathrm{S}} d_2 = \mathrm{Collapse}^{\mathrm{S}}(d_1 \cup d_2)$$

$$d_1 \sqcap^{\mathrm{S}} d_2 = \mathrm{Collapse}^{\mathrm{S}}\left(\left\{ t_1 \sqcap^{\mathrm{Pan}} t_2 \mid \begin{array}{c} t_1 \in d_1 \wedge t_2 \in d_2 \wedge \\ t_1 \sqcap^{\mathrm{Pan}} t_2 \neq \bot^{\mathrm{Pan}} \end{array} \right\} \right)$$

A widening for the structural lifting is given by

$$\begin{array}{l} d_1 \triangledown^{\mathrm{S}} d_2 = \\ \quad \{ t_1 \triangledown^{\mathrm{Pan}} t_2 \mid t_1 \in d_1 \wedge t_2 \in d_2 \wedge t_1 \equiv_{\mathrm{E}}^{\mathrm{Pan}} t_2 \} \\ \quad \cup \{ t_2 \in d_2 \mid \nexists t_1 \in d_1, t_1 \equiv_{\mathrm{E}}^{\mathrm{Pan}} t_2 \} \cup \{ t_1 \in d_1 \mid \nexists t_2 \in d_2, t_1 \equiv_{\mathrm{E}}^{\mathrm{Pan}} t_2 \} \end{array}$$

The other constructions of the structural lifting are the standard ones of disjunctive completion domains. For example, abstract inclusion is given by a Hoare ordering:

$$d_1 \sqsubseteq^{\mathrm{S}} d_2 \text{ iff } \forall a \in d_1, \exists a' \in d_2, a \sqsubseteq^{\mathrm{Pan}} a'$$

5.5 Analysis Result for the `do_ticks` Function

Figure 5.1 shows the result of running our analyser on the example from figure 4.2 (on page 31) using the RAND domain, with polyhedra as the underlying numeric domain. In one sentence, the difference between the RAND domain (Chapter 6) and the structural lifting of this chapter is that the RAND domain duplicates all the variables, to capture input-output relations:

```
Function summary for function do_ticks(p, n) returning p' :
 ( Constructor constraints : p.status@Running; p'.status@Running ...
   Structural equalities   : p = p' ; ...
  Numeric constraints      : n >= 1 ; ...
 )
 Or
 ( Constructor constraints : p.status@Asleep; p'.status@Running ...
   Structural equalities   : p.msg = p'.msg
  Numeric constraints      :
     p.id = p'.id; p'.status@Running.count = p.status@Asleep.count + 1;
     p.status@Asleep.secs >= 0; n >= p.status@Asleep.secs + 1
 )
 Or
 ( Constructor constraints : p.status@Asleep; p'.status@Asleep ...
   Structural equalities   : p.msg = p'.msg
  Numeric constraints      :
     p.id = p'.id; p.status@Asleep.secs >= n; n >= 1;
     p.status@Asleep.count = p'.status@Asleep.count;
     p'.status@Asleep.secs = p.status@Asleep.secs - n )
```

Figure 5.1: Result of our analysis on the example of figure 4.2. Ellipses mark information that is also present in other components of the same case and is elided.

- variables with a prime denote the output,
- whereas variables without a prime denote the input.

In figure 5.1, we see that our disjunctive completion considers three different cases, and contains all five properties that we wanted to infer automatically. In the first case, both the input and the output are running processes and the structural equality $\mathtt{p} = \mathtt{p}'$ tells us that the process remained unchanged (property 1). In the two other cases, the structural equality $\mathtt{p.msg} = \mathtt{p}'\mathtt{.msg}$ conveys that the `msg` field has not changed (property 5) while numeric constraints indicate that the `id` field has not changed (property 4). In the second case, the input process is asleep while the output process is running. The numeric properties tell us that the wake up count has increased by one and the sleeping budget of the input process is lower than argument `n` (property 2). In the third case, both the input and output process are asleep. The numeric relations tell us that the initial sleeping budget was greater than `n` and has decreased by `n`; also, the wake up count remains unchanged (property 3).

A preliminary version of the NPR domain from Chapter 4 was published in [BJM20].

Then, when the approach was implemented, we found several precision concerns, which lead us to modify the definitions in order to improve precision. This inspired the contributions from Chapter 5, which were published in [BJM22b; BJM22a].

Part II

From abstract domain to language analysis

Chapter 6

Turning Relational Domains into Input-Output Relational Domain

The term *"relational analysis"* is widely used in the literature, and may refer to two different notions. In a majority of related works, a "relational analysis" designates a static analysis that infers relations that hold between variables of a *single* program point, *i.e.*, relations *in space*. In other works, a "relational analysis" denotes a static analysis that infers relations between (variables of) *different* states, *i.e.*, relations *in time*. In the rest of this manuscript, the term "relational" mostly refers to *input-output* relational analyses, that compute relations between the input states and the output states of a program.

It is a folklore result, known in the literature, that a non-input-output relational analysis can be turned into an input-output relational analysis by "duplicating the number of variables". In this chapter, we give the formal details that allow to use this folklore result in the context of abstract interpretation. After briefly discussing related work in section 6.1, we present the *relational* collecting semantics that serves as basis for our input-output analysis in section 6.2 and then we formalize the generic approach that allows to build an abstract domain for input-output analyses from a domain for reachability analyses, in section 6.3. In particular, this will allow us to describe in detail (in section 7.2), how we summarize functions during our inter-procedural analysis, and how we instantiate these summaries during function call analysis.

6.1 Related work

The idea of exploiting an input-output *relational* semantics to verify `while` programs was developed by Kozen [Koz97]. He introduced Kleene Algebra with Tests, an extension of relation algebra [Tar41] with co-reflexive relations named *tests*, that serves as a foundation for the semantics of imperative programs, their verification, and as an effective formal tool for proving the correctness of program transformations.

A number of static analyses for approximating the input-output relation of a program have been proposed. Cousot and Cousot [CC02] used abstract interpretation for designing modular and relational analyses, and argue that compositionality can improve the scalability of analysers. Compositional Recurrence Analysis (CRA), by [FK15], is a *compositional* static analysis that infers numeric relations between the inputs and the outputs of programs. CRA first builds a regular expression to describe the set of program paths, that is then interpreted as an input-output *relation* in a compositional way, in a second stage. Their approach is context insensitive, and is similar to the relational semantics of definition 23. Whereas we follow the standard iteration-based analysis of loops, they use a special operator to compute the reflexive transitive closure of a relation, that is specialised on linear recurrence equations. Interestingly, they discuss in their benchmarks a variation of their analysis, named CRA+OCT, that *"uses an intra-procedural octagon analysis to gain some contextual information, but which is otherwise compositional"*, and that leads to more precise results than pure CRA. Although no precise definition is given for CRA+OCT, we believe that it follows our *relational collecting semantics* of definition 24, again with the exception of the treatment of loops. As we have also observed, exploiting the information available at loop entries is crucial to obtain sufficiently precise results. ICRA — by Kincaid, Breck and Bouroujeni [Kin+17] — is an inter-procedural extension of CRA, where function summaries are computed once and for all, independently of their calling contexts—an approach we have followed too in section 7.2. In contrast to CRA and ICRA, our analysis can deal with programs that are not purely numeric, and that can handle algebraic data types. We have not found any detailed description of *how* the function summaries of CRA and ICRA are instantiated. We are therefore not able to compare the way we instantiate function summaries (section 7.2) with CRA or ICRA. In contrast to CRA and ICRA, our analysis does not yet support recursively defined functions.

Other relational analyses were developed in the context of inter-procedural shape analysis [SJ11b; Jea13; ILR21]. They all feature a form of function summary, that helps reduce the analysis cost of large programs, by enabling modular analyses.

6.2 A Collecting Semantics of Relations

In this section, we define an input-output *relational* semantics of programs, that forms the semantic basis of an input-output relational analysis. Our relational semantics determines relations that relate the input stores of a program with its output stores, *i.e.*, the stores

that are obtained when there are no more commands to evaluate.

Definition 23 (Relational semantics)**.** *The relational semantics of a command c is defined as follows:* $\mathbb{S}\,[\![c]\!] = \{(s_1, s_2) \mid (c, s_1) \to^* (\mathsf{skip}, s_2)\}$.

As shown in lemma 11 below, the relational semantics of a compound command can be decomposed in terms of the relational semantics of the command's constituents. We define the relation $Id = \{(s, s)\}$ as the identity relation on stores, and we write $a_1; a_2 = \{(s_1, s_3) \mid \exists s_2, (s_1, s_2) \in a_1 \wedge (s_2, s_3) \in a_2\}$ to denote the composition of the relations a_1 and a_2. We also write a^* to denote the reflexive transitive closure of the relation a, *i.e.*, the least fixed point of the functional $\lambda a'.Id \cup (a; a')$. We have $a^* = Id \cup (a; a^*) = Id \cup (a^*; a)$.

Lemma 11. *The relational semantics enjoys the following identities:*

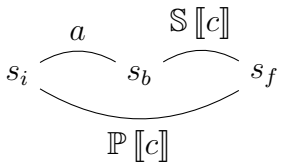
$$\begin{aligned}
\mathbb{S}\,[\![\mathsf{skip}]\!] &= Id \\
\mathbb{S}\,[\![c_1\,;c_2]\!] &= \mathbb{S}\,[\![c_1]\!]\,;\mathbb{S}\,[\![c_2]\!] \\
\mathbb{S}\,[\![\mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end}]\!] &= \textstyle\bigcup_{1\le i\le n} \mathbb{S}\,[\![c_i]\!] \\
\mathbb{S}\,[\![\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}]\!] &= (\mathbb{C}\,[\![b]\!]\,;\mathbb{S}\,[\![c]\!])^*;\mathbb{C}\,[\![\neg b]\!] \\
\mathbb{S}\,[\![\mathsf{assert}(b)]\!] &= \mathbb{C}\,[\![b]\!] \\
\mathbb{S}\,[\![x := t]\!] &= \{(s, s(x \mapsto v)) \mid (s, v) \in \mathbb{E}\,[\![t]\!]\}
\end{aligned}$$

where the semantics of numeric expression and boolean conditions are defined as follows:

$$\mathbb{E}\,[\![t]\!] = \{(s, v) \mid v \in [\![t]\!]_s^{\mathrm{exp}}\} \qquad \mathbb{C}\,[\![b]\!] = \{(s, s) \mid \mathtt{tt} \in [\![b]\!]_s^{\mathrm{bool}}\}$$

Lemma 11 allows for fully compositional analyses, where atomic commands are analysed independently from one another, and the analyses' results are then combined. An example of such a compositional analysis is CRA [FK15; Kin+17]. A drawback of a fully compositional approach, however, is its inability to exploit any information about the states that have been reached so far, which may degrade the precision of an analysis. The following piece of code illustrates this issue: `assert (x > 1 && y > 1); x := y * x`. If we analyse the assignment `x := y * x` with no knowledge that the preceding assertion succeeded, then, using a linear relational domain—*e.g.*, octagons or polyhedra—we will not obtain any precise information about how the value of `x` has changed, as the domain cannot express non-linear relations. The relational collecting semantics $\mathbb{P}\,[\![c]\!]$ from definition 24 below waives this limitation, as it allows to exploit the information that has so far been obtained for the current program point. Our collecting semantics $\mathbb{P}\,[\![c]\!]$ is a function from relations

to relations: given some initial relation a that holds between initial stores s_i and the stores s_b at the current program point (*before* the execution of c), $\mathbb{P}[\![c]\!](a)$ computes a relation between the initial stores s_i and the final stores s_f that are produced by evaluating the command c from the stores s_b. Thus, $\mathbb{P}[\![c]\!]$ *extends* the relations *in time* by composing on the right-hand side with the behaviour of command c.



Definition 24 (Collecting semantics)**.** $\mathbb{P}[\![c]\!](a) = a; \mathbb{S}[\![c]\!]$

$\mathbb{P}[\![c]\!]$ is an abstraction of a semantics of computation traces [Cou97]. Computation traces keeps all the intermediate stores that a program may reach during its execution; whereas $\mathbb{P}[\![c]\!]$ only keeps the initial and final stores. The collecting semantics $\mathbb{P}[\![c]\!]$ enjoys the equations listed in the next lemma, that shows how it decomposes by following the syntax of commands.

Lemma 12 (Inductive Characterisation of the Collecting Semantics). *The following equations hold:*

$$\begin{aligned}
\mathbb{P}[\![\mathit{skip}]\!](a) &= a \\
\mathbb{P}[\![c_1\,;c_2]\!](a) &= \mathbb{P}[\![c_2]\!](\mathbb{P}[\![c_1]\!](a)) \\
\mathbb{P}[\![\mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end}]\!](a) &= \textstyle\bigcup_{1\le i\le n} \mathbb{P}[\![c_i]\!](a) \\
\mathbb{P}[\![\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}]\!](a) &= \mathbb{P}[\![\mathsf{assert}(\neg b)]\!](\mathrm{lfp}\, f_a) \\
&\quad \textit{where } f_a(r) = a \cup \mathbb{P}[\![c]\!](\mathbb{P}[\![\mathsf{assert}(b)]\!](r)) \\
\mathbb{P}[\![\mathsf{assert}(b)]\!](a) &= \{(s_1,s_2) \mid (s_1,s_2) \in a \wedge [\![b]\!]^{\mathrm{bool}}_{s_2} = \{\mathtt{tt}\}\} \\
\mathbb{P}[\![x := t]\!](a) &= \{(s_1, s_2(x \mapsto v)) \mid (s_1,s_2) \in a \wedge v \in [\![t]\!]^{\mathrm{exp}}_{s_2}\}
\end{aligned}$$

Proof. The proof of the three first cases rely on the algebraic properties of relational composition, namely: *Id* is its neutral element, composition is associative, and it distributes over union. To prove the fourth case, we first notice that for any fixed a and b, for the function $g(r) = a \cup (r; b)$, we have $g^i(\bot) = \bigcup_{0\le j<i} a; b^j$. Then, it follows that $\mathrm{lfp}\, g = \bigcup_{i\ge 0} g^i(\bot) = \bigcup_{i\ge 0}\bigcup_{0\le j<i} a; b^j = \bigcup_{j\ge 0}\bigcup_{i>j} a; b^j = \bigcup_{j\ge 0} a; b^j = a; \bigcup_{j\ge 0} b^j = a; b^*$. The proofs of the last two cases proceed by unfolding definitions. □

Lemma 12 will serve as the semantic basis for the analysis that we describe in section 7.1.

Lemma 12 shows that the syntax-directed decomposition of the relation transformer $\mathbb{P}[\![c]\!]$ follows the same *structure* as the standard set-based collecting semantics, that collects the set of reachable states. Most transfer functions of our collecting semantics are the same, but they operate on different objects (binary relations on stores instead of sets of stores). The two transfer functions that are specific to this relational semantics are the ones for assertion and for assignment. We show in section 6.3 how to generically define abstractions for those two transfer functions—that transform relations that relate stores in *different* program points—using *any* relational abstract domain that represents sets of stores for one program point. Using these two results, we can turn a folklore technique into a formal claim: transforming a non input-output relational analysis into an input-output relational one is "as simple as" duplicating variables [BH19; ILR17].

An important difference between the two styles of analyses—set-based *vs.* relational—is the choice of the most precise starting point for the initial store, when no assumption is made on that store. In set-based analysis, that starting point is $\top$ (the set of all stores), whereas in the relational analysis, the starting point is Id (the identity relation on stores).

6.3 Leveraging Relations in Space to Express Relations in Time

In this section we show that any relational domain—*i.e.*, that denotes sets of stores and can express binary relations between different variables of a single store—can be lifted to a domain for pairs of stores, that is able to express relations between input stores and output stores. The main idea is simple: a pair of stores $(s_1, s_2) \in (\text{Vars} \to \text{Values})^2$ can be represented as a single store, provided we can distinguish the variables in s_1 from those in s_2.

Formally, this is achieved by assuming two bijections $\mathit{prime} : \text{Vars} \to \text{Vars}'$ and $\mathit{second} : \text{Vars} \to \text{Vars}''$ where Vars' and Vars'' are disjoint "copies" of Vars, that intuitively contain the "primed" and "seconded" versions of the variables of Vars. We write x' as a shorthand for $\mathit{prime}(x)$, and x'' for $\mathit{second}(x)$, and use the same convention as in [FK15], *i.e.*, we use regular variables for the left-hand sides of relations—the input stores—and primed variables for the right-hand sides—the output stores. We introduce the set Vars'' of "seconded" variables for composition (definition 28 on page 72). Indeed, if a_1 and a_2 abstract two input-output relations that we want to compose, we rename the variables so that

- regular variables refer to the input of the result, which is also the input of a_1,
- primed variables refer to the output of the result, which is also the output of a_2,
- and seconded variables are temporarily used for the intermediate state corresponding both to the output of a_1 and the input of a_2.

For any map f, we write f' as a shorthand for $f \circ \mathit{prime}^{-1}$, and we write $f \cup g$ for the union of maps with disjoint domains. This allows us to represent any pair (s_1, s_2) of stores as a single store $s_1 \cup s_2'$. We use this encoding to transform any relational domain that represents a set of stores into a domain that represents a binary relation over stores.

With this encoding, if some variable x is effectively present both in the input store s_1 and in the output store s_2, it will occur twice—as x and as x'—in the store $s_1 \cup s_2'$.

Definition 25 (Relational lifting)**.** *Let* D *be an abstract domain, such that for any typing context* Γ*,* $\mathrm{D}(\Gamma)$ *is equipped with concretisation function* $\gamma^{\mathrm{D}(\Gamma)} \in \mathrm{D}(\Gamma) \to \mathcal{P}(\mathrm{Vars} \to \mathrm{Values})$*. For any two typing contexts* Γ_1 *and* Γ_2*, the relational lifting* $\mathrm{R}\,(\mathrm{D})(\Gamma_1, \Gamma_2)$ *of* D *and its concretisation function are defined as follows:*

$$\begin{array}{rcl} \mathrm{R}\,(\mathrm{D})(\Gamma_1, \Gamma_2) & = & \mathrm{D}(\Gamma_1 \cup \Gamma_2') \\ \gamma^{\mathrm{R}(\mathrm{D})(\Gamma_1,\Gamma_2)}(a) & = & \{(s_1, s_2) \mid s_1 \cup s_2' \in \gamma^{\mathrm{D}(\Gamma_1 \cup \Gamma_2')}(a)\} \end{array}$$

The relational lifting expects two typing contexts—one for the input stores, and one for the output stores. The fact of allowing different typing environments for the store on the left and the store on the right will prove useful when analysing function calls. Indeed, the same variable name can represent variables with different types for the caller and the callee. Hence, the abstract values representing the transitions from caller to callee and from callee to caller will simultaneously handle two different typing contexts.

