An input–output relational domain for algebraic data types and functional arrays

Santiago Bautista, Thomas Jensen, Benoît Montagu

An Input-Output Relational Domain for Algebraic Data Types and Functional Arrays

Santiago Bautista[1,2*], Thomas Jensen[2,1] and Benoît Montagu[2,1]

[1]Univ Rennes, Rennes, France.
[2]Inria.

*Corresponding author(s). E-mail(s): Santiago.Bautista@ens-rennes.fr;
Contributing authors: Thomas.Jensen@inria.fr; Benoit.Montagu@inria.fr;

Abstract

We define an abstract interpreter for programs manipulating scalars, immutable non-recursive algebraic data types (ADTs), functional arrays and non-recursive functions. We define a first domain that expresses relations between values of ADTs. Our domain extends numeric domains in a generic way, by a disjunctive completion over a reduced product of domains for numeric relations, equalities, and variant constructors. We further extend the segmentation approach for array analysis, in order to allow for values of ADTs inside arrays, and also to capture relations between the content of arrays and the other variables in the program. Our analysis is inter-procedural and modular: it proceeds by computing summaries of the input-output relations of functions, which are then instantiated at call sites. The analysis of ADTs has been implemented in OCaml and tested on a sample of small-to-medium examples, some of which are simplified versions of functions from the formal specification of the seL4 microkernel.

Keywords: Static analysis, Abstract interpretation, Relational abstract domains, Algebraic data types, Input-output relations, Function summaries, Arrays.

1 Introduction

Research in static analysis has successfully developed automatic techniques to ensure the safety and security of programs, by detecting bugs *before* a program actually runs. In particular, there exists a substantial number of analyses that target programs with numeric or pointer-based computations and which can detect frequent bugs that arise from arithmetic overflows or memory safety issues. Another important class of programs

are those manipulating algebraic data types (ADTs). ADTs form the core of modern programming languages—such as OCaml, Haskell, Scala, Rust or Swift—that have been adopted by the software industry. The static analysis of this class of programs has seen important progress too, with the development of type systems [1, 2] or by leveraging tree automata techniques [3] for approximating the tree structures described by ADTs [4–6].

In this paper, we focus on the automatic analysis of programs that perform numeric operations, feature arrays *and* manipulate ADTs. So far, few works [7–10] have put emphasis on the combined analysis of numeric properties and ADT properties; and even fewer [10] have combined these two analyses with array analysis. Such combined analyses can provide additional safety guarantees—such as the unreachability of branches of a pattern matching under some numeric condition, or information on array contents depending on which constructor is used to build an ADT value. Such static analyses can also alleviate the interactive verification of large, critical programs that compute over ADTs, by automatically discharging a substantial number of proof obligations [10].

To this end, we first develop a novel relational abstract domain that can express relations between numeric-algebraic values of a program state (section § 3). We build this abstract domain in a generic way, by taking as a parameter *any* relational abstract domain that fulfils an Apron-like interface [11] to handle the numeric properties. One difficulty in designing this domain is to handle soundly and precisely the mutually exclusive cases that an algebraic value may take. We tackle this issue using *projection paths* that point inside algebraic values, and by devising a notion of *compatibility* between paths: two paths are compatible when they make consistent assumptions over the constructors of variant values. The resulting abstract domain can describe sets of states of algebraic data structures with scalar data.

Then, we show how to turn our abstract domain into RAND—the Relational Algebraic-Numeric Domain—an abstract domain that can express *relations* between *different* states (section § 4). For an example process management program from an idealised operating system (Fig. 4), RAND can express that the input and output processes `p` and `p'` satisfy the constraint `p'.status@Running.count = p.status@Asleep.count + 1`, meaning that the `status` fields of `p` and `p'` differ by 1, whenever the process `p` has a *running* status, and `p'` a *sleeping* status. This is indicated by the projections on constructor cases `@Running` and `@Sleeping`. We discuss this example further in the paper (section § 2.3).

Using RAND, we define a *relational* analysis for a `while` language that features non-recursive ADTs (section § 5). Our analysis infers relations between the inputs and the outputs of programs. In particular, we explain how a standard static analysis for reachable states can be turned into an analysis for input-output relations. This relational analysis is well suited for designing an inter-procedural analysis based on function summaries.

Finally, we describe how to extend our approach to analyse programs with functional arrays (section § 6). Thus, we define the Diorana domain (*Domain for Input-Output Relations on Algebraic types, Numbers and Arrays*), that allows to analyse programs where arrays can contain ADT values.

Our work offers the following contributions:

- We present a novel abstract domain that expresses relations between values of non-recursive immutable ADTs (sections § 3 and 4). Our abstract domain can be instantiated with any numeric relational domain. This offers the possibility to choose domains with different precision *vs* cost balances, and allows to capture numeric inequalities. This improves upon the correlation domain [10], that is restricted to information about equality and reachability.
- Our abstract domain uses a form of *disjunctive completion* (section § 3.6.2), 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 1) and by showing how any relational domain can express relations between different stores (section § 4.2).
- We formally define a relational analysis (section § 5) that infers relations between inputs and outputs of programs, and propose a modular inter-procedural extension that is based on function summaries. We illustrate the analyser's results on a running example taken from an idealised operating system.
- We extend the approach to encompass functional arrays that can contain algebraic data types (section § 6). This extension of our abstract domain is based on the notion of array segmentation [12].
- We provide an OCaml implementation [13] of our analyser, for a `while` language with algebraic types; together with 43 test cases, some of which are inspired from an operating system code (section § 7). Our implementation is limited, as it does not support the extension for functional arrays. We briefly discuss the complexity of our implementation.

We do not handle polymorphism or higher order functions in this work.

This article is an extended version of a paper published at the SAS 2022 conference [14]. The main addition with respect to [14] is the extension to support functional arrays (section § 6). In order to analyse arrays, we build on the work of Cousot, Cousot and Logozzo [12] (CCL). Our approach has three main differences with respect to CCL:

- We allow for arrays to contain values from algebraic types, whereas CCL 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, such as the parameters of functions.
- We have a different definition for the pre-order between array segmentations. Our definition allows the concretisation to be monotonic with respect to the pre-order.

2 Syntax and Semantics

Our programming language is an extension of a classic `while` language with algebraic data types (products and sums). Section § 2.1 presents algebraic types, section § 2.2 presents the language and its semantics, and section § 2.3 introduces our running example.

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:*

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

ε is not a terminal symbol, but denotes the empty path.

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

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

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

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

We use Undef to indicate that a path does not exist in an input value.

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

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

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

Fig. 1: Small-step semantics of commands. $\mathbf{B} = \{\mathtt{tt}, \mathtt{ff}\}$ is the set of boolean values.

2.2 A Language With Algebraic Data Types

The syntax of the language consists of expressions t, boolean conditions b, and commands c. Vars denotes the set of variables that may appear in commands. Expressions include the projection of a variable $x \in \mathrm{Vars}$ over a path $p \in \mathrm{Paths}$, 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.

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

We restrict our attention to well-typed commands (that we call *programs*), following a standard structural type system [1]. For instance, well-typedness ensures that 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 (Fig. 1). The relation $(c, s) \to (c', s')$ states 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,

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

Fig. 2: Denotation of expressions.

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

Fig. 3: Denotation of conditions.

$\mathsf{assert}(b)$ 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 $\mathsf{branch}\ \mathsf{assert}(b)\,;c_1\ \mathsf{or}\ \mathsf{assert}(\neg b)\,;c_2\ \mathsf{end}$.

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 that is identical to the store s, except that it maps the variable x to the value v.

The evaluation $[\![t]\!]_s^{\text{exp}}$ 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 (Fig. 2). $[\![t]\!]_s^{\text{exp}}$ 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 $[\![x@B]\!]_s^{\text{exp}} = \emptyset$, because the constructors A and B are different. The evaluation of booleans $[\![b]\!]_s^{\text{bool}}$ is standard (Fig. 3).

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.

2.3 Running Example

Figure 4 shows an example program for which we would like to infer precise input-output properties. This program 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, 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

```
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
}
```

Fig. 4: Example program performing clock ticks on a process' meta-data.

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:

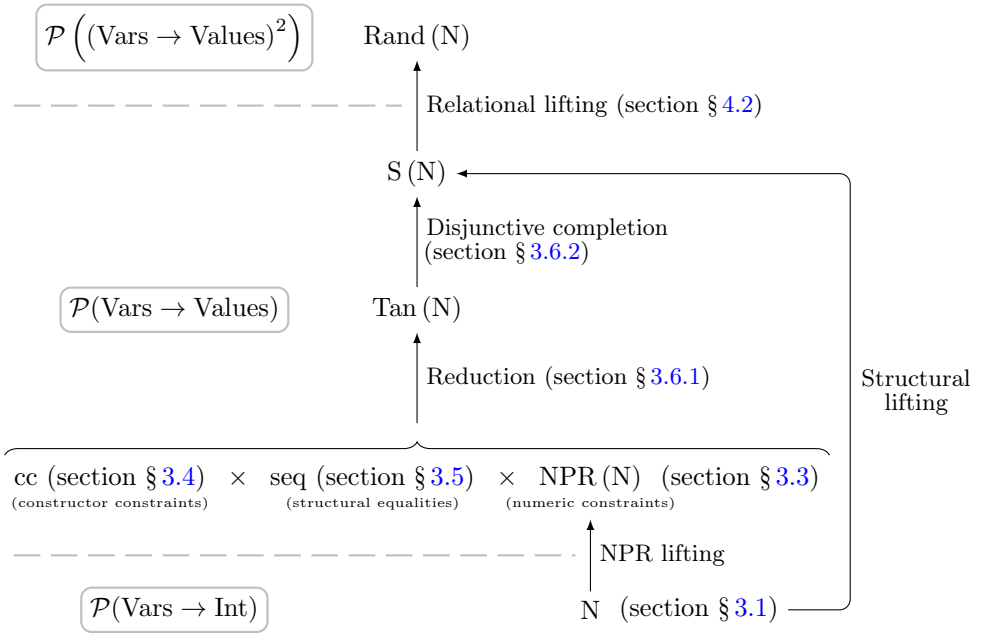

Fig. 5: The construction of the RAND abstract domain. The frame-enclosed sets are the sets the abstract domains concretize to.

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.

Sections § 3 to 5 explain in detail how we express and capture these properties by presenting the structure of the RAND abstract domain. The correlation abstract domain [10] 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.

3 Extending Numeric Domains to Algebraic Types

In this section, we introduce an abstract domain that is able to express equality and numeric constraints between parts of structured values. Our construction is summarised in Fig. 5. It is parametric with respect to a numeric abstract domain N, so that we can instantiate it on domains with different precision versus cost trade-offs. We expect the numeric domain N to provide the operations described in section § 3.1, which are a

subset of the API offered by Apron [11]. An essential ingredient of our construction is the use of *extended variables* (section § 3.2), *i.e.*, regular program variables equipped with a projection path. Our example `do_ticks` on Fig. 4 features extended variables, such as `p.status@Asleep.secs`. Using extended variables, we define in section § 3.3 a first way to lift numeric domains to languages with algebraic types: the *Numeric Path Relations* lifting, or *NPR lifting* for short. This first lifting builds on the ideas of [15], but achieves a better precision. 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 $\texttt{p.status@Asleep.secs} \geq \texttt{p}'\texttt{.status@Asleep.secs}$.

We improve the precision of the *NPR lifting* by combining it with two other domains (sections § 3.4 and 3.5) in a product domain (section § 3.6.2). A first domain of *constructor constraints* tracks which constructors are used for values of sum types (section § 3.4). 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 the output process `p'` is running—*i.e.*, `p'.status@Running` is valid. Another domain, called *structural equalities* (section § 3.5), 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.

In order to obtain additional precision when analysing pattern-matching, we use a *disjunctive completion* of the product of these domains (section § 3.6.2): 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. In sections § 3.3 to 3.5 we define abstractions for assignments and conditionals, that are needed in section § 5 to define the analysis of our language.

Finally, we define the *relational* lifting which extends the intra-procedural analysis to an inter-procedural analysis that computes function summaries of the input-output relation of a function. This lifting extends the relational domain constructions to describe relations between input and output stores, by introducing variables that represent the store at intermediate points in the execution of the function.

3.1 Background: Numeric Abstract Domains

We first review the structure of traditional numeric domains [16] 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 [11]) in such a way that the standard soundness properties of abstract interpretation [17, 18] 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 $\mathrm{Assign}^{\mathrm{N}(V)} \in V \times \mathrm{Arith}\,(V) \to \mathrm{N}(V) \to \mathrm{N}(V)$ (where $\mathrm{Arith}\,(V)$ is the set of arithmetic expressions over the variables V), satisfying the soundness properties:

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

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

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

From this predicate, another predicate $\mathrm{Satisfies}^{\mathrm{N}(V)}$ can be derived, by taking $\mathrm{Satisfies}^{\mathrm{N}(V)}(d, c) = \neg\,\mathrm{CanSat}^{\mathrm{N}(V)}(d, \neg c)$. The predicate $\mathrm{Satisfies}^{\mathrm{N}(V)}$ verifies the property

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

In other words, $\mathrm{Satisfies}^{\mathrm{N}(V)}(d, c)$ 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:

- $\mathrm{Rem}_{V'}^{\mathrm{N}(V)}$ projects an element of $\mathrm{N}(V)$ onto $\mathrm{N}(V \setminus V')$.
- $\mathrm{Add}_{V'}^{\mathrm{N}(V)}$ embeds an element of $\mathrm{N}(V)$ into the domain $\mathrm{N}(V \cup V')$.
- $\mathrm{Rename}_{r}^{\mathrm{N}(V_1)}$ translates elements of $\mathrm{N}(V_1)$ into $\mathrm{N}(V_2)$, given bijection $r : V_1 \to V_2$.

These operators satisfy the soundness properties:

$$\begin{array}{ll} \gamma^{\mathrm{N}(V \setminus V')}(\mathrm{Rem}_{V'}^{\mathrm{N}(V)}(d)) & \supseteq \{\, s|_{(V \setminus V')} \mid s \in \gamma^{\mathrm{N}(V)}(d)\} \\ \gamma^{\mathrm{N}(V \cup V')}(\mathrm{Add}_{V'}^{\mathrm{N}(V)}(a)) & \supseteq \{s : (V \cup V') \to \mathrm{Int} \mid\ s|_V \in \gamma^{\mathrm{N}(V)}(a)\} \\ \gamma^{\mathrm{N}(V_2)}(\mathrm{Rename}_{r}^{\mathrm{N}(V_1)}(d)) & \supseteq \{s \mid s \circ r \in \gamma^{\mathrm{N}(V_1)}(d)\} \end{array}$$

When adding adding variables with $\mathrm{Add}_{V'}^{\mathrm{N}(V)}$, the new variables V' are unconstrained: they can hold any value.

3.2 Extended Variables

We use the name *extended variable* for a pair of a variable and a path. Extended variables designate 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 Fig. 4, the judgements status $\Downarrow^{\text{typ}}$ @Running.count and status $\Downarrow^{\text{typ}}$ @Asleep.secs both hold. However, the judgement 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 $\mathcal{E}(\Gamma) = \{x, x@\mathsf{Running}, x@\mathsf{Asleep}, x@\mathsf{Running.count}, x@\mathsf{Asleep.secs}, x@\mathsf{Asleep.count}\}$.

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 $x@\mathsf{Running.count}$ and $x@\mathsf{Asleep.count}$ are incompatible. Therefore, 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 § 3.4, 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 § 3.6.2.

Assignment decomposition

To simplify the definition of the abstract transfer functions for assignment in the next subsections, 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:*

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

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

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

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

We overload the function $\text{Env}\,(\boldsymbol{.})$ to denote the function that returns the set of extended variables that appear in an expression (boolean or numeric), in a set of commands, or in a set of constraints C.

3.3 Numeric Domains Over Extended Variables

In this section, extending ideas from [15], we define the *Numeric Path Relations lifting* NPR(N) 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). 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 $d \in \text{N}(E)$ whose variables belong to E.

Definition 7. $\text{NPR}\,(\text{N})(\Gamma) = \{(d, E) \mid E \in \mathcal{P}(\mathcal{E}(\Gamma)) \wedge d \in \text{N}(E)\}$

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 § 3.6.2 will only consider sets E that are consistent. When writing examples in the rest of the paper, 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, for the type status defined in Fig. 4, the abstract value $\{\mathsf{st@Asleep.secs} \geq 0;\ \mathsf{st@Asleep.count} \geq 0\}$ is an element of $\text{NPR}\,(\text{N})(\Gamma)$, where $\Gamma = [\mathsf{st} \mapsto \mathsf{status}]$.

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 (Definition 3), 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 $s \in \text{Vars} \to \text{Values}$ on a set of extended variables $E \in \mathcal{P}(\mathcal{E}(\Gamma))$ is a store in $E \to (\text{Values} \cup \{\text{Undef}\})$, written $s \Downarrow^{\text{sto}} E$, and is defined by: $(s \Downarrow^{\text{sto}} E)(x.p) = s(x) \Downarrow^{\text{val}} p$.*

As an example, consider the store $s = [\mathsf{st} \mapsto \mathsf{Asleep}\{\mathsf{secs} = 42; \mathsf{count} = 7\}]$, where the variable st has type status, and the set of extended variables

$E = \{\mathsf{st@Asleep.secs};\ \mathsf{st@Asleep.count}\}$. The projection of s on E is the store $[\mathsf{st@Asleep.secs} \mapsto 42;\ \mathsf{st@Asleep.count} \mapsto 7]$.

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

For example, if the typing context is $\Gamma = [\mathsf{st} \mapsto \mathsf{status}]$, then the concretisation of the abstract value $p = \{\mathsf{st@Asleep.secs} \geq 0;\ \mathsf{st@Asleep.count} \geq 0\}$ contains any store in which the variable st is mapped to a value of type status that is built using constructor Asleep, and where both fields secs and count are non-negative. In particular,

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

We briefly explain how to define the abstract intersection in $\mathrm{NPR}(\mathrm{N})(\Gamma)$. The intersection of (d_1, E_1) and (d_2, E_2) denotes the conjunction of the constraints d_1 and d_2. Therefore, the extended variables that appear in the conjunction are in $E_1 \cup E_2$. Thus, one must inject d_1 and d_2 in $E_1 \cup E_2$ using the $\mathrm{Add}_{E_j}^{\mathrm{N}(E_i)}$ operators, before actually taking their intersection in the numeric domain:

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

The pre-order, union and widening are defined in a similar way:

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

$$\begin{aligned}
(d, E) \sqcap^{\mathrm{NPR}(\mathrm{N})(\Gamma)} (d', E') &= \left(\mathrm{Add}_{E'}^{\mathrm{N}(E)}(d) \sqcap^{\mathrm{N}(E \cup E')} \mathrm{Add}_{E}^{\mathrm{N}(E')}(d'), E \cup E' \right) \\
(d, E) \sqcup^{\mathrm{NPR}(\mathrm{N})(\Gamma)} (d', E') &= \left(\mathrm{Rem}_{E \setminus E'}^{\mathrm{N}(E)}(d) \sqcup^{\mathrm{N}(E \cap E')} \mathrm{Rem}_{E' \setminus E}^{\mathrm{N}(E')}(d'), E \cap E' \right) \\
(d, E) \triangledown^{\mathrm{NPR}(\mathrm{N})(\Gamma)} (d', E') &= \left(\mathrm{Rem}_{E \setminus E'}^{\mathrm{N}(E)}(d) \triangledown^{\mathrm{N}(E \cap E')} \mathrm{Rem}_{E' \setminus E}^{\mathrm{N}(E')}(d'), E \cap E' \right)
\end{aligned}$$

Transfer Functions

Since our values are immutable, all the assignments are of the form $x := t$. In particular, in-place updates of the form $x.f := t$ are not part of our language. In the following definition, E_x is the set of extended variables in E starting with variable x. The transfer function for assignment $x := t$ applied to an abstract element (d, E) 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). In order to do so, it first computes the set E_0 of extended variables appearing in the decomposition of $x' := t$ and adds these to the underlying abstract domain by computing $\mathrm{Add}_{E_0}^{\mathrm{N}(E)}(d)$). This produces the new abstract value d_1. Then, it removes all the references E_x to the paths of x from d_1, 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.

$$\mathrm{Assign}^{\mathrm{NPR(N)}(\Gamma)}(x := t)(d, E) \;=\; \mathrm{Rename}^{\mathrm{NPR(N)}(\Gamma)}_{[x' \mapsto x]}(\mathrm{Rem}^{\mathrm{NPR(N)}(\Gamma)}_{E_x}(d_1, E \cup E_0))$$

where

$$\begin{aligned} E_x &= \{y.p \in E \mid y = x\} \\ E_0 &= \{y.p \in \mathrm{Env}\,(\mathrm{Decomp}\,(x' := t)) \mid \Gamma \vdash y.p : \mathrm{Int}\} \\ d_1 &= \prod_{x'.p := u \in \mathrm{Decomp}(x' := t), \Gamma \vdash u : \mathrm{Int}}^{\mathrm{N}(E \cup E_0)} \mathrm{Assign}^{\mathrm{N}(E \cup E_0)}(x'.p := u)(\mathrm{Add}^{\mathrm{N}(E)}_{E_0}(d)) \end{aligned}$$

If x does not appear in t, a simpler definition for the transfer function could be given, where there is no need for x' and previous constraints on x are forgotten before adding the new ones.

