Compositional Static Value Analysis for Higher-Order Numerical Programs

Milla Valnet
LIP6, Sorbonne Université, F-75005, Paris, France

Raphaël Monat
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Antoine Miné
LIP6, Sorbonne Université, F-75005, Paris, France

Abstract

Static analyzers have been successfully developed to detect runtime errors in many languages. However, the automatic analysis of functional languages remains a challenge due to their recursive functions, recursive algebraic data types, and higher-order functions. Classic type systems provide compositional methods that are in general not precise enough to prove the absence of runtime errors such as assertion failures. At the other end of the spectrum, deductive methods are more expressive but may require user guidance to prove invariants.

Our work describes a static value analysis by abstract interpretation for a higher-order pure functional language. This analysis provides a sound and automatic approach to discover invariants and prevent assertion and match failures. We have designed a compositional analysis: functions are analyzed only once, at their definition site, generating a summary of their behavior. The summaries can be viewed as input-output relations expressed with relational abstract domains. We present two new abstract domains. A first abstract domain summarizes recursive algebraic data types. A second abstract domain lifts existing disjunctive relational summaries to higher-order by formalizing them as domains able to abstract higher-order functions. Both abstractions are parameterized by the abstractions of basic types (strings, integers, ...). Thanks to this parametric nature, both domains can be combined, allowing the analysis of higher-order functions manipulating algebraic data types and, conversely, algebraic data types using functions as first-class values.

We have implemented this analysis in the open-source MOPSA platform. Preliminary evaluation confirms the precision of our approach on a set of 40 handwritten toy programs as well as 20 programs from the state-of-the-art Salto analyzer benchmark.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Theory of computation → Program analysis; Software and its engineering → Functional languages

Keywords and phrases Static Value Analysis, Functional Programming, Abstract Interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.32

Related Version *Full Version*: `https://hal.science/hal-05047369`

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.5`

1 Introduction

Thanks to strong static typing, functional programming languages such as OCaml are safer than traditional imperative languages, as they "reduce the number of runtime errors [...] found in Java and C#" [29]. Still, well-typed functional programs can occasionally go wrong upon encountering runtime errors, such as out-of-bounds array accesses, arithmetic overflows, non-exhaustive pattern matching or uncaught exceptions. In this work, we aim at statically detecting such errors by developing a *compositional static value analysis* and targeting higher-order pure functional languages. Our approach is rooted in the abstract interpretation

framework, and leverages *relational abstract domains* to yield a high expressivity. In particular, we explore a new compositional value analysis that can yield better precision than classic type systems. The analysis is fully automated, contrary to more expressive approaches traditionally based on deductive verification.

Value analyses based on abstract interpretation have been mostly developed to target imperative programming languages [6, 9]. An ongoing research direction focuses on developing *compositional* (or function-modular) analyses. These analyses consist in analyzing functions once, at their definition site, to infer a summary; the summary is then applied to analyze each function call. These methods significantly improve scalability when a function is called multiple times, as summary applications are computationally cheaper than reanalyzing a function at every calling context. One difficulty faced by compositional analyses is precision loss: analyses need to produce a unique summary, correct for every possible input of the analyzed function. A key ingredient to achieve interesting precision levels is to rely on *relational abstract domains* which are able to express relationships between variables. Relational domains have been used extensively in previous works to create input-output relational summaries [26, 14, 17, 25]. However, relational domains are not always sufficient to make up for precision loss, for example when very distinct function behaviors are approximated as a single input-output relation. A second ingredient is the addition of partitioning [30, 7] to create a set of separated summaries for each function, each summary expressing different results on different inputs. Boutonnet and Halbwachs [8] developed a method to compute disjunctive relational summaries that express precise properties of first-order, numeric functions.

The static analysis of functional programs raises new challenges: analyzers need to handle features such as user-defined algebraic data types (ADTs), higher-order functions, partial application, recursive functions and polymorphism. This work presents a compositional analysis which leverages relational abstract domains, exploring a new precision-scalability tradeoff. The analysis supports a pure, monomorphic subset of OCaml, including recursive ADTs, as well as both recursive and higher-order functions.

We present in Figure 1 a motivating example. This program defines a custom algebraic data type, with one constructor containing functions. `to_fun` is a higher-order function, as its result is a function. The definitions of values `f1` and `f2` are performed through partial application. In the case of the program in Figure 1, our approach analyzes function `to_fun` once, at its definition site, and deduces a summary for this function. Under the hood, our compositional analysis relies on two new, key abstract domains: one abstracts user-defined algebraic data types, while the other abstracts higher-order functions. We rely on a cooperation between these two abstract domains to infer a precise, disjunctive [8], summary for `to_fun`. Handling partial applications is seamless, and our analysis is able to infer the following summaries for functions `f1` and `f2`: $\texttt{f1} : x \to 5, \texttt{f2} : x \to x + 4$. The analysis can then perform further applications to conclude that $\texttt{r1} = 5, \texttt{r2} = 9$ fully automatically. Note that both abstract domains can leverage arbitrary abstractions for leaf data types (integers, strings, ...). In particular, the summary of `f2` is expressed precisely thanks to the relational polyhedra domain [15].

In previous work, Lermusiaux and Montagu [27] implemented a value analysis for a large subset of OCaml, which includes recursive ADTs and higher-order constructs. However, this analysis is neither compositional nor relational. This simplifies the analysis of higher-order functions, which can be triggered by analyzing functions only when their parameters are known, at call site. In our example, a non-compositional analysis would analyze `to_fun` twice, when computing the abstract values of `r1` and `r2`. Bautista et al. [5] defined a compositional, relational analysis for a toy imperative language extended with non-recursive ADTs. In particular, their approach does not support recursive functions, nor higher-order ones.

```
type cstorfun = Cst of int | Fun of (int -> int)
let to_fun a =
  match a with
  | Cst n -> fun x -> n
  | Fun f -> f

let f1 = to_fun (Cst 5)
let f2 = to_fun (Fun (fun x -> x + 4))
let r1 = f1 4
let r2 = f2 5
```

Figure 1 Motivating OCaml program with key features from functional programming paradigm.

Contributions. This paper defines the following contributions:

- We introduce a domain summarizing recursive algebraic data types (ADTs). This domain is fully parametric in the abstraction of its base types (integers, strings, ...). Our domain supports relational abstractions.
- Independently, we define an abstract domain for higher-order functions, lifting previous work on disjunctive relational summaries [8]. This allows the creation of a compositional analysis for higher-order functions.
- We illustrate how the two abstract domains can mutually benefit from each other on an example, and provide a soundness statement about the overall analysis.
- We have implemented these domains in the MOPSA platform [22]. Thanks to this implementation, a preliminary evaluation confirms the precision of our approach on a set of 60 toy OCaml programs (handwritten or from previous work [27]).

Outline. Section 2 presents the syntax of a functional language with standard semantics. Section 3 provides background on compositional, relational analyses for the first-order case. Section 4 proposes an abstract domain for recursive algebraic data types, and Section 5 presents our method to analyze higher-order functions. Section 6 highlights the cooperation achieved by combining both abstract domains thanks to their parametricity. Section 7 presents our implementation, benchmarks, and experimental results. Section 8 presents related work and Section 9 concludes.