The lifted domain $\mathrm{R}\,(\mathrm{D})(\Gamma_1, \Gamma_2)$ is naturally equipped with a pre-order relation, abstract union, intersection and widening, by reusing those of $\mathrm{D}(\Gamma_1 \cup \Gamma_2')$.

As we remarked in section 6.2, only two pieces are missing to get a relational input-output analysis: now that we can express relations on stores, the question remains of how to express the transfer functions for conditionals and assignments. We show in figure 6.1 how to do so in a generic way, by exploiting the transfer functions of the underlying domain.

The transfer function for conditionals $\mathrm{Cond}^{\mathrm{R}(\mathrm{D})(\Gamma_1,\Gamma_2)}(b)(a)$ constrains the right-hand side of the relation a to satisfy the boolean condition b. This is achieved by calling the

$$\begin{aligned}\mathrm{Cond}^{\mathrm{R(D)}(\Gamma_1,\Gamma_2)}(b)(a) &= \mathrm{Cond}^{\mathrm{D}(\Gamma_1\cup\Gamma_2')}(b')(a)\\ \mathrm{Assign}^{\mathrm{R(D)}(\Gamma_1,\Gamma_2)}(x := t)(a) &= \mathrm{Assign}^{\mathrm{D}(\Gamma_1\cup\Gamma_2')}(x' := t')(a)\end{aligned}$$

Figure 6.1: Relational transfer functions for conditionals and assignment.

transfer function for conditions of the underlying domain on b', to enforce that the variables of b refer to the outputs of a.

The transfer function for assignment $\mathrm{Assign}^{\mathrm{R(D)}(\Gamma_1,\Gamma_2)}(x := t)(a)$ calls the underlying domain on the primed version of the assignment, $x' := t'$, to ensure that it applies to the outputs of relation a, leaving the inputs unchanged.

This concludes our justification of the folklore claim that, *"to turn a static analysis for the sets of final states into an analysis for input-output relations, it suffices to duplicate variables"*. We have built our justification on the following remarks: 1. Duplicating variables turns a non input-output relational domain—*i.e.*, a relation between variables of the stores of a *single* program point—into a domain of binary relations between stores of *two* different program points. 2. An input-output relational analysis has the same structure as an analysis for final states. 3. The transfer functions that are specific to the input-output relational analysis can be defined in a generic way, using those of the analysis for final states.

In the rest of this manuscript, we use the relational lifting of the abstract domain from Part I, that we call RAND—short for *Relational Algebraic and Numeric Domain*.

Chapter 7

Implementation and Experimental Results

In this chapter, we describe how to use the RAND domain from Part I to analyse programs, and we discuss implementation, experimental results and complexity. Section 7.1 describes intra-procedural analysis, on the language described in section 2.2, while section 7.2 extends the language with function calls and describes inter-procedural analysis. In section 7.3, we present an OCaml implementation, we report on experimental results over several examples and we discuss complexity.

7.1 Intra-Procedural Analysis

We define a function Analyse that takes a program c and an abstract value a—representing the relation gathered so far between the input states and the current state—and returns the abstract value $\text{Analyse}(c)(a)$ that over-approximates the effect of running c *after* a. This section deals with basic constructs, while section 7.2 explains how we analyse functions.

Definition 26 (Intra-procedural version of the analysis function)**.**

$$\begin{aligned}
\text{Analyse}(\mathsf{assert}(b))(a) &= \text{Cond}^{\text{Rand}}(b)(a) \\
\text{Analyse}(x := t)(a) &= \text{Assign}^{\text{Rand}}(x := t)(a) \\
\text{Analyse}(c_1\,;c_2)(a) &= \text{Analyse}(c_2)\,(\text{Analyse}(c_1)(a)) \\
\text{Analyse}(\mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end})(a) &= \bigsqcup\nolimits^{\text{Rand}}_{i\in 1,\ldots,n} \text{Analyse}(c_i)(a) \\
\text{Analyse}(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end})(a) &= \text{Cond}^{\text{Rand}}(\neg b)\,(\lim_{n\to\infty} a_n) \\
\textit{where } a_0 = a \textit{ and } a_{n+1} &= a_n \triangledown^{\text{Rand}} \text{Analyse}(\mathsf{assert}(b); c)(a_n)
\end{aligned}$$

For assertion and assignment, we use the transfer functions built in previous sections. Sequence and branching follow the structure outlined in lemma 12.

We analyse loops in a standard way, using a widening-based Kleene iteration, which

ensures that we reach a post-fixpoint in a finite number of iterations. In practice, our implementation performs a *loop unrolling* [RY20, p.131] of the first iteration, so as to obtain better precision.

The Analyse function is *sound*, in the sense that it over-approximates the relational collecting semantics.

Theorem 1 (Soundness w.r.t. the collecting semantics). *For any numeric domain* N *satisfying the hypotheses from section 2.3, for any command c that type-checks in some typing-context* Γ *and any abstract value* $a \in \mathrm{Rand}\,(\mathrm{N})(\Gamma,\Gamma)$*,*

$$\mathbb{P}\,[\![c]\!]\left(\gamma^{\mathrm{Rand(N)}(\Gamma,\Gamma)}(a)\right) \subseteq \gamma^{\mathrm{Rand(N)}(\Gamma,\Gamma)}\left(\mathrm{Analyse}(c)(a)\right)$$

By instantiating theorem 1 with the abstraction of the identity relation, we get a soundness result with respect to the relational semantics of commands:

Corollary 1 (Soundness w.r.t. the relational semantics). *For any numeric domain* N *satisfying the hypotheses from section 2.3, for any command c that type-checks in some typing-context* Γ *and any abstract value* $a \in \mathrm{Rand}\,(\mathrm{N})(\Gamma,\Gamma)$*,*

$$\mathbb{S}\,[\![c]\!] \subseteq \gamma^{\mathrm{Rand(N)}(\Gamma,\Gamma)}\left(\mathrm{Analyse}(c)\left(\mathrm{Id}^{\mathrm{Rand(N)}(\Gamma,\Gamma)}\right)\right)$$

7.2 Analysis of Function Calls

In this section, we add function definitions and function calls to our language, and extend the intra-procedural analysis of section 7.1 into a modular inter-procedural analysis, based on function summaries.

Extended syntax and semantics for functions We extend our language to support function calls in commands and function declarations:

$$\begin{array}{lll} c \in \mathrm{Cmd} & ::= & \ldots \quad | \quad x := f(x_1, \ldots, x_n) \\ d \in \mathrm{Decl} & ::= & \mathsf{def}\, f(\tau_1\ x_1, \ldots, \tau_n\ x_n) : \tau = \{c\,;\mathsf{return}\,x\} \\ P \in \mathrm{Prog} & ::= & d_1; \ldots; d_n \end{array}$$

For simplicity, the command for function calls $y := f(x_1, \ldots, x_n)$ immediately saves in a variable y the result of calling a function f. This restriction forbids to call functions within expressions, so that the semantics of expressions and the transfer function for assignment

$$\textsc{FunctionCall}\quad \frac{\Delta(f) = ((x_1, \ldots, x_n), c_f, r, \Gamma_f) \qquad \forall i \in \{1, \ldots, n\}, v_i \in [\![z_i]\!]_s^{\exp}}{(y := f(z_1, \ldots, z_n), s, \pi) \to (c_f \,;\, \mathsf{return}\, r, [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n], (y, s) : \pi)}$$

$$\textsc{FunctionReturn}\quad \frac{v_r \in [\![r]\!]_s^{\exp}}{(\mathsf{return}\, r, s, (y_r, s_r) : \pi) \to (\mathsf{skip}, s_r(y_r \mapsto v_r), \pi)}$$

Figure 7.1: Small-step semantics for functions

remain unchanged.

A program is a sequence of function declarations of the form

$$\mathsf{def}\, f(\tau_1\ x_1, \ldots, \tau_n\ x_n) : \tau = \{c \,;\, \mathsf{return}\, r\}$$

that specify, for each function f, what are its formal parameters $x_1, \ldots, x_n$ with their respective types $\tau_1, \ldots, \tau_n$, what is the output type τ, what is the body c and what is the return variable r. A program effectively defines a map Δ, that associates to every declared function f a quadruplet $\Delta(f) = ((x_1, \ldots, x_n), c, r, \Gamma)$ that holds the formal parameters x_i of f, its body c, its formal return variable r, and the typing context Γ that specifies the types of its formal and local variables.

Figure 2.2 on page 15 gives the small-step reduction rules for the language without function calls. We extend the small-step reduction rules from figure 2.2 as follows. First, we add to our semantic states (c, s), that are composed of a program and a store, a third component π that denotes a *call stack*, and is used to properly handle function *returns*. The reduction rule SeqStep that handles sequences of commands simply propagates any change to the stack, whereas the other rules of figure 2.2 leave the stack unchanged. Then, we augment the reduction relation with two new rules (figure 7.1). Rule FunctionCall installs the body of the called function f as the new code to execute, and installs a new store that defines the actual values—read in the caller's store—for the formal parameters of f. Simultaneously, a new element (y, s) is added at the top of the call stack, so as to remember that the caller's store s should be restored upon f's return, and that the value computed by f must be recorded in the variable y. This very action, that must be performed at function return, is specified by rule FunctionReturn.

Analysing functions We have chosen to develop a modular analysis, by analysing each function *only once* and computing a *function summary*, that summarises a function's behaviour. This summary is then reused and instantiated each time that function is called. Such a modular analysis allows to better scale to large code bases [CC02].

Definition 27 (Function summaries)**.** *For a function f defined by*

$$\Delta(f) = ((x_1, \ldots, x_n), c_f, y_f, \Gamma_f)$$

we call summary *of f the quadruplet given by:*

$$\left((x_1, \ldots, x_n), \mathrm{Analyse}(c_f)\left(\mathrm{Id}^{\mathrm{Rand(N)}(\Gamma_f, \Gamma_f)}\right), y_f, \Gamma_f\right)$$

The first, third and fourth components of a function summary are the same than in the function declaration: respectively formal parameters, formal return variable and the function's typing context.

The second component of the summary of a function f is an abstract value summarising f's behaviour by over-approximating the input-output relation between its formal arguments and its formal return variable. Thus, this abstract value deals with the variables that are *local* to the execution of f: no information about the caller's environment is recorded in the summary. This abstract value is obtained by analysing the body of f, starting with the identity relation. This means that we make no assumption on the actual arguments that will be given to f, hence we can reuse the *same* summary in *every* calling context.

To use a function summary at some call site, we *instantiate* the summary on the actual arguments and output variable used at the call site. Our method to instantiate summaries is based on an abstraction of relational composition, that sequentially chains together two abstract values that represent binary relations.

Definition 28 (Abstract composition)**.** *Let Γ_1, Γ_2 and Γ_3 be typing contexts. Let $a_1 \in \mathrm{R}\,(\mathrm{D})(\Gamma_1, \Gamma_2)$ and $a_2 \in \mathrm{R}\,(\mathrm{D})(\Gamma_2, \Gamma_3)$ be two abstract values. The abstract composition $a_1 \,;^{\mathrm{R(D)}}\, a_2$ of the abstract values a_1 and a_2 is defined by:*

$$a_1 \,;^{\mathrm{R(D)}}\, a_2 \;=\; \underset{\Gamma''_2}{\mathrm{Remove}}\left(\underset{\Gamma'_3}{\mathrm{Add}}\, c_1 \;\sqcap^{\mathrm{D}(\Gamma_1 \cup \Gamma''_2 \cup \Gamma'_3)}\; \underset{\Gamma_1}{\mathrm{Add}}\, c_2\right)$$

where $c_1 = \underset{primeToSecond}{\text{Rename}}\ a_1$ *and* $c_2 = \underset{regularToSecond}{\text{Rename}}\ a_2$ *and*

$$\begin{array}{rcccc} primeToSecond & : & \mathrm{dom}(\Gamma_1 \cup \Gamma_2') & \to & \mathrm{dom}(\Gamma_1 \cup \Gamma_2'') \\ & & x \in \mathrm{dom}(\Gamma_1) & \mapsto & x \\ & & x' \in \mathrm{dom}(\Gamma_2') & \mapsto & x'' \\ regularToSecond & : & \mathrm{dom}(\Gamma_2 \cup \Gamma_3') & \to & \mathrm{dom}(\Gamma_2'' \cup \Gamma_3') \\ & & x \in \mathrm{dom}(\Gamma_2) & \mapsto & x'' \\ & & x' \in \mathrm{dom}(\Gamma_3') & \mapsto & x' \end{array}$$

Abstract composition chains the effects of a_1 and a_2 by introducing auxiliary names (seconded variables of the form y'') for the states that are in the output of a_1 and the input of a_2, before taking the intersection, and then removing the temporarily introduced variables. The calls to Add are necessary name management steps, that ensure that the abstract values deal with the same sets of variables. Abstract composition is a sound approximation of relational composition, as stated by the following lemma:

Lemma 13 (Soundness of composition). *For any two abstract values from the relational lifting,* $a_1 \in \mathrm{R}\,(\mathrm{D})(\Gamma_1, \Gamma_2)$ *and* $a_2 \in \mathrm{R}\,(\mathrm{D})(\Gamma_2, \Gamma_3)$*, we have:*

$$\gamma^{\mathrm{R(D)}(\Gamma_1,\Gamma_2)}(a_1)\,;\gamma^{\mathrm{R(D)}(\Gamma_2,\Gamma_3)}(a_2) \subseteq \gamma^{\mathrm{R(D)}(\Gamma_1,\Gamma_3)}(a_1\ ;^{\mathrm{R(D)}}\ a_2)$$

Based on abstract composition, we express summary instantiation as follows:

Definition 29 (Summary instantiation)**.** *The* instantiation *of the function summary* $S_f = ((x_1, \ldots, x_n), a_f, y_f, \Gamma_f)$ *on the actual parameters* $(z_1, \ldots, z_n)$*, the actual return variable* y *and the caller typing context* Γ *is defined as follows:*

$$\begin{array}{l} \mathrm{Inst}(S_f, (z_1, \ldots, z_n), y, \Gamma) \;=\; ins\ ;^{\mathrm{Rand(N)}}\ a_f\ ;^{\mathrm{Rand(N)}}\ outs \\ \quad \textit{where } ins = \mathrm{Cond}^{\mathrm{Rand(N)}(\Gamma,\Gamma_f)}\left(\bigwedge_{i\in\{1,\ldots,n\}} z_i = x_i'\right)\left(\top^{\mathrm{Rand(N)}(\Gamma,\Gamma_f)}\right) \\ \quad \textit{and } outs = \mathrm{Cond}^{\mathrm{Rand(N)}(\Gamma_f,\Gamma)}(y_f = y')\left(\top^{\mathrm{Rand(N)}(\Gamma_f,\Gamma)}\right) \end{array}$$

Summary instantiation simply works by composing three abstract values, using abstract composition. Instantiation first ties each actual parameter to its formal parameter by *pre*-composing the abstract value a_f for f's body with the *ins* abstract value, and then ties the formal output to the actual output by *post*-composing with the *outs* value. The values *ins* and *outs* are simply expressed as mere conjunctions of equalities. The first composition deals with the *call* of the function, whereas the second composition handles the *return*.

During a function call $y := f(z_1, \ldots, z_n)$, the instantiation of f's summary describes how variable y has changed, but does not deal with the fact that *only* the variable y may have changed: every other variable that is available before the call remains the same after the call. Thus, the transfer function for function call augments the instantiation of the function summary S_f with equalities for the unaltered variables, before extending the relation a gathered so far with the effect of the call to f:

$$\begin{aligned}&\text{Analyse}(y := f(z_1, \ldots, z_n))(a) \;=\\ &\qquad a \;;^{\text{Rand(N)}} \left(\begin{array}{l} \text{Inst}(S_f, (z_1, \ldots, z_n), y, \Gamma) \sqcap^{\text{Rand(N)}(\Gamma,\Gamma)} \\ \bigsqcap_{x \neq y}^{\text{Rand(N)}(\Gamma,\Gamma)} \text{Cond}^{\text{Rand(N)}(\Gamma,\Gamma)}(x = x') \end{array} \right)\end{aligned}$$

The transfer function for function calls is sound:

Lemma 14 (Soundness of function call analysis). *For every function definition of the form*

$$\Delta(f) = ((x_1, \ldots, x_n), c_f, y_f, \Gamma_f)$$

and any function summary of the form

$$S_f = ((x_1, \ldots, x_n), a_f, y_f, \Gamma_f)$$

such that $\mathbb{S}\,[\![c_f]\!] \subseteq \gamma^{\text{Rand(N)}(\Gamma_f,\Gamma_f)}(a_f)$*, we have:*

$$\mathbb{P}\,[\![y{:=}f(z_1, \ldots, z_n)]\!](\gamma^{\text{Rand(N)}(\Gamma,\Gamma)}(a)) \subseteq \gamma^{\text{Rand(N)}(\Gamma,\Gamma)}(\text{Analyse}(y{:=}f(z_1, \ldots, z_n))(a))$$

Proof. Let s_0 be a store, and s_2 be the store that results from the call instruction $y{:=}f(z_1, \ldots, z_n)$ on the initial store s_0. We have

$$(s_0, s_2) \in \mathbb{P}\,[\![y{:=}f(z_1, \ldots, z_n)]\!](\gamma^{\text{Rand(N)}(\Gamma,\Gamma)}(a))$$

Unfolding the definition of $\mathbb{P}\,[\![\cdot]\!]$ gives an intermediate store s_1 such that (s_0, s_1) belongs to $\gamma^{\text{Rand(N)}(\Gamma,\Gamma)}(a)$ and (s_1, s_2) belongs to $\mathbb{S}\,[\![y{:=}f(z_1, \ldots, z_n)]\!]$. The function call can be decomposed as a series of reductions

$$\begin{array}{rcl}(y{:=}f(z_1, \ldots, z_n), s_1, \pi) & \to & (c_f; \mathsf{return}\, y_f, s', (y, s_1) : \pi) \\ & \to^* & (\mathsf{return}\, y_f, s'', (y, s_1) : \pi) \\ & \to & (\mathsf{skip}, s_2, \pi)\end{array}$$

where s' is the store that is initialised at the beginning of the execution of f's body, and s'' is the store that is obtained at the end of the execution of f's body. Let *ins* and *outs* be the abstract values used in definition 29. By the definition of s' in rule FunctionCall, we get $(s_1, s') \in \gamma^{\mathrm{Rand(N)}(\Gamma,\Gamma_f)}(ins)$. Moreover, $(s', s'') \in \mathbb{S}[\![c_f]\!]$ is obtained by the hypothesis on f's summary. Finally, we prove $(s'', s_2) \in \gamma^{\mathrm{Rand(N)}(\Gamma_f,\Gamma)}(outs)$ by the definition of s_2 in rule FunctionReturn: s'' and s_2 indeed coincide on every variable other than the actual return variable. We conclude using the soundness of abstract composition, abstract intersection and abstract condition. □

Lemma 14 ensures that the soundness result for the intra-procedural analysis (theorem 1) extends to the language with function calls that we have described in this section.