The transfer function for conditionals is simpler: negation is eliminated using De Morgan laws, whereas conjunctions and disjunctions are handled by abstract intersection and union, respectively. The remaining case is that of a numeric test $t_1 \bowtie t_2$: it suffices to call the transfer function of domain N for conditionals, and to extend the extended variables with those that occur in the test.

$$\begin{aligned} &\mathrm{Cond}^{\mathrm{NPR(N)}(\Gamma)}(b)(d, E) = \\ &\qquad \left(\mathrm{Cond}^{\mathrm{N}(E \cup \mathrm{Env}(b))}(b)(\mathrm{Add}^{\mathrm{N}(E)}_{\mathrm{Env}(b)}(d)), E \cup \mathrm{Env}\,(b)\right) \text{ if } b = t_1 \bowtie t_2 \end{aligned}$$

3.4 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 $E \in \mathrm{cc}(\Gamma)$ is a set of extended variables, that restricts the possible sets of stores to those that are compatible with *every* path in E. In other words, if a path in E mentions some constructor, then the corresponding value in any store of the concretisation must be built using that constructor. Elements of $\mathrm{cc}(\Gamma)$ can contain paths which start with different variables, providing a limited form of relational information. For example, the set $\{x, x@\mathsf{Asleep}, y, y@\mathsf{Asleep}\}$ states that the values in variables x and y both are built using the same constructor Asleep. Constructor constraints are a key ingredient of the disjunctive completion of section § 3.6.2, as they serve as hints for which disjuncts need to be kept separate, and which should be merged.

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

Definition 10 (Constructor constraints)**.** *The domain of constructor constraints is defined by* $\mathrm{cc}(\Gamma) = \{E \subseteq \mathcal{E}(\Gamma) \mid E \preccurlyeq\text{-closed and consistent}\} \cup \{\bot^{\mathrm{cc}(\Gamma)}\}$ *and is equipped with the ordering* $\sqsubseteq^{\mathrm{cc}(\Gamma)}$ *defined as* $E_1 \sqsubseteq^{\mathrm{cc}(\Gamma)} E_2$ *iff* $E_1 = \bot^{\mathrm{cc}(\Gamma)}$ *or* $E_1 \supseteq E_2$.

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 Fig. 4 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 11 (Concretisation for constructor constraints)**.**

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

The concretisation of a set E produces a set of well-typed stores such that the values in the stores can be projected along the paths in E. For example, the concretisation of the abstract value $E = \{x@\mathsf{Asleep.secs}, x@\mathsf{Asleep}, x\}$ contains all the stores in which the variable x holds a value that is built with constructor Asleep. In particular, the store

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

belongs to $\gamma^{\mathrm{cc}(\Gamma)}(E)$, 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:

$$\bot^{\mathrm{cc}(\Gamma)} \sqcup^{\mathrm{cc}(\Gamma)} E = E \sqcup^{\mathrm{cc}(\Gamma)} \bot^{\mathrm{cc}(\Gamma)} = E \qquad E_1 \sqcup^{\mathrm{cc}(\Gamma)} E_2 = E_1 \cap E_2 \text{ otherwise}$$

$$E_1 \sqcap^{\mathrm{cc}(\Gamma)} E_2 = \begin{cases} \bot^{\mathrm{cc}(\Gamma)} & \text{if } E_1 = \bot^{\mathrm{cc}(\Gamma)} \text{ or } E_2 = \bot^{\mathrm{cc}(\Gamma)} \text{ or } E_1 <> E_2 \\ E_1 \cup E_2 & \text{otherwise} \end{cases}$$

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 *"kill-gen"* form as follows:

$$\mathrm{Assign}^{\mathrm{cc}(\Gamma)}(x := t)(E) \;=\; (E \setminus \mathrm{Kill}^{\mathrm{cc}(\Gamma)}(x)(E)) \sqcap^{\mathrm{cc}(\Gamma)} \mathrm{Gen}^{\mathrm{cc}(\Gamma)}(x := t)(E)$$

where

$$\mathrm{Kill}^{\mathrm{cc}(\Gamma)}(x)(E) \;=\; \{y.p \in E \mid y = x\}$$
$$\mathrm{Gen}^{\mathrm{cc}(\Gamma)}(x := t)(E) \;=$$
$$\mathrm{clos}^{\mathrm{cc}(\Gamma)}(\{y.q \in \mathrm{Env}\,(t) \mid y \neq x\} \cup \{x.p \mid \exists t', x.p := t' \in \mathrm{Decomp}\,(x := t)\})$$

The extended variables that must be removed are those that have x as root, since the new value for x might be modified by the assignment. The newly added extended variables are those of t that are still live after x is updated, and the ones that are effectively assigned, as given by the decomposition of the assignment. We ensure that the added variables remain prefix-closed thanks to a call to $\mathrm{clos}^{\mathrm{cc}(\Gamma)}$.

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

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

3.5 Structural Equalities

The NPR lifting of Section § 3.3 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 Fig. 4, 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 $\mathrm{seq}(\Gamma)$, that tracks *structural equalities*. The domain $\mathrm{seq}(\Gamma)$ 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 $\mathrm{seq}(\Gamma)$ 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 $\mathrm{seq}(\Gamma)$ are either the bottom element—denoting the empty set of stores—or a finite set 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 12 (Domain of structural equalities)**.** *The domain of structural equalities* $\mathrm{seq}(\Gamma) = \mathcal{P}(\mathcal{E}(\Gamma) \times \mathcal{E}(\Gamma)) \cup \{\bot^{\mathrm{seq}(\Gamma)}\}$ *is equipped with the concretisation function* $\gamma^{\mathrm{seq}(\Gamma)} \in \mathrm{seq}(\Gamma) \to \mathcal{P}(\mathrm{Vars} \to \mathrm{Values})$ *that is defined as follows:*

$$\begin{aligned}\gamma^{\mathrm{seq}(\Gamma)}(\bot^{\mathrm{seq}(\Gamma)}) &= \emptyset \\ \gamma^{\mathrm{seq}(\Gamma)}(C) &= \{s \mid \Gamma \vdash s \wedge \forall (x.p, y.q) \in C,\ s(x) \Downarrow^{\mathrm{val}} p = s(y) \Downarrow^{\mathrm{val}} q \neq \mathrm{Undef}\}\end{aligned}$$

Abstract values in this domain might carry some implicit information. For example, if x and y have type $\{f \to \mathrm{Int}; g \to \mathrm{Int}\}$, 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 13 (Closure of structural equalities)**.** *The* closure *of a set of structural equalities C with respect to a set of extended variables E, written* $\mathrm{clos}^{\mathrm{seq}(\Gamma)}_{E}(C)$*, is the smallest set that contains C, that contains the equalities $x.p = x.p$ for all $x.p \in E$, that is closed under symmetry, reflexivity and transitivity, and that satisfies the following*

congruence property:

$$(x.p, y.q) \in \mathrm{clos}_E^{\mathrm{seq}(\Gamma)}(C) \wedge x.pr \in \mathrm{Env}\left(\mathrm{clos}_E^{\mathrm{seq}(\Gamma)}(C)\right) \Rightarrow (x.pr, y.qr) \in \mathrm{clos}_E^{\mathrm{seq}(\Gamma)}(C)$$

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 § 3.6.

Definition 14 (Pre-order, abstract union, abstract intersection and widening for the structural equalities domain)**.** *A pre-order for the structural equalities domain is defined by*

$$C_1 \sqsubseteq^{\mathrm{seq}(\Gamma)} C_2 \;\; \textit{iff} \;\; C_1 = \bot^{\mathrm{seq}(\Gamma)} \;\; \textit{or} \; C_2 \subseteq C_1$$

Abstract union for this domain is defined by

$$\begin{array}{lll} \bot^{\mathrm{seq}(\Gamma)} \sqcup^{\mathrm{seq}(\Gamma)} C = C \sqcup^{\mathrm{seq}(\Gamma)} \bot^{\mathrm{seq}(\Gamma)} = C & & \\ C_1 \quad\;\; \sqcup^{\mathrm{seq}(\Gamma)} C_2 = C_1 \cap C_2 & & \textit{otherwise} \end{array}$$

Abstract intersection is given by

$$C_1 \sqcap^{\mathrm{seq}(\Gamma)} C_2 = \begin{cases} \bot^{\mathrm{seq}(\Gamma)} & \textit{if } C_1 = \bot^{\mathrm{seq}(\Gamma)} \;\; \textit{or} \; C_2 = \bot^{\mathrm{seq}(\Gamma)} \\ & \quad \textit{or} \;\; \mathrm{Env}\,(C_1) <> \mathrm{Env}\,(C_2) \\ C_1 \cup C_2 & \textit{otherwise} \end{cases}$$

For widening, we use abstract union

$$C_1 \bigtriangledown^{\mathrm{seq}(\Gamma)} C_2 = C_1 \sqcup^{\mathrm{seq}(\Gamma)} C_2$$

If this domain was used on its own, then the union, intersection and widening given here would be unnecessarily imprecise, because of the lack of closure. But since we use it as part of the reduced product of section § 3.6.1, the closure operator will actually be applied.

The reason why taking the abstract union (that is, intersection of the underlying sets of equalities) as widening works here, is because every element of this domain is a *finite* set of equalities, and abstract union can only decrease it. A decreasing sequence of finite sets necessarily stabilises in finite time.

Transfer Functions

The transfer function for assignment $x := t$ for the seq(Γ) 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 state *before* the assignment.

$$\mathrm{Assign}^{\mathrm{seq}(\Gamma)}(x := t)(C) = \left(C \setminus \mathrm{Kill}^{\mathrm{seq}(\Gamma)}(x)(C)\right) \cup \mathrm{Gen}^{\mathrm{seq}(\Gamma)}(x := t)(C)$$

where

$$\mathrm{Kill}^{\mathrm{seq}(\Gamma)}(x)(C) = \{(y.p, z.q) \in C \mid y = x \vee z = x\}$$
$$\mathrm{Gen}^{\mathrm{seq}(\Gamma)}(x := t)(C) = \bigcup_{x.p:=y.q \in \mathrm{Decomp}(x:=t)} \{(x.p, z.r) \mid z \neq x \wedge (y.q, z.r) \in \mathrm{clos}^{\mathrm{seq}(\Gamma)}{}_{\{y.q\}}(C)\}$$

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

3.6 Bringing Everything Together: Product Domain and Disjunctive Completion

In this section, we describe the remaining steps of our construction, that lead to the *structural lifting* S of the numeric domain N that we have considered. We combine the domains we have defined in the previous sections—the constructor constraints (section § 3.4), the structural equalities (section § 3.5), and the NPR lifting (section § 3.3)—into a *reduced product*, that we call Tan (N), for *Tuple for Algebraic types and Numbers*. Then, we add a disjunctive completion layer on top of that product. We will ultimately obtain the domain of relations RAND once we apply the relational lifting defined in section § 4.2.

3.6.1 Reduced Product

We equip the product domain $\mathrm{Tan}\,(\mathrm{N}) = \mathrm{cc} \times \mathrm{seq} \times \mathrm{NPR}\,(\mathrm{N})$ with a reduction operator ρ, that enables information transfer between the different domains of the product.

Definition 15 (Reduction operator)**.** *The reduction operator ρ for the product of constructor constraints, structural equalities and the NPR lifting is defined as follows:*

$$\rho(E, C, N) = \begin{pmatrix} E \sqcap^{\mathrm{cc}(\Gamma)} \mathrm{clos}^{\mathrm{cc}(\Gamma)}(\mathrm{Env}\,(C')), \\ C', \\ \mathrm{Cond}^{\mathrm{NPR(N)}(\Gamma)}(\bigwedge_{(x.p,y.q) \in C' \wedge \Gamma \vdash x.p:\mathrm{Int}} x.p = y.q)(N) \end{pmatrix}$$

where $C' = \mathrm{clos}^{\mathrm{seq}(\Gamma)}{}_{\mathrm{clos}^{\mathrm{cc}(\Gamma)}(\mathrm{Env}((E,C,N)))}(C)$

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 constrained (this is the role of C').
- 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 [19] for further

details. The transfer functions for assignment and conditionals use the transfer functions of each component.

3.6.2 Using Disjunctions to Handle Incompatible Cases

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 [18] 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 $\mathrm{S}(\mathrm{N}) = \mathcal{P}(\mathrm{Tan}(\mathrm{N}))$. 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 16 defines what it means for two sets of constructor constraints to be equivalent. This definition is used in Definition 17 to describe how, for the elements of the disjunctive completion, cases with equivalent constructor constraints are merged together, using abstract union. Because we merge equivalent cases by computing their *unions*, we cannot produce incorrect results, even when some numeric variables are defined in some cases only. For example, if some variable x is undefined in one case of a disjunction, it is considered as *unconstrained* in the numerical domain for that case of the disjunction, *i.e.*, no numeric information is known about this x. Thus, merging this case with another one will not create any meaningful information on x in the merged abstract value.

Definition 16 (Equivalence of constructor constraints)**.** *Two sets of constructor constraints E_1 and E_2 are said to be* equivalent*, written $E_1 \equiv E_2$, if any path* ending with a constructor *present in one of them is also present in the other. Formally:*

$$E_1 \equiv E_2 \text{ iff } \forall x.(p@A) \in E_1, x.(p@A) \in E_2 \wedge \forall y.(q@B) \in E_2, y.(q@B) \in E_1$$

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

Notice that this equivalence is not an equality because the E_i can contain paths that do not end in a constructor, but in a field selector. For such paths there is no requirement that they either be present in both or none of the E_i. We can extend this into an equivalence relation for elements of the Tan domain, by:

$$(E_1, C_1, N_1) \equiv_{\mathrm{E}}^{\mathrm{Tan}} (E_2, C_2, N_2) \text{ iff } E_1 \equiv E_2$$

This equivalence is restricted to the extended variables, and does not correspond to an equivalence with respect to $\sqsubseteq^{\mathrm{Tan}}$ nor to a semantic equivalence. The sub-script in the notation $\equiv_{\mathrm{E}}^{\mathrm{Tan}}$ reminds of that. Given an element $\mathcal{O}$ of S, we write $\mathcal{O}/\equiv_{\mathrm{E}}^{\mathrm{Tan}}$ for the set of equivalence classes of $\mathcal{O}$ with respect to $\equiv_{\mathrm{E}}^{\mathrm{Tan}}$.

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

Definition 17 (Collapse operator for the disjunctive completion)**.** *We define an operator* $\mathrm{Collapse}^{\mathrm{S}}$ *that takes a set of elements of* Tan *and merges together (by taking the abstract union), the elements that are equivalent with respect to* $\equiv_{\mathrm{E}}^{\mathrm{Tan}}$*. Formally,*

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

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

$$\mathcal{O}_1 \sqcup^{\mathrm{S}} \mathcal{O}_2 = \mathrm{Collapse}^{\mathrm{S}}(\mathcal{O}_1 \cup \mathcal{O}_2)$$
$$\mathcal{O}_1 \sqcap^{\mathrm{S}} \mathcal{O}_2 = \mathrm{Collapse}^{\mathrm{S}}\left(\left\{ t_1 \sqcap^{\mathrm{Tan}} t_2 \mid \begin{array}{c} t_1 \in \mathcal{O}_1 \wedge t_2 \in \mathcal{O}_2 \wedge \\ t_1 \sqcap^{\mathrm{Tan}} t_2 \neq \bot^{\mathrm{Tan}} \end{array} \right\} \right)$$

A widening for the structural lifting is given by

$$\begin{array}{l} \mathcal{O}_1 \triangledown^{\mathrm{S}} \mathcal{O}_2 = \\ \quad \{ t_1 \triangledown^{\mathrm{Tan}} t_2 \mid t_1 \in \mathcal{O}_1 \wedge t_2 \in \mathcal{O}_2 \wedge t_1 \equiv_{\mathrm{E}}^{\mathrm{Tan}} t_2 \} \\ \quad \cup \{ t_2 \in \mathcal{O}_2 \mid \nexists t_1 \in \mathcal{O}_1, t_1 \equiv_{\mathrm{E}}^{\mathrm{Tan}} t_2 \} \cup \{ t_1 \in \mathcal{O}_1 \mid \nexists t_2 \in \mathcal{O}_2, t_1 \equiv_{\mathrm{E}}^{\mathrm{Tan}} t_2 \} \end{array}$$

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:

$$\mathcal{O}_1 \sqsubseteq^{\mathrm{S}} \mathcal{O}_2 \text{ iff } \forall a \in \mathcal{O}_1, \exists a' \in \mathcal{O}_2, a \sqsubseteq^{\mathrm{Tan}} a'$$

4 A Collecting Semantics of Relations

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*. Relations between different versions of variables at several program points are discovered, for instance, during the symbolic expression analysis [20] that is part of the construction of the SSA form of a control-flow graph. In the rest of this paper, the term "relational" refers to to *input-output* relational analysis, that computes relations between the input states and the output states of a program.

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

$\mathbb{S}[\![c]\!]$ is a binary relation that may be employed to derive fully compositional static analyses, such as CRA [21, 22]. Indeed, it enjoys equations (*e.g.*, $\mathbb{S}[\![c_1\,;c_2]\!] = \mathbb{S}[\![c_1]\!]\,;\mathbb{S}[\![c_2]\!]$) that help defining the analysis of a compound command from the *independent* analyses of its constituents. A drawback of this 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 of the next section waives this limitation, as it allows to exploit the information that has so far been obtained for the current program point.

4.1 Relational Collecting Semantics

In this section, we build a collecting relational semantics on top of $\mathbb{S}\,[\![c]\!]$, that can exploit the information about the states that have been reached. Let us consider again the example from the previous paragraph: with the knowledge that the assertion `assert (x > 1 && y > 1)` succeeded, then a linear relational domain will be able to express, for example, that after the assignment `x := y * x`, the value of `x` has strictly increased. 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. Our collecting semantics is defined as follows:

Definition 19 (Collecting semantics)**.** $\mathbb{P}\,[\![c]\!]\,(a) = a; \mathbb{S}\,[\![c]\!]$

$\mathbb{P}\,[\![c]\!]$ is an abstraction of a semantics of computation traces [23], that collects the intermediate stores that a program may reach, that only keeps the initial and the final states of such traces. The collecting semantics $\mathbb{P}\,[\![c]\!]$ satisfies the equations listed in the next lemma, that shows how it decomposes by following the syntax of commands.

Lemma 1 (Inductive Characterisation of the Collecting Semantics)**.** *The following equations hold:*

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

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. □

This inductive characterisation of the collecting semantics will serve as the semantic basis for the analysis that we describe in section § 5.2.

Lemma 1 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 § 4.2 how to define 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 [24–27].

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).

4.2 Leveraging Relations in Space to Express Relations in Time

In this section we briefly recall 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, *i.e.*, using a *two-vocabulary relation* [26].

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. We write x' as a shorthand for $\mathit{prime}(x)$, and x'' for $\mathit{second}(x)$, and use the same convention as in [21, 26], *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. 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.

We can abstract a relation between stores s_1 and s_2 using a single abstract value, that represents the set of stores $s_1 \cup s_2'$, which effectively duplicates a variable into its *input* and its *output* versions.

Definition 20 (Relational lifting)**.** *Let* D *be a family of abstract domains, such that for any* Γ*, the domain* $\mathrm{D}(\Gamma)$ *has a concretisation function* $\gamma^{\mathrm{D}(\Gamma)} \in \mathrm{D}(\Gamma) \to \mathcal{P}(\text{Vars} \to \text{Values})$*. For any* 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:*

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

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

The relational lifting expects two typing contexts—one for the input stores, and one for the output stores. The use of two contexts is necessary to properly handle function calls and returns (section § 5.3), where the variables of the caller's context need to be distinguished from those of the callee's.

To obtain 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. It appears they can both be expressed in a generic way, by exploiting the transfer functions of the underlying domain.

The transfer function for conditionals

$$\mathrm{Cond}^{\mathrm{R(D)}(\Gamma_1,\Gamma_2)}(b)(a) = \mathrm{Cond}^{\mathrm{D}(\Gamma_1 \cup \Gamma_2')}(b')(a)$$

constrains the right-hand side of the relation a to satisfy the boolean expression b. This is achieved by calling the 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 assignments

$$\mathrm{Assign}^{\mathrm{R(D)}(\Gamma_1,\Gamma_2)}(x := t)(a) = \mathrm{Assign}^{\mathrm{D}(\Gamma_1 \cup \Gamma_2')}(x' := t')(a)$$

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.

In the rest of this article, we use the relational lifting of the abstract domain from section § 3, that we call RAND—short for *Relational Algebraic and Numeric Domain*.

5 Analysis

This section explains how to use the abstract domain built in sections § 3 and 4, to analyse the language described in section § 2.2. After providing an example that illustrates what the analysis computes (section § 5.1), we first describe an intra-procedural analysis (section § 5.2) and then extend it to support function calls, yielding a modular, summary-based, inter-procedural analysis (section § 5.3).

The inter-procedural version of our analysis does not currently handle recursive or mutually recursive functions, as we chose to focus solely on the topic of handling algebraic values and arithmetic relations. Nevertheless, we expect that the analysis of recursive functions can be achieved by performing a widened fixpoint iteration sequence at the level of function summaries.

5.1 Analysis Result for the `do_ticks` Function

Before giving the formal description of the analysis, we give an example of the properties that it can infer. Figure 6 shows the result of running our analyser on the example from Section § 2.3 using polyhedra as a numeric domain. 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 $\mathtt{p.msg} = \mathtt{p}'\mathtt{.msg}$

```
Function summary for function do_ticks(p, n) returning p' :
 (Constructor constraints : p.status@Running; p'.status@Running ...
  with structural equalities : p = p' ; ...
  and numeric constraints : n >= 1 ; ... )
 Or (Constructor constraints : p.status@Asleep; p'.status@Running ...
  with structural equalities : p.msg = p'.msg
  and 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 ...
  with structural equalities : p.msg = p'.msg
  and 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 )
```

Fig. 6: Result of our analysis on the example of Fig. 4. Ellipses mark information that is also present in other components of the same case and is elided.

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).

5.2 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 § 5.3 explains how we analyse functions.

