Lifting Numeric Relational Domains to Algebraic Data Types

Santiago Bautista, Thomas Jensen, Benoît Montagu

Lifting Numeric Relational Domains to Algebraic Data Types

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

[1] Univ Rennes, 263 Avenue Général Leclerc, F-35000 Rennes, France,
Santiago.Bautista@ens-rennes.fr
[2] Inria, {Thomas.Jensen, Benoit.Montagu}@inria.fr

Abstract. We present RAND, an input-output relational abstract domain that expresses relations between values of non-recursive algebraic data types (ADTs), and numeric relations between their scalar parts. RAND is parametrised on a user-provided numeric relational domain, that we lift to pairs of variables and projection paths. It is constructed as a disjunctive completion of a reduced product of domains for numeric relations, for equalities, and for cases of variant constructors. Using RAND, we define a modular, inter-procedural, input-output relational analysis for a `while` language with ADTs and function calls. The analysis computes function summaries, that describe relations between the inputs of programs and their outputs.

Keywords: Static Analysis · Abstract interpretation · Relational abstract domains · Algebraic data types · Input-output relations · Function summaries.

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 [38, 39] or by leveraging tree automata techniques [7] for approximating the tree structures described by ADTs [18,30,37].

In this paper, we focus on the automatic analysis of programs that perform numeric operations *and* manipulate ADTs. So far, few works [16, 25, 26] have put emphasis on the analysis of such programs. They provide additional safety guarantees specifically related to this combination—such as the unreachability of branches of a pattern matching. 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 [1].

To this end, we first develop a novel relational abstract domain that can express relations between numeric-algebraic values of a program state (§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 [24] 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 (§4). For an example process management program from an idealised operating system (Fig. 1), 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 (§2.3).

Using RAND, we define a *relational* analysis for a `while` language that features non-recursive ADTs (§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.

Our work offers the following contributions:

- We present a novel abstract domain that expresses relations between values of non-recursive ADTs (§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 [1], that is restricted to information about equality and reachability.
- Our abstract domain uses a form of *disjunctive completion* (§3.6), 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* (Lem. 1) and by showing how any relational domain can express relations between different stores (§4.2).
– We formally define a relational analysis (§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 provide an OCaml implementation [3] 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 (§6). We briefly discuss the complexity of our implementation.

2 Syntax and Semantics

Our programming language is an extension of a classic `while` language with algebraic data types (products and sums). §2.1 presents algebraic types, §2.2 presents the language and its semantics, and §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:*

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

Here, N 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 N and the record type with no fields $\{\}$ are the two base cases for types.

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

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

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

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

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

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

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 $x \in \text{Vars}$ over a path $p \in \text{Paths}$, written $x.p$. The expression $t_1 \boxplus t_2$ denotes some arithmetic operations on the expressions t_1 and t_2, and $t_1 \bowtie t_2$ ranges over arithmetic comparisons.

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

We restrict our attention to well-typed commands (that we call *programs*), following a standard structural type system [39]. 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. The relation $(c, s) \to (c', s')$ tells that the command c transforms the store s into a store s', and that command c'

is to be executed next. We briefly explain the semantics of each command, and refer the reader to the extended version [4] for technical details.