7.3 Experimental Results and Complexity

We have implemented an analyser in approximately 5000 lines of OCaml, to analyse a `while` language with algebraic types. Our implementation together with instructions on how to add new test cases and run the tests cases is packaged and published as a virtual machine artefact [BJM22a]. Similarly to our formal development, our analyser is parametrised by an abstract domain for integers. A command-line option allows to choose among numeric domains provided by Apron [JM09], such as intervals, octagons or polyhedra.

We have tested our analyser on a total of 43 programs, summarised on Tables 7.1 and 7.2. Table 7.1 describes some complex examples: some sorting algorithms, the `do_ticks` function from figure 4.2, and 6 functions inspired from the abstract specification of the seL4 micro-kernel [Kle+09]. Table 7.2 describes a set of simple examples. We now review the results that our analyser computed on the complex examples of Table 7.1, using polyhedra as numeric domain.

Sorting integer arrays Our OCaml implementation does not include the extension for arrays that is presented later in the manuscript, in Part III. To circumvent the absence of support for arrays in our prototype, we modelled arrays of fixed length using tuples, and we defined `get` and `set` functions.

The `do_ticks` function The `do_ticks` function (figure 4.2 on page 31) is inspired from a process scheduler from operating system code. As reported in section 5.5, the analysis

result for `do_ticks` captures all the properties we expected.

seL4-inspired functions We have extracted from the abstract specification of the seL4 formally verified micro-kernel [Kle+09] several functions, that work both on ADTs and on scalar values, and translated them in our `while` language. Specifically, those functions are related to either thread management, capability management or scheduling (`decode_set_priority`, `check_prio`, `mask_cap`, `validate_vm_rights`, `cap_rights_update`, `timer_tick`). Our analyser infers exact abstractions for all of them, except for `timer_tick`. This program is slightly different from `do_ticks`: when a thread's time budget is over, this budget is reset to its original value and the thread is then re-scheduled, in a scheduler queue. Then, a new (possibly different) current thread is chosen from the scheduler queue. During a re-scheduling, the case constraints of our abstract domain cannot distinguish whether the new current thread is the same as the old one or not, so a join of those two cases is performed. This results in some expected information loss on the thread's time budget.

For the `mask_cap` program, we experimented with two encodings of bitmasks, using either integers or ADTs to represent booleans. The integer-based encoding produced a function summary that is compact—only 4 cases—but hard to understand for a human being, whereas the summary produced with the ADT-based encoding is large —it distinguishes 324 disjuncts— but each disjunct is easy to interpret for a human being.

We consider that the precision we obtained on the seL4 examples is satisfying. Still, the last example illustrates a limitation of our approach. Indeed the function summaries can significantly grow when the analysed program pattern matches on many distinct variables. Abstract domains that leverage BDDs have been successfully used to reduce analysis costs by sharing common results [Dim19; DAL22; SJ11a; Jea09], and could also help in our situation.

Complexity of our analysis Each domain that constitutes RAND, with the exception of the disjunctive completion layer, features operators and transfer functions whose complexity is polynomial in program parameters, *e.g.*, the number of variables, or the maximum depth of the defined types. For the disjunctive completion, however, the complexity is polynomial in the number of possible cases, which can itself be exponential in program parameters. The number of cases is asymptotically bounded by c^{xf^p}, where x is the number of variables in the program, c is the maximum number of different constructors per sum type, f is the

Table 7.1: Complex test cases used for experimental evaluation. The columns indicate whether the tests involve **sum** types, **numeric** operations, while **loops** or function **calls**, as well as the analysis **time**, and the maximum number of **cases** per function summary. Analysis times are given in milliseconds, with the exception of longer durations, that are given in seconds and printed with a bold face. Measures were performed on an Intel® Core™ i7 @2.30GHz × 16. The codebase [BJM22a] includes instructions to reproduce the results.

Name	Sums	Numeric	Loops	Calls	Time	Cases
Hand-crafted tests:						
`do_ticks`	Yes	Yes	Yes	No	166 ms	3
`nondeterministic_bubble_sort`	Yes	Yes	Yes	Yes	**2.1 s**	5
`selection_sort`	Yes	Yes	Yes	Yes	**10.9 s**	25
Inspired from SeL4:						
`decode_set_priority`	Yes	Yes	No	Yes	10 ms	2
`mask_cap_boolean`	Yes	No	No	Yes	**7.4 s**	324
`mask_cap_int`	Yes	Yes	No	Yes	**1.5 s**	4
`timer_tick_scheduling`	Yes	Yes	Yes	Yes	**41.2 s**	81

maximum number of fields in any product type and p is the maximum depth of the types being defined. While it is possible to write a program that reaches this bound, we have not found any program, even in seL4, that makes the number of cases explode.

The asymptotic bound for the number of cases can be reached for programs in which all the variables hold a complete f-ary tree of depth p, with all the leaves belonging to a sum type having c possible constructors.

For our analysis to actually consider all those possible cases, the program would also have to perform pattern-matching on all the different leaves, probably using pattern-matching inside a loop or a function call.

There are two different scenarios that render our analysis costly: either when the number of different cases is high—in which case our disjunctive completion can be the bottleneck—or when many *numeric* extended variables are considered—in which case the underlying numeric domain can be the bottleneck. A solution for the first scenario could be to adopt a different merging strategy, so that more cases are merged, at the risk of losing precision. In the second scenario, the generic aspect of our domain allows to choose between numeric domains with different precision versus cost trade-offs. In addition, techniques based on partitioning the set of variables could also be leveraged.

Table 7.2: Simple test cases used for experimental evaluation. We use the ∗ symbol for families of similar tests, whose names start identically. The columns indicate whether the tests involve **sum** types, **numeric** operations, while **loops** or function **calls**, as well as the analysis **time**, and the maximum number of **cases** per function summary. Analysis times are given in milliseconds, with the exception of longer durations, that are given in seconds and printed with a bold face. Measures were performed on an Intel® Core™ i7 @2.30GHz × 16. The codebase [BJM22a] includes instructions to reproduce the results.

Name	Sums	Numeric	Loops	Calls	Time	Cases
Simple tests:						
`assert*`	Yes	Yes	No	No	1 ms	1
`call_inside_loop_*`	No	Yes	Yes	Yes	15 ms	1
`drift`	Yes	Yes	Yes	Yes	24 ms	2
`exchange`	No	No	No	No	2 ms	1
`facto*`	No	Yes	Yes	No	8 ms	1
`false_type_collision`	No	No	No	Yes	3 ms	1
`fibonacci`	No	Yes	Yes	Yes	51 ms	1
`gauss*`	No	Yes	Yes	No	15 ms	1
`ghost_equality`	No	No	No	No	< 1 ms	1
`hidden_incompat`	Yes	No	No	No	2 ms	0
`id`	No	No	No	No	< 1 ms	1
`if`	No	Yes	No	No	2 ms	1
`incompat`	Yes	No	No	No	< 1 ms	0
`indirect_swap`	Yes	No	No	Yes	3 ms	2
`long_id`	Yes	No	No	Yes	5 ms	2
`modulo`	No	Yes	Yes	Yes	33 ms	2
`multiplication_larger`	No	Yes	No	No	2 ms	1
`or_constructor`	Yes	No	No	No	< 1 ms	0
`plus_*`	Yes	Yes	No	No	< 1 ms	1
`record_assignment*`	No	Yes	No	No	2 ms	1
`reduction`	No	Yes	No	No	3 ms	1
`struct_exchange`	Yes	No	No	No	< 1 ms	1
`swap`	Yes	No	No	No	< 1 ms	2
`test_loop`	No	Yes	Yes	No	3 ms	1
`two_by_two`	No	Yes	Yes	No	3 ms	1
`while_true`	No	No	Yes	No	< 1 ms	0
`widening_convergence`	No	Yes	Yes	No	49 ms	1
`xor`	Yes	No	No	Yes	8 ms	3

Part III

Abstract Domain for Arrays of Structured Values

Chapter 8

Related Work on Array Analysis

In this Part III, we extend the work that we have already done, on abstract interpretation for algebraic types (Part I), to programs that manipulate arrays. Hence we are particularly interested in works that use *abstract interpretation* to analyse the contents of *arrays*.

Work by Dietsch et al. [Die+18] uses abstract interpretation to analyse programs that manipulate arrays. However, they do not focus on the *numeric relations* satisfied by the values inside arrays slots. Instead, they focus on the *equality* of arrays up to exceptional indices. They define the *map equality domain*, whose abstract values express the equality of array pairs, except at the indices that satisfy some (automatically established) predicates. By focusing on equality, the perspective that the correlation domain [And+19] takes on algebraic types is similar to the perspective that the map equality domain [Die+18] takes on arrays.

[And+19] also handle arrays, with an expressiveness that is, on purpose, limited. When describing the relation between two arrays, they may specify *one* exceptional index and at most two correlations: one correlation that explains how the arrays relate at the exceptional index, and one that explains how the arrays relate at all other indices. As with algebraic types, correlations between values inside arrays track whether parts of the values are equal to each other; and do not track numeric relations other than equality.

When it comes to analyzing the *values* inside arrays using abstract interpretation, two extremes are *array smashing* and *array expansion*, both discussed by [Bla+03]. Array smashing uses a single abstract value for each array. Array expansion uses a different abstract value for each array slot in each array. Array smashing is very scalable, but not very precise. Array expansion is very precise, but at a cost for scalability. Moreover, array expansion is only feasible when the sizes of the arrays that are analysed are statically known.

A compromise between array smashing and array expansion consists in separating array slots into groups, called *segments*, and associate an abstract value to each segment. This approach was first treated by Gopan, Reps and Sagiv [GRS05]. In order to choose

the way of cutting arrays into segments, they first determine n integer variables (given either by heuristics or by the user), and then they partition the indices of the arrays, according to whether they are less than, equal or greater than each one of the n integer variables. [GRS05] do not allow for the segments to be empty, which implies that they sometimes need to consider a *disjunction*, to distinguish the cases where a segments exists (is not empty) or does not exist (would be empty). In the worst case, their disjunction of partitions can consider more than factorial of n different ways of partitioning an array. This can be very costly. A later work by Halbwachs and Péron [HP08], inspired by [GRS05], allows segments to be empty, and avoids having a disjunction. In [HP08], possible indices are partitioned by a family of predicates $(\varphi_p)_{p\in P}$, and for each predicate φ_p, a predicate on arrays' contents ψ_p must hold. This approach can be seen as a conjunction of implications of the form $\bigwedge_p \varphi_p \Rightarrow \psi_p$. If a segment is empty (that is, some φ_p is false on all indices), then the associated implication vacuously holds. A salient point of [HP08] is the ability to express relations between segments of different arrays. Both [GRS05] and [HP08] separate the step of determining which segments to consider, from the step of capturing information on those segments. Cousot, Cousot and Logozzo [CCL11] improve on the scalability of [GRS05] and [HP08], by performing these two analyses simultaneously: the segments being considered evolve during the analysis, together with the properties on those segments.

For analysing arrays that contain algebraic data types, we build on [CCL11]. [CCL11] in turn improves [HP08] and [GRS05]. Hence, we will present these three works in more detail in this chapter, in sections 8.1 to 8.3.

Works done after [CCL11] extend array analysis in different directions. Fulara [Ful12] presents a more general framework that encompasses dictionaries, in addition to arrays. Liu and Rival [LR15a] have looked at non-contiguous arrays partitions, for the programs where the array slots that share similar properties are not adjacent. This can be the case, for example, when arrays are used to implement dictionaries. Li et al. [Li+17a] have looked at array segments specially tailored for induction loops.

8.1 Array segments from [GRS05]

In order to explain the approach from [GRS05], we take as an example the simple initialization function `array_fill` from figure 8.1. This function takes as arguments an array of integers `a`, an initialization value `x` and `n` the size of array `a`. It fills the array `a` by putting the value of `x` at every slot. We call i the value of variable `i` and x the value of

```
void array_fill(int a[], int x, int n) {
  i <- 0;
  while (i < n) {
    // * We look at the abstract value obtained here, after stabilisation
    a[i] <- x;
    i <- i + 1
  }
}
```

Figure 8.1: A function initializing all the slots of an array to the same value.

variable `x`. Like [GRS05], for this example we assume that:

- the analyser knows that the size of the array `a` is the value of the argument `n`, and
- the size of the array `a` is at least 1.

The first step for the analysis defined by [GRS05] is to determine which integer variables to use in order to partition the array `a` into segments. In their approach, these integer variables can be either manually provided by the user, or found by heuristics. For the simple example of figure 8.1, it is enough to partition according to the single integer variable `i`. This is found by the heuristic that looks at which variables are used to access the array. The analysis will hence consider at most three different segments for array `a`:

- The segment $a_{<i}$ of the slots of `a` that have an index strictly smaller than i.
- The segment a_i that summarizes a single array slot, at index i.
- The segment $a_{>i}$ of array slots with an index strictly greater than i.

For each segment, two variables are introduced: one summarizing the indices of the segment, and the other summarizing the values of the segment. For example, for segment $a_{<i}$, the two variables $a_{<i}$.index and $a_{<i}$.value will be introduced. These variables are called *segment variables*.

The different segments do not necessarily exist, depending on the size of the array and the value of variable `i`, which explains why the abstract value being considered is a disjunction: each disjunct corresponds to a way of partitioning the array, according to hypothesis on the size of the array and the value of `i`. When partitioning `a` according to `i` there can be up to six disjuncts:

Disjunct	$\boxed{a_i}$	$\boxed{a_i}\boxed{a_{>i}}$	$\boxed{a_{<i}}\boxed{a_i}$	$\boxed{a_{<i}}\boxed{a_i}\boxed{a_{>i}}$
Information	$i = 0$ $n = 1$ $a_i.\text{index} = 0$	$i = 0$ $n > 1$ $a_i.\text{index} = 0$ $i < a_{>i}.\text{index} < n$	$i = n - 1$ $n > 1$ $a_i.\text{index} = i$ $a_{<i}.\text{value} = x$ $0 \leq a_{<i}.\text{index} < i$	$0 < i < n$ $0 \leq a_{<i}.\text{index} < i$ $a_{<i}.\text{value} = x$ $a_i.\text{index} = i$ $i < a_{>i}.\text{index} < n$

Figure 8.2: Abstract value obtained at the beginning of the loop, after stabilization, by the method of [GRS05], for the function of figure 8.1. We label disjuncts by the way they partition the array `a`.

- The disjunct $\boxed{a_{>i}}$ corresponds to the case where i is negative. Hence all the array slots have an index strictly larger than i and are in the segment $a_{>i}$. In this case, the segments $a_{<i}$ and a_i do not exist.
- The disjunct $\boxed{a_i}$ corresponds to the case where $i = 0$ and the array is of size 1. In this case, the segments $a_{<i}$ and $a_{>i}$ do not exist.
- The disjunct $\boxed{a_i}\boxed{a_{>i}}$ corresponds to the case where $i = 0$ and the array has strictly more than one slot. In this case, the segment $a_{<i}$ does not exist.
- The disjunct $\boxed{a_{<i}}\boxed{a_i}\boxed{a_{>i}}$ corresponds to the case where the array has strictly more than two slots and i is an index in the array index range, other than the first or the last index.
- The disjunct $\boxed{a_{<i}}\boxed{a_i}$ corresponds to the case where the array has strictly more than one slot, and i is the last index of the array.
- The disjunct $\boxed{a_{<i}}$ corresponds to the case where all the array slots have an index strictly smaller than i.

Figure 8.2 shows the abstract value obtained by the analysis at the beginning of the while loop, after the widening has converged. Like [GRS05], we represent the abstract value as a table, where each column represents a different disjunct. The first line describes how that disjunct partitions the array, and the second line describes the information gathered on the array by that disjunct. The two first disjuncts, $\boxed{a_i}$ and $\boxed{a_i}\boxed{a_{>i}}$, correspond to the first iteration of the loop, since they assume $i = 0$. At the first iteration, no slots

of the array have been initialized yet, hence the only information captured corresponds to the disjuncts' assumptions and the segments' definitions. The third disjunct $\boxed{a_{<i}}\boxed{a_i}$ corresponds to the last iteration of the loop ($i = n - 1$) while the disjunct $\boxed{a_{<i}}\boxed{a_i}\boxed{a_{>i}}$ corresponds to all the iterations of the loop that are neither the first nor the last one. In both the third and the forth disjunct, we see the constraint $a_{<i}.\text{value} = x$, that correctly captures that the array slots at indices strictly smaller than i have been initialized with value x. For this example, only 4 of the 6 possible disjuncts appear in the disjunction. The disjunct $\boxed{a_{>i}}$ does not appear because `i` is initialized to 0 and only increases. Hence i is never negative. Since the array slot at 0 has an index greater than or equal to i, it cannot be part of the segment $a_{>i}$. Therefore the disjunct that considers a single segment $a_{>i}$ does not partition the whole array. The disjunct $\boxed{a_{<i}}$ does not appear because the loop condition $i < n$ ensures that the array slot at $n - 1$ has an index smaller or equal to i. Hence that slot cannot be part of the segment $a_{<i}$. Therefore, the disjunct that considers a single segment $a_{<i}$ does not partition the whole array.