Definition 21 (Intra-procedural version of the analysis function)**.**

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

where $a_0 = a$ *and* $a_{n+1} = a_n \triangledown^{\text{Rand}} \text{Analyse}(\mathsf{assert}(b); c)(a_n)$

The analysis of assertions and assignments use the transfer functions we built in previous sections. Sequence and branching follow the structure outlined in Lemma 1.

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* [28, p.131] of the first iteration, in order to obtain better precision.

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

Theorem 2 (Soundness w.r.t. the collecting semantics)**.** *For any command c and abstract value $a \in$ Rand,*

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

Proof sketch. Given the inductive characterisation of $\mathbb{P}$ from Lemma 1 and the definition of Analyse, Theorem 2 can be proven by induction on the syntax of programs. The different cases rely on the soundness of the operators from the Rand domain. □

By instantiating Theorem 2 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 command c,*

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

Where Id^{Rand} *is the element of* Rand *that has a structural equality $x' = x$ for all variables in the typing context.*

5.3 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 § 5.2 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:

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

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 remain unchanged.

A program is a sequence of function declarations $\mathsf{def}\, f(\tau_1\ x_1, \ldots, \tau_n\ x_n) : \tau = \{c\,;\mathsf{return}\,r\}$, that specify for the function f what are its input and output variables and their types, and defines its body c. 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

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

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

Fig. 7: Small-step semantics for functions

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.

We extend the small-step reduction rules from Fig. 1 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 all changes to the stack, whereas the other rules of Figure 1 leave the stack unchanged. Then, we augment the reduction relation with two new rules (Figure 7). 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 [29].

Definition 22 (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:*

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

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 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. This has the advantage that we can reuse the *same* summary in *every* calling context. The downside of this choice is that the analysis of a function body cannot take advantage of

any contextual information about the arguments to the function. Starting with other relations than the identity relation could possibly remedy this short-coming; this has not been explored further.

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 23 (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 $c_1 = \mathrm{Rename}_{second \circ prime^{-1}}\, a_1$ *and* $c_2 = \mathrm{Rename}_{second}\, a_2$.

Note that some domains like Rand are parametrized by typing contexts and not by sets of variables, since variables of different types are handled differently by the abstract domain. Hence the Add and Remove operators are here parametrized by typing contexts, and not by sets of variables. Additionally, the renaming functions *second* and $second \circ prime^{-1}$ are extended to behave as identity on the variables that are not in their definition domain but appear in a_1 or a_2.

Abstract composition chains the effects of a_1 and a_2 by introducing auxiliary names—*i.e.*, 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 3 (Soundness of composition)**.** *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. 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 24 (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:*

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

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 deals with which variables might have changed and how, 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 so far gathered relation a with the effect of the call to f:

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

The transfer function for function calls is sound:

Lemma 4 (Soundness of function call analysis)**.** *For every function definition* $\Delta(f) = ((x_1, \ldots, x_n), c_f, y_f, \Gamma_f)$ *in a program, and any function summary* $S_f = ((x_1, \ldots, x_n), a_f, y_f, \Gamma_f)$ *such that* $\mathbb{S}\,[\![c_f]\!] \subseteq \gamma^{\mathrm{Rand(N)}(\Gamma_f, \Gamma_f)}(a_f)$*, we have:*

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

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 $(s_0, s_2) \in \mathbb{P}\,[\![y{:=}f(z_1, \ldots, z_n)]\!](\gamma^{\mathrm{Rand}}(a))$. Unfolding the definition of $\mathbb{P}\,[\![.]\!]$ gives an intermediate store s_1 such that (s_0, s_1) belongs to $\gamma^{\mathrm{Rand}}(a)$ 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 $(y{:=}f(z_1, \ldots, z_n), s_1, \pi) \to (c_f; \mathsf{return}\, y_f, s', (y, s_1) : \pi) \to^* (\mathsf{return}\, y_f, s'', (y, s_1) : \pi) \to (\mathsf{skip}, s_2, \pi)$, 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 f's body. Let *ins* and *outs* be the abstract values used in Definition 24. By the definition of s' in rule FunctionCall, we get $(s_1, s') \in \gamma^{\mathrm{Rand(N)}(\Gamma, \Gamma_f)}(\mathit{ins})$. Moreover, $(s', s'') \in \mathbb{S}\,[\![c_f]\!]$ is obtained by the hypothesis on f's summary. Finally, we prove $(s'', s_2) \in \gamma^{\mathrm{Rand(N)}(\Gamma_f, \Gamma)}(\mathit{outs})$ 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 4 ensures that the soundness result for the intra-procedural analysis (Theorem 2) extends to the language with function calls that we have described in this section.

6 Analysis For Arrays and Algebraic Types

In this section, 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 developments, such as seL4 [30], 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.

A number of approaches for analysing programs that manipulate arrays have been proposed [12, 31–41]. We base our development on a proposal by Cousot, Cousot and Logozzo [12], that abstracts arrays using segmentations, and that we refer to as CCL in the rest of the article.

Overview of the CCL domain for arrays

In CCL, each array is abstracted by a *segmentation*. A segmentation groups the array indices into several intervals. These intervals are disjoint, some of them may be empty, and they form a partition of the array indices. The group of array slots that corresponds to each one of these intervals of indices is called a *segment*. 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 entries of 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 $|a|$ expression 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 it specifies that 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 the information for the values stored in this segment is $\top$.

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.

Differences with CCL segmentations

In this article, we extend CCL segmentations in two directions. First, we allow the abstract values for 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 section § 3 (structural lifting).

Additionally, we give a new definition for the pre-order between segmentations, as we have found corner cases in the CCL definitions where the concretisation for segmentations was not monotonic with respect to the pre-order.

Section outline

This section is organised as follows: First, we extend our programming language with primitives for arrays and provide a motivating example (section § 6.1). Second, we give an overview of the overall structure of our abstract domain (section § 6.3). Then, we describe our segmentations (section § 6.4), and we focus on the differences between our definitions and the ones of CCL (section § 6.5). In section § 6.7, we give the soundness theorem of our abstract domain. Then, we give the result of our analysis on the motivating example (section § 6.8). Finally, we summarize our approach and its current limitations (section § 6.9).

6.1 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 or nested 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:

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

Array types and values are defined as follows:

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

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 take type status from Fig. 4, the array

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

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.

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

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 8 shows an example of a program that manipulates arrays, and that is inspired by the functional specification of the seL4 micro-kernel [30]. 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 priorities—although TCBs may exhibit more.

```
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
}
```

Fig. 8: Program that finds a thread descriptor with highest priority in an array.

Our example consists of the `find_max_priority` function, that 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 table with the highest priority.

For the `find_max_priority` function, our analysis computes the abstract summary of Fig. 9. Compared to the summaries of section § 5.3, some new information is available, that associates to array variables an array segmentation.

```
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| } ]
  )
```

Fig. 9: Abstract value for the function `find_max_priority` from Fig. 8.

This summary expresses two cases:

- Either the initial array is empty—it has size 0—and the result is `NoMax{}`, or
- the initial array is non-empty, and the priority of all its elements are less than or equal to the priority of the result.

Although this abstract value expresses useful properties of the `find_max_priority` function, it is not an exact abstraction of the behaviour of the function. Indeed, it does not express the fact that the returned TCB actually belongs to the input array.

In the rest of the section, we explain how we build the Diorana domain, using some of the same ingredients from the RAND domain, to compute input-output summaries of functions that manipulate arrays and algebraic types.

6.2 Array Lengths as Variables

Now that there are arrays in our language, we want our abstract domains to be able to capture information on array lengths. Array lengths are numbers. Hence, if x is an array variable, we will introduce into our abstract domains a numeric variable $|x|$, to represent the length of the array stored in x. We call these numeric variables *array length variables*.

To be clear, the $|\,.\,|$ notation is overloaded and can have three different meanings. If t is an array *value*, then $|t|$ is the length of array t. If x is an array *variable*, then $|x|$ can be either an expression of the language (that appears in conditions, for example), or an array length variable, that will appear in the constraints of our abstract domains.

For any typing context Γ we write $\mathrm{Arr}(\Gamma)$ for the set of variables in Γ that have an array type:

$$\mathrm{Arr}(\Gamma) = \left\{x | \exists \tau^{\mathrm{alg}} \in \mathrm{AlgTypes}, \Gamma(x) = \mathrm{Array}(\tau^{\mathrm{alg}})\right\}$$

We also write $\mathrm{L}(\Gamma)$ for the typing context obtained from Γ by replacing any array variable with the corresponding array length variable:

$$\mathrm{L}(\Gamma) = (\Gamma \setminus \mathrm{Arr}(\Gamma))\,[\ |x| \mapsto \mathrm{Int}\ ]_{x \in \mathrm{Arr}(\Gamma)}$$

Similarly, for any environment ρ that is well-typed in a typing context Γ, we write $\mathrm{L}(\rho)$ for the environment obtained from ρ by replacing any array variable by the

corresponding array length variable:

$$\mathrm{L}(\rho) = (\rho \setminus \mathrm{Arr}(\Gamma))\,[\ |x| \mapsto |\rho(x)|\ ]_{x \in \mathrm{Arr}(\Gamma)}$$

We use these notations to define new abstract domains that behave almost identically to the Tan and S domains from section § 3.6.1 and section § 3.6.2, except in that they are able to capture information on array lengths.

Definition 25. *The* TanL *domain. For any numeric domain* N *and any typing context* Γ*, the* TanL *domain is defined by*

$$\mathrm{TanL}(\mathrm{N})(\Gamma) = \mathrm{Tan}(\mathrm{N})(\mathrm{L}(\Gamma))$$

and its concretisation function is given by

$$\gamma^{\mathrm{TanL}}(t) = \left\{\rho \mid \mathrm{L}(\rho) \in \gamma^{\mathrm{Tan}}(t)\right\}$$

All other operators and transfer functions of the TanL *domain are defined as being those of the* Tan *domain.*

Definition 26. *The* SL *domain. For any numeric domain* N *and any typing context* Γ*, the* SL *domain is defined by*

$$\mathrm{SL}(\mathrm{N})(\Gamma) = \mathrm{S}(\mathrm{N})(\mathrm{L}(\Gamma))$$

and its concretisation function is given by

$$\gamma^{\mathrm{SL}}(d) = \left\{\rho \mid \mathrm{L}(\rho) \in \gamma^{\mathrm{S}}(d)\right\}$$

All other operators and transfer functions of the SL *domain are defined as being those of the* S *domain.*

6.3 Structure of Our Abstract Domain For Arrays

Figure 10 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 §6.4). 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 $\mathrm{TanL}\,(\mathrm{N})$ domain from Definition 25 and the array domain $\mathrm{A}\,(\mathrm{N})$. We call this product domain $\mathrm{Tana}\,(\mathrm{N}) = \mathrm{TanL}\,(\mathrm{N}) \times \mathrm{A}\,(\mathrm{N})$ for *Tuple 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 Tan 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 § 4.2 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*.

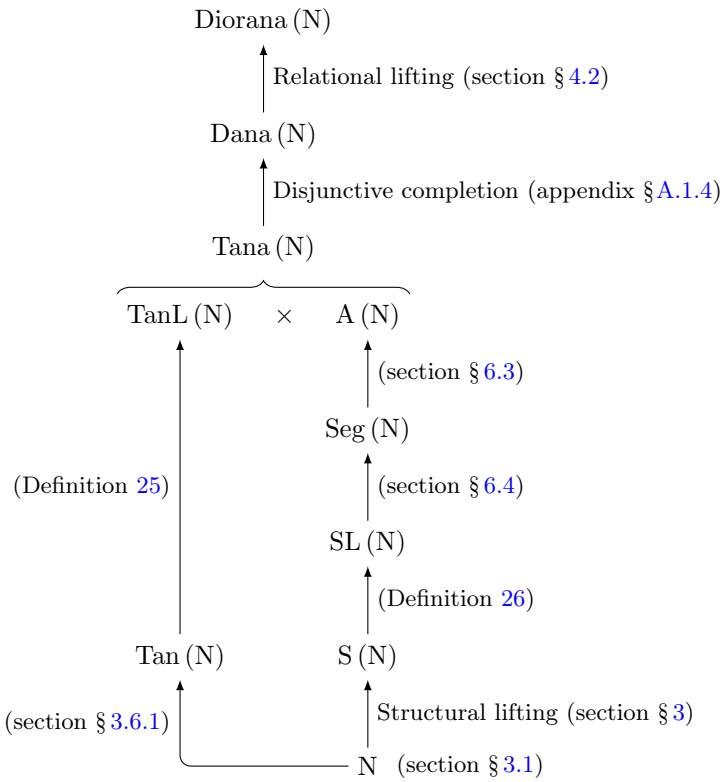

Fig. 10: The construction of the abstract domain for analyzing programs that manipulate both values from algebraic types, and arrays containing values from algebraic types.

6.4 Array Segmentations

We recall that for each array variable x, the numeric variable $|x|$ represents the length of the array contained in x (section § 6.2). For a typing context Γ, we call $\mathrm{V}(\Gamma)$ the set of variables of numeric types and of array lengths variables. Formally, $\mathrm{V}(\Gamma) = \{x \mid \Gamma(x) = \mathrm{Int}\} \cup \{|x| \mid x \in \mathrm{Arr}(\Gamma)\}$. We also call K the set of possible numeric constants in the programming language. In particular, $0 \in \mathrm{K}$.

Definition 27 (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* $\mathcal{P}_{fin}(\mathrm{E}(\Gamma))$*.*

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.

For any set of variables V, let $\mathrm{Fresh}(V) = \mathrm{Vars} \setminus V$ be the set of variables that are fresh with respect to V. 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 28 (Segmentations)**.** *Let x be a variable with an array type in the typing context Γ,* i.e.*, $\Gamma(x) = \text{Array}(\tau)$ for some type τ. A* segmentation $s \in \text{Seg}(\text{N})(\Gamma)(x)$ for variable x *is a quadruplet* $\left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right)$ *where:*

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

In a segmentation $\left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right)$, 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 $i \in \{0, \dots, n\}$ is a boundset that marks the segment limits, and each d_i for $i \in \{1, \dots, n\}$ is a segment summary, that denotes the set of values that a segment can contain. Finally, each m_i for $i \in \{1, \dots, n\}$ 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

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

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

Each segment summary d_i is an abstract value from the SL domain of Definition 26. The segment summaries of a segmentation for the array x can refer to any variable of an algebraic type, any array length variable, 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 $\text{SL}\,(\text{N})\,(\Gamma[l \mapsto \text{Int}; v \mapsto \tau])$, that is the SL built from numeric domain N, for typing context $\Gamma[l \mapsto \text{Int}; v \mapsto \tau]$. This is the typing context obtained from Γ by 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 article, we only consider well-formed segmentations, that are defined as follows.

Definition 29 (Well-formed segmentations)**.** *Let $s = \left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right)$ be a segmentation for the 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*
- *$\forall i \in \{0, \dots, n\}, b_i \neq \emptyset$, and*
- *$\forall i \in \{0, \dots, n\}, \forall j \in \{0, \dots, n\}, i \neq j \Rightarrow b_i \cap b_j = \emptyset$.*

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 § 6.5.1, that the *unification* of segmentations is such an operation [1].

The concretisation of segmentations (Definition 32) 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 30 (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 31 (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 30 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 32 (Concretisation for segmentations)**.** *For a segmentation $s = \big(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\big)$, its concretisation $\gamma^{\mathrm{Seg}}(s)$ is the set of pairs composed of a store ρ and an array value t, that satisfy the following conditions:*

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

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

- *Segment summaries:*

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

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

The five conditions of Definition 32 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*

[1] Allowing empty boundsets could have an interest in itself, but we do not consider that option in the present framework

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 $\{0\} \top^{\mathrm{SL}} \{i; 5\} \top^{\mathrm{SL}} \{|x|\}?$ 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 $[\![b_i]\!]^{\mathrm{exp}}_{\rho}$ 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 29), we know that $0 \in b_0$, hence the first boundset evaluates to zero ($[\![b_0]\!]^{\mathrm{exp}}_{\rho} = 0$).

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

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 $[\![b_{i-1}]\!]^{\mathrm{exp}}_{\rho} < [\![b_i]\!]^{\mathrm{exp}}_{\rho}$ must be satisfied.

The main difference between the segmentations defined here and the ones defined by CCL [12] is that the numeric abstract values used to summarize each segment can refer to the other variables of the program, *in addition* to referring 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.

6.5 Unification and Pre-Order for Segmentations

The goal of this section § 6.5 is to present the differences between our pre-order for segmentations, and the one from CCL [12]. We first give an intuitive explanation of an operator called *unification*, that is used in CCL's definition of segmentation pre-order. Then, we present some corner cases on which the concretisation of CCL's segmentation domain is not monotonic with respect to their definition of segmentation pre-order (section §6.5.2), and we present our definition of segmentation pre-order (section §6.5.3).

6.5.1 Unification of Segmentations

In [12], intersection, union, widening and pre-order on segmentations are defined 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 or pre-order 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 33 (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 *iff the two following conditions are satisfied:*

- $n = p$*, and*
- $\forall i \in \{0, \dots, n\}, b_i^1 = b_i^2$.

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 pre-order—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, splitting a boundset and creating a possibly empty segment, often allows to 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 $\{0\}\ \dots\ \{a\}?$, 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 [12] for a detailed description of the CCL 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 (Lemmas 8 and 9 in appendix §A). For this reason, it is sound to perform a unification before union, intersection and widening (as done both by CCL 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

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

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

However, using unification in the definition of the pre-order, as it is done in CCL, makes concretisation non-monotonic on some corner cases. Indeed, since unification might replace the right-hand side value with a *less precise* one, pre-order is no longer an abstraction of inclusion. We show an example of this in section § 6.5.2 below.

6.5.2 Corner Cases for the Monotonicity of Segmentation Concretisation in CCL

In CCL, the concretisation for segmentations is not monotonic with respect to the pre-order on segmentations. 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 CCL, segmentation pre-order is tested by first unifying segmentations, then testing pre-order segment-wise. The unification of segmentations s^1 and s^2 yields $\{0\}\top\{b\}\top\{|t|\}$ on both sides. Hence, with CCL 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 CCL is not monotonic with respect to the segmentation pre-order of CCL.

We believe there are at least two factors thanks to which this non-monotony of concretisation does not affect soundness in CCL:

- The segmentation domain is part of a product domain that includes a numeric domain. During pre-order testing, the numeric bounds that delimit segments need also to be correctly ordered according to the information captured by the numeric domain. This prevents the counter-example shown above. Indeed the product domain would only state $s_1 \sqsubseteq s_2$ if we also have $a = b = c$, which excludes store ρ_0 from the concretisation.
- Given the way that operators like widening are defined, it is never the case that an argument to the widening and the corresponding result belong to the corner cases that make concretisation non-monotonic. More generally, these corner cases do not show up in places where the monotony of concretisation is needed, thanks to the way the operators are defined.

However, it is a generally expected property for abstract domains that the concretisation be monotonic with respect to the pre-order. Hence we propose a different definition for segmentation pre-order: Definition 34 below. With this definition, segmentation concretisation is monotonic with respect to segmentation pre-order, as proved by Lemma 6 below.

6.5.3 Our 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 34), 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 34 (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:* $\forall i \in \{1,\dots,p\}, \forall j, \phi(i-1) < j \leq \phi(i), \Rightarrow d_j^1 \sqsubseteq^{\text{SL}} d_i^2$
- *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:

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

$$s_2 = \{0\}\ (v \le 0)\ \ \{3\}\ \ \ (v = 0)\ \ \ \{y\}?\ \ \ (v \ge 0)\ \ \ \{7\}?\ (v > 14)\ \{|x|\}?$$

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 34. 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 $3 \le y$, 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 5. *The segmentation inclusion relation $\sqsubseteq^{\text{Seg}}$ is a pre-order on segmentations.*

Proof sketch. For the proof of reflexivity, it suffices to observe that the identity function satisfies all the conditions of Definition 34. The proof of transitivity is based on the fact that if two functions ϕ_1 and ϕ_2 satisfy the properties of Definition 34, so does their composition $\phi_2 \circ \phi_1$. The detailed proof is available in appendix §A.1.1. □

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 6. *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 32, 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 34), 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 32), 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 $i \in \{1, \ldots, n\}$, we have $[\![b^2_{i-1}]\!]^{\exp}_\rho \leq [\![b^2_i]\!]^{\exp}_\rho$. For that, we will take two expressions $e_1 \in b^2_{i-1}$ and $e_2 \in b^2_i$ and prove that $[\![e_1]\!]^{\exp}_\rho \leq [\![e_2]\!]^{\exp}_\rho$. Let $i \in \{1, \ldots, n\}$, $e_1 \in b^2_{i-1}$ 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

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

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

Condition 3: Strict inequalities for non-empty segments.