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

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

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

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

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

Fig. 1 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 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:

```
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. 1: Example: performing clock ticks on a process' meta-data.

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

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 [1] 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

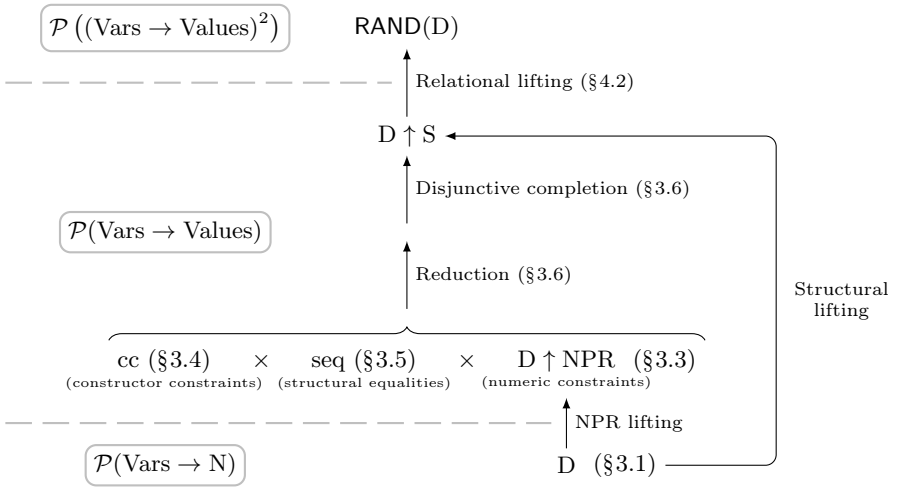

Fig. 2: The construction of the RAND abstract domain. The frame-enclosed sets are the sets the abstract domains concretize to.

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. 2. It is parametric with respect to a numeric abstract domain D, so that we can instantiate it on domains with different precision versus cost trade-offs. We expect the numeric domain D to provide the operations described in §3.1, which are a subset of the API offered by Apron [24]. An

essential ingredient of our construction is the use of *extended variables* (§3.2), *i.e.*, regular program variables equipped with a projection path. Our example `do_ticks` on Fig. 1 features extended variables, such as `p.status@Asleep.secs`. Using extended variables, we define in §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 [2], 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 (§3.4 and 3.5) in a product domain (§3.6). A first domain of *constructor constraints* tracks which constructors are used for values of sum types (§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* (§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. Finally, in order to obtain additional precision when analysing pattern-matching, we use a *disjunctive completion* of the product of these domains (§3.6): 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. §3.3 to 3.5 also define abstractions for assignments and conditionals, that are needed in §5 to define the analysis of our language.

3.1 Background: Numeric Abstract Domains

We first review the structure of traditional numeric domains [35] 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 $\mathrm{D}(V)$ to provide the operations listed below (which are included in the user interface of the Apron library [24]) in such a way that the standard soundness properties of abstract interpretation [9, 10] are met: A set of abstract values $\mathrm{D}(V)$ with a concretisation function $\gamma^{\mathrm{D}(V)} \in \mathrm{D}(V) \to \mathcal{P}(V \to \mathrm{N})$, a pre-order on abstract values $\sqsubseteq^{\mathrm{D}(V)}$, abstract union $\sqcup^{\mathrm{D}(V)}$ and intersection $\sqcap^{\mathrm{D}(V)}$, and a widening operator $\triangledown^{\mathrm{D}(V)}$. The domain must also offer abstractions for boolean conditions $\mathrm{Cond}^{\mathrm{D}(V)} \in \mathrm{BExp} \to \mathrm{D}(V) \to \mathrm{D}(V)$ and for assignment $\mathrm{Assign}^{\mathrm{D}(V)} \in V \times \mathrm{Arith}\,(V) \to \mathrm{D}(V) \to \mathrm{D}(V)$ (where $\mathrm{Arith}\,(V)$ is the set of arithmetic expressions

over the variables V), satisfying the soundness properties:

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

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

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

3.2 Extended Variables

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

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

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

Typing contexts, written Γ, are mappings from variables to types. We write $\mathcal{E}(\Gamma) = \{x.p \mid x \in \mathrm{dom}\,\Gamma \wedge \Gamma(x) \Downarrow^{\mathrm{typ}} p\}$ for the set of extended variables $x.p$ such that p is consistent with the type of x in Γ.

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. Def. 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.*

In §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 §3.6.

Assignment decomposition. To easily define 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:*

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

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

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

3.3 Numeric Domains Over Extended Variables

In this section, extending ideas from [2], we define the *Numeric Path Relations lifting* $\mathrm{D}\uparrow\mathrm{NPR}$ as a generic way to lift a domain D 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* (Def. 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 $\mathrm{D}\uparrow\mathrm{NPR}(\Gamma)$ are pairs of a set E of extended variables that are valid in Γ, and a numeric abstract value from $\mathrm{D}(E)$—*i.e.*, whose variables are the extended variables in E.

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

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 §3.6 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.

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

The concretisation of an element $(d, E) \in \mathrm{D} \uparrow \mathrm{NPR}(\Gamma)$ 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. $\gamma^{\mathrm{D}\uparrow\mathrm{NPR}(\Gamma)}(d, E) = \left\{ s \mid \Gamma \vdash s \ \wedge \ s \Downarrow^{\mathrm{sto}} E \in \gamma^{\mathrm{D}(E)}(d) \right\}$