When considering more than one integer variable to partition arrays, which may be needed for more complicated examples, the number of disjuncts considered by [GRS05] can grow factorially. To avoid this disjunctive aspect, later work by [HP08] and [CCL11] allow segments to be empty.

8.2 Slice variables and shift variables from [HP08]

The approach by [HP08] is inspired from the approach of [GRS05]. However, it allows for empty segments, which avoids having to consider as many cases as [GRS05]. In [HP08], each abstract value is a triple $\left(\Phi, (\varphi_p)_{p\in P}, (\psi_p)_{p\in P}\right)$ where

- Φ is a formula on numeric variables that holds independently of any array segment.
- Each formula in the family $(\varphi_p)_{p\in P}$ describes a set of array indices, by stating numeric inequalities on a special variable l (assumed to be fresh) that represents array indices. For example, the formula $\varphi = (0 \leq l \leq 5)$ denotes the set of array indices $\{0, \ldots, 5\}$. In addition to l, a formula φ_p may also refer to the other integer variables of the program. Because of this, two formulas in $(\varphi_p)_{p\in P}$ may describe the same set of array indices, while making different assumptions on the other program variables. For example, both formulas $\varphi_1 = (1 = l < j < i)$ and $\varphi_2 = (1 = l = j < i)$ describe the singleton $\{1\}$ as the possible values of the array index; but they make different

assumptions on the values of i and j. This is similar to how, in [GRS05], different partitions in the disjunction can make different assumptions on the value of the integer variables of the program. However, unlike [GRS05], in [HP08] the set of indices described by a φ_p might be empty. The abstract domain of [GRS05] *needs* to consider different partitions in the disjunction to distinguish cases where a given segment exists or not. Whereas in [HP08], if the set of indices described by a given φ_p is empty, this is not a problem at all; and it does not require introducing any additional formulas in the conjunction of implications.

- Each formula ψ_p in the family $(\psi_p)_{p \in P}$ describes information that holds for the contents of arrays whenever φ_p holds.

To illustrate this, we take the same example that in section 8.1: the abstract value obtained at the beginning of the while loop, after stabilization of the widening, when analyzing the `array_fill` function from figure 8.1. The abstract value that we get at this program point using the method of [HP08] is the following. The information gathered independently of array segments is $\Phi = (0 \leq i < n)$. Three different sets of array indices are considered. $\varphi_1 = (0 \leq l < i < n)$ for the indices before i, $\varphi_2 = (0 \leq l = i < n)$ for the array slot at index i, and $\varphi_3 = (0 \leq i < l < n)$ for the indices after i. The information captured for segments is given by the three abstract values $\psi_1 = (a = x)$, $\psi_2 = \top$, and $\psi_3 = \top$ respectively. The $(\varphi_p)_{p \in P}$ and $(\psi_p)_{p \in P}$ must be understood as a conjunction of implications. For example, for segment 1, we have $\forall l, \varphi_1(l) \Rightarrow a[l] = x$. The denotation of $(\Phi, (\varphi_p)_{1 \leq p \leq 3}, (\psi_p)_{1 \leq p \leq 3})$ is the same than the disjunction of partitions from figure 8.2. In other words, the approaches by [GRS05] and [HP08] have the same precision on this example program. However, instead of having 4 disjuncts (of 1 to 3 segments each) like [GRS05], [HP08] only considers a conjunction of 3 implications: one per segment. This improved conciseness is possible thanks to the fact that segments are allowed to be empty, in which case the corresponding implications are vacuously true. For example, the case where the array has size 1 — that is, $n = 1$ — corresponds to a case where the formula $\varphi_1 = (0 \leq l < i < n)$ is unsatisfiable, which means that the implication $\varphi_1(l) \Rightarrow a[l] = x$ holds for any index l, since the left-hand side of the implication is false for any index l.

Unlike [GRS05], [HP08] can express relations between segments from two different arrays. For example, if a program manipulates two arrays `a` and `b`, a constraint $\psi_p = (a = b)$ for some segment φ_p would mean $\forall l, \varphi_p(l) \Rightarrow a[l] = b[l]$, *i.e.* the point-wise equality of the two arrays on the given segment. Given that the variables representing the arrays in the

abstract values $(\varphi_p)_{p \in P}$ have this *point-wise* meaning, Halbwachs and Péron call them *slice variables*, to distinguish them from the *segment variables* of [GRS05]. Additionally, [HP08] also introduces *shift variables*, that are like slice variables but shifted by a constant. For example, for an array `a`, the shift array a^{-1} corresponds to the array `a` where array accesses are shifted by the constant -1. Hence, for a segment described by φ_p, the constraint $\psi_p = (a^{-1} \leq a)$ would mean that the array `a` is sorted on segment p. Indeed, the corresponding implication is $\forall l, \varphi_p(l) \Rightarrow a[l-1] \leq a[l]$.

8.3 Segmentations from [CCL11]

Both [GRS05] and [HP08] separate the process of determining which segments to consider, and the analysis on the values in those segments. Inspired by [GRS05; HP08], Cousot, Cousot and Loggozo [CCL11] propose a new approach in which each array is abstracted by a *segmentation*. A segmentation describes both a way of partitioning array indices into segments, and information on the array values inside those segments. As in [HP08], segments are allowed to be empty. Unlike [GRS05] and [HP08], the way of partitioning array indices into segments is not directly determined by the program's syntax nor provided by the user. Instead the way the partition is chosen is guided by the semantic of the program. The abstract operators and transfer functions on segmentations change the partition being considered, during the abstract interpretation of the program. For each segment of the array, a *summary* is provided, that is an abstract value that denotes the possible values that the array may hold at the indices of the segment. For example, the segmentation

$$\{0\} \quad \top \quad \{i\}? \quad [0, +\infty] \quad \{7\} \quad [-\infty, 2] \quad \{|a|\}?$$

denotes a set of arrays of integers, where three segments have been selected:

- The first segment denotes the indices that are greater or equal to 0 and that are strictly less that the value of the variable i. The values in the arrays at such indices may have any value, as indicated by the $\top$ summary. The presence of the ? symbol tells that this segment might be empty.
- The second segment represents the indices that are greater or equal to the value of i and that are strictly smaller than 7. The absence of a ? symbol indicates that this segment cannot be empty. Moreover, the values that are stored at indices between i and 7 must belong to the abstract value $[0, +\infty]$.

- The last segment deals with indices that span from 7 to the end of the array a. The expression $|a|$ denotes the length of the array stored in the variable a. This segment might be empty, as mentioned by the ? symbol, and the values in this segment must belong to $[-\infty, 2]$.

The segments are delimited by *boundsets*—$\{0\}$, $\{i\}$, $\{7\}$ and $\{|a|\}$—that are non-empty sets of expressions. Each boundset might contain more than one expression, and all the expressions a boundset contains must evaluate to the same value. For example, the segmentation

$$\{0\} \quad \top \quad \{i; j+1\}? \quad [0, +\infty] \quad \{7\} \quad [-\infty, 2] \quad \{|a|\}?$$

contains all the information of the previous example, and adds the additional information that the expressions i and $j+1$ must be equal.

The least precise segmentation for an array stored in the variable a is $\{0\}\ \top\ \{|a|\}?$. It only defines one segment (from indices 0 included to $|a|$, the size of the array), that may be empty, and it gives no information for the values stored in this segment, as stated by the abstract value $\top$.

A segmentation also introduces two special variables l and v, that might be used inside segment summaries. The variable l refers to some index of the segment, and v refers to the value that is stored at that index. For example, the segmentation

$$\{0\} \quad l \leq v < l + 3 \quad \{|a|\}$$

describes a non-empty array where each value is greater than or equal to the index at which it is stored, and is less than this index plus 3. The variables l and v are bound by the segmentation, they are not free variables. Therefore, they can be arbitrarily renamed using fresh variables. This domain manages to express point-wise relations between an array index and the value stored at that index in the array. This is done in segment summaries, through relations between the variables l and v (like the relation $l \leq v < l+3$ above). This kind of index-value relations cannot be captured by either [GRS05] nor [HP08]. However, this domain does not manage to express relations between segments of two different arrays, or two segments of the same array; while [HP08] does.

If we look at the same example as in sections 8.1 and 8.2 — the abstract value at the loop head, after stabilization of the widening, for the function `array_fill` from figure 8.1 — we get a more concise abstract value. Indeed, at that program point, the array `a` is

summarized by the segmentation $\{0\}\ v = x\ \{i\}?\ \top\ \{|a|\}$ which only considers two segments: the segment for indices between 0 included and i excluded (which may be empty), and the segment for indices between i included and n excluded (which is necessarily non-empty). Actually, for this example, there are some program points for which this analysis will consider the same three segments as [HP08] would: that is the case after the assignment `a[i] <- x`. However, abstract union, abstract intersection and widening on segmentations can reduce the number of segments. This explains why at the head of the loop, after widening, we get only two segments with [CCL11].

8.4 Works that are not based on abstract interpretation

Some works analyse arrays, but rely on frameworks different from abstract interpretation. Bradley, Manna and Sipma [BMS06] focus on a particular fragment of array theory and translate it to quantifier-free formulas on the theories of un-interpreted functions and Presburger arithmetic. Habermehl, Iosif and Vojnar [HIV08] use Büchi counter automata as models. Jhala and McMillan [JM07] combine Counter-Example Guided Abstraction Refinement with the deduction of Craig Interpolants from proofs of unreachability of certain program paths.

As explained earlier, in next chapter we combine our work from Part I with an analysis of arrays, in order to be able to analyse programs in which arrays contain algebraic types. Since Part I uses abstract interpretation, we have decided to extend an approach for analysing arrays that also uses abstract interpretation; namely the approach of Cousot, Cousot and Logozzo [CCL11].

Chapter 9

Extending Segmentations to Arrays of Structured Values

In this chapter, we extend our abstract domain and analysis to support functional arrays. Unlike arrays found in most programming languages, functional arrays cannot be modified in-place. As such, the update operation creates a new array whose contents are the same as in the original array, except for the cell for which an update has been requested.

Such arrays are used in SMT solvers that handle array theories, but are also available in theorem provers such as Coq. Some formal development, such as seL4 [Kle+09], employ a model of functional arrays—they use functions with natural numbers as a domain—to represent tables that contain thread descriptors in the state of their operating system micro-kernel.

As explained in Chapter 8, in order to analyse programs that manipulate algebraic types and arrays containing values from algebraic types, we extend the segmentation approach from [CCL11], that we presented in section 8.3.

Differences with [CCL11] segmentations In this chapter, we extend [CCL11] segmentations in two directions. First, we allow segment summaries to refer to other program variables, including to parameters of a function. For example, if n is an integer variable, the segmentation $\{0\} \quad v = n - l \quad \{|a|\}?$ describes a set of (possibly empty) arrays of the form $[\ n\ ;\ n-1\ ;\ n-2\ ;\ \ldots\ ;\ n-(|a|-1)\ ]$.

Second, we allow for arrays to contain values from algebraic types. Hence, we use for segment summaries the abstract values of the domain that we introduced in Part I (structural lifting).

Finally, we give a new definition for the abstract inclusion between segmentations, as we have found corner cases in the [CCL11] definitions where the concretisation for segmentations was not monotonic with respect to the inclusion relation.

Chapter outline This chapter is organised as follows: First, we extend our programming language with primitives for arrays and provide a motivating example (section 9.1). Second, we give an overview the overall structure of our abstract domain (section 9.2). Then, we describe our segmentations (section 9.3), and we focus on the differences between our definitions and the ones of [CCL11] (section 9.4). In section 9.7, we give the soundness theorem of our abstract domain. Then, we give the result of our analysis on the motivating example (section 9.8). Finally, we summarize our approach and its current limitations (section 9.9).

9.1 Extension of the Language and Motivating Example

We extend the syntax of types and values with arrays. We allow for arrays to contain algebraic types, but we do not handle, for now, arrays nested inside algebraic types nor arrays nested inside other arrays. In order to enforce this, we separate, in the definitions, algebraic types $\tau^{\text{alg}} \in \text{AlgTypes}$ from array types $\tau^{\text{arr}} \in \text{ArrTypes}$, and values of algebraic types $v^{\text{alg}} \in \text{AlgValues}$ from values of array types $v^{\text{arr}} \in \text{ArrValues}$. The definition of algebraic types and values remains exactly the same as in definition 1 on page 13:

$$\begin{array}{rcl} \tau^{\text{alg}} \in \text{AlgTypes} & ::= & \text{Int} \quad | \quad \{\overline{f_i \to \tau_i^{\text{alg}}}^{i\in I}\} \quad | \quad [\overline{A_i \to \tau_i^{\text{alg}}}^{i\in I}] \\ v^{\text{alg}} \in \text{AlgValues} & ::= & \underline{n} \quad | \quad \{\overline{f_i = v_i^{\text{alg}}}^{i\in I}\} \quad | \quad A(v^{\text{alg}}) \end{array}$$

Array types and values are defined as follows:

$$\begin{array}{rcl} \tau^{\text{arr}} \in \text{ArrTypes} & ::= & \text{Array}\left(\tau^{\text{alg}}\right) \\ v^{\text{arr}} \in \text{ArrValues} & ::= & [v_1^{\text{alg}} \ ; \ \ldots \ ; \ v_k^{\text{alg}}] \end{array}$$

Each variable has either an algebraic type or an array type, hence the types of the language are $\text{Types} = \text{AlgTypes} \cup \text{ArrTypes}$. The values are $\text{Values} = \text{AlgValues} \cup \text{ArrValues}$. For example, if we consider the type `status` defined by

```
type status = [Running of { count: int } | Asleep of { secs: int; count: int }]
```

then an example of an array of type Array(`status`) is

$$[\ \text{Running}\,\{\text{count} = 5\}\ ;\ \ \text{Asleep}\,\{\text{secs} = 42; \text{count} = 7\}\ ]$$

which is an array of size two, and of type Array (status).

We add three new commands to the language of section 2.2, that deal with array creation, array access, and array updates, respectively. We also add a new case of expressions, written $|x|$, that denotes the length of the array that is stored in a variable x.

$$
\begin{array}{lcll}
c \in \text{Cmd} & ::= & \ldots & | \quad y := \mathsf{new\_array}(\tau^{\text{alg}}, e_1, e_2) \quad \text{(array creation)} \\
 & & & | \quad y := x[e] \quad \text{(array access)} \\
 & & & | \quad y := x[e_1 \to e_2] \quad \text{(array update)} \\
e \in \text{Exp} & ::= & \ldots & | \quad |x| \quad \text{(array length)}
\end{array}
$$

The creation of a new array $x := \mathsf{new\_array}(\tau^{\text{alg}}, e_1, e_2)$ initialises the variable x with a new array. This construct takes as parameters the type τ^{alg} of the values contained in the array, an expression e_1 for the size of the array, and an expression e_2 for the initial values of all the array slots. The array access $y := x[e]$ loads in the variable y the contents of the array that is stored in x at the index e. The array update $y := x[e_1 \to e_2]$ stores in the variable y a new array that differs from the array stored in x at one index only: the new array contains at index e_1 the result of the evaluation of e_2. The array length $|x|$ refers to the length of the array stored inside variable x.

The restriction that the main array operations (creation, access, update) should be assigned to a variable before further manipulation does not restrict the expressiveness of the language, but simplifies the analysis.

Motivating example Figure 9.1 recalls the function `find_max_priority` that was also presented in the introduction (figure 1.2). Function `find_max_priority` is an example of a function that manipulates arrays. It is inspired by a function from the functional specification of the seL4 micro-kernel [Kle+14]. It involves thread descriptors—named *Thread Control Blocks*, or TCBs for short—that represent information about the threads that are managed by an operating system kernel. TCBs are records of properties. To keep the example short, we only exhibit one property of TCBs—their priority—although TCBs may have more.

The `find_max_priority` function searches in an array of TCBs one TCB whose priority is the highest. It returns an option type, such that either `NoMax{}` is returned if the array is empty, or `SomeMax d` is returned, where `d` is a TCB in the array, with the highest priority.

In the rest of the chapter, we explain how we build the Diorana abstract domain, that allows to obtain the abstract value from figure 9.3. The construction of this domain uses

```
type unit = {} (* Record type with no fields *)

(* Thread descriptors (Thread Control Block) *)
type tcb =
{ prio      : int; (* Priority *)
  ... (* Other fields of the TCB are elided *)
}

(* An array of TCBs. Represents a scheduler queue. *)
type queue = tcb[]

(* Options of TCBs. Serves as a return type for find_max_priority *)
type max_result = [ NoMax of unit | SomeMax of tcb ]

(* Returns the TCB with the hightest priority in the queue, if any. *)
def find_max_priority(queue q) : max_result = {
  max_result res
  unit       case
  int        i
  tcb        challenger

  i = 0
  res = NoMax{}
  while (i < |q|) do (* Iterate over the queue *)
    challenger = q[i]
    branch
      case = res@NoMax (* First iteration *)
      res  = SomeMax challenger
    or
      assert(challenger.prio > res@SomeMax.prio) (* Higher priority found *)
      res = SomeMax challenger
    or
      assert(res@SomeMax.prio >= challenger.prio) (* No change needed *)
    end
    i = i + 1
  end
  return res
}
```

Figure 9.1: Program that finds a thread descriptor with highest priority in an array.