Now, we need to prove that for any index $i \in \{1, \ldots, n\}$ such that the boolean m^2_i is false, we have $[\![b^2_{i-1}]\!]^{\exp}_\rho < [\![b^2_i]\!]^{\exp}_\rho$. Let $i \in \{1, \ldots, n\}$. For the same reasons than in the previous condition, we have $[\![b^2_{i-1}]\!]^{\exp}_\rho = \left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\exp}_\rho$ and $[\![b^2_i]\!]^{\exp}_\rho = \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\exp}_\rho$. 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 $[\![b^1_{j-1}]\!]^{\exp}_\rho < [\![b^1_j]\!]^{\exp}_\rho$. 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

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

Therefore, $\left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\exp}_\rho < \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\exp}_\rho$. We recall that $[\![b^2_{i-1}]\!]^{\exp}_\rho = \left[\!\left[b^1_{\phi(i-1)}\right]\!\right]^{\exp}_\rho$ and $[\![b^2_i]\!]^{\exp}_\rho = \left[\!\left[b^1_{\phi(i)}\right]\!\right]^{\exp}_\rho$. Hence, we have proven $[\![b^2_{i-1}]\!]^{\exp}_\rho < [\![b^2_i]\!]^{\exp}_\rho$; 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 $[\![b^2_{i-1}]\!]^{\exp}_\rho \leq k < [\![b^2_i]\!]^{\exp}_\rho$, we have $\rho[l \mapsto k][v \mapsto t[k]] \in$

$\gamma^{\mathrm{SL}}(d_i^2)$. Let $i \in \{1, \ldots, n\}$ and k be an index such that $[\![b_{i-1}^2]\!]_\rho^{\exp} \leq k < [\![b_i^2]\!]_\rho^{\exp}$ (if no such index exists, what we want to prove is vacuously true). Like in conditions 3 and 4, the "boundset inclusion" condition of $s^1 \sqsubseteq^{\mathrm{Seg}} s^2$ allows us to deduce that $[\![b_{i-1}^2]\!]_\rho^{\exp} = \left[\!\left[b_{\phi(i-1)}^1\right]\!\right]_\rho^{\exp}$ and $[\![b_i^2]\!]_\rho^{\exp} = \left[\!\left[b_{\phi(i)}^1\right]\!\right]_\rho^{\exp}$. Hence, $\left[\!\left[b_{\phi(i-1)}^1\right]\!\right]_\rho^{\exp} \leq k < \left[\!\left[b_{\phi(i)}^1\right]\!\right]_\rho^{\exp}$. Let's consider the sequence of integer intervals

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

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

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

to which k belongs. Hence, there exists a j such that $\phi(i-1) < j \leq \phi(i)$ and $k \in \left\{ [\![b_{j-1}^1]\!]_\rho^{\exp}, \ldots, [\![b_j^1]\!]_\rho^{\exp} - 1 \right\}$. Using the "segment summaries" condition of $(\rho, t) \in \gamma^{\mathrm{Seg}}(s^1)$ for index j, we have $\rho[l \mapsto k][v \mapsto t[k]] \in \gamma^{\mathrm{SL}}(d_j^1)$. Then, using the "segment summaries"condition of $s^1 \sqsubseteq^{\mathrm{Seg}} s^2$, we know that $d_j^1 \sqsubseteq^{\mathrm{SL}} d_i^2$. Hence, using the monotony of γ^{SL} with respect to $\sqsubseteq^{\mathrm{SL}}$, we can deduce $\rho[l \mapsto k][v \mapsto t[k]] \in \gamma^{\mathrm{SL}}(d_i^2)$, 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

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}. □

6.6 Comparison With the CCL Domain For Arrays

We have explained how our definition of segmentation pre-order differs from the version in CCL. 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 CCL. Then, we handle

arrays that can contain values of algebraic data types—as opposed to scalar values only in CCL.

In this section, we discuss the main three 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 CCL.
- Then, we describe the three types of assignments that we need to distinguish—whereas CCL distinguish only two types of assignments.
- Finally, we illustrate how we transfer information between segment summaries and the TanL component of the Tana 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 CCL's. The operators are described in details in appendix § A.

Differences in Segmentations and Segmentation Concretisation

In CCL, 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 28), 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 28), 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 CCL 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 CCL: array updates, and scalar updates. We handle these two kinds of assignments similarly as CCL 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 a numeric variable. Like in CCL, 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 $y = e$.

Assignment to an array variable. In CCL, when updating an array variable, the only segmentation that changes is the one for that variable, and the other segmentations remain unaffected. In our analysis this is no longer true. Array lengths may appear in the boundsets and segment summaries of other segmentations. Hence assignment to an array variable x behaves in part as an assignment to the array length variable $|x|$ and can introduce changes in every segmentation.

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 and are not extended variables.

Conversion: Transfer Function for Array Creation

When converting an element of the TanL 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 35 (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* $(t, a) \in \text{Tana}$*, the transfer function for array creation is defined by*

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

This is a simplified version of the definition. The complete version can be found in Definition 54 in appendix §A. The operation $\text{Assign}^{\text{A}}(|y| = e_1)(a)$, that updates all the boundsets in which $|y|$ might appear, is an instance of a non-array assignment defined in appendix §A.2.2. Indeed, the array length variable $|y|$ is a numeric variable and e_1 is a numeric expression.

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 TanL. 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.

The complete 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 $\bot^{\text{Tana}}$, 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 35.

Conversion: Transfer Function for Array Access at Non-Bound Expressions

When converting a segmentation summary back into an element of the TanL 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 TanL 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 36 (Transfer function for an array access at a non-bound expression)**.** *For any abstract value $(t, a) \in \text{Tana}$ and any array access instruction $y := x[e]$ where e is not a bound expression, the abstraction for array access is given by*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Assign}^{\text{Tana}}(y := x[e])(t, a) = (t', a')} \\
\quad \textit{where} & a' = \text{Assign}^{\text{A}}(y := x[e])(a) & \\
\quad \textit{and} & t' = \text{Rem}^{\text{TanL}}_{\{l,v\}}\left(\bigsqcup_{t'' \in d}^{\text{TanL}} t''\right) & \\
\quad \textit{and} & d = \bigsqcup_{i \in I}^{\text{SL}} \text{Assign}^{\text{SL}}(y := v)\left(\text{Cond}^{\text{SL}}(l = e \wedge 0 \leq e)(d_i)\right) & \\
\quad \textit{and} & I = \left\{i \in \{1, \ldots, n\} \mid \exists e_1 \in b_{i-1}, \exists e_2 \in b_i, \text{CanSat}^{\text{TanL}}(t, e_1 \leq e < e_2)\right\} & \\
\quad \textit{and} & \left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\ldots,n\}}\right) = a(x) &
\end{array}
$$

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 TanL domain. Then we remove the special variables l and v, and we get t', that we use as new TanL component in the Tana domain. As explained earlier at page 45, 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 Assign^{A} operator.

6.7 Soundness Theorem For the Array Domain

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

Theorem 7 (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 Tana 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 5. The fact that γ^{Seg} is monotonic with respect to segmentation inclusion is the object of Lemma 6. 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 (Lemma 10). For example, for 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

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

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 CCL. □

6.8 Result of Analysis on The Motivating Example

Appendix § B details the execution of our analysis on the `find_max_priority` from Fig. 8. This analysis was executed by hand, as it is not yet implemented. The result of this analysis is the abstract value of Fig. 9. 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 output 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 (here the variable containing the future output), in addition to referring to the array's index and value.

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.

6.9 Conclusive Remarks on Array Analysis

We have described in this section 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 CCL [12]. Unlike CCL, we allow arrays to contain values from algebraic types, we allow segment summaries to refer to arbitrary program variables, and we have changed the definition of segmentation pre-order, 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.

7 Implementation, Experimental Results and Complexity

We have implemented the analysis of section § 5 in approximately 5000 lines of OCaml. Our prototype computes input-output summaries for non-recursive monomorphic fist-order functions manipulating non-recursive algebraic data types. However, this implementation does not support arrays. The source code of our implementation together with instructions on how to add new test cases and run the existing tests cases is packaged and published as a virtual machine artefact [13]. Similarly to our formal development, our analyser is parametrised by an abstract domain for integers. A command-line option allows to choose among the numeric domains of intervals, octagons or polyhedra, provided by Apron [11].

We have tested our analyser on a total of 43 programs (summarised on Table 1). The "extended variables" column of Table 1 gives, for each test file, the maximum number of extended variables in the summaries computed for each function in the file. This number is usually higher than what a user might see when looking at the results ([13]) because when printing the results the analyser uses a concise version of the summary, where some redundant extended variables are elided (for example, the cc domain is prefix-closed, but not all prefixes are shown in the concise version). Our test cases comprise some complex examples: some sorting algorithms, the `do_ticks` function from section § 2.3, and 6 functions inspired from the abstract specification of the seL4 micro-kernel [30]. We now review the results that our analyser computed for these examples, using polyhedra as numeric domain.

Sorting integer arrays

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. With this encoding, we wrote two sorting algorithms for arrays of integers, for arrays of size 5. For one of the sorting algorithms (selection sort), our analyser was able to infer that the result is sorted. For the second sorting algorithm (bubble sort), our analyser was not able to infer that the result is sorted. It was however able to infer a weaker property: that the sum of the elements before and after the sort remains unchanged. In both cases our analyser does not manage to infer that the multi-set of elements before and after the sort remains unchanged. Our abstract domains cannot express such properties. Some domains have been designed to express properties over multi-sets, like the domain introduced by Cezara Drăgoi during her PhD [42].

The `do_ticks` function

The `do_ticks` function (section § 2.3) is inspired from a process scheduler from operating system code. As reported in section § 5.1, 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 microkernel [30] 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, which might select a new current thread. The case constraints of our abstract domain cannot distinguish whether the current thread remains the same or not, so a join of those two cases is performed. This results in some expected information loss on the thread's time.

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 involves 324 cases in the disjunction — but each one of its cases is easy to understand. An improvement that could be imagined is to have a built-in boolean type that is internally encoded as an integer during the analysis — so that the result is concise — then translated back in terms of truth value and logic formulas — so that it is easy to interpret by a human reading the results.

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 [43–46], 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 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 the different leaves, probably using pattern-matching inside a loop or a function call.

Table 1: Test cases used for experimental evaluation. We use the ∗ symbol for families of similar tests, whose names start identically. The columns indicate the number of **lines of code** in the test's source code, the maximum number of **variables** per function in the source code, the **execution time** of the analysis, the maximum number of **cases** per function summary and the maximum number of **extended variables** in the verbose version of each 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 accompanying artefact [13] includes instructions to reproduce the results.

Name	Lines of code	Variables	Execution Time	Cases	Extended Variables
Hand-crafted tests:					
`do_ticks`	48	3	166 ms	3	19
`nondeterministic_bubble_sort`	167	10	**2.1 s**	5	19
`selection_sort`	137	11	**28.9 s**	25	24
Inspired from SeL4:					
`decode_set_priority`	45	6	10 ms	2	10
`mask_cap_boolean`	266	16	**7.4 s**	324	112
`mask_cap_int`	284	16	**1.5 s**	4	47
`timer_tick_scheduling`	179	8	**41.2 s**	81	52
Simple tests:					
`assert*`	22	2	1 ms	1	4
`call_inside_loop_*`	24	3	15 ms	1	3
`drift`	26	2	24 ms	2	6
`exchange`	6	3	2 ms	1	6
`facto*`	20	2	8 ms	1	4
`false_type_collision`	19	3	3 ms	1	3
`fibonacci`	24	3	51 ms	1	7
`gauss*`	16	2	15 ms	1	7
`ghost_equality`	11	3	< 1 ms	1	6
`hidden_incompat`	12	4	2 ms	0	0
`id`	8	2	< 1 ms	1	4
`if`	9	2	2 ms	1	4
`incompat`	9	2	< 1 ms	0	0
`indirect_swap`	19	1	3 ms	2	6
`long_id`	20	1	5 ms	2	6
`modulo`	32	4	33 ms	2	7
`multiplication_larger`	5	6	2 ms	1	6
`or_constructor`	7	1	< 1 ms	0	0
`plus_*`	18	1	< 1 ms	1	4
`record_assignment*`	25	2	2 ms	1	11
`reduction`	10	2	3 ms	1	10
`struct_exchange`	11	3	< 1 ms	1	6
`swap`	14	1	< 1 ms	2	6
`test_loop`	3	1	3 ms	1	2
`two_by_two`	11	1	3 ms	1	2
`while_true`	1	0	< 1 ms	0	0
`widening_convergence`	34	5	49 ms	1	8
`xor`	39	4	8 ms	3	9

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.

8 Related Work

In seminal work on the static analysis of recursive procedures, Cousot and Cousot [47] define a collecting semantics for an imperative language with recursive procedures and no global state. This semantics computes the set of output states—now deemed standard—for *almost* all the instructions of their language. There are two exceptions: Procedure calls and *while* loops. In these two cases only, the semantics computes a set of states that involve the input variables and the output variables, *i.e.*, a *relation* between inputs and outputs. We presume that procedures and loops are treated in a similar manner because the authors wanted to highlight that loops could be given the semantics of ordinary recursive functions.

The idea of exploiting an input-output relational semantics also appears in Kozen's work on the verification of `while` programs [48]. In this work, a semantics of relations is employed for *every* instruction of the language. Kozen introduced Kleene Algebra with Tests—an extension of relation algebra [49] 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 [29] used abstract interpretation for designing modular and relational analyses, and argue that compositionality can improve the scalability of analysers. Compositional Recurrence Analysis (CRA), by Farzan and Kincaid [21], 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 18. 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 19, 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, Bouroujeni and Reps [22] — 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 § 5.3. 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 § 5.3) with CRA or ICRA. In contrast to CRA and ICRA, our analysis does not yet support recursively defined functions.

The same approach of computing context-insensitive function summaries was followed in the context of correlation analysis, by Andreescu et al. [10]. This analysis infers binary equalities between the parts of structured inputs and outputs of programs, using the *correlation abstract domain*. We improve on that work because we can also express numeric relations between parts of structured values, and n-ary equalities. Our domain differs significantly from the correlation domain, in the sense that correlations are recursively defined so that parts of abstract values relate parts of structured values, whereas our domain is not a recursive structure, and instead exploits *extended variables* to relate the parts of structured values that are accessible through projection paths.

A preliminary version of our abstract numerical domains over extended variables was used to define an analysis which was not input-output relational [15]. This analysis inferred approximations of the final states, as opposed to the relations between input and output states that the current paper is dealing with. Moreover, our previous work did not include the domain for structural equalities, and was thus unable to express concisely n-ary equality relations between parts of structured values. Finally, no implementation and experimental evaluation was provided. Our implementation effort helped identify several precision issues in our previous approach, that motivated the addition of the structural equality domain (section § 3.5) and of the relational lifting (section § 4.2).

Several relational analyses were developed for the inter-procedural analysis of numeric programs [24, 50–52], and in the context of inter-procedural shape analysis [53–55]. They all feature a form of function summary, that helps reduce the analysis cost of large programs, by enabling modular analyses. A domain that supports both shape abstraction and numeric constraints was developed by Chang and Rival [56]. It is defined in a modular fashion, based on the cofibered abstract domain of Venet [57]. As in our construction, theirs also features a disjunctive completion, but leaves open the question of how to keep the number of disjuncts under control.

In the context of the static analysis of languages with algebraic data types, techniques based on tree automata [3] have been developed. Tree automata are well suited to represent regular sets of trees, and several works propose to extend their expressive power further. Lattice tree automata by Genet et al. [7, 58] augment tree automata with elements of an arbitrary abstract domain at their leaves, and allow to express *non-relational* integer constraints on the leaves of trees. More recently, Journault *et al.* [8, 9] use a combination of tree automata and of a relational domain whose keys are regular expressions to express relational constraints between the numeric leaves of trees. They use regular expressions to denote sets of access paths within those trees, and thus to support structures of unbounded heights.

Valnet, Monat and Miné [59] have developed another 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. 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 their approach to ours, since these aspects are not yet part of their implementation.

As a particular case of algebraic values, the analysis of programs with *optional* numerical values was handled by Liu and Rival [60], by associating to optional variables two *avatars*, that respectively model lower- and upper-constraints on that variable. When the avatars of some variable x induce a contradictory constraint, this denotes that x is in the None case. It is unclear how this approach generalises to deeply nested algebraic values.

Controlling the number of disjuncts in a disjunctive completion is admittedly difficult, as a cost *vs* precision balance must be found. Since we deal with finite types only, our number of disjuncts is bounded by the product of the sizes of types used in a program. Other works have used *silhouettes* (Li et al. [61]) —abstractions of the shapes of the abstract values— to control disjunctions. Following Kim, Rival and Ryu [62], our disjunctions, that are guided by paths in values, can be understood as a form of *control sensitivity*. It is worth noticing that our disjuncts do not form a *partition* since some disjuncts may overlap—a degree of freedom that is advocated by [62]. Based on our present work, we will investigate whether we can re-cast our disjuncts as *conjunctions of implications*, which could both improve precision and lead to a more parsimonious representations of abstract values.

Our Rand and Diorana domains could be seen as being both a *covering* in the sense of Rival and Mauborgne [63] and a disjunctive completion. A covering because our different cases allow to group and distinguish possible executions by some properties of the input states, by specifying extended variables that must be well-defined. Some overlapping between our cases is possible, hence it is a covering and not a partitioning. A disjunctive completion because our cases also group and distinguish possible executions according to properties of the *output* states. However, we prefer to see our domains as a disjunction over input-output relations, instead of seeing them as both a covering and a disjunctive completion.

When it comes to the automatic analysis of programs that manipulate arrays (in particular to deduce properties on the contents of arrays) several approaches have been explored. Some works look at the problem from a logics perspective. The approach of Bradley, Manna and Sipma [33] consists in restricting to a particular fragment of array theory and translating it to quantifier-free formulas on the theories of uninterpreted functions and Presburger arithmetic. Other approaches use Büchi counter automata as models (Habermehl, Iosif and Vojnar [35]), or combine Counter-Example Guided Abstraction Refinement with the deduction of Craig Interpolants from proofs of unreachability of certain program paths (Jhala and McMillan [34]).

In order to combine the work that we have already done on abstract interpretation for algebraic types with the analysis of arrays, we examined existing approaches that use *abstract interpretation* to analyse array contents. Two extremes are *array smashing*—used by Blanchet et al. [31]—that uses a single abstract value for each array, and *array expansion* [31], that 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 [32]. In order to choose the way of cutting arrays into segments, they first determine n partition functions (given either by heuristics or by the user), and then they consider a disjunction of up to 3^n different ways to partition arrays. This can be very costly. A later work by Halbwachs and Péron [36], inspired by [32], uses a conjunction of implications instead of a disjunction: possible indices are partitioned by a family of predicates $(\phi_p)_{p\in P}$, and for each predicate ϕ_p, a predicate on arrays' contents ψ_p must hold. Unlike [32], [36] allows for segments to be empty, in which case the corresponding implication vacuously holds. A salient point of [36] is the ability to express relations between segments of different arrays. Cousot, Cousot and Logozzo [12] improve on the scalability of [32] and [36], by intertwining the determination of which segments to consider with the analysis of the segments' contents. We have presented this approach in section § 6, as it forms the basis for extending our abstract domain for algebraic values with arrays. Later work by Fulara [38] presents a more general framework that encompasses dictionaries, in addition to arrays. Liu and Rival [39] 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. [40] have looked at array segments specially tailored for induction loops.

Other work by Dietsch et al. [41] has looked at the way different arrays relate to each other, without abstracting the contents of each array. Their approach works on predicates that express the equality of arrays, except for the indices that satisfy some (automatically established) predicates.

9 Conclusion and Future Work

In the context of programs that combine arithmetic operations, algebraic data types and functional arrays, we have shown how to construct an abstract domain that extends *any* abstract domain for numeric relations into an abstract domain for relations between numeric, algebraic and array values.

For ADTs, the main idea is to consider extended variables—*i.e.*, a variable prefixed by an access path—as the entities that are related in the numeric abstract domain. To reduce the size of abstract values, we add a domain that keeps track of equalities between non-numeric values. The domains are combined using a reduced product that propagates equalities. Additional expressiveness and precision is obtained using an adaptation of disjunctive completion for handling the different, incompatible cases that

an algebraic value can exhibit. This abstract domain is called RAND—the Relational Algebraic Numeric Domain.

We have given a formal justification 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 state into an analysis that computes a relation between input and output. 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 evaluated the feasibility of our abstract domain construction by designing and implementing [13] 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. 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. To support this extension, we designed Diorana: the *Domain for Input-Output Relations on Algebraic types, Numbers and Arrays*. This domain is based on the notion of array segmentation [12], and enables the analysis of programs that manipulate arrays whose cells may contain values of algebraic data types.

An obvious next challenge to address is to handle *recursive* algebraic data types. To that end, we will need to adapt our language of *paths*, *e.g.*, by using regular languages, or by extending techniques based on tree automata [8]. Another direction of research is to analyse recursive programs, which will require the computation of a fixpoint at the level of function summaries for groups of mutually defined functions.

The support of polymorphism seems mostly orthogonal to the work that we have done so far for algebraic data types and arrays. In the context of parametric polymorphism, our projections paths would not be able to look inside the values whose type is a type parameter. Indeed, knowing which paths make sense for a given type requires to know the type's definition. However, the domain of structural equalities would prove useful, since tracking equalities between two extended variables does not require to know their precise type.

In order to handle higher-order functions, an interesting direction of research would be to investigate how our work combines with the stable relations framework from Montagu and Jensen [64].

The overall practical purpose of our domain construction is to apply our analyser to help the verification of programs, by mixing automatic techniques based on abstract interpretation with standard deductive verification tools, such as Why3 [65]. Previous work on correlation analysis [10] has demonstrated that a large number of proof obligations could be discharged automatically in such a way, thereby alleviating the manual effort in the verification of large programs. The combination of algebraic data types and (functional) arrays paves the way for applying such automatic techniques to assist formal development projects of safety-critical software such as the formally verified operating system seL4 [30].

Declarations

Competing interests

None of the authors has any financial nor personal competing interest with the subject or content of this research.

Authors' contributions

All authors wrote the main manuscript.

Availability of data and materials

The implementation of the RAND domain, together with the test cases of section § 7 are available as a virtual machine artefact [13].