2 Syntax of the Considered Functional Language

In this section, we present the syntax of a generic functional language, on which we will define our analysis. Its key features are recursion, algebraic data types, and higher-order. We limit ourselves to a monomorphic higher-order language without side-effects. This language is described in Figure 2. $\mathbb{V}$ is the set of variables, $\oplus$ any operator of the language ($+$, $-$, $\times$, $/$, $=$, etc.). $\mathcal{E}$ is the set of expressions. $fv(e)$ is the set of free variables of expression e. Π, the set of patterns, contains the wildcard _, variables, integers, `when` clauses, and type constructors. Free variables from the same pattern are required to be disjoint, as is the case in OCaml: the same variable cannot be bound twice. Figure 3 describes the semantic domain of the language. $\mathbb{C}$ is the set of all possible constructors. Semantic values of the language $\mathcal{V}$ are integers, type constructors containing values, error ω and continuous functions from values to values, to which we add $\perp$ (for non-termination). Finally, Σ is the set of concrete

$$
\begin{array}{rll}
\mathcal{E} ::= & x \in \mathbb{V} & \\
& \mid n \in \mathbb{Z} & \mid e_1 \oplus e_2 \\
& \mid e_0\ e_1\ \ldots\ e_n & \mid \text{fun } x_1 \ldots x_n \to e \\
& \mid \text{let } x = e_1 \text{ in } e_2 & \mid \text{let rec } x = e_1 \text{ in } e_2 \\
& \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \mid \text{assert } e \\
& \mid C(e_1, \ldots, e_n) & \mid \text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m \\
\Pi ::= & \_ \mid\ x \in \mathbb{V} \mid n \in \mathbb{Z} \mid p_1 \texttt{ when } e_1 \mid C(p_1, \ldots, p_n) & \\
& \text{where } \forall i \neq j, fv(p_i) \cap fv(p_j) = \emptyset &
\end{array}
$$

Figure 2 Syntax of the functional language.

$$\mathcal{V} ::= \mathbb{Z}^\perp \cup \{\ C(v_1, \ldots, v_n) \mid C \in \mathbb{C}, v_i \in \mathcal{V}\ \}^\perp \cup \{\omega\}^\perp \cup \bigcup_{n \in \mathbb{N}} [\mathcal{V}^n \to \mathcal{V}]^\perp$$

$$\Sigma = \mathbb{V} \to \mathcal{V}$$

$$\mathbb{E}[\![\cdot]\!] : \mathcal{E} \times \Sigma \to \mathcal{V}$$

Figure 3 Semantic domain of the functional language.

environments, associating a value to each variable. The concrete semantics of an expression e is $\mathbb{E}[\![e]\!] \in \Sigma \to \mathcal{V}$. It associates a semantic value to an expression in an environment. This semantics follows the standard for eagerly evaluated functional languages (such as OCaml). It is standard, and not detailed here.

We work under the hypothesis that programs are well-typed. The type inference works as usual on functional languages [20, 31]. This way, some situations where an expression evaluates to the error ω are statically detected. However, well-typedness is not enough to prevent all possible kinds of errors, such as assertion failures, or match failures in the presence of `when` clauses. Our analysis focuses on those. During the analysis, we use type information, as well as user-defined type definitions, to select relevant abstract domains.

To simplify presentation, `int` type represents mathematical integers: working on machine integers and statically detecting overflows would only require adding assertions when performing arithmetic operations. Similarly, even though the language is pure, we can still handle array bound checks with a simple translation of programs with arrays to array-less programs: array accesses are replaced with assertions that indices are within the array range, abstracting away the content of the array.

3 Compositional and Relational Analysis at First-Order

Since the concrete semantics is uncomputable, we propose an abstract semantics for our language. This section provides background on relational analyses, and compositional interprocedural analyses for first-order programs with integers and pairs. We start by introducing an abstract semantics compatible with numerical relational analyses through the introduction of symbolic expressions (Section 3.1), and then describe how this abstract semantics can be lifted to perform compositional analysis of recursive functions (Section 3.2).

The analyses we describe feature two notable strengths: their definition is *domain-modular* – i.e. fully parametric in the underlying abstract domains – and supports precise *relational* inference – allowing us to express relationships between variables. We believe that this approach, originally formulated by the authors of the MOPSA framework [22], provides greater expressivity than traditional static analysis definitions. However, this generic and domain-modular approach requires additional formalization efforts to define domains.

3.1 Relational Analysis

We define a computable abstract semantics $\mathbb{E}^\sharp[\![e]\!]$ of an expression e, based on abstract domains. It delegates the abstraction of an object of type τ (e.g. integers, pairs, and after Section 4, ADTs) to the abstract domain $\mathcal{D}_\tau$. We denote the union of those domains as $\mathcal{D}$. Usually, such a semantics takes as parameter an expression e and an abstract environment (abstracting the possible values of all the variables), and returns an abstract value representing the set of possible values of the expression (in the abstract domain corresponding to the type of the expression). However, this formulation cannot easily express a relationship between the value of the expression and those of the variables. Indeed, let us consider the program in Figure 4. A numerical domain such as the interval domain can abstract `x` and `y` independently, as $\texttt{x} \in [1, 10]$ and $\texttt{y} \in [2, 11]$, but cannot infer the relation `y = x + 1`. To overcome this limitation, we use an abstract numerical domain able to store relations between variables – e.g. the polyhedra domain [15]. However, expressing relations between numeric variables from the program, that is, `x` and `y`, is not sufficient to leverage those relational information to `pair`, since we have no way to express a relation between the second field of `pair` and `y`.

Following the works of Chevalier and Feret [10], Journault et al. [22], we introduce *intermediate variables* (also named ghost variables), in addition to the initial program variables. Here, we introduce two intermediate variables p_1 and p_2, which respectively represent the first and second fields of `pair`. In practice, we keep separate the set of program variables $\mathbb{V}$ and the set $\mathbb{V}_\mathbb{Z}$ of intermediate variables on which a (possibly relational) numerical domain will express semantic constraints. We then keep a map from program variables in $\mathbb{V}$ to their *abstract values* in $\mathcal{V}^\sharp$, which can either be an element of an abstract domain, or an intermediate variable. In our example, $\mathbb{V}_\mathbb{Z}$ is $\{x, y, p_1, p_2\}$ whereas $\mathbb{V} = \{\texttt{x}, \texttt{y}, \texttt{pair}\}$. The map is then $m = \texttt{x} \to x, \texttt{y} \to y, \texttt{pair} \to (p_1, p_2)$. The possible values of variables in $\mathbb{V}_\mathbb{Z}$ are then represented as an abstract element from a (possibly relational) abstract domain $\mathcal{D}_\mathbb{Z}$. If the chosen numerical domain $\mathcal{D}_\mathbb{Z}$ is the polyhedra domain – which can express conjunctions of linear constraints – we can infer: $d = 1 \leq x \leq 10 \wedge y = x + 1 \wedge p_1 = y \wedge 1 \leq p_2 \leq y \in \mathcal{D}_\mathbb{Z}$. Note that the relation between the second field of `pair` and `y` was not expressible without introducing p_2. Such intermediate variables will be especially useful when abstracting algebraic data types in Section 4.

An abstract environment in $\Sigma^\sharp$ is then the pair of a map $m \in \mathbb{V} \to \mathcal{V}^\sharp$ and an element of the possibly relational numerical domain $d \in \mathcal{D}_\mathbb{Z}$. Thus, $\sigma^\sharp = (m, d) \in \Sigma^\sharp$.