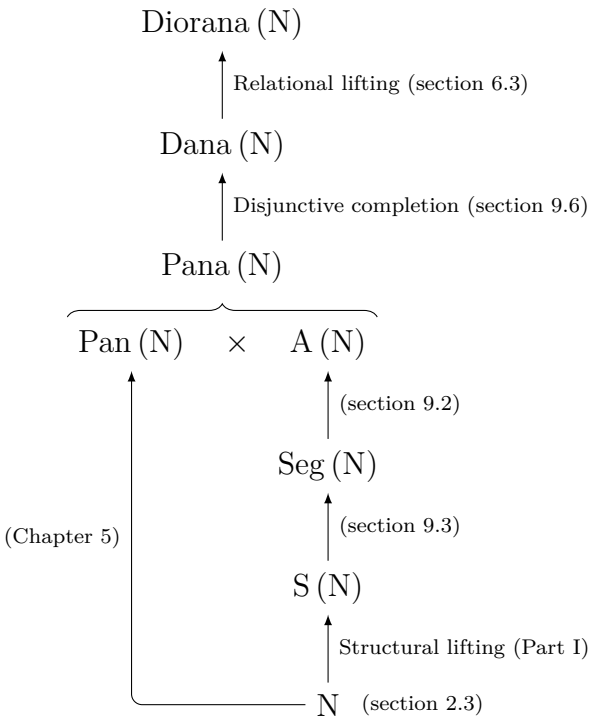


Figure 9.2: The construction of the abstract domain for analyzing programs that manipulate both values from algebraic types, and arrays containing values from algebraic types.

some of the same ingredients than the RAND domain. It allows to compute input-output summaries of functions that manipulate arrays and algebraic types.

9.2 Structure of our Abstract Domain for Arrays

Figure 9.2 summarizes the different components of the construction. Extending the structural lifting allows us to define segmentations for abstract arrays whose contents are values from algebraic types (section 9.3). Then, a domain associates a segmentation to each array variable. We call this domain $\mathrm{A}\,(\mathrm{N})$. To also handle non-array variables, we take a product between the $\mathrm{Pan}\,(\mathrm{N})$ domain from section 5.3 and the array domain $\mathrm{A}\,(\mathrm{N})$. We call this product domain $\mathrm{Pana}\,(\mathrm{N}) = \mathrm{Pan}\,(\mathrm{N}) \times \mathrm{A}\,(\mathrm{N})$ for *Product domain for Algebraic types, Numbers and Arrays*. Then, we take a disjunctive completion of this product domain, to handle incompatible cases for constructors, like we did for the Pan domain. We call this disjunctive completion $\mathrm{Dana}\,(\mathrm{N})$, for *Disjunction for Algebraic types, Numbers and Arrays*. Finally, we apply the relational lifting of section 6.3 to get a domain that expresses relations between two different program states (input and output). We call this domain

Diorana, for *Domain for Input-Output Relations on Algebraic types, Numbers and Arrays.*

9.3 Array Segmentations

For each variable x, the expression $|x|$ represents the length of the array contained in x, if any. For a typing context Γ, we call $\mathrm{V}(\Gamma)$ the set of variables of numeric types and of array lengths expressions. Formally, $\mathrm{V}(\Gamma) = \{x \mid \Gamma(x) = \mathrm{Int}\} \cup \bigcup_{\tau \in \mathrm{AlgTypes}} \{|x| \mid \Gamma(x) = \mathrm{Array}(\tau)\}$. We also call K the set of possible numeric constants in the programming language. In particular, $0 \in \mathrm{K}$.

Definition 30 (Bound expressions and boundsets)**.** *We call* bound expression *any element of the set* $\mathrm{E}(\Gamma) = \mathrm{K} \cup \{x + k \mid x \in \mathrm{V}(\Gamma) \wedge k \in \mathrm{K}\}$*. We call* boundset *any finite set of bound expressions,* i.e.*, an element of* $\mathcal{P}_{\mathit{fin}}(\mathrm{E}(\Gamma))$*.*

In the examples that follow, we will write x instead of $x + 0$, when $x + 0 \in \mathrm{E}(\Gamma)$. The bound expressions $\mathrm{E}(\Gamma)$ will be used as formal bounds, that delimit the array segments in segmentations.

We write $(z_i)_{i \in \{1,\ldots,n\}}$ for the finite sequence of elements $z_1, z_2, \ldots, z_n$. The definition of segmentations follows.

Definition 31 (Segmentations)**.** *Let x be a variable with an array type in the typing context Γ,* i.e.*, $\Gamma(x) = \mathrm{Array}(\tau)$ for some algebraic type $\tau \in \mathrm{AlgTypes}$. A* segmentation $s \in \mathrm{Seg}(\mathrm{N})(\Gamma)(x)$ for variable x *is a quadruplet* $\left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\ldots,n\}}\right)$ *where:*

- *l and v are distinct fresh variables,* i.e.*, elements of* $\mathrm{Fresh}(\mathrm{dom}(\Gamma))$ *such that $l \neq v$, and*
- *each b_i for $i \in \{0, \ldots, n\}$ is a boundset, and*
- *each d_i for $i \in \{1, \ldots, n\}$ belongs to* $\mathrm{S}\,(\mathrm{N})\,((\Gamma \setminus \{x\})[l \mapsto \mathrm{Int}; v \mapsto \tau])$*, and*
- *each m_i for $i \in \{1, \ldots, n\}$ is a boolean.*

In a segmentation $\left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\ldots,n\}}\right)$, l is a variable that refers to the indices of the array inside segment summaries, whereas the variable v refers to the values of the array inside segment summaries. Segmentations behave like binders for the special variables l and v, in the same way a λ-abstraction would, in a λ-calculus. Hence, these

variables can be replaced by any other variables, as long as they are sufficiently *fresh*, so that accidental captures are avoided.

Each b_i for $i \in \{0, \ldots, n\}$ is a boundset that marks the segment limits, and each d_i for $i \in \{1, \ldots, n\}$ is a segment summary, that denotes the set of values that a segment can contain. Finally, each m_i for $i \in \{1, \ldots, n\}$ is a boolean that indicates whether the preceding segment is allowed to be empty.

In the examples that follow, we omit the special variables l and v, and we write boolean markers as ? when they are equal to $\mathtt{tt}$ and omit them when they are equal to $\mathtt{ff}$. For example, we write

$$\{0\}\ (\mathbf{NPR} : \{0 \leq l < i; v = l\})\ \{i; 5\}\ \top^{\mathrm{S}}\ \{|x|\}?$$

for the segmentation $(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,2\}})$ where $b_0 = \{0\}$, $b_1 = \{i; 5\}$, $b_2 = \{|x|\}$, $d_1 = (\mathbf{NPR} : \{0 \leq l < i; v = l\})$, $d_2 = \top^{\mathrm{S}}$, $m_1 = \mathtt{ff}$ and $m_2 = \mathtt{tt}$.

Each segment summary d_i is an abstract value from the structural lifting of Part I. The segment summaries of a segmentation for the array x cannot refer to the variable x, but they can refer to other program variables, as well as to the special variables l and v, that represent the index and the value of the different array slots, respectively. This is why we take segment summaries in $\mathrm{S}(\mathrm{N})((\Gamma \setminus \{x\})[l \mapsto \mathrm{Int}; v \mapsto \tau])$, that is the structural lifting of numeric domain N, for typing context $(\Gamma \setminus \{x\})[l \mapsto \mathrm{Int}; v \mapsto \tau]$. This is the typing context obtained from Γ by removing variable x, and adding variable l of type Int and variable v of type τ (the type of the values inside the array).

In the rest of this manuscript, we only consider well-formed segmentations, that are defined as follows.

Definition 32 (Well-formed segmentations)**.** *Let* $s = \left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\ldots,n\}}\right)$ *be a segmentation for the array variable* x*. We say that the segmentation* s *is* well-formed *if it satisfies the following properties:*

- $0 \in b_0$*, and*
- $|x| \in b_n$*, and*
- $\forall i \in \{0, \ldots, n\}, b_i \neq \emptyset$*, and*
- $\forall i \in \{0, \ldots, n\}, \forall j \in \{0, \ldots, n\}, i \neq j \Rightarrow b_i \cap b_j = \emptyset$.

The indices of an array always range from 0 to the length of the array minus one. This is why we require that the first boundset of a well-formed segmentation contains 0 and the last boundset contains the length of the array (the segments include their left boundset and exclude their right boundset).

Boundsets are required to be non-empty. Indeed, boundsets should evaluate to array indices, as they delimit the range of array indices of each segment, and an empty boundset cannot be evaluated. To prevent an operation on segmentations from creating an empty boundset, we may need to merge the segment summaries on each side of a boundset using abstract union. We will see later in section 9.4.1, that the *unification* of segmentations is such an operation.

The concretisation of segmentations (definition 35) will enforce that all the expressions of a given boundset must evaluate to the *same* concrete value. Therefore, we require that distinct boundsets do not intersect, in order to avoid having boundsets that are artificially split.

The constraint that all the expressions of a boundset must evaluate to the same value is formalised as follows, with the definition of the concretisation of boundsets.

Definition 33 (Concretisation for boundsets)**.** *The concretisation of a boundset b is the set of environments defined by* $\gamma^{\mathrm{B}}(b) = \{\rho \mid \forall e_1 \in b, \forall e_2 \in b, [\![e_1]\!]^{\mathrm{exp}}_{\rho} = [\![e_2]\!]^{\mathrm{exp}}_{\rho} \neq \emptyset\}$.

For a boundset b and an environment $\rho \in \gamma^{\mathrm{B}}(b)$ that belongs to the concretisation of b, all the expressions in the boundset, if any, must evaluate to the same value. Hence, it makes sense to talk about the *evaluation of a boundset* b, as the evaluation of any bound expression contained in b.

Definition 34 (Evaluation of a boundset)**.** *For any non-empty boundset b and any store $\rho \in \gamma^{\mathrm{B}}(b)$, we call* evaluation of boundset b in store ρ*, written $[\![b]\!]^{\mathrm{exp}}_{\rho}$, the value v such that $[\![e]\!]^{\mathrm{exp}}_{\rho} = \{v\}$, for $e \in b$.*

Definition 33 guarantees both that $[\![e]\!]^{\mathrm{exp}}_{\rho}$ is a singleton for any $e \in b$, and that any choice of e in b gives the same result.

The definition of the concretisation of abstract segmentation follows.

Definition 35 (Concretisation for segmentations)**.** *For a segmentation*

$$s = \left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\ldots,n\}}\right)$$

its concretisation $\gamma^{\text{Seg}}(s)$ *is the set of pairs composed of an environment* ρ *and an array value* t*, that satisfy the following conditions:*

- *Equalities in each boundset:* $\forall i \in \{0, \ldots, n\}, \rho \in \gamma^{\text{B}}(b_i)$
- *Inequalities between boundsets:* $\forall i \in \{1, \ldots, n\}, [\![b_{i-1}]\!]_\rho^{\text{exp}} \leq [\![b_i]\!]_\rho^{\text{exp}}$
- *Strict inequalities for non-empty segments:*

$$\forall i \in \{1, \ldots, n\}, (\neg m_i) \Rightarrow [\![b_{i-1}]\!]_\rho^{\text{exp}} < [\![b_i]\!]_\rho^{\text{exp}}$$

- *Segment summaries:*

$$\forall i \in \{1, \ldots, n\}, \forall j,\ [\![b_{i-1}]\!]_\rho^{\text{exp}} \leq j < [\![b_i]\!]_\rho^{\text{exp}} \Rightarrow \rho[l \mapsto j; v \mapsto t[j]] \in \gamma^{\text{S}}(d_i)$$

- *Array size:* $|t| = [\![b_n]\!]_\rho^{\text{exp}}$.

The five conditions of definition 35 guarantee that the array t corresponds to the information described by the segmentation. But the segmentation also gives information on store ρ, through the "*Equalities in each boundset*" and "*Segment summaries*" conditions. This is why the concretisation is a set of pairs composed of a store and of an array, instead of only a set of arrays. For example, the segmentation $\{0\}\ \top^{\text{S}}\ \{i; 5\}\ \top^{\text{S}}\ \{|x|\}?$ tells us that the value stored in variable i must be 5, and that the length of the array stored in variable x must be greater than or equal to 5.

The "*Equalities in each boundset*" condition guarantees that for each boundset, all the bound expressions concretise to the same integer value. Without this condition, the other four conditions of the definition would not be well-defined, as they refer to the evaluation $[\![b_i]\!]_\rho^{\text{exp}}$ of the different boundsets b_i.

The "*Inequalities between boundsets*" guarantees that the boundsets are in increasing order, with respect to their evaluation. Since we only consider well-formed segmentations (definition 32), we know that $0 \in b_0$, hence the first boundset evaluates to zero ($[\![b_0]\!]_\rho^{\text{exp}} = 0$).

The last condition "*Array size*" guarantees that the last boundset evaluates to the size of the array t ($[\![b_n]\!]_\rho^{\text{exp}} = |t|$).

For well formed segmentations, the conditions "*Equalities in each boundset*", "*Inequalities between boundsets*" and "*Array size*" guarantee that the boundsets describe a *partition* into intervals of the indices of array t. These intervals of indices might be empty.

The "*Strict inequalities for non-empty segments*" condition enforces that, whenever the emptiness marker of a segment is false, then the corresponding interval of array indices is not empty, *i.e.*, the strict inequality $[\![b_{i-1}]\!]_\rho^{\text{exp}} < [\![b_i]\!]_\rho^{\text{exp}}$ must be satisfied.

The main difference between the segmentations defined here and the ones defined by [CCL11] is that the numeric abstract values used to summarize each segment can refer to the other variables of the program, *in addition* to refering to the index and value of the array slots. For this purpose, the "*Segment summaries*" condition guarantees that the information given by an array summary d_i expresses the relations that hold between any array index j, the value stored at that index in array t, and all other variables in store ρ.

9.4 Unification and Inclusion for Segmentations

This section starts by discussing the differences between our definition of segmentation inclusion, and the one from [CCL11]. We first give an intuitive explanation of an operator called *unification*, that is used in [CCL11]'s definition of segmentation inclusion. Then, we present the problem of monotonicity for concretisation with respect to inclusion that we found in [CCL11] definitions (section 9.4.2), and we present our definition of segmentation inclusion (section 9.4.3). Lastly, we discuss the two other differences between our definitions and the one from [CCL11]: the presence of other program variables inside segmentation summaries, and values of algebraic types inside arrays (section 9.5).

9.4.1 Unification of Segmentations

In [CCL11], intersection, union, widening and inclusion test on segmentations are performed in two steps: first, *unification* is performed to obtain two segmentations that might be less precise, but that share the same boundsets to delimit segments; then, intersection, union, widening and inclusion test are performed segment per segment.

We say that two segmentations are *unified* when they share the same boundsets. In other words, two unified segmentations may only differ by their segment summaries and their emptiness markers.

Definition 36 (Unified segmentations)**.** *Let* $s^1 = \left(l, v, b_0^1, (d_i^1, b_i^1, m_i^1)_{i\in\{1,\dots,n\}}\right)$ *and* $s^2 = \left(l, v, b_0^2, (d_i^2, b_i^2, m_i^2)_{i\in\{1,\dots,p\}}\right)$ *be two segmentations.* s^1 *and* s^2 *are called* unified *if and only if the two following conditions are satisfied:*

- $n = p$*, and*

- $\forall i \in \{0, \ldots, n\}, b_i^1 = b_i^2$.

In order to transform two arbitrary segmentations into two *unified* segmentations, two basic transformations on segmentations can be applied:

Removal of bound expressions This amounts to forgetting equalities between expressions. If the removal of bound expressions might create an empty boundset, then the two enclosing summaries are joined, so that no empty boundset is created.

Split of a boundset into two parts Again, this amounts to forgetting equalities between expressions. It has the effect of creating a new segment with an emptiness marker set to $\mathtt{tt}$, and with a summary that has to be chosen, depending on what operation—union, intersection, widening, or inclusion test—is to be performed.

For example, in order to unify segmentation $s^1 = \{0; a; b\}\ d^1\ \{|x|\}$ with segmentation $s^2 = \{0; a\}\ d^2\ \{|x|\}$, we can remove the bound expression b from s^1. This yields $s^{1'} = \{0; a\}\ d^1\ \{|x|\}$, which is indeed unified with s^2. If instead we had started with $s^1 = \{0; a\}\ d_1^1\ \{b\}\ d_2^1\ \{|x|\}$, then removing bound expression b from s^1 remains possible, but it implies merging together, with abstract union, the segment summaries d_1^1 and d_2^1. We get $s^{1'} = \{0; a\}\ d_1^1 \sqcup d_2^1\ \{|x|\}$, which is unified with s^2.

Removing bound expressions is sufficient to unify two segmentations for the same array variable. Indeed, it is always possible to remove all bound expressions except the ones for 0 and the array length, that are necessarily common to both segmentations. When doing so, the unified segmentations contain a single segment. However, by splitting a boundset into a possibly empty segment, we might obtain more precise unifications. For example, let us consider the two segmentations $s^1 = \{0; a\}\ d^1\ \{|x|\}$ and $s^2 = \{0\}\ d_1^2\ \{a\}\ d_2^2\{|x|\}$. Instead of removing the bound expression a from both segmentations, we can split the boundset $\{0; a\}$ of segmentation s^1 into a possibly empty segment $\{0\}\ \ldots\ \{a\}?$, where the choice of the abstract value that serves as a summary for this new segment depends on what operation the unification is performed for. For example, if the goal is to compute the abstract union of s^1 and s^2, then their unification can choose $\bot$—*i.e.*, the neutral element of abstract union—as summaries for the new segments that might be created by a split. We would get, for this unification, $s^{1'} = \{0\}\ \bot\ \{a\}?\ d^1\{|x|\}$, which is unified with s^2.

We refer the reader to section 11.4 of [CCL11] for a detailed description of the [CCL11] algorithm for unification, that uses bound expression removal and boundset splitting. Their algorithm works by doing a parallel traversal of the two segmentations. Any bound

expression that is only present in one of the two segmentations is removed. As long as common bound expressions are found at the current traversal position, the algorithm splits the boundsets to keep the bound expressions that are present in the two segmentations, and then continues the traversal. However, if at some point the traversal arrives at a position where there are no bound expressions in common, it backtracks in order to remove bound expressions that resulted from previous splits.