References

[1] Pierce, B.C.: Types and Programming Languages. The MIT Press, Cambridge, Massachusetts (2002)

[2] Pierce, B.: Advanced Topics in Types and Programming Languages. The MIT Press, Cambridge, Massachusetts (2005)

[3] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications, (2008). https://hal.inria.fr/hal-03367725

[4] Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: POPL (2010). https://doi.org/10.1145/1706299.1706355

[5] Ong, C.-L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: POPL (2011). https://doi.org/10.1145/1926385.1926453

[6] Haudebourg, T., Genet, T., Jensen, T.: Regular language type inference with term rewriting. In: ICFP (2020). https://doi.org/10.1145/3408994

[7] Genet, T., Le Gall, T., Legay, A., Murat, V.: A completion algorithm for lattice tree automata. In: CIAA (2013). https://doi.org/10.1007/978-3-642-39274-0_13

[8] Journault, M., Miné, A., Ouadjaout, A.: An abstract domain for trees with numeric relations. In: ESOP (2019). https://doi.org/10.1007/978-3-030-17184-1_26

[9] Journault, M.: Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference. PhD thesis, Sorbonne University, France (2019). https://tel.archives-ouvertes.fr/tel-02947214

[10] Andreescu, O.F., Jensen, T., Lescuyer, S., Montagu, B.: Inferring frame conditions with static correlation analysis. In: POPL (2019). https://doi.org/10.1145/3290360

[11] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV (2009). https://doi.org/10.1007/978-3-642-02658-4_52

[12] Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL (2011). https://doi.org/10.1145/1925844.1926399

[13] Bautista, S., Jensen, T., Montagu, B.: Artifact for the "Lifting Numeric Relational Domains to Algebraic Data Types" article of the SAS 2022 symposium. Zenodo (2022). https://doi.org/10.5281/zenodo.6977156

[14] Bautista, S., Jensen, T., Montagu, B.: Lifting numeric relational domains to algebraic data types. In: SAS (2022). https://doi.org/10.1007/978-3-031-22308-2_6

[15] Bautista, S., Jensen, T., Montagu, B.: Numeric domains meet algebraic data types. In: NSAD (2020). https://doi.org/10.1145/3427762.3430178

[16] Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Found. Trends Program. Lang. **4**(3-4), 120–372 (2017) https://doi.org/10.1561/2500000034

[17] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977). https://doi.org/10.1145/512950.512973

[18] Cousot, P.: Principles of Abstract Interpretation, p. 832. The MIT Press, Cambridge, Massachusetts (2021)

[19] Bautista, S., Jensen, T., Montagu, B.: Lifting Numeric Relational Domains to Algebraic Data Types (extended version) (2022). https://hal.inria.fr/hal-03765357

[20] Lemerre, M.: Ssa translation is an abstract interpretation. Proceedings of the ACM on Programming Languages **7**(POPL), 1895–1924 (2023) https://doi.org/10.1145/3571258

[21] Farzan, A., Kincaid, Z.: Compositional Recurrence Analysis. In: FMCAD (2015). https://doi.org/10.1109/FMCAD.2015.7542253

[22] Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.: Compositional recurrence analysis revisited. In: PLDI (2017). https://doi.org/10.1145/3062341.3062373

[23] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation (extended abstract). In: MFPS (1997). https://doi.org/10.1016/s1571-0661(05)80168-9

[24] Boutonnet, R., Halbwachs, N.: Disjunctive relational abstract interpretation for interprocedural program analysis. In: VMCAI. Lecture note in Computer Science (2019). https://doi.org/10.1007/978-3-030-11245-5_7

[25] Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. In: NASA Formal Methods (2017). https://doi.org/10.1007/978-3-319-57288-8_15

[26] Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract domains of affine relations. ACM Transactions on Programming Languages and Systems **36**(4), 1–73 (2014) https://doi.org/10.1145/2651361

[27] Delmas, D., Miné, A.: Analysis of Software Patches Using Numerical Abstract Interpretation, pp. 225–246. Springer, ??? (2019). https://doi.org/10.1007/978-3-030-32304-2_12

[28] Rival, X., Yi, K.: Introduction to Static Analysis: an Abstract Interpretation Perspective. The MIT Press, Cambridge, Massachusetts (2020)

[29] Cousot, P., Cousot, R.: Modular static program analysis. In: CC (2002). https://doi.org/10.1007/3-540-45937-5_13

[30] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP (2009). https://doi.org/10.1145/1629575.1629596

[31] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003). https://doi.org/10.1145/780822.781153

[32] Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL (2005). https://doi.org/10.1145/1040305.1040333

[33] Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI (2006). https://doi.org/10.1007/11609773_28

[34] Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: CAV (2007). https://doi.org/10.1007/978-3-540-73368-3_23

[35] Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: FOSSACS (2008). https://doi.org/10.1007/978-3-540-78499-9_33

[36] Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI (2008). https://doi.org/10.1145/1379022.1375623

[37] Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL (2008). https://doi.org/10.1145/1328438.1328468

[38] Fulara, J.: Generic abstraction of dictionaries and arrays. Electronic Notes in Theoretical Computer Science **287**, 53–64 (2012) https://doi.org/10.1016/j.entcs.2012.09.006

[39] Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: VMCAI, pp. 282–299 (2015). https://doi.org/10.1007/978-3-662-46081-8_16

[40] Li, B., Zhai, J., Tang, Z., Tang, E., Zhao, J.: A framework for array invariants synthesis in induction-loop programs. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC) (2017). https://doi.org/10.1109/apsec.2017.8

[41] Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: The map equality domain. In: VSTTE (2018). https://doi.org/10.1007/978-3-030-03592-1_17

[42] Drăgoi, C.: Automated verification of heap-manipulating programs with infinite data. PhD thesis, Université Paris Diderot - Paris 7 (2011)

[43] Dimovski, A.S.: Lifted static analysis using a binary decision diagram abstract domain. In: GPCE (2019). https://doi.org/10.1145/3357765.3359518

[44] Dimovski, A.S., Apel, S., Legay, A.: Several lifted abstract domains for static analysis of numerical program families. Science of Computer Programming **213** (2022) https://doi.org/10.1016/j.scico.2021.102725

[45] Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. In: SAS (2011). https://doi.org/10.1007/978-3-642-23702-7_19

[46] Jeannet, B.: The BDDAPRON logico-numerical abstract domains library. https://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/ (2009)

[47] Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Neuhold, E.J. (ed.) Formal Description of Programming Concepts: Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, pp. 237–278. North-Holland, St. Andrews, NB, Canada (1977)

[48] Kozen, D.: Kleene Algebra with Tests. In: TOPLAS (1997). https://doi.org/10.1145/256167.256195

[49] Tarski, A.: On the calculus of relations. Journal of Symbolic Logic **6** (1941) https://doi.org/10.2307/2268577

[50] Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. In: SAS (2004). https://doi.org/10.1007/978-3-540-27864-1_19

[51] Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: TOPLAS (2007).

https://doi.org/10.1145/1275497.1275504

[52] Sharma, T., Reps, T.W.: A new abstraction framework for affine transformers. Form. Methods in Syst. Des. **54** (2019) https://doi.org/10.1007/s10703-018-0325-z

[53] Sotin, P., Jeannet, B.: Precise interprocedural analysis in the presence of pointers to the stack. In: ESOP (2011). https://doi.org/10.1007/978-3-642-19718-5_24

[54] Jeannet, B.: Relational interprocedural verification of concurrent programs. Softw. Syst. Model. **12**(2), 285–306 (2013) https://doi.org/10.1007/s10270-012-0230-7

[55] Illous, H., Lemerre, M., Rival, X.: Interprocedural shape analysis using separation logic-based transformer summaries. In: SAS (2020). https://doi.org/10.1007/978-3-030-65474-0_12

[56] Chang, B.-Y.E., Rival, X.: Modular construction of shape-numeric analyzers. Festschrift for Dave Schmidt (2013)

[57] Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: SAS (1996). https://doi.org/10.1007/3-540-61739-6_53

[58] Genet, T., Le Gall, T., Legay, A., Murat, V.: Tree regular model checking for lattice-based automata. Technical report, Inria (2012). https://inria.hal.science/hal-00687310/

[59] Valnet, M., Monat, R., Miné, A.: 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). https://inria.hal.science/hal-03936718

[60] Liu, J., Rival, X.: Abstraction of optional numerical values. In: APLAS (2015). https://doi.org/10.1007/978-3-319-26529-2_9

[61] Li, H., Berenger, F., Evan Chang, B., Rival, X.: Semantic-directed clumping of disjunctive abstract states. In: POPL (2017). https://doi.org/10.1145/3009837.3009881

[62] Kim, S., Rival, X., Ryu, S.: A theoretical foundation of sensitivity in an abstract interpretation framework. In: TOPLAS (2018). https://doi.org/10.1145/3230624

[63] Rival, X., Mauborgne, L.: The trace partitioning abstract domain. TOPLAS **29**(5), 26 (2007) https://doi.org/10.1145/1275497.1275501

[64] Montagu, B., Jensen, T.: Stable Relations and Abstract Interpretation of Higher-Order Programs. In: ICFP. Association for Computing Machinery, New York, NY, USA (2020)

[65] Filliâtre, J.-C., Paskevich, A.: Why3 — Where programs meet provers. In: ESOP (2013). https://doi.org/10.1007/978-3-642-37036-6_8

Appendices

A Abstract Operators and Transfer Functions for Array Analysis

A.1 Operators of the Abstract Domains

In this appendix, we give technical details on the operators of the segmentation domain (appendix §A.1.1), the array domain (appendix §A.1.2), the Tana product domain (appendix §A.1.3) and Dana disjunctive completion (appendix §A.1.4).

A.1.1 Segmentations

Inclusion test

We provide a detailed proof of reflexivity and transitivity of the inclusion test between segmentations.

Proof of Lemma 5 ($\sqsubseteq^{\text{Seg}}$ is a pre-order). In order to prove that the relation $\sqsubseteq^{\text{Seg}}$ is a pre-order, we need to prove that it is both reflexive and transitive.

Reflexivity.

Let $s = \big(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\big)$ be a segmentation. The goal here is to prove that $s \sqsubseteq^{\text{Seg}} s$. For that, we take the identity function over $\{0,\dots,n\}$ as the function ϕ that will allow us to check the conditions of Definition 34. The "first and last indexes" condition holds because $\phi(0) = 0$ and $\phi(n) = n$. The "boundset inclusion" condition holds because for any index $i \in \{0,\dots,n\}$, we have $b_i \subseteq b_i$. The "segment summaries" condition holds because, for any index $i \in \{1,\dots,n\}$, the integer set $\{\phi(i-1)+1,\dots,\phi(i)\}$ is the singleton $\{i\}$, and by reflexivity of $\sqsubseteq^{\text{SL}}$ we have $d_i \sqsubseteq^{\text{SL}} d_i$. The "emptiness markers" condition holds because, for any $i \in \{1,\dots,n\}$, the conjunction $\bigwedge_{j=\phi(i-1)+1}^{\phi(i)} m_j$ is m_i and we have indeed $m_i \Rightarrow m_i$.

Transitivity.

We consider three segmentations $s^1 = \big(l, v, b_0^1, (d_j^1, b_j^1, m_j^1)_{j\in\{1,\dots,n\}}\big)$, $s^2 = \big(l, v, b_0^2, (d_k^2, b_k^2, m_k^2)_{k\in\{1,\dots,m\}}\big)$ and $s^3 = \big(l, v, b_0^3, (d_i^3, b_i^3, m_i^3)_{i\in\{1,\dots,p\}}\big)$ such that $s^1 \sqsubseteq^{\text{Seg}} s^2$ and $s^2 \sqsubseteq^{\text{Seg}} s^3$. We want to prove that $s^1 \sqsubseteq^{\text{Seg}} s^3$. By hypothesis, we know that there exists two functions $\phi^{2\to1} : \{0,\dots,m\} \to \{0,\dots,n\}$ and $\phi^{3\to2} : \{0,\dots,p\} \to \{0,\dots,m\}$ that satisfy the conditions of Definition 34. We now prove that $s^1 \sqsubseteq^{\text{Seg}} s^2$ by taking $\phi = \phi^{2\to1} \circ \phi^{3\to2}$ and by verifying the four conditions of Definition 34.

Condition 1: first and last indices.

We need to prove that $\phi(0) = 0$ and $\phi(p) = n$. By the "first and last indices" condition of $s^1 \sqsubseteq^{\text{Seg}} s^2$, we have both $\phi^{2\to1}(0) = 0$ and $\phi^{2\to1}(m) = n$. Additionally, by the "first and last indices" condition of $s^2 \sqsubseteq^{\text{Seg}} s^3$, we have both $\phi^{3\to2}(0) = 0$ and $\phi^{3\to2}(p) = m$. Hence $\phi(0) = \phi^{2\to1}(\phi^{3\to2}(0)) = \phi^{2\to1}(0) = 0$ and $\phi(p) = \phi^{2\to1}(\phi^{3\to2}(p)) = \phi^{2\to1}(m) = n$.

Condition 2: boundset inclusion.

We want to prove that for any index $i \in \{0, \ldots, p\}$ we have $b_i^3 \subseteq b_{\phi(i)}^1$. Let $i \in \{0, \ldots, p\}$. By the "boundset inclusion" condition of $s^3 \sqsubseteq^{\text{Seg}} s^2$ applied at index i, we know that $b_i^3 \subseteq b_{\phi^{3\to 2}(i)}^2$. Besides, by the "boundset inclusion" condition of $s^2 \sqsubseteq^{\text{Seg}} s^1$ applied at index $\phi^{3\to 2}(i)$, we have $b_{\phi^{3\to 2}(i)}^2 \subseteq b_{\phi^{2\to 1}(\phi^{3\to 2}(i))}^1$. By the definition of ϕ, we have $b_{\phi^{2\to 1}(\phi^{3\to 2}(i))}^1 = b_{\phi(i)}^1$; hence $b_i^3 \subseteq b_{\phi^{3\to 2}(i)}^2 \subseteq b_{\phi(i)}^1$.

Condition 3: segment summaries.

The goal here is to prove that for any index $i \in \{1, \ldots, p\}$ and any index j such that $\phi(i-1) < j \leq \phi(i)$, we have $d_j^1 \sqsubseteq^{\text{SL}} d_i^3$. Let $i \in \{1, \ldots, p\}$ be an index of the segmentation s^3 and j with $\phi(i-1) < j \leq \phi(i)$ be one of the corresponding indices of the segmentation s^1 (if no such j exists, the property that we are trying to prove is vacuously true). Let's consider the sequence of integer intervals

$$\left(\left\{\phi^{2\to 1}(k-1)+1, \ldots, \phi^{2\to 1}(k)\right\}\right)_{\phi^{3\to 2}(i-1)<k\leq\phi^{3\to 2}(i)}$$

These integer intervals are contiguous. Their left-most bound is

$$\phi^{2\to 1}\left(\phi^{3\to 2}(i-1)+1-1\right)+1 = \phi(i-1)+1$$

while their right-most bound is $\phi(i)$. Hence, this sequence of integer intervals forms a partition of the integer interval $\{\phi(i-1)+1, \ldots, \phi(i)\}$. Therefore, since index j belongs to that integer interval, we know that there exists an index k such that $\phi^{3\to 2}(i-1) < k \leq \phi^{3\to 2}(i)$ and $\phi^{2\to 1}(k-1) < j \leq \phi^{2\to 1}(k)$. Then, by the "segment summaries" property of $s^1 \sqsubseteq^{\text{Seg}} s^2$ applied to index k, we have $d_j^1 \sqsubseteq^{\text{SL}} d_k^2$. Similarly, by the "segment summaries" property of $s^2 \sqsubseteq^{\text{Seg}} s^3$ applied at index i, we have $d_k^2 \sqsubseteq^{\text{SL}} d_i^3$. Therefore, using the transitivity of $\sqsubseteq^{\text{SL}}$, we deduce $d_j^1 \sqsubseteq^{\text{SL}} d_i^3$.

Condition 4: emptiness markers.

We want to prove that for any index $i \in \{1, \ldots, p\}$ the following implication holds :

$$\left(\bigwedge_{j=\phi(i-1)+1}^{\phi(i)} m_j^1\right) \Rightarrow m_i^3$$

Let $i \in \{1, \ldots, p\}$. Given the partition that we exhibited for the previous condition, we know that:

$$\left(\bigwedge_{j=\phi(i-1)+1}^{\phi(i)} m_j^1\right) = \bigwedge_{k=\phi^{3\to 2}(i-1)+1}^{\phi^{3\to 2}(i)} \left(\bigwedge_{j=\phi^{2\to 1}(k-1)+1}^{\phi^{2\to 1}(k)} m_j^1\right)$$

using the "emptiness markers" condition of $s^1 \sqsubseteq^{\text{Seg}} s^2$ for all the indices k such that $\phi^{3\to2}(i-1) < k \leq \phi^{3\to2}(i)$, we know that for every one of those indices k we have

$$\left(\bigwedge_{j=\phi^{2\to1}(k-1)+1}^{\phi^{2\to1}(k)} m_j^1\right) \Rightarrow m_k^2$$

Additionally, using the "emptiness markers" condition of $s^2 \sqsubseteq^{\text{Seg}} s^3$ at index i, we know that

$$\left(\bigwedge_{k=\phi^{3\to2}(i-1)+1}^{\phi^{3\to2}(i)} m_k^2\right) \Rightarrow m_i^3$$

Hence,

$$\left(\bigwedge_{j=\phi(i-1)+1}^{\phi(i)} m_j^1\right) \Rightarrow \left(\bigwedge_{k=\phi^{3\to2}(i-1)+1}^{\phi^{3\to2}(i)} m_k^2\right) \Rightarrow m_i^3$$

□

Unification of segmentations

In order to more easily define the two operations used during unification (boundset splitting and bound expression removal), we are going to introduce two notations on sequences: sequence slicing and sequence concatenation. If σ is a sequence, then $\sigma[i:j]$ is the sequence that contains the elements of σ between indices i and j. If σ_1 and σ_2 are two sequences, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2.

Definition 37 (Boundset splitting)**.** *For any segmentation $s = (l, v, b_0, \sigma)$, with a sequence of segments $\sigma = (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}$, any segmentation index $i_0 \in \{0,\dots,n\}$, any two non-empty boundsets b^1 and b^2 that form a partition of b_{i_0}, and any segment summary d_0 the* splitting of s at index i_0 on boundsets b^1 and b^2 with summary d_0 *is the segmentation defined by*

$$\text{Split}(s, i_0, b^1, b^2, d_0) = \begin{cases} (l, v, b^1, (d_0, b^2, \mathtt{tt}) \cdot \sigma) & \text{if } i_0 = 0 \\ (l, v, b_0, \sigma[1:i_0-1] \cdot \sigma' \cdot \sigma[i_0+1:n]) & \text{if } i_0 \neq 0 \\ \quad \text{where } \sigma' = (d_{i_0}, b^1, m_{i_0}) \cdot (d_0, b^2, \mathtt{tt}) & \end{cases}$$

Splitting a boundset amounts to relaxing an equality constraint, which explains why the result is less precise.

Lemma 8 (Boundset splitting yields a value less precise than the initial one)**.** *For any segmentation $s = \big(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\big)$, any segmentation index $i_0 \in \{0,\dots,n\}$, any two non-empty boundsets b^1 and b^2 that form a partition of b_{i_0}, and any segment summary d_0 such that $d_0 \sqsubseteq^{\text{SL}} d_{i_0}$, we have*

$$s \sqsubseteq^{\text{Seg}} \text{Split}(s, i_0, b^1, b^2, d_0)$$

When defining the removal of bound expressions from a segmentation, we proceed in two steps, by first defining how to remove them at a particular index, and then defining the same operation for all indices at once.

Definition 38 (Bound expression removal)**.** *For any segmentation $s = (l, v, b_0, \sigma)$, with a sequence of segments $\sigma = (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}$, any set of bound expressions b and any segmentation index $i_0 \in \{0, \dots, n\}$, the* removal of bound expressions b from segmentation s at index i_0 *is given by*

$$\text{RemBexpAtIndex}(b, s, i_0) = \begin{cases} (l, v, b_0 \setminus b, \sigma) & \text{if } i_0 = 0 \\ (l, v, b_0, \sigma[1 : i_0 - 1] \cdot (d_{i_0}, b_{i_0} \setminus b, m_{i_0}) \cdot \sigma[i_0 + 1 : n]) & \text{if } b_{i_0} \setminus b \neq \emptyset \\ (l, v, b_0, \sigma[1 : i_0 - 1] \cdot (d_{i_0} \sqcup^{\mathrm{S}} d_{i_0+1}, b_{i_0+1}, m_{i_0} \wedge m_{i_0+1}) \cdot \sigma[i_0 + 2 : n]) & \text{if } b_{i_0} \setminus b = \emptyset \end{cases}$$

Then, the removal of bound expressions b from segmentation s *is given by*

$$\text{RemoveBexp}(b, s) = \text{RemBexpAtIndex}\left(b, \dots \text{RemBexpAtIndex}(b, s, 0) \dots, n\right)$$

In order to keep well-formed segmentations, we do not allow removing the constant 0 from b_0, nor the length variable from b_n. This ensures that the cases where $b_{i_0} \setminus b = \emptyset$ with $i_0 = 0$ or $i_0 = n$ are not possible. Removing bound expressions amounts to relaxing equality and inequality constraints. Which is why the result is less precise.