As a last step, we define *abstract expressions* $\mathcal{E}^\sharp$, a lifting of abstract values where numerical intermediate variables can instead be *numeric expressions* $\mathcal{E}_\mathbb{Z}$, that is, symbolic operations between intermediate variables and integers. The abstract semantics of an expression e is then: $\mathbb{E}^\sharp[\![e]\!] : \Sigma^\sharp \to \mathcal{E}^\sharp \times \Sigma^\sharp$. It evaluates an expression e in an environment and returns an abstract expression into a new environment. Let p be the program in Figure 4, and $\sigma_0^\sharp$ a starting environment with no information on program variables. The analysis of p is:

$$\mathbb{E}^\sharp[\![p]\!]\sigma_0^\sharp = (p_1, p_2), (\texttt{x} \to x, \texttt{y} \to y, \texttt{pair} \to (p_1, p_2), \\ 1 \leq x \leq 10 \wedge y = x + 1 \wedge p_1 = y \wedge 1 \leq p_2 \leq y)$$

```
1 let x = random 1 10 in
2 let y = x + 1 in
3 let pair = (y, random 1 y) in
4  pair
```

Figure 4 Example program motivating relational analysis.

Set of program variables	$\mathbb{V}$
Set of intermediate variables	$\mathbb{V}_{\mathbb{Z}}$
Union of non-numerical domains	$\mathcal{D}$
Numerical domain	$\mathcal{D}_{\mathbb{Z}}$
Abstract values	$\mathcal{V}^\sharp = \mathcal{D} \cup \mathbb{V}_{\mathbb{Z}}$
Abstract environments	$\Sigma^\sharp = (\mathbb{V} \to \mathcal{V}^\sharp) \times \mathcal{D}_{\mathbb{Z}}$
Concrete numerical environments	$\Sigma_{\mathbb{Z}} = \mathbb{V}_{\mathbb{Z}} \to \mathbb{Z}$
Numeric expressions	$\mathcal{E}_{\mathbb{Z}} ::= n \in \mathbb{Z} \mid v \in \mathbb{V}_{\mathbb{Z}} \mid e_1 \oplus e_2,$ where $e_1, e_2 \in \mathcal{E}_{\mathbb{Z}}$
Abstract expressions	$\mathcal{E}^\sharp = \mathcal{D} \cup \mathcal{E}_{\mathbb{Z}}$
Abstract semantics of expr. e	$\mathbb{E}^\sharp[\![e]\!] : \Sigma^\sharp \to \mathcal{E}^\sharp \times \Sigma^\sharp$

Figure 5 Formalization of the abstract semantics.

The abstract expression (p_1, p_2) is a symbolic pair of intermediate variables $p_1, p_2 \in \mathbb{V}_{\mathbb{Z}}$, therefore $(p_1, p_2) \in \mathcal{E}_{\mathbb{Z}}$. Figure 5 summarizes all abstract sets.

Assignments. Our abstract environments are made of two components $(m, d) \in \Sigma^\sharp$: a map m from variables to abstract values and an element d of the relational domain. To define assignments in such an environment, we write:

- $m[\mathtt{x} \to v]$ the map $m \in (\mathbb{V} \to \mathcal{V}^\sharp)$ where variable $\mathtt{x} \in \mathbb{V}$ is now bound to the value $v \in \mathcal{V}^\sharp$;
- $\mathbb{E}^\sharp_{\mathbb{Z}}[\![x \to v]\!]d \in \mathcal{D}_{\mathbb{Z}}$ the result of the assignment in an element $d \in \mathcal{D}_{\mathbb{Z}}$ of variable $x \in \mathbb{V}_{\mathbb{Z}}$ to a value or a numeric expression v.

▶ **Definition 3.1** (Abstract assignment). *The abstract assignment of a program variable* $\mathtt{x} \in \mathbb{V}$ *with abstract value* $v \in \mathcal{V}^\sharp$ *in environment* $(m, d) \in \Sigma^\sharp$ *is then:*

$$(m, d)[\mathtt{x} \to v] = \begin{cases} \left(m[\mathtt{x} \to x], \mathbb{E}^\sharp_{\mathbb{Z}}[\![x \to v]\!]d\right) & \text{if } x : \texttt{int} \\ (m[\mathtt{x} \to v], d) & \text{otherwise} \end{cases}$$

Intuitively, the assignment is delegated to the numerical domain for integer variables, and it is a simple map update otherwise.

Join. We denote the join between two environments as $\sqcup^\sharp$, which lifts the join operators from underlying components. The join is defined pointwise between two maps in $\mathbb{V} \to \mathcal{V}^\sharp$: each variable maps to a join of its abstract values from both maps in the relevant domain. We rely on the join of the numerical abstraction for the second component of the abstract environment. Note that this join can be *heterogeneous* [24], if both elements are not defined on the same set of variables (this will be further developed in Section 4). The meet $\sqcap^\sharp$ is defined similarly.

Concretization. The concretization of an abstract environment is based on the concretization $\gamma_{\mathbb{Z}}$ of the numerical domain, and on the concretizations γ_τ of the domains $\mathcal{D}_\tau$ abstracting objects of type τ. Those domains can be relational, so the concretization of a value can depend on other variables' concretization, following a construction defined by Monat [32]. Consequently, γ_τ takes as parameter an element of the domain $\mathcal{D}_\tau$ *together with a concrete numerical environment* $\sigma_{\mathbb{Z}} \in \Sigma_{\mathbb{Z}}$.

▶ **Definition 3.2** (Environment concretization). *Given* $\sigma^\sharp = (m, d) \in \Sigma^\sharp$*, its concretization is:*

$$\gamma_{\Sigma^\sharp}(m, d) = \{\sigma \in \Sigma \mid \exists \sigma_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d), \forall x \in \mathbb{V}, x : \tau, \sigma(x) \in \gamma_\tau(m(x), \sigma_{\mathbb{Z}})\}$$

This concretization thus depends on the concretizations for ground types (such as integers, strings, etc.), but also on the concretizations for functions and algebraic data types. Those will be defined in Section 4 and Section 5.

▶ Remark 3.3 (Implicit concretization signature). Even though the concretization γ_τ takes as parameter a *concrete environment*, we can lift this definition so that it takes as parameter an *abstract environment* by writing:

$$\gamma_\tau(e^\sharp, (m, d)) = \bigcup_{\sigma_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d)} \gamma_\tau(e^\sharp, \sigma_{\mathbb{Z}})$$

We simply join concretizations for every concrete numerical environment abstracted by d. Our abstract semantics takes as input an abstract environment and returns an abstract value in an abstract environment. With this notation, given $e : \tau$ and $\sigma^\sharp \in \Sigma^\sharp$, we can write the concretization of the abstract semantics $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp \in \mathcal{D}_\tau \times \Sigma^\sharp$, and we can write $\gamma_\tau(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$. In particular, this will be useful when stating the soundness theorem (Section 6.2).

Error propagation. To simplify the presentation, errors are implicitly propagated: if a sub-expression evaluates to the error ω, then so does the whole expression.

3.2 Compositional Function Analysis

Since an important contribution of this article is to design a *compositional* analysis, we first introduce how we analyze first-order functions in a compositional fashion, before extending our analysis in Section 5 to higher-order.

Compositional analysis aims at over-approximating all the possible behaviors of a function at definition site. The function is analyzed once and for all, and we then store the result of the analysis, its *summary*. Since the language is pure, the function does not modify the environment (no side-effects). Its behavior can then be represented by a relation between its inputs and its output, using a relational domain.

Let $\mathcal{F}^\tau$ be the set of functions of type $\tau = \tau_1 \to \cdots \to \tau_{n+1}$. Since we are restricted to first-order in this section, τ_i are base types such as integers or pairs. Given a set V of variables for inputs and output, we denote $\mathcal{R}(V)$ as the abstract domain chosen to represent relations between those. It abstracts $\Sigma|_V$, i.e. concrete environments restricted to V. It is provided with its concretization $\gamma_r : \mathcal{R}(V) \to \Sigma|_V$, with a lifting $\lambda_r : \mathcal{R}(V) \to \Sigma^\sharp$ and with a projection operator proj_r such that $\text{proj}_r(\sigma^\sharp) \in \mathcal{R}(V)$ only keeps information on variables in V.