We briefly explain how to define the abstract intersection in $\mathrm{D} \uparrow \mathrm{NPR}(\Gamma)$. 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 $\mathrm{Add}_{E_j}^{\mathrm{D}(E_i)}$ operators, before actually taking their intersection in the numeric domain:

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

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

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

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

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

$$\mathrm{Assign}^{\mathrm{D}\uparrow\mathrm{NPR}(\Gamma)}(x := t)(d, E) \ = \ \mathrm{Rename}_{[x' \mapsto x]}^{\mathrm{D}\uparrow\mathrm{NPR}(\Gamma)}(\mathrm{Rem}_{E_x}^{\mathrm{D}\uparrow\mathrm{NPR}(\Gamma)}(d_1, E_1))$$

$$\text{where } \begin{cases}
E_x = \{ y.p \in E \mid y = x \} \\
d_1 = \bigsqcap_{x'.p := u \in \mathrm{Decomp}(x' := t), \Gamma \vdash u : \mathrm{N}}^{\mathrm{D}(E_1)} \mathrm{Assign}^{\mathrm{D}(E_1)}(x'.p := u)(\mathrm{Add}_{E_0}^{\mathrm{D}(E)}(d)) \\
E_1 = E \cup E_0 \\
E_0 = \{ y.p \in \mathrm{Env}\,(\mathrm{Decomp}\,(x' := t)) \mid \Gamma \vdash y.p : \mathrm{N} \}
\end{cases}$$

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 the one of a numeric test $t_1 \bowtie t_2$: it suffices to call the transfer function of domain D for conditionals, and to extend the extended variables with those that occur in the test.

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

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. Constructor constraints are a key ingredient of the disjunctive completion of §3.6, as they serve as hints for which disjuncts need to be kept separate, and which should be merged.

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

Definition 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 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.

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 standard *"kill-gen"* form as follows:

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

The extended variables that must be removed are those that have x as root, since the 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 §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. 1, 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. The extended version [4] provides an example of this decomposition of equalities into a conjunction of equalities.

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:*

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

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

Abstract values in this domain might carry some implicit information. For example, if x and y have type $\{f \to \mathrm{N}; g \to \mathrm{N}\}$, 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 $\mathrm{clos}_E^{\mathrm{seq}(\Gamma)}(C)$, is the smallest set that is larger than C, that mentions the variables in E, is closed under symmetry, reflexivity and transitivity, and satisfies the following congruence property:*

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

The need for a closure operator is not surprising, as it occurs in other relational domains, like octagons [34]. We use this closure operator to gain precision in the transfer function for assignment, and in the reduction operator of the product domain of §3.6.

Transfer Functions. The transfer function for assignment $x := t$ for the $\mathrm{seq}(\Gamma)$ domain exploits the decomposition of assignments from Def. 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.

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

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* $\mathrm{D} \uparrow \mathrm{S}$ of the numeric domain D that we have considered. We combine the domains we have defined in the previous sections—the constructor constraints (§3.4), the structural equalities (§3.5), and the NPR lifting (§3.3)—into a *reduced product*, and then 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 §4.2.

Reduced Product Our reduced product is based on a reduction operator ρ, that enables information transfer between the different domains of the product.

Definition 14 (Reduction operator). *The reduction operator ρ for the product of constructor constraints, structural equalities and the NPR lifting is defined as follows:*

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

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 mentioned (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 the extended version [4] for further details. The transfer functions for assignment and conditionals use the transfer functions of each component.

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 D, we take the disjunctive completion [9] of the reduced product of constructor constraints, structural equalities and the NPR lifting of D. We call this the *structural lifting* of D, written $\mathrm{D}\uparrow\mathrm{S}(\Gamma)$, and defined as $\mathrm{D}\uparrow\mathrm{S}(\Gamma) = \mathcal{P}(\mathrm{cc}(\Gamma)\times\mathrm{seq}(\Gamma)\times\mathrm{D}\uparrow\mathrm{NPR}(\Gamma))$. 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. The technical details are provided in the extended version [4].