Lemma 9 (Bound expression elimination yields a value less precise than the initial one)**.** *For any segmentation $s = \left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right)$ and any set of bound expressions b,*

$$s \sqsubseteq^{\mathrm{Seg}} \text{RemoveBexp}(b, s)$$

Here, segment-wise operations on unified segmentations are written with an overline:

Definition 39 (Segment-wise operations)**.** *For any two unified segmentations $s^1 = \left(l, v, b_0, (d_i^1, b_i, m_i^1)_{i\in\{1,\dots,n\}}\right)$ and $s^2 = \left(l, v, b_0, (d_i^2, b_i, m_i^2)_{i\in\{1,\dots,n\}}\right)$ union, intersection and widening are defined as follows:*

$$\begin{aligned} s^1 \mathbin{\overline{\sqcup^{\mathrm{seg}}}} s^2 &= \left(l, v, b_0, \left(d_i^1 \sqcup^{\mathrm{S}} d_i^2, b_i, m_i^1 \vee m_i^2\right)_{i\in\{1,\dots,n\}}\right) \\ s^1 \mathbin{\overline{\sqcap^{\mathrm{seg}}}} s^2 &= \left(l, v, b_0, \left(d_i^1 \sqcap^{\mathrm{S}} d_i^2, b_i, m_i^1 \wedge m_i^2\right)_{i\in\{1,\dots,n\}}\right) \\ s^1 \mathbin{\overline{\triangledown^{\mathrm{seg}}}} s^2 &= \left(l, v, b_0, \left(d_i^1 \triangledown^{\mathrm{S}} d_i^2, b_i, m_i^1 \vee m_i^2\right)_{i\in\{1,\dots,n\}}\right) \end{aligned}$$

Lemma 10 (Soundness of segment-wise operations)**.** *The segment-wise operations are sound on unified segmentations:*

- *If s_1 and s_2 are two unified segmentations, then*

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

- *If s_1 and s_2 are two unified segmentations, then*

$$\gamma^{\mathrm{Seg}}(s_1) \cap \gamma^{\mathrm{Seg}}(s_2) \subseteq \gamma^{\mathrm{Seg}}(s_1 \mathbin{\overline{\sqcap^{\mathrm{seg}}}} s_2)$$

- *The widening* $\overline{\nabla^{\text{seg}}}$ *computes upper-bounds on unified segmentations. If* s_1 *and* s_2 *are two unified segmentations, then*

$$s_1 \sqsubseteq^{\text{Seg}} s_1 \,\overline{\nabla^{\text{seg}}}\, s_2 \qquad\qquad s_2 \sqsubseteq^{\text{Seg}} s_1 \,\overline{\nabla^{\text{seg}}}\, s_2$$

- *Widening enforces convergence in finite time. If* $(s_i)_{i\in\mathbf{N}}$ *is a sequence of unified segmentations, then the sequence* $(s'_i)_{i\in\mathbf{N}}$ *defined by* $s'_0 = s_0$ *and, for all* $n \in \mathbf{N}$, $s'_{n+1} = s'_n \,\overline{\nabla^{\text{idisj}}}\, s_{n+1}$, *then there exists an index* $n_0 \in \mathbf{N}$ *such that, for any* $n \geq n_0$, *we have* $s'_n = s'_{n_0}$.

Proof that segment-wise union is sound on unified segmentations. Let $s^1 = \big(l, v, b^1_0, (d^1_i, b^1_i, m^1_i)_{i\in\{1,\dots,n\}}\big)$ and $s^2 = \big(l, v, b^2_0, (d^2_i, b^2_i, m^2_i)_{i\in\{1,\dots,n\}}\big)$ be two unified segmentations. Let $(\rho, t) \in \gamma^{\text{Seg}}(s^1) \cup \gamma^{\text{Seg}}(s^2)$ be an environment-array pair in the concretisation of either s^1 or s^2. The definition of $s^1 \,\overline{\sqcup^{\text{seg}}}\, s^2$ being symmetric on s^1 and s^2, we can assume without loss of generality that $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$. The goal is to prove that $(\rho, t) \in \gamma^{\text{Seg}}(s^1 \,\overline{\sqcup^{\text{seg}}}\, s^2)$. In order to prove that, we need to check the five conditions of Definition 32. Since the boundsets of $s^1 \,\overline{\sqcup^{\text{seg}}}\, s^2$ are exactly the same ones of s^1, the "Equalities in each boundset", "Inequalities between boundsets" and "Array size" conditions stem directly from $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$. The conditions that are left to verify are "Strict inequalities for non-empty segments" (which involves emptiness markers) and "Segment summaries" (which involves segment summaries). We write $s' = (l, v, b_0, (d'_i, b_i, m'_i)_{i\in\{1,\dots,n\}})$ for the segmentation $s^1 \,\overline{\sqcup^{\text{seg}}}\, s^2$.

Strict inequalities for non-empty segments. Let $i \in \{1, \dots, n\}$ be an index of segmentation s'. For this condition, we get to assume that m'_i is false, and we need to show that $[\![b_{i-1}]\!]^{\text{exp}}_\rho < [\![b_i]\!]^{\text{exp}}_\rho$. By definition, we have $m'_i = m^1_i \vee m^2_i$. Hence, since m'_i is false, we know that m^1_i and m^2_i are both false. By using the "Strict inequalities for non-empty segments" property from $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$ we have the desired inequality: $[\![b_{i-1}]\!]^{\text{exp}}_\rho < [\![b_i]\!]^{\text{exp}}_\rho$.

Segment summaries. Let $i \in \{1, \dots, n\}$ be a segment index and j be an array index for that segment (that is, such that $[\![b_{i-1}]\!]^{\text{exp}}_\rho \leq j < [\![b_i]\!]^{\text{exp}}_\rho$). The goal here is to prove that the environment ρ extended with value j for variable l and value $t[j]$ for variable v, belongs to the concretisation of $d'_i = d^1_i \sqcup^{\text{S}} d^2_i$. In other words, the goal is to prove $\rho[l \mapsto j][v \mapsto t[j]] \in \gamma^{\text{S}}(d'_i)$. Using the "Segment summaries" condition from $(\rho, t) \in \gamma^{\text{Seg}}(s^1)$, we have $\rho[l \mapsto j][v \mapsto t[j]] \in \gamma^{\text{Seg}}(d^1_i)$. Then, using the soundness of $\sqcup^{\text{S}}$, we have that $\gamma^{\text{S}}(d^1_i) \cup \gamma^{\text{S}}(d^2_i) \subseteq \gamma^{\text{S}}(d^1_i \sqcup^{\text{S}} d^2_i)$; hence in particular $\rho[l \mapsto j][v \mapsto t[j]] \in \gamma^{\text{S}}(d^1_i \sqcup^{\text{S}} d^2_i) = \gamma^{\text{S}}(d'_i)$. □

The proofs for intersection and widening use similar arguments.

For intersection, union and widening, we use the same unification algorithm as [12]. We refer the reader to section 11.4 of [12] for details.

A.1.2 Array environments

For a typing context Γ, we recall that $\text{Arr}(\Gamma)$ is the set of variables that have an array type.

The domain for abstracting arrays is defined by associating a segmentation to each variable that has an array type.

Definition 40 (Abstract array environments)**.** *For any numeric domain* N*, we define a domain for arrays, written* $\mathrm{A}(\mathrm{N})$*. For any typing context* Γ*, the elements* $a \in \mathrm{A}(\mathrm{N})(\Gamma) = \prod_{x \in \mathrm{Arr}(\Gamma)} \mathrm{Seg}(\mathrm{N})(\Gamma)(x)$ *of the domain for arrays are called* abstract array environments*, and they associate, to every variable* $x \in \mathrm{Arr}(\Gamma)$ *with an array type, a segmentation* $a(x) \in \mathrm{Seg}(\mathrm{N})(\Gamma)(x)$*.*

Definition 41 (Concretisation for abstract array environments)**.** *The concretisation for abstract array environments* γ^{A} *is defined by:*

$$\gamma^{\mathrm{A}}(a) = \left\{\rho \mid \forall x \in \mathrm{Arr}(\Gamma), (\rho, \rho(x)) \in \gamma^{\mathrm{Seg}}(a(x))\right\}$$

All operations on abstract array environments (pre-order, intersection, union and widening) are performed element-wise.

Definition 42 (Operators on abstract array environments)**.** *For two abstract array environments* a_1 *and* a_2 *on the same typing context* Γ*, we define the relation* $\sqsubseteq^{\mathrm{A}}$ *and the operators* $\sqcap^{\mathrm{A}}$*,* $\sqcup^{\mathrm{A}}$ *and* $\triangledown^{\mathrm{A}}$ *as follows:*

$$\begin{array}{rcl} a_1 \sqsubseteq^{\mathrm{A}} a_2 & \textit{iff} & \forall x \in \mathrm{Arr}(\Gamma), a_1(x) \sqsubseteq^{\mathrm{Seg}} a_2(x) \\ a_1 \sqcap^{\mathrm{A}} a_2 & = & \left[x \mapsto a_1(x) \sqcap^{\mathrm{Seg}} a_2(x)\right] \\ a_1 \sqcup^{\mathrm{A}} a_2 & = & \left[x \mapsto a_1(x) \sqcup^{\mathrm{Seg}} a_2(x)\right] \\ a_1 \triangledown^{\mathrm{A}} a_2 & = & \left[x \mapsto a_1(x) \triangledown^{\mathrm{Seg}} a_2(x)\right] \end{array}$$

Since they share the same typing context Γ, the two abstract array environments a_1 and a_2 always have the same domain: $\mathrm{Arr}(\Gamma)$. When defining functions, we use the notation

$$f = [x \mapsto \ldots]$$

to mean

$$\forall x \in \mathrm{dom}(f), f(x) = \ldots$$

Lemma 11 (Soundness of the array domain)**.** *The operations on the array domain are sound:*

- $\sqsubseteq^{\mathrm{A}}$ *is a pre-order.*
- *The concretisation* γ^{A} *is monotonic with respect to the pre-order* $\sqsubseteq^{\mathrm{A}}$*.*
- *Abstract union and abstract intersection are sound over-approximations of their concrete counter-parts.*
- *Widening computes upper bounds and enforces convergence.*

A.1.3 The Tana Product Domain

As explained in Fig. 10, we take a product domain of $\mathrm{TanL}(\mathrm{N})$ and $\mathrm{A}(\mathrm{N})$. We call this product domain $\mathrm{Tana}(\mathrm{N})$, for *Tuple for Algebraic types, Numbers and Arrays*. We do not define a reduction operator for this product domain. When not explicitly defined in this appendix, operators of the product domain are defined component-wise.

A.1.4 Disjunctive completion

In this appendix, we give the detailed definitions for the Dana domain, which is obtained from the Tana 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 Tan to S (section §3.6.2).

Abstract inclusion is given by a Hoare order:

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

Abstract intersection for the disjunctive completion is given by:

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

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 previous appendix (Definition 16); but this time we extend it for the quadruplets of the Tana domain:

$$(c_1, e_1, p_1, a_1) \equiv_{\text{E}}^{\text{Tana}} (c_2, e_2, p_2, a_2) \text{ iff } c_1 \equiv c_2$$

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

Definition 43. *We define an operator* $\text{Collapse}^{\text{Dana}}$ *that takes a set of elements of* Tana *and merges together (by taking the abstract union), the elements that are equivalent with respect to* $\equiv_{\text{E}}^{\text{Tana}}$*. Formally,*

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

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

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

A.2 Analysing arrays: transfer functions

In this appendix, we give the transfer functions for Conditions (appendix §A.2.1) and Assignments (appendix §A.2.2) for the domain of segmentations Seg and the domains built on top of it.

A.2.1 Abstraction for conditions

When analysing a condition (whether it comes from a while loop guard or an assertion), there are up to three things that change in segmentations:

- The segment summaries are modified to integrate the new condition. Here, to keep definitions simple, nothing happens if a segment summary becomes $\bot^{\mathrm{S}}$. We could however, when a segment summary becomes $\bot^{\mathrm{S}}$, have a more precise treatment by looking at the emptiness marker of the segment. If a segment that is not allowed to be empty has a $\bot^{\mathrm{S}}$ summary, then the whole segmentation can be turned into $\bot^{\mathrm{Seg}}$. If a segment that is allowed to be empty has a $\bot^{\mathrm{S}}$, then the segment is indeed empty in the concrete, and it can be removed, by merging the boundsets to its left and right.
- Two boundsets may be merged, if the condition implies that they are equal. If this contradicts the (lack of) emptiness markers between the two boundsets, then the whole segmentation becomes $\bot^{\mathrm{Seg}}$.
- An emptiness marker may become false, if the condition implies that the associated segment is necessarily non-empty.

We are going to formalize each one of those effects separately, before defining abstraction of conditions for segmentations as a whole.

Definition 44 (Abstraction of conditions for segmentation summaries)**.** *For any condition c, any array variable x and any segmentation s for variable x given by $s = \left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right) \in \mathrm{Seg}(\mathrm{N})(\Gamma)(x)$, we define*

$$\mathrm{CondSum}(c)(s) = \left(l, v, b_0, \left(\mathrm{Cond}^{\mathrm{SL}}(c)(d_i), b_i, m_i\right)_{i\in\{1,\dots,n\}}\right)$$

Before describing the effect of conditions on boundsets, we define what it means to merge two boundsets together.

Definition 45 (Boundset Fusion)**.** *For any segmentation $s = (l, v, b_0, \sigma)$, with a sequence of segments $\sigma = (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}$; and any two segment indices $i_1 \in \{1,\dots,n\}$ and $i_2 \in \{1,\dots,n\}$ such that $i_1 \le i_2$, the fusion of all boundsets between indices i_1 and i_2 of s is the segmentation defined by*

$$\mathrm{FuseBoundSets}(i_1, i_2, s) = \begin{cases} \bot^{\mathrm{Seg}} & \text{if } \exists j, i_1 < j \le i_2, m_j = \mathtt{ff} \\ \left(l, v, \bigcup_{i_1\le j\le i_2} b_j, \sigma[i_2+1:n]\right) & \text{if } i_1 = 0 \text{ and} \\ & \forall j, i_1 < j \le i_2 \Rightarrow m_j = \mathtt{tt} \\ (l, v, b_0, \sigma[1:i_1-1] \cdot \left(d_{i_1}, \bigcup_{i_1\le j\le i_2} b_j, m_{i_1}\right) \cdot \sigma[i_2+1:n]) & \text{otherwise} \end{cases}$$

Merging boundsets assumes that they are equal in the concrete. If any of the emptiness markers between the boundsets being merged is false, this is a contradiction and yields the bottom segmentation.

In the following definitions, we assume that conditions do not include conjunctions or disjunctions. Conjunctions and disjunctions are treated generically, by using abstract intersection (resp. abstract union) on the result of analysing the sub-formulas separately.

Similarly, we assume that the conditions has been pre-treated by applying De Morgan's laws to eliminate negation as much as possible.

Definition 46 (Abstraction of conditions for segmentation boundsets)**.** *For any segmentation* $s = \left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right)$ *and any condition* c*, we define*

$$\mathrm{CondBsets}(c)(s) = \begin{cases} \mathrm{FuseBoundSets}(i_1, i_2, s) & \text{if } \exists i_1 \in \{0,\dots,n\}, \exists i_2 \in \{0,\dots,n\}, \\ & \quad \exists e_1 \in b_{i_1}, \exists e_2 \in b_{i_2}, \\ & \quad c = (e_1 = e_2) \vee (i_1 \leq i_2 \wedge c = (e_1 \geq e_2)) \\ \bot^{\mathrm{Seg}} & \text{if } \exists i_1 \in \{0,\dots,n\}, \exists i_2 \in \{0,\dots,n\}, \\ & \quad \exists e_1 \in b_{i_1}, \exists e_2 \in b_{i_2}, \\ & \quad i_1 \leq i_2 \wedge c = (e_1 > e_2) \\ s & \text{otherwise} \end{cases}$$

Definition 47 (Abstraction of conditions for emptiness markers)**.** *For any condition* c *and any segmentation* $s = (l, v, b_0, \sigma)$ *with a segment sequence* $\sigma = (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}$*, we define*

$$\mathrm{CondMark}(c)(s) = \begin{cases} (l, v, b_0, \sigma[1 : i_1 - 1] \cdot (d_{i_1}, b_{i_1}, \mathtt{ff}) \cdot \sigma[i_1 + 1; n]) & \text{if } \exists i_1 \in \{1,\dots,n\}, \\ & \quad \exists e_1 \in b_{i_1-1}, \exists e_2 \in b_{i_1}, \\ & \quad c = (e_1 < e_2) \\ s & \text{otherwise} \end{cases}$$

These three auxiliary definitions allow us to define the abstraction of conditions for segmentations

Definition 48 (Abstraction of conditions for segmentation)**.** *For any condition* c *and any segmentation* s*, we define the* abstraction of condition c for segmentation s *by*

$$\mathrm{Cond}^{\mathrm{Seg}}(c)(s) = \left(\mathrm{CondSum}(c) \circ \mathrm{CondMark}(c) \circ \mathrm{CondBsets}(c)\right)(s)$$

The order of composition does not affect the final result. It can, however, impact the execution time. We have chosen this order since CondBsets can reduce the number of segments, and both CondMark and CondSum do a traversal of all segments.

This definition is lifted to the A, Tana, Dana and Diorana domains in the standard way. That is, respectively, element-wise, component-wise, disjunct-wise before collapsing

and by priming the condition:

$$\mathrm{Cond}^{\mathrm{A}}(c)(a) = \Big[x \mapsto \mathrm{Cond}^{\mathrm{Seg}}(c)(a(x))\Big]$$

$$\mathrm{Cond}^{\mathrm{Tana}}(c)(t,a) = \Big(\mathrm{Cond}^{\mathrm{TanL}}(c)(t), \mathrm{Cond}^{\mathrm{A}}(c)(a)\Big)$$

$$\mathrm{Cond}^{\mathrm{Dana}}(c)(\mathcal{O}) = \mathrm{Collapse}^{\mathrm{Dana}}\left(\Big\{\mathrm{Cond}^{\mathrm{Tana}}(c)(o) \mid o \in \mathcal{O}\Big\}\right)$$

$$\mathrm{Cond}^{\mathrm{Diorana}}(c)(r) = \mathrm{Cond}^{\mathrm{Dana}}(c')(r)$$

A.2.2 Abstraction for assignments

Abstraction for an assignment $y := e$ behaves differently according to whether a variable of array type is involved (either because y is of array type or because a variable in e has an array type) or not.

Abstraction for non-array assignments

Here we look at the assignments of the form $y := e$ where y is not an array variable and no variable appearing in e is an array variable. This means that e is not an array access expression.

We start by formalizing variable replacement inside a bound expression.

Definition 49 (Variable replacement for bound expressions, boundsets and segmentations)**.** *For any variable y and any two bound expressions e_1 and e_2, the* replacement of variable y by bound expression e_1 inside bound expression e_2 *is given by*

$$\mathrm{Replace}^{\mathrm{E}}(y, e_1, e_2) = \begin{cases} x + (k_1 + k_2) & \text{if } e_1 = x + k_1 \text{ and } e_2 = y + k_2 \\ k_1 + k_2 & \text{if } e_1 = k_1 \text{ and } e_2 = y + k_2 \\ e_2 & \text{otherwise} \end{cases}$$

This definition is first extended to boundsets. For any boundset b, we define

$$\mathrm{Replace}^{\mathrm{B}}(y, e_1, b) = \big\{\mathrm{Replace}^{\mathrm{E}}(y, e_1, e_2) \mid e_2 \in b\big\}$$

Then, the definition is extended to segmentations in a point-wise manner. Formally, for any segmentation $s = \big(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\big)$, we define

$$\mathrm{Replace}^{\mathrm{Seg}}(y, e_1, s) = \Big(l, v, \mathrm{Replace}^{\mathrm{B}}(y, e_1, b_0), \big(d_i, \mathrm{Replace}^{\mathrm{B}}(y, e_1, b_i), m_i\big)_{i\in\{1,\dots,n\}}\Big)$$

Definition 50 (Conditional addition of a bound expression to a boundset)**.** *For any variable y, any bound expression e and any boundset b, we define the* addition of y to b

conditionally to the presence of e *as being the boundset*

$$\mathrm{AddBexp}^{\mathrm{B}}(y, e, b) = \begin{cases} b \cup \{y\} & \textit{if } e \in b \\ b & \textit{otherwise} \end{cases}$$

We extend this definition to segmentations in a point-wise manner. Formally, for any segmentation $s = \big(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\ldots,n\}}\big)$ *we define*

$$\mathrm{AddBexp}^{\mathrm{Seg}}(y, e, s) = \big(l, v, \mathrm{AddBexp}^{\mathrm{B}}(y, e, b_0), (d_i, \mathrm{AddBexp}^{\mathrm{B}}(y, e, b_i), m_i)_{i\in\{1,\ldots,n\}}\big)$$

This allows us to formalize the effect of assignment on the boundsets of an entire segmentation.

Definition 51 (Abstraction of non-array assignments for segmentation boundsets)**.** *For any segmentation* s *and any non-array assignment* $y := e$*, we define*

$$\mathrm{AssignBsets}(y := e)(s) = \begin{cases} \mathrm{Replace}^{\mathrm{Seg}}(y, y - k, s) & \textit{if } \exists k \in \mathrm{K}, e = y + k \\ \mathrm{AddBexp}^{\mathrm{Seg}}\left(y, e, \mathrm{RemoveBexp}\left(\{y + k \mid k \in \mathrm{K}\}, s\right)\right) & \textit{otherwise} \end{cases}$$

For summaries, assignment is propagated to each segment.

Definition 52 (Abstraction of non-array assignments for segmentation summaries)**.** *For any segmentation* $s = \big(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\ldots,n\}}\big)$ *and any non-array assignment* $y := e$*, we define*

$$\mathrm{AssignSum}(y := e)(s) = \big(l, v, b_0, \big(\mathrm{Assign}^{\mathrm{S}}(y := e)(d_i), b_i, m_i\big)\big)$$

The two previous definitions allow us to define what happens to all segmentations when analysing a non-array assignment.