We denote the domain chosen for functions of type τ as $\mathcal{D}_\tau = \mathbb{V}^n \times \mathbb{V} \times \mathcal{R}(V)$. This way, a function is abstracted as: the names $x_1, \ldots, x_n$ of its formal inputs, a result variable, and a relation between those variables.

▶ **Definition 3.4** (Functions concretization). *The concretization* $\gamma_\tau : \mathcal{D}_\tau \times \Sigma_\mathbb{Z} \to \mathcal{F}^\tau$ *is:*

$$\gamma_\tau((x_1,\dots,x_n,r,p),\_) = \{f : \tau \mid \forall(a_1,\dots,a_n) : \tau_1 \times \cdots \times \tau_n, \\ [x_1 = a_1] \cdots [x_n = a_n][r = f(a_1,\dots,a_n)] \in \gamma_r(p)\}$$

A function f is abstracted as $(x_1,\dots,x_n,r,p)$ if the relation between its inputs and its output can be described by p. $\_ \in \Sigma_\mathbb{Z}$ is not used by the concretization, as the summary relies on its own relational abstract element p.

We now describe the transfer functions used by the compositional analysis.

Function definition. Intuitively, the analysis of functions needs to generate a summary valid for all inputs. Thus, the body is analyzed with the most general inputs: the inputs are initialized at $\top$, the domain maximum (meaning we have no information on their values). We project the result on input and output variables. This way, we get the most general input-output relation, usable at any call site. Note that relationality is critical there, enabling us to discover relevant information on the function behavior despite making no hypothesis on the values of the inputs.

$$\mathbb{E}^\sharp[\![\text{fun } x_1 \cdots x_n \to \mathit{body}]\!]\sigma^\sharp = \text{let } e^\sharp, \sigma_0^\sharp = \mathbb{E}^\sharp[\![\mathit{body}]\!]\sigma^\sharp[x_1 \to \top,\dots,x_n \to \top] \text{ in} \\ ((x_1,\dots,x_n), r, \text{proj}_r(\sigma_0^\sharp[r \to e^\sharp])), \sigma^\sharp$$

Recursive functions. For recursive functions, the concrete semantics has to compute a least fixpoint of the function concrete semantics F:

$$\mathbb{E}[\![\text{let rec } f = e_1 \text{ in } e_2]\!]\sigma = \text{ let } F(v) = \mathbb{E}[\![e_1]\!]\sigma[f \to v] \text{ in } \mathbb{E}[\![e_2]\!]\sigma[f \to \text{lfp}(F)]$$

We use a standard technique from abstract interpretation [12] to obtain a computable abstract semantics, in three steps. First, thanks to Kleene's theorem, we rewrite the least fixpoint as the union of iterations of the analysis of the function, starting from $f \to \bot$, i.e. $\text{lfp}(F) = \bigcup_{k\in\mathbb{N}} F^k(\bot)$. Second, we move to the abstract semantics by using an iteration from $f \to \bot_\tau$, but evaluate the function body in the abstract domain with F. Third, we use a widening operator ∇_τ instead of a join to enforce the convergence in finite time. The widening operator is a component-wise lift, similar to the definition of the join operator on abstract environments in Section 3.1.

$$\mathbb{E}^\sharp[\![\text{let rec } f = e_1 \text{ in } e_2]\!]\sigma^\sharp = \text{let } \bot_\tau = ((x_1,\dots,x_n), r, [r \to \bot][x_i \to \top]_{i\le n}) \text{ in} \\ \text{let } F : \mathcal{D}_\tau \to \mathcal{D}_\tau = v^\sharp \to \text{fst}(\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp[f \to v^\sharp]) \text{ in} \\ \text{let } s = \nabla_\tau \bigcup_{k\in\mathbb{N}} F^k(\bot_\tau) \text{ in} \\ \mathbb{E}^\sharp[\![e_2]\!]\sigma^\sharp[f \to s]$$

This reasoning extends to mutually recursive functions by iterating on a vector of function abstractions.

Function application. When applying a function e_0 to inputs of abstract values $e_i^\sharp$, we intuitively combine the current environment and the summary relation p. Before the combination, both abstract environments need to be defined on the same set of variables. We rely on two auxiliary functions: `add_vars` : $\Sigma^\sharp \times \mathcal{P}(\mathbb{V}_\mathbb{Z}) \to \Sigma^\sharp$ which extends the definition domain of an abstract environment with a set of variables, and `dom` : $\Sigma^\sharp \to \mathcal{P}(\mathbb{V}_\mathbb{Z})$ which

provides the definition domain of an abstract environment. The combination also requires additional equality constraints $[x_i = e_i^\sharp]$ between formal arguments x_i and values $e_i^\sharp$. These equality constraints are delegated to the filter function[1] of the relevant abstract domain.

$$\begin{aligned}\mathbb{E}^\sharp[\![e_0 \ \dots \ e_n]\!]\sigma_0^\sharp = \ & \text{let } ((x_1,\dots,x_n),r,p),\sigma_0^\sharp = \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp \text{ in}\\ & \text{let } \forall i \in [\![0,n-1]\!] \ e_{i+1}^\sharp,\sigma_{i+1}^\sharp = \mathbb{E}^\sharp[\![e_{i+1}]\!]\sigma_i^\sharp \text{ in}\\ & \text{let } \sigma_n^\sharp = \texttt{add\_vars}(\sigma_n^\sharp,\texttt{dom}(p)) \text{ in}\\ & \text{let } p = \texttt{add\_vars}(\lambda_r(p),\texttt{dom}(\sigma_n^\sharp)) \text{ in}\\ & r,(\sigma_n^\sharp \sqcap^\sharp p)[x_1 = e_1^\sharp,\dots,x_n = e_n^\sharp]\end{aligned}$$

Compositionality. A key observation on our abstract semantics is that the function is analyzed at *definition site*. Consequently, we analyze it only once. At any call site, we only instantiate the resulting summary with input values. At first-order, Boutonnet and Halbwachs [8] proved that such a *compositional analysis* leads to a more efficient analysis for a function used at many call sites. Therefore, it can help the analysis to scale up.

Note that the purity hypothesis is necessary here: abstracting functions as input-output relations is correct only because the function does not modify the memory state. To handle side-effects, we would need to modify this abstraction to track mutability of the memory state, e.g., using extra input-output arguments to functions to model the mutable part of the state. Keeping compositionality while overcoming this limitation is left for future work.

In conclusion, this section defined a compositional analysis for first-order functions, both relational and fully parametric in the choice of abstract domains used to represent ground types. These features will be at the core of our abstractions for ADTs and higher-order functions.

4 Abstracting Recursive Algebraic Data Types

Algebraic objects are basic blocks of functional languages. Lists, trees, abstract syntax trees, etc. are defined using algebraic data types. To analyze them, we need a specific domain for objects of user-defined type:

$$\texttt{type } \tau = C_1 \text{ of } \tau_{1,1} * \cdots * \tau_{1,m_1} \mid \cdots \mid C_n \text{ of } \tau_{n,1} * \cdots * \tau_{n,m_n}$$

This section defines an abstraction for user-defined algebraic data types which is fully parametric in the abstraction of base types (integers, strings, etc). We illustrate our construction on integer base types, as these are very common, and many numerical abstractions already exist to serve as parameter. It also allows us to show that our analysis can be relational when parameterized with relational base domains. This relationality is a crucial factor to make compositional analyses precise, as exemplified in Section 4.3.