Removing bound expressions and splitting boundsets yields segmentations that are less precise than the initial ones. For this reason, it is sound to perform a unification before union, intersection and widening (as done both by [CCL11] and by us). For example, when taking the abstract union of two segmentations s^1 and s^2, if we call $s^{1'}$ and $s^{2'}$ the result of unifying them, and if we write $s^{1'} \mathbin{\overline{\sqcup^{\text{seg}}}} s^{2'}$ for the segment-wise union after unification, then we have

$$\gamma^{\text{Seg}}\left(s^1\right) \subseteq \gamma^{\text{Seg}}\left(s^{1'}\right) \qquad \gamma^{\text{Seg}}\left(s^2\right) \subseteq \gamma^{\text{Seg}}\left(s^{2'}\right)$$

$$\gamma^{\text{Seg}}\left(s^{1'}\right) \cup \gamma^{\text{Seg}}\left(s^{2'}\right) \subseteq \gamma^{\text{Seg}}\left(s^{1'} \mathbin{\overline{\sqcup^{\text{seg}}}} s^{2'}\right)$$

which allows to conclude $\gamma^{\text{Seg}}\left(s^1\right) \cup \gamma^{\text{Seg}}\left(s^2\right) \subseteq \gamma^{\text{Seg}}\left(s^1 \sqcup^{\text{Seg}} s^2\right)$, which is the soundness lemma for abstract union.

It can be incorrect, however, to apply unification for segmentation inclusion, as it is done in [CCL11]. Indeed, testing the inclusion with a less precise value on the right-hand side of the inclusion does not guarantee that the inclusion still holds for the initial, more precise value. We show an example of this issue below, in section 9.4.2.

9.4.2 On the Unsoundness of Segmentation Inclusion in [CCL11]

In [CCL11], the concretisation for segmentations is not monotonic with respect to the abstract inclusion. The following example illustrates this issue. Let s_1 be the segmentation $\{0\}\top\{a\}\top\{b\}\top\{c\}\top\{|t|\}$ and s_2 be the segmentation $\{0\}\top\{c\}\top\{b\}\top\{a\}\top\{|t|\}$. In [CCL11], segmentation inclusion is tested by first unifying segmentations, then testing inclusion segment-wise. The unification of segmentations s^1 and s^2 yields $\{0\}\top\{b\}\top\{|t|\}$ on both sides. Hence, with [CCL11] definitions, we have $s_1 \sqsubseteq^{\text{Seg}} s_2$. We will show, however, that $\gamma^{\text{Seg}}(s_1) \not\subseteq \gamma^{\text{Seg}}(s_2)$. Let $t_0 = [0;0;0;0]$ be the array of size 4 filled with zeros, and let $\rho_0 = [a \mapsto 1; b \mapsto 2; c \mapsto 3; t \mapsto t_0]$ be an environment. We have $(\rho_0, t_0) \in \gamma^{\text{Seg}}(s_1)$. However, $(\rho_0, t_0) \notin \gamma^{\text{Seg}}(s_2)$, because otherwise, we would have $\rho_0(c) < \rho_0(a)$, *i.e.*, $3 < 1$.

Thus, $\gamma^{\text{Seg}}(s_1) \not\subseteq \gamma^{\text{Seg}}(s_2)$, which proves that the segmentation concretisation from [CCL11] is not monotonic with respect to the segmentation inclusion of [CCL11].

Our definition of segmentation inclusion (definition 37 below) fixes this issue, as proved by lemma 16 below.

9.4.3 A Sound Definition For Segmentation Inclusion

When we define operations that involve two segmentations, we always assume, without loss of generality, that they have the same index and value variables. It is indeed always possible to rename those variables with fresh ones, so that the two segmentations use the same index and value variables.

To define the inclusion test between two segmentations s^1 and s^2, we will avoid computing their unification. Testing inclusion between s^1 and s^2 is not straightforward, since the two segmentations may have different numbers of segments. For this reason, in our definition (definition 37), we introduce a map ϕ between the indices of the boundsets of s^2, and the indices of the boundsets of s^1, that keeps track of which boundsets and segments of s^1 correspond to the boundsets and segments of s^2. The function ϕ therefore identifies how the segments of the two segmentations can be *aligned* with each other.

Definition 37 (Segmentation inclusion)**.** *Let* $s_1 = \left(l, v, b_0^1, (d_i^1, b_i^1, m_i^1)_{i\in\{1,\dots,n\}}\right)$ *and* $s_2 = \left(l, v, b_0^2, (d_i^2, b_i^2, m_i^2)_{i\in\{1,\dots,p\}}\right)$ *be two segmentations for the same array variable, with the same index and value variables. We say that* s_1 *is included in* s_2*, written* $s_1 \sqsubseteq^{\text{Seg}} s_2$*, if and only if there exists a non-decreasing function* $\phi : \{0,\dots,p\} \to \{0,\dots,n\}$ *such that the following conditions are satisfied:*

- *First and last indices:* $\phi(0) = 0 \wedge \phi(p) = n$
- *Boundset inclusion:* $\forall i \in \{0,\dots,p\}, b_i^2 \subseteq b_{\phi(i)}^1$
- *Segment summaries:* $\forall i \in \{1,\dots,p\}, \forall j, \phi(i-1) < j \leq \phi(i), \Rightarrow d_j^1 \sqsubseteq^{\text{S}} d_i^2$
- *Emptiness markers:* $\forall i \in \{1,\dots,p\}, \left(\bigwedge_{j=\phi(i-1)+1}^{\phi(i)} m_j^1\right) \Rightarrow m_i^2$

The smaller a boundset, the less equality constraints it implies on its concretisation. This is why, if s^2 is less precise than s^1 (that is $s^1 \sqsubseteq^{\text{Seg}} s^2$), then the boundsets of s^2 must be smaller than their s^1 counterpart. This is stated by the "*Boundset inclusion*" condition.

The "*Segment summaries*" condition enforces that the aligned segment summaries must be related by the inclusion relation of the abstract domain used for array cells. Several

summaries on the left-hand side might correspond to the summary d_i^2 at index i on the right-hand side. Since d_i^2 is delimited by the boundsets b_{i-1}^2 and b_i^2, the corresponding summaries on the left-hand side are delimited by the boundsets $b_{\phi(i-1)}^1$ and $b_{\phi(i)}^1$. Therefore, the summaries on the left-hand side that are aligned with d_i^2 are the d_j^1 for $\phi(i-1)+1 \leq j \leq \phi(i)$.

The "*Emptiness markers*" condition is similar and considers the markers for the same segments as for the "Segment summaries" condition. It states that if all the segments on the left-hand side might be empty, then the aligned segment on the right-hand side might be empty too.

For example, let us define two segmentations s_1 and s_2 as follows:

$$s_1 = \quad \{0\} \quad (v \leq 0) \quad \{3; y\} \quad \begin{pmatrix} 3 \leq l < 5 \\ v = 2 \times l \end{pmatrix} \quad \{5\} \quad \begin{pmatrix} 5 \leq l < 7 \\ v = 3 \times l \end{pmatrix} \quad \{7\} \quad (v > 14) \quad \{|x|\}?$$

$$s_2 = \quad \{0\} \quad (v \leq 0) \quad \{3\} \quad (v = 0) \quad \{y\}? \quad (v \geq 0) \quad \{7\}? \quad (v > 14) \quad \{|x|\}?$$

We have $s_1 \sqsubseteq^{\text{Seg}} s_2$ because function $\phi = [0 \mapsto 0;\ 1 \mapsto 1;\ 2 \mapsto 1;\ 3 \mapsto 3;\ 4 \mapsto 4]$ is non-decreasing and satisfies all the conditions of definition 37. The function ϕ maps both 1 and 2 to 1. This reflects the fact that s_1 assumes one more equality than s_2 on bound expressions: the equality $y = 3$. The "*Emptiness markers*" condition verifies that this additional equality is allowed by s_2. The fact that 2 is not in the range of ϕ reflects the fact that multiple segments in s_1—namely, segments 2 and 3—correspond to a single segment in s_2—namely, segment 3.

Intuitively, there are multiple reasons why s_1 is more precise than s_2:

- The segmentation s_1 states that y and 3 *must* be equal, because y and 3 belong to the same boundset. The segmentation s_2, however, only requires that $3 \leq y$, because 3 and y are in two boundsets that delimit a segment, that might be empty.
- For the indices between y and 7, the segmentation s_2 only states that the values in the array are non-negative, whereas segmentation s_1 states more precise conditions. Segmentation s_1, indeed, states that between indices y and 5 the values are equal to twice their index, and that the values that lie between indices 5 and 7 are equal to three times their index.

Lemma 15. *The relation $\sqsubseteq^{\text{Seg}}$ is a pre-order.*

Proof sketch. For the proof of transitivity, it suffices to check that the identity function satisfies all the conditions of definition 37. The proof of transitivity is based on the fact that

if two functions ϕ_1 and ϕ_2 satisfy the properties of definition 37, so does their composition $\phi_2 \circ \phi_1$. □

The $\sqsubseteq^{\text{Seg}}$ pre-order is sound, in the sense that the concretisation for segmentations is monotonic with respect to this pre-order.

Lemma 16. *If $s^1 \sqsubseteq^{\text{Seg}} s^2$, then $\gamma^{\text{Seg}}(s^1) \subseteq \gamma^{\text{Seg}}(s^2)$.*

Proof. We write $s^1 = \left(l, v, b^1_0, (d^1_j, b^1_j, m^1_j)_{j\in\{1,\dots,n\}}\right)$ for the different components of s_1 and $s^2 = \left(l, v, b^2_0, (d^2_i, b^2_i, m^2_i)_{i\in\{1,\dots,p\}}\right)$ for the different components of s_2. Let $\phi : \{0,\dots,p\} \to \{0,\dots,n\}$ be the function given by the fact that $s^1 \sqsubseteq^{\text{Seg}} s^2$.

Let $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$. Given definition 35, there are five conditions that we need to prove to show that $(\rho, t) \in \gamma^{\text{Seg}}(s^2)$.

Condition 1 : Equalities in each boundset. We need to prove that for any index $i \in \{0,\dots,n\}$ and any two bound expressions $e_1 \in b^2_i$ and $e_2 \in b^2_i$, we have $[\![e_1]\!]^{\text{exp}}_\rho = [\![e_2]\!]^{\text{exp}}_\rho$. Let $i \in \{0,\dots,n\}$, $e_1 \in b^2_i$ and $e_2 \in b^2_i$. By the "boundset inclusion" property that stems from $s^1 \sqsubseteq^{\text{Seg}} s^2$ (definition 37), we know that $b^2_i \subseteq b^1_{\phi(i)}$. Hence, using the "equalities in each boundset" property of $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$ (definition 35), we have $[\![e_1]\!]^{\text{exp}}_\rho = [\![e_2]\!]^{\text{exp}}_\rho$, which is what we wanted.

Condition 2: Inequalities between boundsets. Here, we want to prove that for any index $i \in \{1,\dots,n\}$, we have $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho \leq [\![b^2_i]\!]^{\text{exp}}_\rho$. For that, we will take two expressions $e_1 \in b^2_{i-1}$ and $e_2 \in b^2_i$ and prove that $[\![e_1]\!]^{\text{exp}}_\rho \leq [\![e_2]\!]^{\text{exp}}_\rho$. Let $i \in \{1,\dots,n\}$, $e_1 \in b^2_{i-1}$ and $e_2 \in b^2_i$. By the "boundset inclusion" property of $s^1 \sqsubseteq^{\text{Seg}} s^2$, we have $b^2_{i-1} \subseteq b^1_{\phi(i-1)}$ and $b^2_i \subseteq b^1_{\phi(i)}$. Hence $e_1 \in b^1_{\phi(i-1)}$ and $e_2 \in b^1_{\phi(i)}$. We recall that the function ϕ is non-decreasing, therefore $\phi(i-1) \leq \phi(i)$. By using the "inequalities between boundsets" condition of $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$ for all the indices between $\phi(i-1)+1$ and $\phi(i)$, we have

$$\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho \leq \left[\!\left[b^1_{\phi(i-1)+1}\right]\!\right]^{\text{exp}}_\rho \leq \dots \leq \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$$

and hence $\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho \leq \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$. Since $e_1 \in b^1_{\phi(i-1)}$ and $e_1 \in b^2_{i-1}$, we have $\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho = [\![e_1]\!]^{\text{exp}}_\rho = \left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho$. Similarly, $\left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho = [\![e_2]\!]^{\text{exp}}_\rho = [\![b^2_i]\!]^{\text{exp}}_\rho$. This allows to deduce that $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho \leq [\![b^2_i]\!]^{\text{exp}}_\rho$, which is what we wanted to prove.

Condition 3: Strict inequalities for non-empty segments. Now, we need to prove that for any index $i \in \{1,\dots,n\}$ such that the boolean m^2_i is false, we have $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho < [\![b^2_i]\!]^{\text{exp}}_\rho$. Let $i \in \{1,\dots,n\}$. For the same reasons than in the previous condition,

we have $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho = \left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho$ and $\left[\!\left[b^2_i\right]\!\right]^{\text{exp}}_\rho = \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$. From the "emptiness markers" condition of $s^1 \sqsubseteq^{\text{Seg}} s^2$ we know that the implication $\left(\bigwedge_{j=\phi(i-1)+1}^{\phi(i)} m^1_j\right) \Rightarrow m^2_i$ holds. The right-hand side of the implication being false, we know that the left-hand side must be false as well. A conjunction of booleans is only false if it is not empty and one of the booleans is false. Hence, $\phi(i-1) \neq \phi(i)$ and $\exists j, \phi(i-1) < j \leq \phi(i) \wedge \neg m^1_j$. Using the "strict inequalities for non-empty segments" condition of $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$, we deduce that $\left[\!\left[b^1_{j-1}\right]\!\right]^{\text{exp}}_\rho < \left[\!\left[b^1_j\right]\!\right]^{\text{exp}}_\rho$. Combining this with the "inequalities between boundsets" condition of $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$, for all the indices between $\phi(i-1)+1$ and $\phi(i)$, we have

$$\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho \leq \ldots \leq \left[\!\left[b^1_{j-1}\right]\!\right]^{\text{exp}}_\rho < \left[\!\left[b^1_j\right]\!\right]^{\text{exp}}_\rho \leq \ldots \leq \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$$

Therefore, $\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho < \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$. We recall that $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho = \left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho$ and $\left[\!\left[b^2_i\right]\!\right]^{\text{exp}}_\rho = \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$. Hence, we have proven $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho < \left[\!\left[b^2_i\right]\!\right]^{\text{exp}}_\rho$; which is what we needed to prove.

Condition 4: Segment summaries. We want to prove that for any index $i \in \{1, \ldots, n\}$ of the the segmentation s^2, and for any index k of array t such that $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho \leq k < \left[\!\left[b^2_i\right]\!\right]^{\text{exp}}_\rho$, we have $\rho[l \mapsto k][v \mapsto t[k]] \in \gamma^{\text{D}}(d^2_i)$. Let $i \in \{1, \ldots, n\}$ and k be an index such that $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho \leq k < \left[\!\left[b^2_i\right]\!\right]^{\text{exp}}_\rho$ (if no such index exists, what we want to prove is vacuously true). Like in conditions 3 and 4, the "boundset inclusion" condition of $s^1 \sqsubseteq^{\text{Seg}} s^2$ allows us to deduce that $\left[\!\left[b^2_{i-1}\right]\!\right]^{\text{exp}}_\rho = \left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho$ and $\left[\!\left[b^2_i\right]\!\right]^{\text{exp}}_\rho = \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$. Hence, $\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho \leq k < \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho$. Let's consider the sequence of integer intervals

$$\left(\left\{\left[\!\left[b^1_{j-1}\right]\!\right]^{\text{exp}}_\rho, \ldots, \left[\!\left[b^1_j\right]\!\right]^{\text{exp}}_\rho - 1\right\}\right)_{\phi(i-1)<j\leq\phi(i)}$$

These integer intervals are contiguous, their left-most bound is $\left[\!\left[b^1_{\phi(i-1)+1-1}\right]\!\right]^{\text{exp}}_\rho$ which is equal to $\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho$, and their right-most bound is $\left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho - 1$, hence they form a partition of the integer interval

$$\left\{\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\text{exp}}_\rho, \ldots, \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\text{exp}}_\rho - 1\right\}$$

to which k belongs. Hence, there exists a j such that $\phi(i-1) < j \leq \phi(i)$ and $k \in \left\{\left[\!\left[b^1_{j-1}\right]\!\right]^{\text{exp}}_\rho, \ldots, \left[\!\left[b^1_j\right]\!\right]^{\text{exp}}_\rho - 1\right\}$. Using the "segment summaries" condition of $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$ for index j, we have $\rho[l \mapsto k][v \mapsto t[k]] \in \gamma^{\text{D}}(d^1_j)$. Then, using the "segment summaries"condition of $s^1 \sqsubseteq^{\text{Seg}} s^2$, we know that $d^1_j \sqsubseteq^{\text{D}} d^2_i$. Hence, using the monotony of γ^{D}

with respect to $\sqsubseteq^{\mathrm{D}}$, we can deduce $\rho[l \mapsto k][v \mapsto t[k]] \in \gamma^{\mathrm{D}}(d_i^2)$, which is what we wanted to prove.

Condition 5: Array size. Here, the goal is to prove that $[\![b_p^2]\!]_\rho^{\exp} = |t|$. Since $\phi(p) = n$, the "boundset inclusion" condition of $s^1 \sqsubseteq^{\mathrm{Seg}} s^2$ gives us that $b_p^2 \subseteq b_n^1$ and hence $[\![b_p^2]\!]_\rho^{\exp} = [\![b_n^1]\!]_\rho^{\exp}$. Using the "array size" condition of $(\rho, t) \in \gamma^{\mathrm{Seg}}(s^1)$, we have $[\![b_n^1]\!]_\rho^{\exp} = |t|$. Hence $[\![b_p^2]\!]_\rho^{\exp} = |t|$, which is what we wanted.

Conclusion of the proof. Since these five conditions are satisfied, we have proved that $(\rho, t) \in \gamma^{\mathrm{Seg}}(s^2)$, for any (ρ, t) in $\gamma^{\mathrm{Seg}}(s^1)$. Thus, we have proved that $\gamma^{\mathrm{Seg}}(s^1) \sqsubseteq^{\mathrm{Seg}} \gamma^{\mathrm{Seg}}(s^2)$, for any segmentations s^1 and s^2. Therefore, we have proved that $s^1 \sqsubseteq^{\mathrm{Seg}} s^2$ implies $\gamma^{\mathrm{Seg}}(s^1) \sqsubseteq^{\mathrm{Seg}} \gamma^{\mathrm{Seg}}(s^2)$. This concludes the proof of monotonicity of γ^{Seg}. □

9.5 Comparison with the [CCL11] Domain For Arrays

We have explained how our definition of segmentation inclusion differs from the version in [CCL11]. As we have already stated, our domain for array differs in two more aspects. First, we allow segment summaries to refer to any other program variables—as opposed to only the special l and v variables for index and cell value in [CCL11]. Then, we handle arrays that can contain values of algebraic data types—as opposed to scalar values only in [CCL11].