Definition 53 (Abstraction of non-array assignments for segmentations)**.** *For any segmentation* s *and any non-array assignment* $y := e$ *the* abstraction of assignment $y := e$ for segmentation s *is given by*

$$\mathrm{Assign}^{\mathrm{Seg}}(y := e)(s) = \left(\mathrm{AssignSum}(y := e) \circ \mathrm{AssignBsets}(y := e)\right)(s)$$

When analysing assignments where no variable has an array type (that is, y is not an array variable, and none of the variables appearing in e are either), the transfer function $\mathrm{Assign}^{\mathrm{Seg}}$ can be lifted to the A, Tana, Dana and Diorana domains in the standard way. That is, respectively, element-wise, component-wise, disjunct-wise before

collapsing and by priming all the variables:

$$\text{Assign}^{\text{A}}(y := e)(a) = \left[x \mapsto \text{Assign}^{\text{Seg}}(y := e)(a(x))\right]$$

$$\text{Assign}^{\text{Tana}}(y := e)(t, a) = \left(\text{Assign}^{\text{TanL}}(c)(t), \text{Assign}^{\text{A}}(y := e)(a)\right)$$

$$\text{Assign}^{\text{Dana}}(y := e)(\mathcal{O}) = \text{Collapse}^{\text{Dana}}\left(\left\{\text{Assign}^{\text{Tana}}(y := e)(o) \mid o \in \mathcal{O}\right\}\right)$$

$$\text{Assign}^{\text{Diorana}}(c)(r) = \text{Assign}^{\text{Dana}}(y' := e')(r)$$

Remember that this kind of assignments can change the segmentations inside the array domain. Indeed, the segment summaries change, and (if y is a numeric variable) the boundsets might also change (Definition 51).

Abstraction for Array Creation

Here, we look at the assignments of the form $y := \mathsf{new\_array}(\tau^{\text{alg}}, e_1, e_2)$; where y is necessarily an array variable.

Definition 54 (Transfer Function for Array Creation [Complete 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* $(t, a) \in \text{Tana}$*, the transfer function for array creation is defined by*

$$\begin{aligned}
&\text{Assign}^{\text{Tana}}\left(y := \mathit{new_array}(\tau^{\text{alg}}, e_1, e_2)\right)(t, a) = \\
&\quad \begin{cases} \bot^{\text{Tana}} & \text{if } \text{Satisfies}^{\text{TanL}}(t, e_1 < 0) \\ \left(\text{Cond}^{\text{TanL}}(e_1 \geq 0)(t'), a'[y \mapsto s']\right) & \text{otherwise} \end{cases} \\
&\text{where } t' = \begin{cases} \text{Cond}^{\text{TanL}}(|y| = 0)(t) & \text{if } \text{Satisfies}^{\text{TanL}}(t, e_1 \leq 0) \\ \text{Cond}^{\text{TanL}}(|y| = e_1)(t) & \text{otherwise} \end{cases} \\
&\text{and } a' = \text{Assign}^{\text{A}}(|y| := e_1)(a) \\
&\text{and } s' = \begin{cases} \{0;\ |y|\} & \text{if } \text{Satisfies}^{\text{TanL}}(t, e_1 \leq 0) \\ \{0\}\ d\ \{|y|\} & \text{if } \text{Satisfies}^{\text{TanL}}(t, e_1 > 0) \\ \{0\}\ d\ \{|y|\}? & \text{otherwise} \end{cases} \\
&\text{and } d = \left\{\text{Cond}^{\text{TanL}}(0 \leq l < e_1 \wedge v = e_2)\left(\text{Add}^{\text{TanL}}_{\{l,v\}}(t)\right)\right\}
\end{aligned}$$

As stated after Definition 35, we distinguish four cases:

- The case where e_1 is known to be negative (that is $\text{Satisfies}^{\text{TanL}}(t, e_1 < 0)$) and the result is $\bot^{\text{Tana}}$, since any code after this array creation assignment is unreachable.
- The case where e_1 is known to be non-positive (that is $\text{Satisfies}^{\text{TanL}}(t, e_1 \leq 0)$), in which case the new array is known to have size 0, which corresponds with to a segmentation with no segments and a single boundset that contains both zero and the length of the array : $\{0;\ |y|\}$.
- The case where e_1 is known to be positive (that is $\text{Satisfies}^{\text{TanL}}(t, e_1 > 0)$). In this case, the emptiness marker of the segmentation is false (as indicated by the absence of a question mark), because we know for certain that the array is not empty.
- All the other cases, where we have no particular prior knowledge on e_1, and the result is as described in Definition 35.

Abstraction for array update

In order to define the transfer functions for array update and array access, we start by describing, for a segmentation, the creation of a new segment at a given bound expression. We will use this when the index used to access or update an array is a bound expression.

Definition 55 (Insert a segment at a bound expression)**.** *Given a segmentation* $s = (l, v, b_0, \sigma)$*, with segment sequence* $\sigma = (d_i, b_i, m_i)_{i \in \{1,\dots,n\}}$*, a bound expression* $e \in \mathrm{E}(\Gamma)$*, a segment summary* $d \in \mathrm{S}$ *and an abstract value* $o \in \mathrm{Tana}$*, the* insertion of segment d at bound expression e in segmentation s, given information o *is given by*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{Insert}^{\mathrm{Seg}}(d, e, s, o) = (l, v, b'_0, \sigma[1 : i_1 - 1] \cdot \sigma' \cdot \sigma[i_2 + 1 : n])} \\
\quad \textit{where} & i_1 = \max \left\{ i \in \{0, \dots, n\} \mid \exists e_1 \in b_i, \mathrm{Satisfies}^{\mathrm{Tana}}(o, e_1 \leq e) \right\} \\
\quad \textit{and} & i_2 = \min \left\{ i \in \{0, \dots, n\} \mid \exists e_2 \in b_i, \mathrm{Satisfies}^{\mathrm{Tana}}(o, e < e_2) \right\} \\
\quad \textit{and} & \sigma' = \begin{cases} \sigma_1 \cdot \sigma_e \cdot \sigma_2 & \textit{if } i_1 = 0 \\ (d_{i_1}, b'_{i_1}, m_{i_1}) \cdot \sigma_1 \cdot \sigma_e \cdot \sigma_2 & \textit{otherwise} \end{cases} \\
\quad \textit{and} & b'_{i_1} = \begin{cases} b_{i_1} \cup \{e\} & \textit{if } \exists e_1 \in b_{i_1}, \mathrm{Satisfies}^{\mathrm{Tana}}(o, e_1 = e) \\ b_{i_1} & \textit{otherwise} \end{cases} \\
\quad \textit{and} & b'_0 = \begin{cases} b'_{i_1} & \textit{if } i_1 = 0 \\ b_0 & \textit{otherwise} \end{cases} \\
\quad \textit{and} & \sigma_1 = \begin{cases} \varepsilon & \textit{if } \exists e_1 \in b_{i_1}, \mathrm{Satisfies}^{\mathrm{Tana}}(o, e_1 = e) \\ (d', \{e\}, m_L) & \textit{otherwise} \end{cases} \\
\quad \textit{and} & \sigma_e = (d, b_e, \mathsf{ff}) \\
\quad \textit{and} & b_e = \begin{cases} b_{i_2} \cup \{e + 1\} & \textit{if } \exists e_2 \in b_{i_2}, \mathrm{Satisfies}^{\mathrm{Tana}}(o, e_2 = e + 1) \\ \{e + 1\} & \textit{otherwise} \end{cases} \\
\quad \textit{and} & \sigma_2 = \begin{cases} \varepsilon & \textit{if } \exists e_2 \in b_{i_2}, \mathrm{Satisfies}^{\mathrm{Tana}}(o, e_2 = e + 1) \\ (d', b_{i_2}, m_R) & \textit{otherwise} \end{cases} \\
\quad \textit{and} & m_L = \exists e_1 \in b_{i_1}, \mathrm{CanSat}^{\mathrm{Tana}}(o, e_1 = e) \\
\quad \textit{and} & m_R = \exists e_2 \in b_{i_2}, \mathrm{CanSat}^{\mathrm{Tana}}(o, e_2 = e + 1) \\
\quad \textit{and} & d' = \bigsqcup_{i_1 < i \leq i_2}^{\mathrm{SL}} d_i
\end{array}
$$

Note that for arbitrary abstract values o, the segment indices i_1 and i_2 of Definition 55 might not exist. However, we will always use $\mathrm{Insert}^{\mathrm{Seg}}$ in contexts where b_0 contains 0, b_n contains some length variable $|x|$ and o contains the information that $0 \leq e < |x|$, which guarantees the existence of i_1 and i_2.

For array updates, we distinguish two cases, like [12], depending on whether the expression used as index can be put in the form of a bound expression or not.

Definition 56 (Array update at a bound expression)**.** *For any abstract value* (t, a)*, and any bound expression* $e_1 \in \mathrm{E}(\Gamma)$*, the transfer function for the array update instruction*

$y := x[e_1 \to e_2]$ *is given by*

$$\begin{array}{ll}
\mathrm{Assign}^{\mathrm{Tana}}(y := x[e_1 \to e_2])(t,a) = (t_2, a_3) & \\
\quad \textit{where} & (t_2, a_3) = \mathrm{Assign}^{\mathrm{Tana}}(|y| := |x|)(t_1, a_2) \\
\quad \textit{and} & a_2 = a_1[y \mapsto s'] \\
\quad \textit{and} & s' = \mathrm{Insert}^{\mathrm{Seg}}(d, e_1, a_1(x), (t_1, a_1)) \\
\quad \textit{and} & d = \left\{ \mathrm{Cond}^{\mathrm{TanL}}(l = e_1 \wedge v = e_2) \left(\mathrm{Add}^{\mathrm{TanL}}_{\{l,v\}}(t_1) \right) \right\} \\
\quad \textit{and} & \left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\dots,n\}}\right) = a_1(x) \\
\quad \textit{and} & (t_1, a_1) = \mathrm{Cond}^{\mathrm{Tana}}(0 \leq e_1 < |x|)(t,a)
\end{array}$$

We are going to explain the formula of array update at a bound expression (Definition 56) from bottom to top. Since expression e_1 is used as an index in the array x, it must evaluate to an integer between the bounds of the array. Hence we can add the constraint $0 \leq e_1 < |x|$ to the information we already have, which is done by the last line: $(t_1, a_1) = \mathrm{Cond}^{\mathrm{Tana}}(0 \leq e_1 < |x|)(t,a)$. Then, we get the names for the special index variable l and value variable v by looking at the current segmentation for x, which is done in the second-to-last line: $\left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\dots,n\}}\right) = a_1(x)$. We use these special variables l and v to build the segment summary for index e_1 for the new array: $d = \left\{ \mathrm{Cond}^{\mathrm{TanL}}(l = e_1 \wedge v = e_2) \left(\mathrm{Add}^{\mathrm{TanL}}_{\{l,v\}}(t_1) \right) \right\}$. This allows to define the segmentation s' for the new array, by inserting the segment summary d at index e_1 (which is possible since e_1 is a bound expression): $s' = \mathrm{Insert}^{\mathrm{Seg}}(d, e_1, a_1(x), (t_1, a_1))$. The new array environment is obtained by associating this new segmentation s' to the variable y being assigned: $a_2 = a_1[y \mapsto s']$. The length of the array now stored in y is the same as the array stored in x, and this must be taken into account both inside numeric constraints but also in boundsets. This information is updated by $(t_2, a_3) = \mathrm{Assign}^{\mathrm{Tana}}(|y| := |x|)(t_1, a_2)$, which yields our final result.

Definition 57 (Array update at a non-bound expression)**.** *For any abstract value* (t,a)*, and any expression* e_1 *that is* not *a bound expression, the transfer function for the array update instruction* $y := x[e_1 \to e_2]$ *is given by*

$$\begin{array}{ll}
\mathrm{Assign}^{\mathrm{Tana}}(y := x[e_1 \to e_2])(t,a) = (t_2, a_3) & \\
\quad \textit{where} & (t_2, a_3) = \mathrm{Assign}^{\mathrm{Tana}}(|y| := |x|)(t_1, a_2) \\
\quad \textit{and} & a_2 = a[y \mapsto s'] \\
\quad \textit{and} & s' = \left(l, v, b_0, (d'_i, b_i, m_i)_{i \in \{1,\dots,n\}}\right) \\
\quad \textit{and} & \forall i \in I, d'_i = d_i \sqcup^{\mathrm{SL}} \left\{ \mathrm{Cond}^{\mathrm{TanL}}(l = e_1 \wedge v = e_2) \left(\mathrm{Add}^{\mathrm{TanL}}_{\{l,v\}}(t_1) \right) \right\} \\
\quad \textit{and} & \forall i \in \{1, \dots, n\} \setminus I, d'_i = d_i \\
\quad \textit{and} & I = \left\{ i \in [\![1, n]\!] \mid \exists (e_L, e_R) \in b_{i-1} \times b_i, \mathrm{CanSat}^{\mathrm{Tana}}((t_1, a_1), e_L \leq e_1 < e_R) \right\} \\
\quad \textit{and} & \left(l, v, b_0, (d_i, b_i, m_i)_{i \in \{1,\dots,n\}}\right) = a_1(x) \\
\quad \textit{and} & (t_1, a_1) = \mathrm{Cond}^{\mathrm{Tana}}(0 \leq e_1 < |x|)(t,a)
\end{array}$$

Definition 57 is very similar to Definition 56. The difference lies in the segment summaries. Since e_1 is not a bound expression, it is not possible to create a segment of length exactly one to store in a precise way the information about the new value. Instead, the set I identifies all possible segments that might be indexed by e_1, and for

these segments a union between the old value and the new value is taken. Said otherwise, Definition 56 performs a strong update while Definition 57 performs a weak update.

Abstraction for array accesses

When analysing an array access instruction $y := x[e]$, three different things happen:

- The segmentation for variable x is used to deduce information on the new value of variable y.
- If e is a bound expression, a new segment is created inside the segmentation for x, of size one, whose only possible index is e. For this new segment, we know that its value is equal to the new value of y. Hence any information that we learn from y going forward, also enriches the segmentation for x.
- If y appears in bound expressions inside the boundsets of segmentations, the corresponding bound expressions are removed.

Let's look at an example of array access. In this example, we will omit the disjunctive completion, constructor constraints and structural equalities, to focus on the NPR and array components. For example, if we analyse the assignment $y := t[x]$ while already having the abstract value

$$\left(\begin{array}{l}\{5 \leq x; |t| = 10\},\\ [t \mapsto (l, v, \{0\}\ v \leq 0\ \{5\}\ v = l + 1\ \{7\}\ v = 2 \times l\ \{|t|\})]\end{array}\right)$$

then, the abstract value after the assignment should be

$$\left(\begin{array}{l}\{5 \leq x < 10; 6 \leq y \leq 18; |t| = 10\},\\ [t \mapsto (l, v, \{0\}\ v \leq 0\ \{5\}\ v = l + 1\ \{7\}\ v = 2 \times l\ \{|t|\})]\end{array}\right)$$

where we have gained information *both* on variable x and on variable y. Indeed, x being used as index tells us that x is smaller than $|t|$; while the information for y is obtained from joining what would happen for two different segments: the segment between indices 5 and 7 (5 included and 7 excluded) and the segment between indices 7 and $|t|$ (7 included and $|t|$ excluded). This results in joining the information $\{y = x+1; 5 \leq x < 7\}$ with the information $\{y = 2 \times x; 7 \leq x < 10\}$ which results in $\{6 \leq y \leq 18; 5 \leq x < 10\}$.

We start by defining the removal of the bound expressions containing y. This is similar to what is done for non-array assignments (Definition 51), but simpler, since we only remove, without adding nor replacing.

Definition 58 (Abstraction of array access for segmentation boundsets)**.** *For any segmentation s and any array access assignment $y := x[e]$, we define*

$$\text{AssignBsets}(y := x[e])(s) = \text{RemoveBexp}\,(\{y + k \mid k \in \text{K}\}, s)$$

We then define the effect of array access on abstract array environments: in addition to the removal of y from the boundsets it appears to, the segmentation associated to x is used to deduce information about y that will be added to the segment summaries of the other segmentations.

Definition 59 (Abstraction of array access for abstract array environments)**.** *For any abstract array environment a and any array access assignment $y := x[e]$, the*

abstraction of array access $y := x[e]$ for abstract array environment a *is the abstract array environment such that, for any array variable* z,

$$
\begin{aligned}
&\mathrm{Assign}^{\mathrm{A}}(y := x[e])(a,t)(z) = \\
&\quad \left(l, v, b_0^z, (d_i'^z, b_i^z, m_i^z)_{i\in\{1,\dots,p\}}\right) \\
&\quad \textit{where} \quad \forall i \in \{1,\dots,p\}\,, d_i'^z = d_e \sqcap^{\mathrm{SL}} \mathrm{Add}^{\mathrm{SL}}_{\{y\}}\left(\mathrm{Rem}^{\mathrm{SL}}_{\{y\}}(d_i^z)\right) \\
&\quad \textit{and} \quad d_e = \bigsqcup_{i=i_1+1}^{i_2\,\mathrm{SL}} \mathrm{Assign}^{\mathrm{SL}}(y := v)\left(\mathrm{Cond}^{\mathrm{SL}}(l = e)(d_i^x)\right) \\
&\quad \textit{and} \quad i_1 = \max\left\{i \in \{0,\dots,n\} \mid \exists e_1 \in b_i^x, \mathrm{Satisfies}^{\mathrm{Tana}}((t,a), e_1 \le e)\right\} \\
&\quad \textit{and} \quad i_2 = \min\left\{i \in \{0,\dots,n\} \mid \exists e_2 \in b_i^x, \mathrm{Satisfies}^{\mathrm{Tana}}((t,a), e < e_2)\right\} \\
&\quad \textit{and} \quad \left(l, v, b_0^x, (d_i^x, b_i^x, m_i^x)_{i\in\{1,\dots,n\}}\right) = a(x) \\
&\quad \textit{and} \quad \left(l, v, b_0^z, (d_i^z, b_i^z, m_i^z)_{i\in\{1,\dots,p\}}\right) = \mathrm{AssignBsets}(y := x[e])(a(z))
\end{aligned}
$$

In the detailed example of appendix §B, the fact of creating a new segment at array access proves very useful. Indeed, it allows, after assignment `challenger = q[i]`, to take into account inside the segmentation the assertions made on variable `challenger`, like, for example, the assertion `assert(res@SomeMax.prio >= challenger.prio)` on the third branch. The knowledge of how the array content relates to the value of variable `res` is what allows to capture into the function summary the information that the result has a priority higher to any of the TCBs in the queue.

Because of this creation of a new segment, array access definition is similar to the one for array update. In particular, there are two cases according to whether the index being accessed is a bound expression or not.

Definition 60 (Abstraction of array access at a bound expression)**.** *For any abstract value* $(t,a) \in \mathrm{Tana}$ *and any array access instruction* $y := x[e]$ *where* e *is a bound expression, the abstraction for array access is given by*

$$
\begin{aligned}
&\mathrm{Assign}^{\mathrm{Tana}}(y := x[e])(t,a) = (t',a') \\
&\quad \textit{where} \quad a' = \mathrm{Assign}^{\mathrm{A}}(y := x[e])(a'',t') \\
&\quad \textit{and} \quad a'' = a[y \mapsto s'] \\
&\quad \textit{and} \quad s' = \mathrm{Insert}^{\mathrm{Seg}}(d, e, a(x), (t',a)) \\
&\quad \textit{and} \quad t' = \mathrm{Rem}^{\mathrm{TanL}}_{\{l,v\}}\left(\bigsqcup_{t''\in d}^{\mathrm{TanL}} t''\right) \\
&\quad \textit{and} \quad d = \bigsqcup_{i=i_1+1}^{i_2\,\mathrm{SL}} \mathrm{Assign}^{\mathrm{SL}}(y := v)\left(\mathrm{Cond}^{\mathrm{SL}}(l = e \wedge 0 \le e < |x|)(d_i)\right) \\
&\quad \textit{and} \quad i_1 = \max\left\{i \in \{0,\dots,n\} \mid \exists e_1 \in b_i, \mathrm{Satisfies}^{\mathrm{Tana}}((t,a), e_1 \le e)\right\} \\
&\quad \textit{and} \quad i_2 = \min\left\{i \in \{0,\dots,n\} \mid \exists e_2 \in b_i, \mathrm{Satisfies}^{\mathrm{Tana}}((t,a), e < e_2)\right\} \\
&\quad \textit{and} \quad \left(l, v, b_0, (d_i, b_i, m_i)_{i\in\{1,\dots,n\}}\right) = a(x)
\end{aligned}
$$

The abstraction of array access at a non-bound expression is given in Definition 36, Page 47, on the main text of the article.

B Detailed example of analysis with arrays

In this appendix, we manually execute the analysis on the `find_max_priority` function from Fig. 8. To alleviate notation, we use the following conventions in the description that follows:

- If a set of abstract values in a disjunctive completion is a singleton, then we omit the braces. For example, in a structural equalities component, we write $q' = q$ instead of $\{q' = q\}$.
- If a constraint can be deduced from other constraints already written, it can be omitted. For example, if we have a structural equalities component $q' = q$ and the NPR component can be described by the set of constraints $\{|q| > 0; |q'| > 0\}$, then one of the two constraints of the NPR component can be omitted, and we write $|q'| > 0$.
- We may omit an entire component of a product domain when the component is $\top$, or when all its constraints can be deduced from other components.
- We use labels to better identify components of an abstract value: we use the label **cc** for the constructor constraints component, **seq** for the structural equalities component, **NPR** for the NPR component, **A** for the array component.

We start the analysis of the function with the identity relation between the primed and un-primed versions of the arguments, together with the top segmentation for arrays. The top segmentation for arrays implies that every array length is non-negative. Hence, the initial abstract value is

$$r_0 = \begin{pmatrix} \mathbf{seq} : q' = q, \quad \mathbf{NPR} : |q'| \geq 0 \\ \mathbf{A} : \ [q' \mapsto \{0\} \top \{|q'|\}?] \end{pmatrix}$$

After the first two assignments `i = 0; res = NoMax {}`, we get the abstract value

$$r_1 = \begin{pmatrix} \mathbf{cc} : res'@\text{NoMax}, \quad \mathbf{seq} : q' = q, \quad \mathbf{NPR} : \{|q'| \geq 0; i' = 0\} \\ \mathbf{A} : \ [q' \mapsto \{0; i'\} \top \{|q'|\}?] \end{pmatrix}$$

We see that, in addition to gaining the constructor constraint $res'@\text{NoMax}$, the bound expression i' was added to the set of bound expressions that contained 0, since, after the assignment, we have $i' = 0$. The variable i' is added by the first boundset thanks to the $\text{AddBexp}^{\text{B}}$ function, that is used by the transfer function for assignments. This is formalised in Definitions 42, 50, 51 and 53.