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*. In the rest of this paper, the term "relational" mostly 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 15 (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 [14,28]. 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 succeeded, then a linear relational domain will be able to express that `x` has strictly increased, for example. 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 16 (Collecting semantics). $\mathbb{P}[\![c]\!](a) = a;\mathbb{S}[\![c]\!]$

$\mathbb{P}[\![c]\!]$ is an abstraction of a semantics of computation traces [8], that collects the intermediate stores that a program may reach. The collecting semantics $\mathbb{P}[\![c]\!]$

enjoys the equations listed in the next lemma, that shows how it decomposes by following the syntax of commands.

Lemma 1. *The following equations hold:*

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

Lem. 1 will serve as the semantic basis for the analysis that we describe in §5.2. The proof of Lem. 1 is available in the extended version [4].

Lem. 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 §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 [5, 19].

4.2 Leveraging Relations in Space to Express Relations in Time

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

Formally, this is achieved by assuming two bijections $\mathit{prime} : \mathrm{Vars} \to \mathrm{Vars}'$ and $\mathit{second} : \mathrm{Vars} \to \mathrm{Vars}''$ where Vars' and Vars'' are disjoint "copies" of Vars, that intuitively contain the "primed" and "seconded" versions of the variables of Vars. We write x' as a shorthand for $\mathit{prime}(x)$, and x'' for $\mathit{second}(x)$, and use the same convention as in [14], *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. This allows us to represent any pair (s_1, s_2) of stores as a single store $s_1 \cup s_2'$. We use this encoding to transform any relational domain that represents a set of stores into a domain that represents a binary relation over stores.

Definition 17 (Relational lifting). *Let* A *be an abstract domain, such that for any typing context* Γ, $\mathrm{A}(\Gamma)$ *is equipped with concretisation function* $\gamma^{\mathrm{A}(\Gamma)} \in \mathrm{A}(\Gamma) \to \mathcal{P}(\mathrm{Vars} \to \mathrm{Values})$. *For any two typing contexts* Γ_1 *and* Γ_2, *the relational lifting* $\mathrm{A} \uparrow \mathrm{R}(\Gamma_1, \Gamma_2)$ *of* A *and its concretisation function are defined as follows:*

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

The relational lifting expects two typing contexts—one for the input stores, and one for the output stores. This flexibility will prove useful in §5.3 to define the abstract relational composition in order to analyse function calls.

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

As we remarked in § 4.1, only two pieces are missing to get a relational input-output analysis: now that we can express relations on stores, the question remains of how to express the transfer functions for conditionals and assignments. We show in Fig. 3 how to do so in a generic way, by exploiting the transfer functions of the underlying domain.

$$\begin{aligned} \mathrm{Cond}^{\mathrm{A}\uparrow\mathrm{R}(\Gamma_1,\Gamma_2)}(b)(a) &= \mathrm{Cond}^{\mathrm{A}(\Gamma_1\cup\Gamma_2')}(b')(a) \\ \mathrm{Assign}^{\mathrm{A}\uparrow\mathrm{R}(\Gamma_1,\Gamma_2)}(x := t)(a) &= \mathrm{Rem}^{\mathrm{A}(\Gamma)}_{\Gamma_2''} \begin{pmatrix} \mathrm{Assign}^{\mathrm{A}(\Gamma)}(x' := t'') \left(\mathrm{Add}^{\mathrm{A}(\Gamma_1\cup\Gamma_2'')}_{\Gamma_2'} c\right) \\ \sqcap^{\mathrm{A}(\Gamma)} \prod^{\mathrm{A}(\Gamma)}_{y\in\mathrm{dom}\,\Gamma\setminus\{x\}} \mathrm{Cond}^{\mathrm{A}(\Gamma)}(y'' = y') \end{pmatrix} \\ &\quad \text{where } \Gamma = \Gamma_1 \cup \Gamma_2' \cup \Gamma_2'' \text{ and } c = \underset{second \circ prime^{-1}}{\mathrm{Rename}}(a) \end{aligned}$$

Fig. 3: Relational transfer functions for conditionals and assignment.

The transfer function for conditionals $\mathrm{Cond}^{\mathrm{A}\uparrow\mathrm{R}(\Gamma_1,\Gamma_2)}(b)(a)$ 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 assignment $\mathrm{Assign}^{\mathrm{A}\uparrow\mathrm{R}(\Gamma_1,\Gamma_2)}(x := t)(a)$ first renames the output variables y' of a into y''. The variables y'' belong to the state that lies just *before* the assignment. Then, we call the transfer function for assignment from the underlying domain for the command $x' := t''$. This has the effect of extending a with relations that express the link between t and the new variable x. Then, we add the equalities $y'' = y'$ for all the variables other than x, because none of them was modified by the assignment. Finally, we eliminate the auxiliary variables y''. This effectively builds a relation between the input state and the state that is obtained *after* the assignment.

This concludes our justification of the folklore claim that, *"to turn a static analysis for the sets of final states into an analysis for input-output relations, it suffices to duplicate variables"*. We have built our justification on the following remarks: 1. Duplicating variables turns a non input-output relational domain—*i.e.*, a relation between variables of the stores of a *single* program point—into a