In this section, we discuss the main four ways in which these differences impact our definitions:

- First, we discuss in more details the differences in our definitions of segmentations and segmentation concretisation, compared with [CCL11].
- Then, we describe the three types of assignments that we need to distinguish—whereas [CCL11] distinguish only two types of assignments.
- Finally, we illustrate how we transfer information between segment summaries and the Pan component of the Pana product domain, by looking at the transfer functions for array creation, and array access at non-bound expressions.

All the aspects of abstract domain operators and transfer functions that we do not mention in this section—such as the transfer function for array update—are similar to [CCL11]'s.

Differences in Segmentations and Segmentation Concretisation In [CCL11], the segment summaries of segmentations can *only* talk about the special variables l and v for array index and array value. In our definition (definition 31), segment summaries can talk about *any* variable of the program—except the one containing the array that is summarized by the segmentation—*in addition* to the same special variables l and v.

More precisely, in the definition of concretisation (definition 31), the difference lies in the "*Segment summaries*" condition. Let t be an array, j an index of t and d_i a segment summary for some segment that contains the index j. The part that deals with the segment summary d_i in the [CCL11] concretisation enforces that the pair $(j, t[j])$ belongs to the concretisation of d_i. Our definition of concretisation, however, constrains the *whole* store ρ, by checking that the extended store $\rho[l \mapsto j; v \mapsto t[j]]$ belongs to the concretisation of d_i. We consider the store—instead of just the pair $(j, t[j])$—precisely because d_i might impose some constraints on other program variables, that are recorded in the store. We also allow the index l and the value at this index v to be constrained by d_i—and possibly related to other program variables—by adding l and v to the store.

Different Types of Assignments Two different kinds of assignments are described in [CCL11]: array updates, and scalar updates. We handle these two kinds of assignments similarly as [CCL11] do. Because we also handle algebraic values, we need to support another kind of assignment, for variables that are neither scalar nor arrays. We briefly review how to handle the different kinds of assignments.

Assignment to an array variable Like in [CCL11], when updating an array variable, the only segmentation that changes is the one for that variable. The other segmentations remain unaffected.

Assignment to a numeric variable Like in [CCL11], the update of a numeric variable $y := e$ may have two sorts of consequences:

- The assignment might be propagated inside the segment summaries of arrays.
- The boundsets of segmentations may change. Indeed, the variable y can occur inside bound expressions. Hence these bound expressions need to be either updated or removed to remain valid after the assignment. If the assignment is of the form $y := y + k$ where $k \in \mathrm{K}$ is a constant, then the bound expressions where y occurs are updated, by replacing y with $y - k$. Otherwise, the bound expressions where y

occurs are removed. Additionally, the variable y might also be added (because of its new value) to boundsets. Indeed, if the expression e in the assignment $y := e$ is a bound expression that occurs in a boundset, then variable y can be added to that boundset, so as to record that $y = e$.

Other assignments When updating a variable that has neither a numeric nor an array type, then the segment summaries are updated, but the boundsets remain unchanged. The boundsets cannot be affected by the variable update, since the expressions in a boundset necessarily have a scalar type.

Conversion: Transfer Function for Array Creation When converting an element of the Pan domain into a segment summary, we take three steps:

- We add the special variables for array index and array value
- We add information on those special variables, if we have any
- We embed this abstract value into the disjunctive completion of the structural lifting, by creating a singleton out of it

This can be seen, for example, in the transfer function for array creation:

Definition 38 (Transfer Function for Array Creation [Simplified Version])**.** *For any algebraic type* $\tau^{\text{alg}} \in \text{AlgTypes}$*, for any variable* y *of type* $\text{Array}(\tau^{\text{alg}})$*, for any numeric expression* e_1*, any expression* e_2 *of type* τ^{alg} *and any abstract value* $(t, a) \in \text{Pana}$*, the transfer function for array creation is defined by*

$$\begin{aligned}&\text{Assign}^{\text{Pana}}\left(y := \mathit{new\_array}(\tau^{\text{alg}}, e_1, e_2)\right)(t, a) = \\ &\quad\left(\text{Cond}^{\text{Pan}}(e_1 \geq 0)(t),\ a\,[y \mapsto \{0\}\ d\ \{|y|\}?\ ]\right) \\ &\quad \textit{where } d = \left\{\text{Cond}^{\text{Pan}}(0 \leq l < e_1 \wedge v = e_2)\left(\text{Add}^{\text{Pan}}_{\{l,v\}}(t)\right)\right\}\end{aligned}$$

Indeed, in this transfer function we see in the array component of the result, that variable y is associated to the segmentation $\{0\}\ d\ \{|y|\}?$, where the segment summary d used to summarise all the array slots of y, is a singleton, that contains the information that the array indices are non-negative (given by condition $0 \geq l$) and also abstracts the fact that the array slots contain e_2 (given by condition $v = e_2$). The abstraction that the array slots contain e_2 may not be an exact one, depending on expression e_2 and on the

underlying numeric domain N used to build Pan. This definition also takes into account the fact that if the instruction succeeds, then the expression e_1 given as the size of the new array is non-negative. Which is why component t of the abstract value is enriched with condition $e_1 \geq 0$ in the result.

This is a simplified version of the definition. The full definition queries the abstract value for previous knowledge on e_1, and distinguishes four cases as a result:

- The case where e_1 is known to be negative and the result is $\bot^{\text{Pana}}$, since any code after this is unreachable.
- The case where e_1 is known to be non-positive, in which case the new array is known to have size 0.
- The case where e_1 is known to be positive, in which case the emptiness marker of the segmentation is false, as we know for certain the array is not empty.
- All the other cases, where we have no particular prior knowledge on e_1, and the result is the one described in definition 38.

Conversion: Transfer Function for Array Access at Non-Bound Expressions When converting a segmentation summary back into an element of the Pan domain, two steps need to be taken:

- Abstract union is used to turn an element of the disjunctive completion into a single element of the Pan domain
- The special variables for array index and array value are removed

This can be seen, for example, in the transfer function for array access at a non-bound expression:

Definition 39 (Transfer function for an array access at a non-bound expression)**.** *For any abstract value* $(t, a) \in \text{Pana}$ *and any array access instruction* $y := x[e]$ *where* e *is not*

a bound expression, the abstraction for array access is given by

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{Assign}^{\mathrm{Pana}}(y := x[e])(t, a) = (t', a')} \\
\quad \textit{where} & a' = \mathrm{Assign}^{\mathrm{A}}(y := x[e])(a) \\
\quad \textit{and} & t' = \mathrm{Rem}^{\mathrm{Pan}}_{\{l,v\}} \left(\bigsqcup^{\mathrm{Pan}}_{t'' \in d} t'' \right) \\
\quad \textit{and} & d = \bigsqcup^{\mathrm{S}}_{i \in I} \mathrm{Assign}^{\mathrm{S}}(y := v) \left(\mathrm{Cond}^{\mathrm{S}}(l = e \wedge 0 \leq e)(d_i) \right) \\
\quad \textit{and} & I = \left\{ i \in \{1, \ldots, n\} \mid \exists e_1 \in b_{i-1}, \exists e_2 \in b_i, \mathrm{CanSat}^{\mathrm{Pan}}(t, e_1 \leq e < e_2) \right\} \\
\quad \textit{and} & \left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\ldots,n\}} \right) = a(x)
\end{array}
$$

We will explain the different lines of this definition from bottom to top. When performing the assignment $y := x[e]$, the variable y receives the value that is stored in array x, at the array index that expression e evaluates to. Hence, to know any information on the new value of variable y, we look at what information we had for the array stored in variable x. In other words, we look at what segmentation was stored for variable x inside the array component a of the abstract value (t, a). Let $\left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\ldots,n\}} \right)$ be this segmentation $a(x)$, as stated by the last line of the definition. In this segmentation, the segments that might talk about array index e are the segments i such that e might be delimited by boundsets b_{i-1} and b_i. In other words, the segments i such that there exists two expressions, $e_1 \in b_{i-1}$ and $e_2 \in b_i$ such that, given the knowledge we have so far in abstract value t, it is possible that $e_1 \leq e < e_2$. The set of these segment indices is called I, as stated in the second-to-last line of the definition. We then use this set I to extract from the segment summaries the information we want: the result of assigning to variable y the content of the array — as represented by the special variable v — when the array index is e, as represented by the condition $l = e$. Abstract union is used to merge the result for the different segment summaries that might be involved, yielding abstract value d. Then, the conversion step takes place: d is an element of the structural lifting. We take the abstract union of its disjuncts to obtain a single element of the Pan domain. Then we remove the special variables l and v, and we get t', that we use as new Pan component in the Pana domain. As explained earlier at page 108, if y is a numeric variable then the boundsets of all the segmentations need to be updated, and if y has a product type or a sum type, then the segment summaries of segmentations are updated. This is done by the $\mathrm{Assign}^{\mathrm{A}}$ operator.

9.6 Disjunctive Completion for the Pana Domain

In this section, we give the detailed definitions for the Dana domain, which is obtained from the Pana domain by taking a disjunctive completion where cases that have equivalent constructor constraints are merged together using abstract union. This construction follows the exact same structure than the construction that allows to go from the reduced product to the structural lifting in section 5.4.

Abstract inclusion is given by a Hoare order:

$$\mathcal{O}_1 \sqsubseteq^{\text{Dana}} \mathcal{O}_2 \text{ iff } \forall o \in \mathcal{O}_1, \exists o' \in \mathcal{O}_2, o \sqsubseteq^{\text{Pana}} o'$$

Abstract intersection for the disjunctive completion is given by:

$$\mathcal{O}_1 \sqcap^{\text{Dana}} \mathcal{O}_2 = \text{Collapse}^{\text{Dana}} \left(\left\{ o_1 \sqcap^{\text{Pana}} o_2 \mid \begin{array}{l} o_1 \in \mathcal{O}_1 \wedge o_2 \in \mathcal{O}_2 \wedge \\ o_1 \sqcap^{\text{Pana}} o_2 \neq \bot^{\text{Pana}} \end{array} \right\} \right)$$

We take a disjunction because for certain programs we want to be able to extract information for different cases, in which incompatible constructor names are involved. However, when two abstract values are equivalent with respect to the constructors that they mention, they should be merged, to limit the size of the disjunction. We use the same definition of constructor constraints equivalence as in section 5.4; but this time we extend it for the quadruplets of the Pana domain:

$$(c_1, e_1, n_1, a_1) \equiv_{\text{E}}^{\text{Pana}} (c_2, e_2, n_2, a_2) \text{ iff } c_1 \equiv c_2$$

This notion of equivalence allows us to define an operator that collapses together equivalent quadruplets in an element of Dana $= \mathcal{P}(\text{Pana})$.

Definition 40. *We define an operator* $\text{Collapse}^{\text{Dana}}$ *that takes a set of elements of* Pana *and merges together (by taking the abstract union), the elements that are equivalent with respect to* $\equiv_{\text{E}}^{\text{Pana}}$*. Formally,*

$$\text{Collapse}^{\text{Dana}}(\mathcal{O}) = \left\{ \bigsqcup_{o \in \overline{o}}^{\text{Pana}} o \mid \overline{o} \in \mathcal{O}/\equiv_{\text{E}}^{\text{Pana}} \right\}$$

We use this collapse operator to provide an abstract union for the Dana domain:

$$\mathcal{O}_1 \sqcup^{\text{Dana}} \mathcal{O}_2 = \text{Collapse}^{\text{Dana}}(\mathcal{O}_1 \cup \mathcal{O}_2)$$

A widening for the Dana domain is given by

$$\begin{aligned}&\mathcal{O}_1 \triangledown^{\text{Dana}} \mathcal{O}_2 = \\ &\quad \{o_1 \triangledown^{\text{Pana}} o_2 \mid o_1 \in \mathcal{O}_1 \wedge o_2 \in \mathcal{O}_2 \wedge o_1 \equiv_{\text{E}}^{\text{Pana}} o_2\} \\ &\quad \cup \{o_2 \in \mathcal{O}_2 \mid \nexists o_1 \in \mathcal{O}_1, o_1 \equiv_{\text{E}}^{\text{Pana}} o_2\} \cup \{o_1 \in \mathcal{O}_1 \mid \nexists o_2 \in \mathcal{O}_2, o_1 \equiv_{\text{E}}^{\text{Pana}} o_2\}\end{aligned}$$

9.7 Soundness Theorem tor the Diorana Domain

We now state the main soundness theorem for our abstract domain for arrays, and give a sketch of its proof.

Theorem 2 (Soundness of the Diorana domain). *The operators and transfer functions of the Diorana domain are sound :*

- $\sqsubseteq^{\text{Diorana}}$ *is a pre-order.*
- γ^{Diorana} *is monotonic with respect to the pre-order.*
- *The abstract union* $\sqcup^{\text{Diorana}}$*, and abstract intersection* $\sqcap^{\text{Diorana}}$ *are sound over-approximations of their concrete counter-parts.*
- *Widening* $\triangledown^{\text{Diorana}}$ *computes upper-bounds and enforces convergence.*
- *The transfer functions for assignment* $\text{Assign}^{\text{Diorana}}$ *and conditions* $\text{Cond}^{\text{Diorana}}$ *are sound.*

Proof sketch. For pre-order, concretisation, abstract union, abstract intersection and widening, the soundness of the Diorana domain is deduced from the one of the Seg domain. Indeed, from Seg to A the proofs are transported element-wise, from A to Pana the standard arguments of non-reduced products apply, and for the disjunctive layer of Dana and the relational lifting layer of Diorana, the arguments are the same than for RAND.

The fact that $\sqsubseteq^{\text{Seg}}$ is a pre-order is the object of lemma 15. The fact that γ^{Seg} is monotonic with respect to segmentation inclusion is the object of lemma 16. For abstract union and abstract intersection of segmentations, the soundness is a combination of the fact that unification yields less precise abstract values, and the fact that segment-wise operations are sound. For example, for abstract union, if s_1 and s_2 are two segmentations, and s'_1 and s'_2 are the result of their unification, then we have

$$s_1 \sqsubseteq^{\text{Seg}} s'_1 \qquad\qquad s_2 \sqsubseteq^{\text{Seg}} s'_2$$

```
Function summary for function find_max_priority(p) returning res' :
  ( Constructor constraints : res'@NoMax
    Numeric constraints : |q| = 0
    Abstract array environment : [ q -> { 0 |q| } ]
  )
  Or
  ( Constructor constraints : res'@SomeMax
    Numeric constraints : |q| > 0
    Abstract array environment :
      [ q -> { 0 } v.prio <= res'@SomeMax.prio { |q| } ]
  )
```

Figure 9.3: Abstract value for the function `find_max_priority` from figure 9.1.

Which implies, by the monotonicity of γ^{Seg} with respect to $\sqsubseteq^{\text{Seg}}$, that

$$\gamma^{\text{Seg}}(s_1) \sqsubseteq^{\text{Seg}} \gamma^{\text{Seg}}(s_1') \qquad\qquad \gamma^{\text{Seg}}(s_2) \sqsubseteq^{\text{Seg}} \gamma^{\text{Seg}}(s_2')$$

Then, by the soundness of segment-wise operations

$$\gamma^{\text{Seg}}(s_1') \cup \gamma^{\text{Seg}}(s_2') \subseteq \gamma^{\text{Seg}}(s_1' \mathbin{\overline{\sqcup^{\text{seg}}}} s_2')$$

Hence,

$$\gamma^{\text{Seg}}(s_1) \cup \gamma^{\text{Seg}}(s_2) \subseteq \gamma^{\text{Seg}}(s_1' \mathbin{\overline{\sqcup^{\text{seg}}}} s_2')$$

Since $s_1 \sqcup^{\text{Seg}} s_2$ is defined as $s_1' \mathbin{\overline{\sqcup^{\text{seg}}}} s_2'$, this proves the soundness of $\sqcup^{\text{Seg}}$.

For the convergence of widening, the process of unification can only *remove*, not add, bound expressions; and there is only a finite number to begin with. Hence, from a certain index, the results of unification always yield segmentations with the same boundsets, and we can rely on the convergence of segment-wise widening.

For the transfer functions for assignment and conditions, we proceed in the same way than [CCL11]. □

9.8 Result of Analysis on the Motivating Example

The Diorana domain is not implemented in our OCaml prototype. However, we have used the Diorana domain to execute by hand an analysis of the `find_max_priority` function from figure 9.1. We obtained as a result the abstract value from figure figure 9.3. This

input-output summary of function `find_max_priority` distinguishes two cases:

- either the input queue is empty ($|q| = 0$) and the result is built using constructor NoMax;
- or the input queue is not empty ($|q| \geq 1$), the result is built using constructor SomeMax and the TCB that is returned as a result (via the constructor SomeMax) has a priority that is greater than the priority of any of the TCBs in the input queue.

In the second case, the relation between the output and the contents of the input array (namely that the priority of the ouput is greater than the priorities of any of the array values) was captured thanks to the fact that we allow our segment summaries to refer to program variables in addition to referring to the array's index and value. Here, in particular, it is the relations captured between variable `res` and the values inside array `q` that carry interesting information on the function's behaviour.

This input-output summary is not exact: it does not state that, when the result is a TCB wrapped in constructor SomeMax, this TCB belongs to the input queue.

9.9 Conclusive Remarks on Array Analysis

We have described in this chapter an abstract domain that allows us to compute input-output summaries of functions that may manipulate arrays that contain values from algebraic types. For this purpose, we have built on the segmentation approach of [CCL11]. Unlike [CCL11], we allow arrays to contain values from algebraic types, we allow segment summaries to refer to arbitrary program variables, and we have fixed the definition of segmentation inclusion, so that it makes the concretisation function monotonic.

One limitation of our approach is that we only allow arrays in top-level program variables: we do not allow arrays nested inside values of algebraic types nor inside other arrays. Additionally, we have not yet implemented our abstract domain for arrays.