4.1 Parametric Relational Domain with Symbolic Variables

4.1.1 Domain

Let n be the number of constructors and m_i the number of fields of constructor C_i. Let $\mathbb{C}_\tau = \{C_i \mid 1 \leq \mathtt{i} \leq n\}$ be the set of constructors of type τ. Let $\mathbb{T}_\tau$ be the set of finite concrete objects of type τ. For recursive types, $\mathbb{T}_\tau$ can be infinite. To provide a computable

[1] The filter function models the effect of a test or a guard in the domain, by selecting environments satisfying a conditional – here, $[x_i = e_i^\sharp]$

abstract semantics, we chose to summarize (or "smash") together all the elements of a given field that are nested in an abstract data type. For instance, when abstracting a list of integers, a single abstract integer variable will be used to represent all the possible values of all the elements of the list. More generally, infinite sets of unbounded data-structures can be abstracted using a finite number of dimensions – here, one per field.

We construct systematically an abstraction for values of type τ from existing abstractions for values of types $\tau_{i,j}$ used in the definition of τ. However, in order to avoid cyclic definitions in the case of recursive types, we start from given domains $\mathcal{D}^\sharp_{\tau_{i,j}}$ defined only when $\tau_{i,j} \neq \tau$. The case of recursive types is well-founded, as is the case of user-defined types using fields of previously defined user-defined types. For simplicity, this definition omits the case of mutually recursive types, which does not pose additional theoretical issues.

▶ **Definition 4.1** (ADTs domain). *We derive domains $\mathcal{D}^\sharp_{i,j}$ used to abstract the value of the j-th field of the i-th constructor as follows:*

$$\mathcal{D}^\sharp_{i,j} = \begin{cases} \mathcal{P}(\mathbb{C}_\tau) & \text{if } \tau_{i,j} = \tau \\ \mathbb{V}_\mathbb{Z} & \text{if } \tau_{i,j} = \texttt{int} \\ \mathcal{D}^\sharp_{\tau_{i,j}} & \text{otherwise} \end{cases}$$

Finally, values of type τ are abstracted in $\mathcal{D}^\sharp_\tau$, defined as follows:

$$\mathcal{D}^\sharp_\tau = \prod_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m_i}} (\mathcal{D}^\sharp_{i,j})^\perp \times \mathcal{P}(\mathbb{C}_\tau)$$

Intuitively, $\mathcal{D}^\sharp_\tau$ abstracts an object x of type τ as $((g_{i,j})_{i,j}, \mathscr{C})$ such that:

- $g_{i,j} \in (\mathcal{D}^\sharp_{i,j})^\perp$ abstracts every j-th field of C_i nested in x;
- $\mathscr{C} \in \mathcal{P}(\mathbb{C}_\tau)$ is the set of possible constructors C_i for x.

$\mathcal{D}^\sharp_\tau$ delegates the abstraction of field (i,j) to $\mathcal{D}^\sharp_{i,j}$. If this field is recursive, $\mathcal{D}^\sharp_{i,j} = \mathcal{P}(\mathbb{C}_\tau)$, meaning that we only keep track of its possible start constructors. If this is an integer field, $\mathcal{D}^\sharp_{i,j} = \mathbb{V}_\mathbb{Z}$, meaning that we create a symbolic variable $x.i.j \in \mathbb{V}_\mathbb{Z}$, to represent every field (i,j) nested in x. The numerical domain can then infer relations between them. When the field is neither recursive nor an integer, we delegate the abstraction to the domain chosen to abstract objects of type $\tau_{i,j}$. In this case, $\mathcal{D}^\sharp_{i,j} = \mathcal{D}^\sharp_{\tau_{i,j}}$.

By smashing every constructor field in one summary variable, we use an abstraction of fixed size to represent unbounded objects of recursive type. This approach shares similarities with the approach of Gopan et al. [18]. They use a fixed number of dimensions to over-approximate numeric values in unbounded collections, to analyze e.g. dynamic arrays or heap-allocated structures. Summarization variables consequently abstract more than one object. Note that smashing is done at the variable allocation site: each variable of type τ defines a set of numerical variables – the abstraction is called *object-sensitive* [39].

▶ **Example 4.2.** Let us illustrate this abstraction on lists of integers, defined as:

```
type list = Cons of int * list | Nil
```

We use the interval domain (i.e., $\mathcal{D}_\mathbb{Z} = \mathbb{I}$) to abstract integers. The set of possible constructors is $\mathbb{C}_{\texttt{list}} = \{\texttt{Cons}, \texttt{Nil}\}$. The constructor `Cons` has two fields. The first one, containing integers, is represented by $\mathcal{D}^\sharp_{1,1} = \mathbb{V}_\mathbb{Z}$, that is, a numeric variable, abstracted in $\sigma^\sharp$ as an interval. The second, recursive, one is represented by $\mathcal{D}^\sharp_{1,2} = \mathcal{P}(\mathbb{C}_{\texttt{list}})$. The constructor `Nil` has no field to be represented. The global abstraction for lists of integers is then:

$$\mathcal{D}^\sharp_{\texttt{list}} = \left((\mathcal{D}^\sharp_{1,1})^\perp \times (\mathcal{D}^\sharp_{1,2})^\perp\right) \times \mathcal{P}(\mathbb{C}_{\texttt{list}}) = \left(\mathbb{V}^\perp_\mathbb{Z} \times \mathcal{P}(\mathbb{C}_{\texttt{list}})^\perp\right) \times \mathcal{P}(\mathbb{C}_{\texttt{list}})$$

4.1.2 Concretization

Let $\gamma_{\tau_{i,j}}$ be the concretization for $\mathcal{D}^\sharp_{\tau_{i,j}}$ when $\tau_{i,j} \neq \tau$. It is provided by the domain $\mathcal{D}^\sharp_{\tau_{i,j}}$.

▶ **Definition 4.3** (ADTs concretization). *We define* $\gamma_\tau : \mathcal{D}^\sharp_\tau \times \Sigma_\mathbb{Z} \to \mathcal{P}(\mathbb{T}_\tau)$. *For* $(g, \mathscr{C}) \in \mathcal{D}^\sharp_\tau$, $\sigma_\mathbb{Z} \in \Sigma_\mathbb{Z}$*:*

$$\gamma_\tau((g,\mathscr{C}),\sigma_\mathbb{Z}) = \left\{ x : \tau \;\middle|\; \exists i,\, x = C_i(x_{i,1},\ldots,x_{i,m_i}) \;\wedge\; C_i \in \mathscr{C} \;\wedge\; \forall j, x_{i,j} \in \begin{cases} \gamma_\tau((g,g_{i,j}),\sigma_\mathbb{Z}) & \text{if } \tau_{i,j} = \tau \\ \gamma_{\tau_{i,j}}(g_{i,j},\sigma_\mathbb{Z}) & \text{otherwise} \end{cases} \right\}$$

Since type τ is finite, recursion is well-founded ($x_{i,j}$ contains strictly less constructors than x). An object $x : \tau$ is abstracted as $(g, \mathscr{C})$ if:

- it starts with $C_i \in \mathscr{C}$,
- and every j-th field of C_i accessible in x
  - either starts with constructor in $g_{i,j} \subseteq \mathbb{C}_\tau$, in the recursive case ($\tau_{i,j} = \tau$),
  - or can be abstracted as $g_{i,j}$ otherwise.

$g_{i,j}$ represents every j-th field from C_i nested in x, while $\Sigma_\mathbb{Z}$ keeps track of numeric variables.