```
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. 4: Result of our analysis on the example of Fig. 1. Ellipses mark information that is also present in other components of the same case and is elided.

domain of binary relations between stores of *two* different program points. 2. An input-output relational analysis has the same structure as an analysis for final states. 3. The transfer functions that are specific to the input-output relational analysis can be defined in a generic way, using those of the analysis for final states.

In the rest of this article, we use the relational lifting of the abstract domain from §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 §3 and 4, to analyse the language described in §2.2. After providing an example that illustrates what the analysis computes (§5.1), we first describe an intra-procedural analysis (§5.2) and then extend it to support function calls, yielding a modular, summary-based, inter-procedural analysis (§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. Fig. 4 shows the result of running our analyser on the example from §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 $\mathtt{p.msg} = \mathtt{p}'\mathtt{.msg}$ 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 Analyze 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 $\mathrm{Analyze}(c)(a)$ that over-approximates the effect of running c *after* a. This section deals with basic constructs, while §5.3 explains how we analyse functions.

Definition 18 (Intra-procedural version of the analysis function).

$$\begin{aligned}
\mathrm{Analyze}(\mathsf{assert}(b))(a) &= \mathrm{Cond}^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}(b)(a)\\
\mathrm{Analyze}(x := t)(a) &= \mathrm{Assign}^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}(x := t)(a)\\
\mathrm{Analyze}(c_1\,;c_2)(a) &= \mathrm{Analyze}(c_2)\,(\mathrm{Analyze}(c_1)(a))\\
\mathrm{Analyze}(\mathsf{branch}\ c_1\ \mathsf{or}\ \ldots\ \mathsf{or}\ c_n\ \mathsf{end})(a) &= \textstyle\bigsqcup_{i\in 1,\ldots,n}^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)} \mathrm{Analyze}(c_i)(a)\\
\mathrm{Analyze}(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end})(a) &= \mathrm{Cond}^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}(\neg b)\,(\lim_{n\to\infty} a_n)\\
\textit{where } a_0 = a \textit{ and } a_{n+1} &= a_n \,\triangledown^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}\, \mathrm{Analyze}(\mathsf{assert}(b); c)(a_n)
\end{aligned}$$

Assertion and assignment use the transfer functions we built in previous sections. Sequence and branching follow the structure outlined in Lem. 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* [40, p.131] of the first iteration, so as to obtain better precision.

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

Theorem 1 (Soundness w.r.t. the collecting semantics). *For any command c and abstract value $a \in \mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)$,*

$$\mathbb{P}\,[\![c]\!]\left(\gamma^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}(a)\right) \subseteq \gamma^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}\,(\mathrm{Analyze}(c)(a))$$

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

Corollary 1 (Soundness w.r.t. the relational semantics). *For any command* c, $\mathbb{S}[\![c]\!] \subseteq \gamma^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}\left(\mathrm{Analyze}(c)\left(\mathrm{Id}^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}\right)\right)$.

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 §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:

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

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

The operational semantics is extended in a standard manner to support function calls and returns, by augmenting states with a call stack. The new rules are given in the extended version [4].

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 [11].

Definition 19 (Function summaries). *For a function* f *defined by* $\Delta(f) = ((x_1, \ldots, x_n), c_f, y_f, \Gamma_f)$*, we call* summary *of* f *the quadruplet given by:*

$$\left((x_1, \ldots, x_n), \mathrm{Analyze}(c_f)\left(\mathrm{Id}^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma_f)}\right), y_f, \Gamma_f\right)$$

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: no information about the caller's environment is recorded in the summary. This abstract value

is obtained by analysing the body of f, starting with the identity relation. This means that we make no assumption on the actual arguments that will be given to f, hence we can reuse the *same* summary in *every* calling context.

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

Definition 20 (Abstract composition). *Let Γ_1, Γ_2 and Γ_3 be typing contexts. Let $a_1 \in \mathrm{A} \uparrow \mathrm{R}(\Gamma_1, \Gamma_2)$ and $a_2 \in \mathrm{A} \uparrow \mathrm{R}(\Gamma_2, \Gamma_3)$ be two abstract values. The abstract composition $a_1 \mathbin{;^{\mathrm{A}}} a_2$ of the abstract values a_1 and a_2 is defined by:*

$$a_1 \mathbin{;^{\mathrm{A}}} a_2 = \operatorname*{Remove}_{\Gamma_2''} \left(\operatorname*{Add}_{\Gamma_3'} c_1 \sqcap^{\mathrm{A}(\Gamma_1 \cup \Gamma_2'' \cup \Gamma_3')} \operatorname*{Add}_{\Gamma_1} c_2 \right)$$