Part IV

Conclusion

Chapter 10

Conclusion

We have shown that it is possible to define an abstract domain that is expressive enough to capture numeric relations between parts of values from algebraic types; even when these values are stored inside arrays of an unknown size. We have partially implemented our approach (Parts I and II) and we obtain a satisfying precision and reasonable execution times for the kind of programs that we are interested in (section 7.3, in particular Table 7.1).

For algebraic types, the main idea is to consider extended variables—*i.e.*, pairs of a variable and an access path—as the entities that are related in a numeric abstract domain (Chapter 4). To reduce the size of abstract values, we add a domain that keeps track of equalities between non-numeric values (section 5.2). The domains are combined using a reduced product that propagates equalities (section 5.3). Additional expressiveness and precision is obtained using an adaptation of disjunctive completion for handling the different, incompatible cases that an algebraic value can exhibit (section 5.4). This abstract domain is called RAND—the Relational Algebraic Numeric Domain.

We have given a formal justification, in the context of abstract interpretation, to the folklore result of static analysis that *"an intra-procedural analysis can be made input-output relational by duplicating variables"*, by effectively turning an analysis that relates different parts of a store into an analysis that computes a relation between input and output stores (Chapter 6). One key observation is that the input-output relational analyser and the non-input-output relational one share the same *structure*: only a few transfer functions need to be redefined. The second observation is that any relational domain can easily be used to express relations between different stores: the necessary transfer functions can be redefined once and for all, in a generic manner.

We have used the RAND abstract domain to implement [BJM22a] a static analyser for a `while` language with algebraic data types and function calls that exploits the relational feature of RAND to infer function summaries (Chapter 7). Summaries express the input-output behaviours of functions, and enable a *modular* inter-procedural analysis of programs: every function is analysed *exactly once*.

Finally, we have shown how to extend RAND to handle functional arrays (Chapter 9). This extension is based on the notion of array segmentation [CCL11], and enables the analysis of programs that manipulate arrays whose cells may contain values of algebraic data types. Our array extension improves on [CCL11], by fixing a problem on the abstract inclusion operator, that made the concretisation operator not monotonic on some edge cases. Additionally, our extension allows to capture relations between array contents and other program variables.

The main limitation of our RAND abstract domain is that it does not handle *recursive* algebraic types. Two possible ideas for overcoming this limitation are to allow for regular expressions inside paths, or to use tree automata. Both of these ideas are used in the work done by Journault, Miné and Ouadjaout [JMO19]. It would also be an interesting line of work to see whether replacing our disjunctive completion by a conjunction of implications, as advocated by Liu and Rival [LR15b], could improve either the performance or the precision of our analysis of algebraic types.

The main limitation of our inter-procedural analysis is that it is restricted to *non-recursive* functions. Analysing recursive functions will require the computation of a fixpoint at the level of function summaries.

When it comes to arrays, an obvious next step is to implement our abstract domain, and to study its complexity and scalability. Beyond the lack of implementation, the approach we suggest for arrays has two main limitations.

The first limitation is that we forbid array types as record fields or constructor arguments inside algebraic types. At first sight it might look like the only change needed in order to overcome this limitation is to associate segmentations to *extended* variables, instead of just variables. However, we already allow algebraic types inside array types. Hence, the most significant challenge in having array types inside algebraic types is the possibility of *arrays nested inside arrays*. In particular, this could imply having an abstract domain definition that is recursive: the domain for arrays would map (extended) variables into segmentations, which in turn use the domain for arrays inside the summaries of array segments. We believe this would actually work, because the number of (extended) variables having an array type decreases at each stage, thus the abstract values are finite. Nevertheless, this could have a significant cost on performance and scalability, especially since the domain we use to summarize each array segment contains a disjunctive completion. Hence there would be disjunctions inside disjunctions, which could be very costly. Allowing for nested arrays would be useful. For example, the functional specification of seL4 in Isabelle does

feature arrays inside arrays. Indeed, the kernel heap is represented as an array that may contain thread control blocks, and each thread control block in turn contains an array of capabilities describing the permissions of the thread.

The second limitation is that we do not capture relations *between arrays*. Indeed, we associate a segmentation to each variable with an array type, and the segmentations do not interact with each other. Overcoming this limitation would allow, for example, to describe the way in which an array changes between the input and the output of a function. This might prove useful when analysing functions that modify a scheduler queue, for example. One possibility would be to map each *pair* of array variables to a segmentation, and duplicate, in the summaries of array segments, the special variable v that refers to the array's content. Each copy of v would refer to the content of one of the arrays in the pair, and the segment summaries would capture relations between them. Our intuition is that only some pairs of arrays would be related in a meaningful way. It would then be interesting to try to determine statically which pairs of array variables to analyse, and avoid useless computations.

As mentioned in the introduction (Chapter 1), this work was motivated by the long-term goal of alleviating the human effort required by interactive theorem proving, by mixing automatic and interactive forms of formal verification. This thesis focuses on developing abstract domains whose expressiveness would be useful for this endeavour; but it does not study the question of *how* to integrate such abstract domains with interactive theorem proving. One possibility would be to formally verify our analysis inside the proof assistant. However, this approach would require for the numeric abstract domains that we extend to be verified as well. In order to maintain a choice of instantiation for the underlying numeric domain, a significant verification effort would be required. A second possibility would be to have our analyser generate a certificate that explains how the analysis result was obtained, and have a verifier (itself formally verified inside the proof assistant) check the certificate. More precisely, this certificate could consist of the different intermediate results of the analysis, and the verifier could then use a set of lemmas about the language semantics to try to check that the different intermediate results are correct, until the final result is reached. A third possibility could be to have our analyser generate a proof script, and have the proof assistant check that the provided proof script indeed allows to prove the final result of the analysis. In a way, this third possibility follows the same principle as the second possibility, except the certificate is a proof script and the verifier is the proof assistant itself.

Bibliography

[And+19] Oana F. Andreescu, Thomas Jensen, Stéphane Lescuyer, and Benoît Montagu, "Inferring Frame Conditions with Static Correlation Analysis", *in*: *POPL*, 2019, DOI: 10.1145/3290360.

[BH19] Rémy Boutonnet and Nicolas Halbwachs, "Disjunctive Relational Abstract Interpretation for Interprocedural Program Analysis", *in*: *VMCAI*, Lecture note in Computer Science, 2019, DOI: 10.1007/978-3-030-11245-5_7.

[BJM20] Santiago Bautista, Thomas Jensen, and Benoît Montagu, "Numeric Domains Meet Algebraic Data Types", *in*: *NSAD*, 2020, DOI: 10.1145/3427762.3430178.

[BJM22a] Santiago Bautista, Thomas Jensen, and Benoît Montagu, *Artifact for the "Lifting Numeric Relational Domains to Algebraic Data Types" article of the SAS 2022 symposium*, 2022, DOI: 10.5281/zenodo.6977156.

[BJM22b] Santiago Bautista, Thomas Jensen, and Benoît Montagu, "Lifting Numeric Relational Domains to Algebraic Data Types", *in*: *SAS*, 2022, DOI: 10.1007/978-3-031-22308-2_6.

[BJM22c] Santiago Bautista, Thomas Jensen, and Benoît Montagu, *Lifting Numeric Relational Domains to Algebraic Data Types (extended version)*, 2022, URL: https://hal.inria.fr/hal-03765357.

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival, "A static analyzer for large safety-critical software", *in*: *PLDI*, 2003, DOI: 10.1145/780822.781153.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma, "What's Decidable About Arrays?", *in*: *VMCAI*, 2006, DOI: 10.1007/11609773_28.

[CC02] Patrick Cousot and Radhia Cousot, "Modular Static Program Analysis", *in*: *CC*, 2002, DOI: 10.1007/3-540-45937-5_13.

[CC77] Patrick Cousot and Radhia Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”, *in*: *POPL*, 1977, DOI: 10.1145/512950.512973.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo, “A parametric segmentation functor for fully automatic and scalable array content analysis”, *in*: 2011, DOI: 10.1145/1925844.1926399.

[Com+08] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi, *Tree Automata Techniques and Applications*, 2008, URL: https://hal.inria.fr/hal-03367725.

[Cou21] Patrick Cousot, *Principles of Abstract Interpretation*, Cambridge, Massachusetts: The MIT Press, 2021, p. 832, ISBN: 9780262044905.

[Cou97] Patrick Cousot, “Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract)”, *in*: *MFPS*, 1997, DOI: 10.1016/s1571-0661(05)80168-9.

[DAL22] Aleksandar S. Dimovski, Sven Apel, and Axel Legay, “Several lifted abstract domains for static analysis of numerical program families”, *in*: *Science of Computer Programming* 213 (2022), DOI: 10.1016/j.scico.2021.102725.

[Die+18] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski, “The Map Equality Domain”, *in*: *VSTTE*, 2018, DOI: 10.1007/978-3-030-03592-1_17.

[Dim19] Aleksandar S. Dimovski, “Lifted static analysis using a binary decision diagram abstract domain”, *in*: *GPCE*, 2019, DOI: 10.1145/3357765.3359518.

[FK15] Azadeh Farzan and Zachary Kincaid, “Compositional Recurrence Analysis”, *in*: *FMCAD*, 2015, DOI: 10.1109/FMCAD.2015.7542253.

[Ful12] Jędrzej Fulara, “Generic Abstraction of Dictionaries and Arrays”, *in*: *Electronic Notes in Theoretical Computer Science* 287 (Nov. 2012), pp. 53–64, DOI: 10.1016/j.entcs.2012.09.006.

[Gen+13] Thomas Genet, Tristan Le Gall, Axel Legay, and Valérie Murat, “A Completion Algorithm for Lattice Tree Automata”, *in*: *CIAA*, 2013, DOI: 10.1007/978-3-642-39274-0_13.

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv, “A framework for numeric analysis of array operations”, *in*: *POPL*, 2005, DOI: 10.1145/1040305.1040333.

[HIV08] Peter Habermehl, Radu Iosif, and Tomáš Vojnar, "What Else Is Decidable about Integer Arrays?", *in*: *FOSSACS*, 2008, DOI: 10.1007/978-3-540-78499-9_33.

[HP08] Nicolas Halbwachs and Mathias Péron, "Discovering properties about arrays in simple programs", *in*: *PLDI*, 2008, DOI: 10.1145/1379022.1375623.

[ILR17] Hugo Illous, Matthieu Lemerre, and Xavier Rival, "A Relational Shape Abstract Domain", *in*: *NASA Formal Methods*, 2017, DOI: 10.1007/978-3-319-57288-8_15.

[ILR21] Hugo Illous, Matthieu Lemerre, and Xavier Rival, "A relational shape abstract domain", *in*: *Formal Methods in System Design* 57.*3* (Apr. 2021), pp. 343–400, DOI: 10.1007/s10703-021-00366-4.

[Jea09] Bertrand Jeannet, *The BDDAPRON logico-numerical abstract domains library*, https://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/, 2009.

[Jea13] Bertrand Jeannet, "Relational interprocedural verification of concurrent programs", *in*: *Softw. Syst. Model.* 12.*2* (May 2013), pp. 285–306, ISSN: 1619-1366, DOI: 10.1007/s10270-012-0230-7.

[JM07] Ranjit Jhala and Kenneth L. McMillan, "Array Abstractions from Proofs", *in*: *CAV*, 2007, DOI: 10.1007/978-3-540-73368-3_23.

[JM09] Bertrand Jeannet and Antoine Miné, "Apron: A Library of Numerical Abstract Domains for Static Analysis", *in*: *CAV*, 2009, ISBN: 978-3-642-02658-4, DOI: 10.1007/978-3-642-02658-4_52.

[JMO19] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout, "An Abstract Domain for Trees with Numeric Relations", *in*: *ESOP*, 2019, DOI: 10.1007/978-3-030-17184-1_26.

[Jou+19] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout, "Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer", *in*: *VSTTE*, 2019, DOI: 10.1007/978-3-030-41600-3_1.

[Kin+17] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps, "Compositional Recurrence Analysis Revisited", *in*: *PLDI*, 2017, DOI: 10.1145/3062341.3062373.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood, "seL4: Formal Verification of an OS Kernel", *in*: *SOSP*, 2009, DOI: 10.1145/1629575.1629596.

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser, "Comprehensive formal verification of an OS microkernel", *in*: *ACM Trans. Comput. Syst.* 32.*1* (Feb. 2014), DOI: 10.1145/2560537.

[Koz97] Dexter Kozen, "Kleene Algebra with Tests", *in*: *TOPLAS*, 1997, DOI: 10.1145/256167.256195.

[KRR18] Se-Won Kim, Xavier Rival, and Sukyoung Ryu, "A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework", *in*: *TOPLAS*, 2018, DOI: 10.1145/3230624.

[Ler+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand, "CompCert - A Formally Verified Optimizing Compiler", *in*: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, SEE, Toulouse, France, Jan. 2016, URL: https://inria.hal.science/hal-01238879.

[Les15] Stéphane Lescuyer, "ProvenCore: Towards a Verified Isolation Micro-Kernel", *in*: *MILS@HiPEAC*, 2015, DOI: 10.5281/zenodo.47990.

[LGJ23] Théo Losekoot, Thomas Genet, and Thomas Jensen, "Automata-Based Verification of Relational Properties of Functions over Algebraic Data Structures", *in*: *FSCD*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, DOI: 10.4230/LIPICS.FSCD.2023.7.

[Li+17a] Bin Li, Juan Zhai, Zhenhao Tang, Enyi Tang, and Jianhua Zhao, "A Framework for Array Invariants Synthesis in Induction-Loop Programs", *in*: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2017, DOI: 10.1109/apsec.2017.8.

[Li+17b] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival, "Semantic-directed clumping of disjunctive abstract states", *in*: *POPL*, 2017, DOI: 10.1145/3009837.3009881.

[LR15a] Jiangchao Liu and Xavier Rival, "Abstraction of Arrays Based on Non Contiguous Partitions", *in*: *VMCAI*, 2015, pp. 282–299, DOI: 10.1007/978-3-662-46081-8_16.

[LR15b] Jiangchao Liu and Xavier Rival, "Abstraction of Optional Numerical Values", *in*: *APLAS*, 2015, DOI: 10.1007/978-3-319-26529-2_9.

[Mat18] Daniel Matichuk, "Automation for Proof Engineering: Machine-Checked Proofs At Scale", PhD thesis, Sydney, Australia: UNSW, 2018, DOI: 10.26190/unsworks/20637.

[Min06] Antoine Miné, "The octagon abstract domain", *in*: *High. Order Symb. Comput.* 19.*1* (2006), pp. 31–100, DOI: 10.1007/s10990-006-8609-1.

[Min17] Antoine Miné, "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation", *in*: *Found. Trends Program. Lang.* 4.*3-4* (2017), pp. 120–372, DOI: 10.1561/2500000034.

[Pie02] Benjamin C. Pierce, *Types and Programming Languages*, Cambridge, Massachusetts: The MIT Press, 2002, ISBN: 978-0-262-16209-8.

[RY20] Xavier Rival and Kwangkeun Yi, *Introduction to static analysis: an abstract interpretation perspective*, Cambridge, Massachusetts: The MIT Press, 2020, ISBN: 9780262043410.

[SJ11a] Peter Schrammel and Bertrand Jeannet, "Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs", *in*: *SAS*, 2011, DOI: 10.1007/978-3-642-23702-7_19.

[SJ11b] Pascal Sotin and Bertrand Jeannet, "Precise Interprocedural Analysis in the Presence of Pointers to the Stack", *in*: *ESOP*, 2011, DOI: 10.1007/978-3-642-19718-5_24.

[Tar41] Alfred Tarski, "On the Calculus of Relations", *in*: *Journal of Symbolic Logic* 6 (Sept. 1941), DOI: 10.2307/2268577.

[VMM23] Milla Valnet, Raphaël Monat, and Antoine Miné, "Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs (Static analysis of values by abstract interpretation of functional programs manipulating recursive algebraic types)", *in*: *JFLA*, 2023, URL: https://inria.hal.science/hal-03936718.

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

école normale supérieure rennes

Titre : Analyse Statique de Types Algébriques et de Tableaux

Mot clés : Interprétation abstraite, types algébriques, tableaux, segmentations

Résumé : Pour assurer l'absence d'erreurs dans les logiciels critiques, la vérification formelle peut s'appuyer sur des méthodes d'analyse statique, telle que l'interprétation abstraite, qui déduisent des propriétés sur les programmes à partir de leur code source. Cependant, les domaines d'interprétation abstraite existants se concentrent majoritairement sur des programmes où les variables contiennent des nombres ou des pointeurs. Cette thèse étudie la possibilité de développer une interprétation abstraite pour des langages de programmation où les variables contiennent des valeurs de types algébriques, ou des tableaux de telles valeurs. Nous présentons une façon générique d'étendre les domaines numériques existants pour calculer des résumés du comportement entrée-sortie des fonctions manipulant des valeurs de types algébriques et des tableaux ; pourvu que les types algébriques en question soient non-récursifs. L'aspect entrée-sortie de notre analyse lui permet d'être modulaire, n'analysant qu'une seule fois chaque fonction. Un prototype de notre analyse pour les types algébriques (mais pas pour les tableaux) a été implémenté en OCaml et testé sur 43 exemples, dont certains inspirés du code du micro-noyeau seL4.

Title: Static Analysis of Algebraic Data Types and Arrays

Keywords: Abstract interpretation, algebraic types, arrays, segmentations

Abstract: In order to ensure that critical software has no error, formal verification may rely on static analysis. Static analysis methods, such as abstract interpretation, infer the properties of a program by analysing its source code. However, the existing domains of abstract interpretation mainly focus on programs where the variables hold numbers or pointers. In this thesis, we study the possibility of developping an abstract interpretation for programming languages with algebraic types and arrays of values from algebraic types. We present a generic way of extending already existing numeric domains in order to infer input-output summaries of functions manipulating both numbers, values of algebraic types and arrays. A limitation of our approach is that we only handle non-recursive algebraic types. The input-output aspect of our function summaries allows for a modular analysis, where each function is analysed only once. A prototype of our analysis of algebraic types (without the analysis of arrays) was implemented in OCaml and tested on 43 exemples. Some of the examples are inspired from the code of the seL4 micro-kernel.