▶ **Example 4.4.** Following Example 4.2, we consider the integer list abstract value $(g, \mathscr{C}) = ((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$ with numerical environment $\sigma_\mathbb{Z} = [r \to [1, 10]]$. Its concretization $\gamma_{\texttt{list}}((g, \mathscr{C}), \sigma_\mathbb{Z})$ is the set of non-empty lists containing integers between 1 and 10:

$$\{\, x : \texttt{int list} \mid x = \texttt{Cons}(h, q) \wedge h \in [1, 10] \wedge q \in \gamma_{\texttt{list}}(((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma_\mathbb{Z})\}$$

Let us consider $l = \texttt{Cons}(1, \texttt{Cons}(4, \texttt{Cons}(10, \texttt{Nil})))$. We have $l \in \gamma_{\texttt{list}}((g, \mathscr{C}), \sigma_\mathbb{Z})$, meaning that $(g, \mathscr{C})$ is a correct over-approximation of l.

4.1.3 Lattice Operators

We now define lattice operators. Since our domain is relational, operators are defined on environments. As seen in Section 3, environments are pairs, composed of a map from program variables to objects from their abstract domains, together with an element from the numerical relational domain. We denote lattice operators for the numerical relational domain as $(\sqsubseteq_r, \cup_r, \cap_r, \nabla_r)$. We denote operators for $\mathcal{D}^\sharp_{i,j}$ as $(\sqsubseteq_{\mathcal{D}_{i,j}}, \cup_{\mathcal{D}_{i,j}}, \cap_{\mathcal{D}_{i,j}}, \nabla_{\mathcal{D}_{i,j}})$. We define an intermediate inclusion between two ADT abstractions, interpreted outside their abstract environment. We simply lift inclusions $\sqsubseteq_{\mathcal{D}_{i,j}}$.

▶ **Definition 4.5** (Environment-free inclusion). *For* $(g_1, \mathscr{C}_1)$, $(g_2, \mathscr{C}_2) \in \mathcal{D}^\sharp_\tau$ *abstract elements:*

$$(g_1,\mathscr{C}_1) \subseteq^\sharp_\tau (g_2,\mathscr{C}_2) \iff \mathscr{C}_1 \subseteq \mathscr{C}_2 \wedge \forall i,j, g^1_{i,j} \sqsubseteq_{\mathcal{D}_{i,j}} g^2_{i,j}$$

$(g_1, \mathscr{C}_1) \subseteq^\sharp_\tau (g_2, \mathscr{C}_2)$ then means that the non-numerical fields of $(g_1, \mathscr{C}_1)$ are more precise than those of $(g_2, \mathscr{C}_2)$. To compare numerical fields, we need to check inclusion on environments:

▶ **Definition 4.6** (Inclusion). *We define* $\sqsubseteq^\sharp : \Sigma^\sharp \to \Sigma^\sharp$*:*

$$(m_1, d_1) \sqsubseteq^\sharp (m_2, d_2) \iff \forall v : \tau, m_1(v) \subseteq^\sharp_\tau m_2(v) \wedge d_1 \sqsubseteq_r d_2$$

An environment (m_1, d_1) is more precise than (m_2, d_2) if for every program variable, the abstraction of this variable is more precise in m_1 than in m_2, and its abstraction for numerical variables, d_1, is more precise than d_2.

▶ **Example 4.7.** Following Example 4.2 on integer lists, we have for instance:

$$((x.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}) \subseteq^\sharp_{\texttt{list}} ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$$

So we can deduce:

$$\begin{aligned}&(x \to ((x.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), \{3 \leq x.1.1 \leq 6\})\\ \sqsubseteq^\sharp\ &(x \to ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), \{2 \leq x.1.1 \leq 7\})\end{aligned}$$

To define the meet, join and widening operators, we also start by defining those operators on the environment-free ADT abstractions, that is, ignoring their numerical part. Similarly, we lift operators from domains $\mathcal{D}^\sharp_{i,j}$. Those operators are defined between objects giving the same name to their summarization variables.

▶ **Definition 4.8** (Environment-free operators). *For* $\square \in \{\cup, \cap, \nabla\}$, *for* $(g_1, \mathscr{C}_1), (g_2, \mathscr{C}_2) \in \mathcal{D}^\sharp_\tau$*:*

$$(g_1, \mathscr{C}_1)\ \square^\sharp_\tau\ (g_2, \mathscr{C}_2) = (g, \mathscr{C}_1\ \square\ \mathscr{C}_2) \text{ where } g \text{ is defined by } \forall i, j, g_{i,j} = g^1_{i,j}\ \square_{\mathcal{D}_{i,j}}\ g^2_{i,j}$$

$(g_1, \mathscr{C}_1)\ \square^\sharp_\tau\ (g_2, \mathscr{C}_2)$ performs a join (resp. meet or widening) between non-numerical fields by delegating the operation to the relevant domains. Afterwards, to perform the operation on numerical fields too, we need to perform it on environments:

▶ **Definition 4.9** (Operators). *For* $\square \in \{\cup, \cap, \nabla\}$,*we build* $\square^\sharp : \Sigma^\sharp \to \Sigma^\sharp$*:*

$$(m_1, d_1)\ \square^\sharp\ (m_2, d_2) = (v : \tau \to m_1(v)\ \square^\sharp_\tau\ m_2(v),\ d_1\ \square^\sharp_r\ d_2)$$

Performing an operation between two environments consists in performing it on each variable abstraction in both maps, pointwise, and between both numerical abstract elements.

Note that the inclusion test, meet, join, and widening for the numerical relational domain operate on *heterogeneous* environments, that is, environments that may be defined on different sets of variables. We rely on the technique from Journault et al. [24] to lift classic relational domains (such as polyhedra) to the case of heterogeneous environments.. The soundness proof of our operators is a direct consequence of their soundness proofs [21].

4.1.4 Constructor Transfer Function

We then define the transfer function of constructors. Given an abstract environment $\sigma^\sharp_0$ and a concrete expression $C_k(e_1, \ldots, e_n)$, we compute a sound abstraction $(g, \mathscr{C})$ in a new abstract environment:

$$\begin{aligned}
\mathbb{E}^\sharp[\![\ C_k(e_1, \ldots, e_n)\ ]\!]\sigma^\sharp_0 = {}& \text{let } \forall i \in [\![1, n]\!]\ v_i, \sigma^\sharp_i = \mathbb{E}^\sharp[\![e_i]\!]\sigma^\sharp_{i-1} \text{ in} && (1)\\
& \text{let } (g^0, _), \sigma^\sharp = \mathbf{fold}\left((g^0, \emptyset) \leftarrow (v_j)_{\tau_{k,j}=\tau}\right)\sigma^\sharp_n \text{ in} && (2)\\
& \text{let } g^0, \sigma^\sharp = \bigcirc_{j, \tau_{k,j} \neq \tau}\left(\mathbf{fold}(g^0 \leftarrow_{k,j} v_j)\right)\ \sigma^\sharp \text{ in} && (3)\\
& \text{let } g = \left(\begin{cases} g^0_{k,j} \cup \mathrm{snd}(v_j) & \text{if } i = k \wedge \tau_{k,j} = \tau \\ g^0_{i,j} & \text{otherwise} \end{cases}\right)_{i,j} \text{ in} && (4)\\
& (g, \{C_k\}), \sigma^\sharp
\end{aligned}$$

The abstract semantics of $C_k(e_1, \ldots, e_n)$ is an element of $\mathcal{D}^\sharp_\tau$. The only possible start constructor is C_k. We abstract its fields as g, which is computed as follows. First, we recursively compute abstractions for e_i in v_i (1). We then define $g^0 = (g^0_{i,j})_{i,j}$. We use the

fold operator, defined by Gopan et al. [18] for summarization variables of arrays and lifted there for ADTs. It folds inside every $g^0_{i,j}$ all abstractions for the j-th fields of C_i nested inside every recursive field abstraction v_j (2). After the first folding, $g^0_{i,j}$ abstracts possible values in the j-th field of C_i nested at a depth of at least 1 in x. Then, we need to update it with every possible values in j-th fields of C_i nested at *any depth* in x. First, we add non-recursive information at depth 0, that is, updating $g^0_{k,j}$ with j-th field of C_k at depth 0, v_j (3). To do so, we compose ($\bigcirc$), for every non-recursive field (k, j), the **fold** operator to add v_j as a possible abstract values for field $g^0_{k,j}$. Finally, the possible start constructors of e_j, that is, $\mathrm{snd}(v_j)$, are added as possible start constructors for the j-th field of C_k (4).