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

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 2 (Soundness of composition). *Let $a_1 \in \mathrm{A} \uparrow \mathrm{R}(\Gamma_1, \Gamma_2)$ and $a_2 \in \mathrm{A} \uparrow \mathrm{R}(\Gamma_2, \Gamma_3)$ be two abstract values. We have:*

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

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

Definition 21 (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:*

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

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:

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

The transfer function for function calls is sound:

Lemma 3 (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 $\mathbb{S}[\![c_f]\!] \subseteq \gamma^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma_f,\Gamma_f)}(a_f)$, we have:*

$$\mathbb{P}[\![y{:=}f(z_1, \ldots, z_n)]\!](\gamma^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}(a)) \subseteq \gamma^{\mathrm{D}\uparrow\mathrm{S}\uparrow\mathrm{R}(\Gamma,\Gamma)}(\mathrm{Analyze}(y{:=}f(z_1, \ldots, z_n))(a))$$

Lem. 3 ensures that the soundness result for the intra-procedural analysis (Thm. 1) extends to the language with function calls that we have described in this section. We give in the extended version [4] a proof of Lem. 3.

6 Implementation, Experimental Results and Complexity

We have implemented our analyser in approximately 5000 lines of OCaml. Our implementation together with instructions on how to add new test cases and run the tests cases is packaged and published as a virtual machine artefact [3]. Similarly to our formal development, our analyser is parametrised by an abstract domain for integers. A command-line option allows to choose among numeric domains provided by Apron [24], such as intervals, octagons or polyhedra.

We have tested our analyser on a total of 43 programs (summarised on Table 1), that comprise some complex examples: some sorting algorithms, the `do_ticks` function from §2.3, and 6 functions inspired from the abstract specification of the seL4 micro-kernel [29]. 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 language and in our abstract domain, we modelled arrays of fixed length using tuples, and we defined `get` and `set` functions. With this encoding, we wrote several sorting algorithms for arrays of integers, for arrays of size 5. The analyser could not infer that the output array was sorted. Still, it was able to infer that the sum of the values of the array was preserved by the sorting function.

The `do_ticks` function. The `do_ticks` function (§2.3) is inspired from a process scheduler from operating system code. As reported in §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 micro-kernel [29] 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 was easy to interpret, but large—it involved 324 disjunctions.

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 [12,13,21,41], 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.

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

Table 1: Test cases used for experimental evaluation. We use the ∗ symbol for families of similar tests, whose names start identically. The columns indicate whether the tests involve **sum** types, **numeric** operations, while **loops** or function **calls**, as well as the analysis **time**, and the maximum number of **cases** per function summary. Analysis times are given in milliseconds, with the exception of longer durations, that are given in seconds and printed with a bold face. Measures were performed on an Intel® Core™ i7 @2.30GHz × 16. The accompanying artefact [3] includes instructions to reproduce the results.

Name	Sums	Numeric	Loops	Calls	Time	Cases
Hand-crafted tests:						
`do_ticks`	Yes	Yes	Yes	No	166 ms	3
`nondeterministic_bubble_sort`	Yes	Yes	Yes	Yes	**2.1 s**	5
`selection_sort`	Yes	Yes	Yes	Yes	**10.9 s**	25
Inspired from SeL4:						
`decode_set_priority`	Yes	Yes	No	Yes	10 ms	2
`mask_cap_boolean`	Yes	No	No	Yes	**7.4 s**	324
`mask_cap_int`	Yes	Yes	No	Yes	**1.5 s**	4
`timer_tick_scheduling`	Yes	Yes	Yes	Yes	**41.2 s**	81
Simple tests:						
`assert*`	Yes	Yes	No	No	1 ms	1
`call_inside_loop_*`	No	Yes	Yes	Yes	15 ms	1
`drift`	Yes	Yes	Yes	Yes	24 ms	2
`exchange`	No	No	No	No	2 ms	1
`facto*`	No	Yes	Yes	No	8 ms	1
`false_type_collision`	No	No	No	Yes	3 ms	1
`fibonacci`	No	Yes	Yes	Yes	51 ms	1
`gauss*`	No	Yes	Yes	No	15 ms	1
`ghost_equality`	No	No	No	No	< 1 ms	1
`hidden_incompat`	Yes	No	No	No	2 ms	0
`id`	No	No	No	No	< 1 ms	1
`if`	No	Yes	No	No	2 ms	1
`incompat`	Yes	No	No	No	< 1 ms	0
`indirect_swap`	Yes	No	No	Yes	3 ms	2
`long_id`	Yes	No	No	Yes	5 ms	2
`modulo`	No	Yes	Yes	Yes	33 ms	2
`multiplication_larger`	No	Yes	No	No	2 ms	1
`or_constructor`	Yes	No	No	No	< 1 ms	0
`plus_*`	Yes	Yes	No	No	< 1 ms	1
`record_assignment*`	No	Yes	No	No	2 ms	1
`reduction`	No	Yes	No	No	3 ms	1
`struct_exchange`	Yes	No	No	No	< 1 ms	1
`swap`	Yes	No	No	No	< 1 ms	2
`test_loop`	No	Yes	Yes	No	3 ms	1
`two_by_two`	No	Yes	Yes	No	3 ms	1
`while_true`	No	No	Yes	No	< 1 ms	0
`widening_convergence`	No	Yes	Yes	No	49 ms	1
`xor`	Yes	No	No	Yes	8 ms	3