When analysing the loop, we use loop unrolling. Hence, the results of not executing the loop at all, and executing it at least once, are computed separately, and then joined afterwards. The result of not executing the loop at all is obtained by applying the transfer function for condition $i' \geq |q'|$ to the abstract value before the loop, and produces the following abstract value:

$$r_2 = \text{Cond}^{\text{Dana}}(i' \geq |q'|)(r_1) = \\ \begin{pmatrix} \mathbf{cc} : res'@\text{NoMax}, \quad \mathbf{seq} : q' = q, \quad \mathbf{NPR} : i' = 0 = |q'| \\ \mathbf{A} : \ [q' \mapsto \{0; i'; |q'|\}] \end{pmatrix}$$

In the NPR component, the new condition $i' \geq |q'|$ combined with the information that we already had $i' \leq |q'|$ allows to deduce $i' = |q'|$. In the array component, in the segmentation for q', the two former boundsets $\{0; i'\}$ and $\{|q'|\}$ are merged by the FuseBoundSets function. Indeed, there is no segment between i' and $|q'|$, since $i' = |q'|$. This is formalized in Definitions 45, 46 and 48. We get for q' a segmentation with only one set of segment bounds and no segment summary; in other words, a segmentation for the empty array.

To obtain the result of executing the loop at least once, we start by analysing one execution of the loop body, then we look for a fixpoint with a Kleene iteration with widening. When entering the loop the first time, we get

$$r_3 = \mathrm{Cond}^{\mathrm{Dana}}(i' < |q'|)(r_1) = \left(\begin{array}{lll} \mathbf{cc} : res'@\mathrm{NoMax}, & \mathbf{seq} : q' = q, & \mathbf{NPR} : \{|q'| > 0; i' = 0\} \\ \mathbf{A} : [q' \mapsto \{0; i'\} \top \{|q'|\}] & & \end{array} \right)$$

The condition tells us that the segment between $\{0; i'\}$ and $\{|q'|\}$ is not empty, since $i' < |q'|$. Hence the question mark indicating that the segment can be empty disappears from the array segmentation. This is formalized in Definition 47.

Then, analysing the assignment `challenger = q[i]` creates a new segment, between bound expressions i and $i + 1$, for the segmentation of q'.

$$r_4 = \mathrm{Assign}^{\mathrm{Dana}}(challenger' := q'[i'])(r_3) = \left(\begin{array}{lll} \mathbf{cc} : res'@\mathrm{NoMax}, & \mathbf{seq} : q' = q, & \mathbf{NPR} : \{|q'| > 0; i' = 0\} \\ \mathbf{A} : \left[q' \mapsto \{0; i'\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' = 0; \\ v = challenger' \end{array} \right\} \right) \{i' + 1\} \top \{|q'|\}? \right] & & \end{array} \right)$$

The insertion of this segment of size 1 is performed by the $\mathrm{Insert}^{\mathrm{Seg}}$ function, and is formalized in Definitions 55 and 60.

To analyse the branching, the tree branches are analysed separately, and then the results are joined together. Here, for the first iteration of the loop, only the first branch of the branching is reachable. Indeed, for the second and third branches, the assertions introduce the path `res'@SomeMax.prio`, which is incompatible with the path `res'@NoMax` already present. The analysis therefore infers that the second and third branches are unreachable and produces $\bot^{\mathrm{Diorana}}$. Hence, the result of the whole branching construct for this first loop iteration is the result of analysing the first branch. The first assignment of the first branch adds a structural equality to our abstract value:

$$r_5 = \mathrm{Assign}^{\mathrm{Dana}}(case' := res'@\mathrm{NoMax})(r_4) = \left(\begin{array}{lll} \mathbf{cc} : res'@\mathrm{NoMax}, & \mathbf{seq} : \left\{ \begin{array}{l} q' = q; \\ case' = res'@\mathrm{NoMax} \end{array} \right\}, & \mathbf{NPR} : \left\{ \begin{array}{l} |q'| > 0; \\ i' = 0 \end{array} \right\} \\ \mathbf{A} : \left[q' \mapsto \{0; i'\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' = 0; \\ v = challenger' \end{array} \right\} \right) \{i' + 1\} \top \{|q'|\}? \right] & & \end{array} \right)$$

The second assignment of the first branch changes the constructor constraint on `res'`, and allows us to gain some information for the array segment contained between the

bounds i' and $i'+1$:

$$r_6 = \mathrm{Assign}^{\mathrm{Dana}}(res' := \mathrm{SomeMax}(challenger'))(r_5) = \left(\begin{array}{l} \mathbf{seq} : \{res'@\mathrm{SomeMax} = challenger'; q' = q\}, \quad \mathbf{NPR} : \{|q'| > 0; i' = 0\} \\ \mathbf{A} : \left[q' \mapsto \{0; i'\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' = 0; \\ v = challenger' \\ v = res'@\mathrm{SomeMax} \end{array} \right\} \right) \{i'+1\} \top \{|q'|\}? \right] \end{array} \right)$$

After the branching, the next instruction is the assignment $i := i + 1$. This assignment is invertible, in the sense that the value of variable i after the assignment is an expression on the value of i before the assignment. The inverse of this assignment is $i := i - 1$. The expression $i - 1$ in this inverse assignment can be used as a segment bound. Hence we update the bound segments by replacing i' with $i' - 1$ (in addition to updating the NPR components and the segment summaries). This is obtained thanks to the function $\mathrm{Replace}^{\mathrm{Seg}}$ and formalized in Definitions 49 and 51. We get

$$r_7 = \mathrm{Assign}^{\mathrm{Dana}}(i' := i' + 1)(r_6) = \left(\begin{array}{l} \mathbf{seq} : \{res'@\mathrm{SomeMax} = challenger'; q' = q\}, \quad \mathbf{NPR} : \{|q'| > 0; i' = 1\}, \\ \mathbf{A} : \left[q' \mapsto \{0; i' - 1\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' - 1 = 0; \\ v = challenger' \\ v = res'@\mathrm{SomeMax} \end{array} \right\} \right) \{i'\} \top \{|q'|\}? \right] \end{array} \right)$$

This abstract value r_7, that corresponds to the end of the first abstract execution of the loop, will not only be used to compute the second abstract execution of the loop, but also to take a widening with the result of that second execution.

The second abstract execution of the loop body starts with the loop condition

$$r_8 = \mathrm{Cond}^{\mathrm{Dana}}(i' < q')(r_7) = \left(\begin{array}{l} \mathbf{seq} : \{res'@\mathrm{SomeMax} = challenger'; q' = q\}, \quad \mathbf{NPR} : \{|q'| > 1; i' = 1\} \\ \mathbf{A} : \left[q' \mapsto \{0; i' - 1\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' - 1 = 0; \\ v = challenger' \\ v = res'@\mathrm{SomeMax} \end{array} \right\} \right) \{i'\} \top \{|q'|\} \right] \end{array} \right)$$

As before, the loop condition allows to stablish that the last segment of the segmentation is not empty.

Then we have to analyse the assignment to variable `challenger`, and we obtain

$$\begin{array}{l} r_9 = \mathrm{Assign}^{\mathrm{Dana}}(challenger' := q'[i'])(r_8) = \\ \quad \left(\begin{array}{l} \mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{seq} : q' = q, \quad \mathbf{NPR} : \{|q'| > 1; i' = 1\} \\ \mathbf{A} : \left[q' \mapsto \{0; i' - 1\} \; d_1^9 \; \{i'\} \; d_2^9 \; \{i'+1\} \top \{|q'|\}? \right] \end{array} \right) \\ \quad \text{where } d_1^9 = \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' - 1 = 0; \\ v = res'@\mathrm{SomeMax} \end{array} \right\} \right) \\ \quad \text{and} \quad d_2^9 = \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' = 1; \\ v = challenger \end{array} \right\} \right) \end{array}$$

As before, this assignment creates a new segment in the segmentation, between bounds i' and $i'+1$.

During this second abstract execution of the loop, the first branch of the branching is unreachable and its analysis yields $\bot^{\text{Diorana}}$, while the other two branches are reachable. The assertion on the second branch yields

$$
\begin{array}{l}
r_{10} = \text{Cond}^{\text{Dana}}(\mathit{challenger}'.\text{prio} > \mathit{res}'@\text{SomeMax.prio})(r_9) = \\
\left(\begin{array}{l}
\mathbf{cc} : \mathit{res}'@\text{SomeMax}, \mathbf{seq} : q' = q, \\
\mathbf{NPR} : \{|q'| > 1; i' = 1; \mathit{challenger}'.\text{prio} > \mathit{res}'@\text{SomeMax.prio}\} \\
\mathbf{A} : \left[q' \mapsto \{0; i'-1\}\ d_1^{10}\ \{i'\}\ d_2^{10}\ \{i'+1\} \top \{|q'|\}?\right]
\end{array}\right) \\
\text{where } d_1^{10} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i'-1 = 0; v = \mathit{res}'@\text{SomeMax} \\ v.\text{prio} < \mathit{challenger}'.\text{prio} \end{array}\right\}\right) \\
\text{and } d_2^{10} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' = 1; \\ v = \mathit{challenger}' \end{array}\right\}\right)
\end{array}
$$

We see that the fact that the segment summaries mention variables of the program other than l and v (like res' and $\mathit{challenger}'$ here) allows to capture interesting information on the array contents. For example, here, the inequality $v.\text{prio} < \mathit{challenger}'.\text{prio}$ inside the first segment summary d_1^{10}, that will allow, as the analysis continues, to deduce that the priorities of all TCBs in the array are less than or equal to the one of the final result.

Then, after the assignment of the second branch, we have

$$
\begin{array}{l}
r_{11} = \text{Assign}^{\text{Dana}}(\mathit{res}' := \text{SomeMax}(\mathit{challenger}'))(r_{10}) = \\
\left(\begin{array}{l}
\mathbf{cc} : \mathit{res}'@\text{SomeMax}, \quad \mathbf{seq} : \{q' = q; \mathit{res}'@\text{SomeMax} = \mathit{challenger}'\}, \\
\mathbf{NPR} : \{|q'| > 1; i' = 1\} \\
\mathbf{A} : \left[q' \mapsto \{0; i'-1\}\ d_1^{11}\ \{i'\}\ d_2^{11}\ \{i'+1\} \top \{|q'|\}?\right]
\end{array}\right) \\
\text{where } d_1^{11} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i'-1 = 0; \\ v.\text{prio} < \mathit{res}'@\text{SomeMax.prio} \end{array}\right\}\right) \\
\text{and} \quad d_2^{11} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' = 1; \\ v = \mathit{challenger}' \end{array}\right\}\right)
\end{array}
$$

If we compare the first segment summary in this abstract value r_{11} with the previous abstract value r_{10}, we see that the previous equality $v = \mathit{res}'@\text{SomeMax}$ has disappeared, since the value of res' has changed; but a new inequality $v.\text{prio} < \mathit{res}'@\text{SomeMax.prio}$ has been deduced from the previous inequality $v.\text{prio} < \mathit{challenger}'.\text{prio}$.

For the third branch, the constraint $\mathit{res}'@\text{SomeMax.prio} \geq \mathit{challenger}'.\text{prio}$ is captured by the NPR component of the Tana product. This is the only change with

respect to the abstract value before the branching.

$$
\begin{array}{l}
r_{12} = \mathrm{Cond}^{\mathrm{Dana}}(res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio})(r_9) = \\
\left(\begin{array}{l}
\mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{seq} : q' = q, \\
\mathbf{NPR} : \{|q'| > 1; i' = 1; res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio}\} \\
\mathbf{A} : \ \left[q' \mapsto \{0; i'-1\}\ d_1^{12}\ \{i'\}\ d_2^{12}\ \{i'+1\}\top\{|q'|\}?\right]
\end{array}\right) \\
\text{where } d_1^{12} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' - 1 = 0; \\ v = res'@\mathrm{SomeMax} \end{array}\right\}\right) \\
\text{and} \quad d_2^{12} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' = 1; \\ v = challenger' \end{array}\right\}\right)
\end{array}
$$

At the end of the branching, we get the union of the abstract value for the second branch r_{11}, and the abstract value for the third branch r_{12} (recall the first branch is unreachable and its abstract value $\bot^{\mathrm{Diorana}}$ at this point).

$$
\begin{array}{l}
r_{13} = r_{11} \sqcup^{\mathrm{Diorana}} r_{12} = \\
\left(\begin{array}{l}
\mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{seq} : q' = q, \\
\mathbf{NPR} : \{|q'| > 1; i' = 1; res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio}\} \\
\mathbf{A} : \ \left[q' \mapsto \{0; i'-1\}\ d_1^{13}\ \{i'\}\ d_2^{13}\ \{i'+1\}\top\{|q'|\}?\right]
\end{array}\right) \\
\text{where } d_1^{13} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' - 1 = 0; \\ v.\mathrm{prio} \leq res'@\mathrm{SomeMax.prio} \end{array}\right\}\right) \\
\text{and} \quad d_2^{13} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' = 1; \\ v = challenger' \end{array}\right\}\right)
\end{array}
$$

One interesting thing to note here, is the value of the first segment summary. Indeed, in the second branch we have $v.\mathrm{prio} < res'@\mathrm{SomeMax.prio}$ whereas in the third branch we have both $v = res'@\mathrm{SomeMax}$ and $res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio}$, hence the union of the two yields $v.\mathrm{prio} \leq res'@\mathrm{SomeMax.prio}$. Intuitively, we go from the two cases "A TCB with a higher priority has been found in the array, and stored in variable res" and "The TCB stored in res is still the one with the highest priority so far" to the invariant "Whatever the case, the priority of the TCB stored in res is higher or equal to the ones in the array, between indices 0 included, and i excluded".

The last instruction of the loop body is the increment of variable i. As before, this updates the bound expressions used to delimit segments, in addition to update segment summaries.

$$
\begin{array}{l}
r_{14} = \mathrm{Assign}^{\mathrm{Dana}}(i' := i' + 1)(r_{13}) = \\
\left(\begin{array}{l}
\mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{seq} : q' = q, \\
\mathbf{NPR} : \{|q'| > 1; i' = 2; res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio}\} \\
\mathbf{A} : \ \left[q' \mapsto \{0; i'-2\}\ d_1^{14}\ \{i'-1\}\ d_2^{14}\ \{i'\}\top\{|q'|\}?\right]
\end{array}\right) \\
\text{where } d_1^{14} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' - 2 = 0; \\ v.\mathrm{prio} \leq res'@\mathrm{SomeMax.prio} \end{array}\right\}\right) \\
\text{and} \quad d_2^{14} = \left(\mathbf{NPR} : \left\{\begin{array}{c} l = i' - 1 = 1; \\ v = challenger' \end{array}\right\}\right)
\end{array}
$$

The next step of the analysis is to take a widening between abstract value r_7 that was obtained at the end of the first execution of the loop, and abstract value r_{14}, obtained at the end of the second execution of the loop. In order to perform the widening, we start by transforming the segmentations that appear in these two abstract values, so that they have the same sets of bound expressions; this is called *unification*. In this detailed example, when unifying segmentations for widening, we will only use bound expression removal, and not boundset splitting (section § 6.5.1). Although less precise in general, it will be precise enough for this example, and allows for faster convergence, which is useful in hand-executed analyses. In this example, not using boundset splitting forces the removal of bound expression $i' - 1$, which means that the two first segments of r_{14} are merged together, using abstract union. After unification we get

$$r'_7 = \left(\begin{array}{l} \mathbf{seq} : \{res'@\text{SomeMax} = challenger'; q' = q\}, \quad \mathbf{NPR} : \{|q'| > 0; i' = 1\}, \\ \mathbf{A} : \left[q' \mapsto \{0\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} l = i' - 1 = 0; \\ v = challenger' \\ v = res'@\text{SomeMax} \end{array} \right\} \right) \{i'\} \top \{|q'|\}? \right] \end{array} \right)$$

and

$$r'_{14} = \left(\begin{array}{l} \mathbf{cc} : res'@\text{SomeMax}, \quad \mathbf{seq} : q' = q, \\ \mathbf{NPR} : \{|q'| > 1; i' = 2; res'@\text{SomeMax.prio} \geq challenger'.\text{prio}\} \\ \mathbf{A} : \left[q' \mapsto \{0\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} i' - 2 \leq l \leq i' - 1; i' = 2; \\ v.\text{prio} \leq res'@\text{SomeMax.prio} \end{array} \right\} \right) \{i'\} \top \{|q'|\}? \right] \end{array} \right)$$

We see that, as intended, the segmentations share the same sets of bound expressions after unification: $\{0\}$, $\{i'\}$ and $\{|q'|\}$. To go from r_7 to r'_7 the bound expression $i' - 1$ was removed from the boundset $\{0; i' - 1\}$. To go from r_{14} to r'_{14}, the bound expressions $i' - 2$ and $i' - 1$ were removed from their respective bound expressions, and the two first segments were merged. Everything else remained unchanged. Both of these transformations yield abstract values that are less precise than the initial ones. In other words, $r_7 \sqsubseteq^{\text{Diorana}} r'_7$ and $r_{14} \sqsubseteq^{\text{Diorana}} r'_{14}$. This is the reason why it is sound to perform unification for widening.

After unification, widening is done segment-wise. The result of the widening is

$$r_{15} = \left(\begin{array}{l} \mathbf{cc} : res'@\text{SomeMax}, \quad \mathbf{seq} : q' = q, \\ \mathbf{NPR} : \{|q'| > 0; i' \geq 1; res'@\text{SomeMax.prio} \geq challenger'.\text{prio}\} \\ \mathbf{A} : \left[q' \mapsto \{0\} \left(\mathbf{NPR} : \begin{array}{c} 0 \leq l \leq i' - 1; i' \geq 1; \\ v.\text{prio} \leq res'@\text{SomeMax.prio} \end{array} \right) \{i'\} \top \{|q'|\}? \right] \end{array} \right)$$

We see that during unification and widening, the distinction between the last slot of the array to be treated (the one at $i - 1$) and the others before it has disappeared. Instead we have a single segment for all the slots treated so far (between indices 0 included and i excluded), with the invariant that their TCB's priority is smaller or equal to the one in variable *res*.

This is a fixpoint. We recall that in the context of loop unrolling, we computed separately the abstract value for the executions where the loop is not taken at all (r_2), from the abstract value for all the executions that take the loop at least once. The latter is obtained from fixpoint r_{15} by applying the negation of the loop condition:

$$r_{16} = \mathrm{Cond}^{\mathrm{Dana}}(i' \geq |q'|)(r_{15}) = \left(\begin{array}{l} \mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{seq} : q' = q, \\ \mathbf{NPR} : \{|q'| = i'; i' \geq 1; res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio}\} \\ \mathbf{A} : \left[q' \mapsto \{0\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} 0 \leq l \leq i' - 1; i' \geq 1; \\ v.\mathrm{prio} \leq res'@\mathrm{SomeMax.prio} \end{array} \right\} \right) \{i'; |q'|\} \right] \end{array} \right)$$

Like before for r_2, the condition $i' \geq |q'|$ together with the information already present $i' \leq |q'|$ allows to deduce that $i' = |q'|$, and hence allows to merge the two sets of bound expressions $\{i'\}$ and $\{|q'|\}$.

The abstract value after the loop, that accounts for the execution that do not take the loop at all, and those that take it at least once, is

$$r_{17} = r_2 \sqcup^{\mathrm{Diorana}} r_{16} = \left\{ \begin{array}{l} \left(\begin{array}{l} \mathbf{cc} : res'@\mathrm{NoMax}, \quad \mathbf{seq} : q' = q, \quad \mathbf{NPR} : i' = 0 = |q'| \\ \mathbf{A} : \ [q' \mapsto \{0; i'; |q'|\}] \end{array} \right); \\ \left(\begin{array}{l} \mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{seq} : q' = q, \\ \mathbf{NPR} : \{|q'| = i'; i' \geq 1; res'@\mathrm{SomeMax.prio} \geq challenger'.\mathrm{prio}\} \\ \mathbf{A} : \left[q' \mapsto \{0\} \left(\mathbf{NPR} : \left\{ \begin{array}{c} 0 \leq l \leq i' - 1; i' \geq 1; \\ v.\mathrm{prio} \leq res'@\mathrm{SomeMax.prio} \end{array} \right\} \right) \{i'; |q'|\} \right] \end{array} \right) \end{array} \right\}$$

Since the two abstract values r_2 and r_{16} have incompatible constructor constraints $res'@\mathrm{NoMax}$ and $res'@\mathrm{SomeMax}$; they are both kept in the disjunction, as separate cases.

The input-output summary for the function is then obtained by removing all the variables except the input variable q and the output variable res'. The structural equality $q' = q$ allows to transport all the information gathered on q' to q. The summary that we get for the `find_max_priority` function is

$$\left\{ \begin{array}{l} \left(\mathbf{cc} : res'@\mathrm{NoMax}, \quad \mathbf{NPR} : |q| = 0, \quad \mathbf{A} : \ [q \mapsto \{0; |q|\}] \right); \\ \left(\begin{array}{l} \mathbf{cc} : res'@\mathrm{SomeMax}, \quad \mathbf{NPR} : |q| \geq 1, \\ \mathbf{A} : \ [q \mapsto \{0\} \ \ (\mathbf{NPR} : l \geq 0; \ v.\mathrm{prio} \leq res'@\mathrm{SomeMax.prio}) \ \ \{|q|\}] \end{array} \right) \end{array} \right\}$$

As stated in section § 6.8, this input-output summary 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.

As we stated before, 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.