▶ **Example 4.10.** Following previous examples on integer lists, given $\sigma^\sharp \in \Sigma^\sharp$:

$$\mathbb{E}^\sharp[\![\texttt{Nil}]\!]\sigma^\sharp = ((r.1.1, \bot), \{\texttt{Nil}\}), \sigma^\sharp[r.1.1 \to \bot]$$
$$\mathbb{E}^\sharp[\![\texttt{Cons}(10, \texttt{Nil})]\!]\sigma^\sharp = ((r.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), \sigma^\sharp[r.1.1 \to \bot \cup_{\mathbb{I}} [10, 10]]$$
$$\mathbb{E}^\sharp[\![\texttt{Cons}(1, \texttt{Cons}(10, \texttt{Nil}))]\!]\sigma^\sharp = ((r.1.1, \{\texttt{Nil}, \texttt{Cons}\}), \{\texttt{Cons}\}), \sigma^\sharp[r.1.1 \to [1, 10]]$$

Note that we rely on type information to delegate abstraction of a given field to the relevant domain. We discuss related implementation details in Section 7.1.

4.1.5 Abstraction Precision

▶ **Example 4.11** (Abstraction of trees)**.** We start with $\sigma^\sharp_0$ an empty environment. As an illustration for this abstract domain, consider:

```
1  type tree = Node of tree * int * tree | Leaf of int
2  let x = Node(Node(Leaf(250), 100, Leaf(251)), 1, Leaf(252))
```

Here, $\mathbb{E}^\sharp[\![\texttt{Node}(\texttt{Node}, (\texttt{Leaf}(250), 100, \texttt{Leaf}(251)), 1, \texttt{Leaf}(252))]\!]\sigma^\sharp_0 = ((g, \mathscr{C}), \sigma^\sharp)$ with:

$$(g, \mathscr{C}) = ((\{\texttt{Node}, \texttt{Leaf}\}, x.1.2, \{\texttt{Leaf}\}, x.2.1), \{\texttt{Node}\})$$
$$\sigma^\sharp = ([x.1.2 \to v_{x.1.2}], [x.2.1 \to v_{x.2.1}], [v_{x.1.2} \to [1, 100]][v_{x.2.1} \to [250, 252]])$$

$$\gamma_{\texttt{tree}}((g, \mathscr{C}), \sigma^\sharp) = \{x \mid x = \texttt{Node}(t_1, n, t_2) \ \wedge \ n \in [1, 100] \\ \wedge\, t_1 \in \gamma_{\texttt{tree}}((g, \{\texttt{Node}, \texttt{Leaf}\}))t_2 \in \gamma_{\texttt{tree}}(g, \{\texttt{Leaf}\})\}$$

It abstracts the set of trees starting with `Node`, only growing to the left, where the content of `Leaf` fields are between 250 and 252, and the content of `Node` between 1 and 100.

Note that our summarization abstraction can quickly lose precision, even when the structure size is known statically. To limit this, we could keep track of `x`'s exact content and summarize only when needed for convergence (recursive call), or for a certain depth of `x`. Such features are not yet implemented. Note that though of identical type, `Node` and `Leaf`'s integer fields are summed up separately.

▶ **Example 4.12** (Relationality)**.** Consider this program manipulating integer variables `a` and `y` of unknown value:

```
1  let z = Cons(a, Nil) in
2  let x = if y <= 2*a + 1 then Cons(y, z) else Cons(2 * a + 1, z)
```

With the polyhedra domain of Cousot and Halbwachs [15], the analysis is able to infer relational properties:

$$
\begin{aligned}
z \to & ((z.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), [z.1.1 = a] \\
x \to & (((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), [x.1.1 = z.1.1 \cup y][y \leq 2a+1][z.1.1 = a]) \cup^\sharp_{\texttt{list}} \\
& (((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), [x.1.1 = z.1.1 \cup 2a+1][y > 2a+1][z.1.1 = a]) \\
\to & ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), [x.1.1 < 2a+1][z.1.1 = a]
\end{aligned}
$$

We can then prove: $x_{1,1} \leq 2a+1$, that is, every element of the list is smaller than $2a+1$, without assuming any bound on a. This relational aspect is crucial to infer input-output function summaries for generic inputs.

4.2 Pattern-Matching Abstract Semantics

Pattern-matching is a key construct to manipulate ADTs. We define their abstract semantics. We use as intermediate function $\texttt{match}^\sharp : \mathcal{V}^\sharp \times \Sigma^\sharp \times \Pi \to \Sigma^\sharp \times \Sigma^\sharp$. Given an abstract value $v^\sharp$, an abstract environment $\sigma^\sharp$ and a pattern p, it returns two abstract environments. The first is an over-approximation of environments from $\sigma^\sharp$, in which p and $v^\sharp$ can match. The second is an over-approximation of environments in which they cannot. We rely on a filter function $\mathcal{F}^\sharp$ from the numerical domain. It restricts an abstract environment to keep only those where a given boolean predicate evaluates to true. It is useful to model the `when` guards appearing in patterns:

$$
\begin{aligned}
\texttt{match}^\sharp(v^\sharp, \sigma^\sharp, p \ \texttt{when}\ e_1) = \ & \text{let } \sigma^\sharp_0, \sigma^\sharp_{\neg,0} = \texttt{match}^\sharp(\sigma^\sharp, v^\sharp, p) \text{ in} \\
& \text{let } e^n_1, \sigma^\sharp_1 = \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp_0 \text{ in} \\
& \left(\mathcal{F}^\sharp[\![e^n_1 \neq 0]\!]\sigma^\sharp_1\right), \left(\sigma^\sharp_{\neg,0} \cup^\sharp \mathcal{F}^\sharp[\![e^n_1 = 0]\!]\sigma^\sharp_1\right)
\end{aligned}
$$

Thus, $v^\sharp$ matches p `when` e_1 in environments where $v^\sharp$ and p match, and when e^n_1 is non-zero. The other cases are simpler.

$$
\begin{aligned}
\mathbb{E}^\sharp[\![\text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m]\!]\sigma^\sharp = \ & \\
& \text{let } \sigma^\sharp_1, \sigma^\sharp_{\neg,1} = \texttt{match}^\sharp(\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp, p_1) \text{ in} \\
& \text{let } e'_1 = \text{match } e_0 \text{ with } p_2 \to e_2 \mid \cdots \mid p_m \to e_m \text{ in} \\
& \begin{cases} \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp_1 \cup^\sharp \mathbb{E}^\sharp[\![e'_1]\!]\sigma^\sharp_{\neg,1} & \text{if } m \geq 1 \\ \{\omega\}, \sigma^\sharp & \text{if } m = 0 \wedge \sigma^\sharp \neq \bot \end{cases}
\end{aligned}
$$