7 Related Work

The idea of exploiting an input-output *relational* semantics to verify `while` programs was developed by Kozen [31]. He introduced Kleene Algebra with Tests, an extension of relation algebra [44] 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 [11] used abstract interpretation for designing modular and relational analyses, and argue that compositionality can improve the scalability of analysers. Compositional Recurrence Analysis (CRA) [14] 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 Def. 15. 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 Def. 16, 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 [28] 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 §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 (§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 [1]. 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. We published a preliminary version of our approach in [2]. In this previous work, the analysis was not input-output

relational, since it 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 (§3.5) and of the relational lifting (§4.2).

Several relational analyses were developed for the inter-procedural analysis of numeric programs [5, 23, 36, 42], and in the context of inter-procedural shape analysis [20,22,43]. 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 [6]. It is defined in a modular fashion, based on the cofibered abstract domain [45]. 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 [7] 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 [16, 17] 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, [25, 26] 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.

As a particular case of algebraic values, the analysis of programs with *optional* numerical values was handled in [33] 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 products of the sizes of types used in a program. Other works have used *silhouettes* [32]—abstractions of the shapes of the abstract values—to control disjunctions. Following [27], 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 [27]. 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.

8 Conclusion and Future Work

In the context of programs that combine arithmetic operations and algebraic data types, we have shown how to construct an abstract domain that extends *any* abstract domain for numeric relations into an abstract domain for relations between algebraic values. The main idea is to consider extended variables—*i.e.*, a variable, and an access path—as the variables used 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—and can be exploited in a static analyser.

We have given a formal justification to the folklore result of static analysis that *"an analysis can be made relational by duplicating variables"*, by effectively turning a non input-output relational analysis into an input-output relational one. 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.

Finally, we have exploited our abstract domain to design and implement [3] 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*.

Further work will address the challenging problem of handling *recursive* algebraic data types and functional arrays. 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 [26]. 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.

Finally, we intend 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 [15]. Previous work [1] have indeed demonstrated that a large number of proof obligations could be discharged automatically in such a way, and could alleviate the verification of large programs.

References

1. Andreescu, O.F., Jensen, T., Lescuyer, S., Montagu, B.: Inferring frame conditions with static correlation analysis. POPL (2019). https://doi.org/10.1145/3290360
2. Bautista, S., Jensen, T., Montagu, B.: Numeric domains meet algebraic data types. NSAD (2020). https://doi.org/10.1145/3427762.3430178

3. Bautista, S., Jensen, T., Montagu, B.: Artifact for the "Lifting Numeric Relational Domains to Algebraic Data Types" article of the SAS 2022 symposium (2022). https://doi.org/10.5281/zenodo.6977156
4. Bautista, S., Jensen, T., Montagu, B.: Lifting Numeric Relational Domains to Algebraic Data Types (extended version) (2022), https://hal.inria.fr/hal-03765357
5. Boutonnet, R., Halbwachs, N.: Disjunctive relational abstract interpretation for interprocedural program analysis. VMCAI (2019). https://doi.org/10.1007/978-3-030-11245-5_7
6. Chang, B.Y.E., Rival, X.: Modular construction of shape-numeric analyzers. Festschrift for Dave Schmidt (2013), https://hal.inria.fr/hal-00926948
7. 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
8. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation (extended abstract). MFPS (1997). https://doi.org/10.1016/s1571-0661(05)80168-9
9. Cousot, P.: Principles of Abstract Interpretation. The MIT Press (2021)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. POPL (1977). https://doi.org/10.1145/512950.512973
11. Cousot, P., Cousot, R.: Modular static program analysis. CC (2002). https://doi.org/10.1007/3-540-45937-5_13
12. Dimovski, A.S.: Lifted static analysis using a binary decision diagram abstract domain. GPCE (2019). https://doi.org/10.1145/3357765.3359518
13. 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
14. Farzan, A., Kincaid, Z.: Compositional Recurrence Analysis. FMCAD (2015). https://doi.org/10.1109/FMCAD.2015.7542253
15. Filliâtre, J.C., Paskevich, A.: Why3 — Where programs meet provers. ESOP (2013). https://doi.org/10.1007/978-3-642-37036-6_8
16. Genet, T., Le Gall, T., Legay, A., Murat, V.: A completion algorithm for lattice tree automata. CIAA (2013). https://doi.org/10.1007/978-3-642-39274-0_13
17. Genet, T., Le Gall, T., Legay, A., Murat, V.: Tree regular model checking for lattice-based automata. CIAA (2013), https://hal.inria.fr/hal-00924849
18. Haudebourg, T., Genet, T., Jensen, T.P.: Regular language type inference with term rewriting. ICFP (2020). https://doi.org/10.1145/3408994
19. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. NASA Formal Methods (2017). https://doi.org/10.1007/978-3-319-57288-8_15
20. Illous, H., Lemerre, M., Rival, X.: Interprocedural shape analysis using separation logic-based transformer summaries. SAS (2020). https://doi.org/10.1007/978-3-030-65474-0_12
21. Jeannet, B.: The BDDAPRON logico-numerical abstract domains library. https://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/ (2009)
22. Jeannet, B.: Relational interprocedural verification of concurrent programs. Software & Systems Modeling **12** (2013). https://doi.org/10.1007/s10270-012-0230-7
23. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. SAS (2004). https://doi.org/10.1007/978-3-540-27864-1_19
24. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. CAV (2009). https://doi.org/10.1007/978-3-642-02658-4_52

25. Journault, M.: Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference. (Analyse statique modulaire précise par interprétation abstraite pour la preuve automatique de correction de programmes et pour l'inférence de contrats.). Ph.D. thesis, Sorbonne University, France (2019), https://tel.archives-ouvertes.fr/tel-02947214
26. Journault, M., Miné, A., Ouadjaout, A.: An abstract domain for trees with numeric relations. ESOP (2019). https://doi.org/10.1007/978-3-030-17184-1_26
27. Kim, S., Rival, X., Ryu, S.: A theoretical foundation of sensitivity in an abstract interpretation framework. TOPLAS (2018). https://doi.org/10.1145/3230624
28. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.: Compositional recurrence analysis revisited. PLDI (2017). https://doi.org/10.1145/3062341.3062373
29. 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. SOSP (2009). https://doi.org/10.1145/1629575.1629596
30. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. POPL (2010). https://doi.org/10.1145/1706299.1706355
31. Kozen, D.: Kleene Algebra with Tests. TOPLAS (1997). https://doi.org/10.1145/256167.256195
32. Li, H., Berenger, F., Evan Chang, B., Rival, X.: Semantic-directed clumping of disjunctive abstract states. POPL (2017). https://doi.org/10.1145/3009837.3009881
33. Liu, J., Rival, X.: Abstraction of optional numerical values. APLAS (2015). https://doi.org/10.1007/978-3-319-26529-2_9
34. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19** (2006). https://doi.org/10.1007/s10990-006-8609-1
35. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Found. Trends Program. Lang. **4** (2017). https://doi.org/10.1561/2500000034
36. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. TOPLAS (2007). https://doi.org/10.1145/1275497.1275504
37. Ong, C.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. POPL (2011). https://doi.org/10.1145/1926385.1926453
38. Pierce, B.: Advanced topics in types and programming languages. MIT Press, Cambridge, Mass. (2005)
39. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, Mass. (2002)
40. Rival, X., Yi, K.: Introduction to static analysis: an abstract interpretation perspective. The MIT Press (2020)
41. Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. SAS (2011). https://doi.org/10.1007/978-3-642-23702-7_19
42. 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
43. Sotin, P., Jeannet, B.: Precise interprocedural analysis in the presence of pointers to the stack. ESOP @ ETAPS (2011). https://doi.org/10.1007/978-3-642-19718-5_24
44. Tarski, A.: On the calculus of relations. Journal of Symbolic Logic **6** (Sep 1941). https://doi.org/10.2307/2268577
45. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. SAS (1996). https://doi.org/10.1007/3-540-61739-6_53