The abstract semantics of pattern matching is the abstract union of all interpretations of branches e_i, in an environment in which e_0 matches the pattern p_i but not the previous patterns p_j for $j < i$. We add $\{\omega\}$ (match failure) if some part of the environment did not match any pattern. Our analysis can thus target match failure, i.e. when no pattern caught the value of e_0. This is the case if $\sigma^\sharp \neq \bot$ when no pattern is left. To the best of our knowledge, this is the first value analysis handling `when` clauses in pattern-matching exploiting value information to precisely detect non-exhaustive pattern-matching. The OCaml compiler does warn on non-exhaustive pattern-matching but, since its analysis is not value-sensitive, it is more conservative. For the following program, it will issue a "non-exhaustive pattern matching" warning, whereas our analysis, knowing that `l` starts with `Cons`, can prove that no case is left unmatched:

```
let l = Cons(1, Nil) in let x = match l with Cons(h,Nil)-> h.
```

```
let rec filter_le inf l =
  match l with
  | Cons(h, q) when h > inf -> filter_le inf q
  | Cons(h, q) when h <= inf -> Cons(h, filter_le inf q)
  | Nil -> Nil
in
let x = Cons(0, Cons(5, Cons(11, Nil))) in
let y =  filter_le 4 x in
let hd = match y with Cons(h,q) -> h | Nil -> 0 in
assert(hd <= 4)
```

Figure 6 A recursive function manipulating recursive algebraic objects.

4.3 Example

Before defining a domain for higher-order functions, we show with an example how our compositional function analysis works together with our ADT summarization domain and the abstract semantics of pattern-matching. We consider the program in Figure 6. It defines a function `filter_le`. This function of type $\tau = \texttt{int} \to \texttt{list} \to \texttt{list}$ takes as inputs an integer *inf* and a list l and keeps only the elements of l that are lower than *inf*.

Let *body* be the body of the function `filter_le`. We analyze it with unknown inputs, in $\sigma^\sharp = [l \to \top][inf \to \top]$. Its abstract semantics is $F : v^\sharp \to \text{fst}(\mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[f \to v^\sharp])$. To get the abstract semantics of the recursive function, we compute the abstract fixpoint of the body semantics (Section 3.2). Starting with `filter_le` at $\bot_\tau = ((inf, l), r, \sigma^\sharp[r \to \bot])$, we compute the fixpoint $\nabla_{n\in\mathbb{N}} F^n(\bot_\tau)$:

$$\begin{aligned}
F(\bot_\tau) &= \mathbb{E}^\sharp[\![\text{fun } l \ inf \ \to body]\!]\sigma^\sharp[\texttt{filter_le} \to \bot_\tau] \\
&= ((inf, l), ((r.1.1, \bot), \{\texttt{Cons}\}, \sigma^\sharp[r.1.1 \to \bot])) \cup^\sharp_\tau \bot \\
&= (\bot_\tau, \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \to \bot]) \\
F^2(\bot_\tau) &= F(((inf, l), (r.1.1, \bot), \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \to \bot])) \\
&= (((inf, l), (r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \leq inf]) \\
F^3(\bot_\tau) &= F^2(\bot_\tau) \\
\nabla_{n\in\mathbb{N}} F^n(\bot_\tau) &= (((inf, l), (r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \leq inf])
\end{aligned}$$

By iteratively computing the summary of the function, we are finally able to infer that the elements in the output list ($r.1.1$) are lower than input *inf*. We denote this result as V_1, assign it to `filter_le` and continue the analysis:

$$\mathbb{E}^\sharp[\![\text{let rec } \texttt{filter\_le} = body \text{ in } e]\!]\sigma^\sharp = \text{ let } \sigma_1^\sharp = \sigma^\sharp[\texttt{filter\_le} \to V_1] \text{ in } \mathbb{E}^\sharp[\![e]\!]\sigma_1^\sharp$$

We abstract the assignment into x:

$$\begin{aligned}
\mathbb{E}^\sharp[\![\texttt{Cons}(0, \texttt{Cons}(5, \texttt{Cons}(11, \texttt{Nil})))]\!]\sigma_1^\sharp = \ & \text{let } \sigma_2^\sharp = \sigma_1^\sharp[x.1.1 \to 0 \cup_\mathbb{Z} 5 \cup_\mathbb{Z} 11] \text{ in} \\
& ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), \sigma_2^\sharp = V_2, \sigma_2^\sharp \\
\mathbb{E}^\sharp[\![\text{let } x = \texttt{Cons}(0, \texttt{Cons}(5, \texttt{Cons}(11, \texttt{Nil}))) \text{ in } e_2]\!]\sigma_3^\sharp = \ & \text{let } \sigma_3^\sharp = \sigma_2^\sharp[x \to V_2] \text{ in } \mathbb{E}^\sharp[\![e_2]\!]\sigma_3^\sharp
\end{aligned}$$

We then abstract the assignment into y by applying `filter_le`.

$$\mathbb{E}^\sharp[\![\texttt{filter\_le}\ 4\ x]\!]\sigma_3^\sharp = ((r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}),$$
$$(\sigma^\sharp[r.1.1 \leq inf])[inf = 4][l = \sigma_3^\sharp(x)]$$
$$((r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^\sharp[r.1.1 \leq 4] = V_3, \sigma_4^\sharp$$

$$\mathbb{E}^\sharp[\![\text{let } y = \texttt{filter\_le}\ 4\ x \text{ in } e_2]\!]\sigma_4^\sharp = \text{ let } \sigma_5^\sharp = \sigma_4^\sharp[x.1.1 \to 0 \cup_{\mathbb{Z}} 5 \cup_{\mathbb{Z}} 11][y \to V_3] \text{ in}$$
$$\mathbb{E}^\sharp[\![e_2]\!]\sigma_5^\sharp$$

We then evaluate $e_1 =$ match y with $\mid \texttt{Cons}(h, q) \to h \mid \texttt{Nil} \to 0$. We assign the result to hd.

$$\mathbb{E}^\sharp[\![e_1]\!]\sigma_1^\sharp = h, \sigma_5^\sharp[hd \leq 4]$$

Since $hd \leq 4$, the assertion is proven. More generally, the summary for `filter_le` proves that for any list l and integer inf, `filter_le` inf l is a list whose content is lower than inf. The analysis also proves that the pattern-matching inside the function body is exhaustive.

5 Higher-Order, Disjunctive Relational Summaries

In a higher-order setting, functions can be inputs or output of other functions. This is a key concept of functional programming, making programs harder to analyze. We consider the program from Figure 1 and highlight the benefits of compositional analyses on this example.

Non-compositional analysis. Different choices were made in the literature to analyze higher-order functions. A method giving precise results [27] is simply to defer the analysis of `to_fun`'s body until we know its parameters, or an abstraction of them. Consequently, the body analysis is performed at every call site. Thus, the definitions of `f1` and `f2` only trigger a store of their definition code. When we compute `r1` and `r2`, all inputs are known, yielding: $\texttt{r1} = 5, \texttt{r2} = 9$. This method is able to infer precise results for `r1` and `r2`, with the drawback that the function body is reanalyzed at each call site. This is costly, and hinders scalability.

Compositional analysis. To circumvent this cost, we can perform a compositional analysis, where a function is analyzed only once, at its definition site. This is how `filter_le` was analyzed in Section 4.3, at first-order. Since functions are abstracted as points in the polyhedra space, intuitively, manipulating them as first class values for higher order does not add further complexity. The function domain is simply based on the polyhedra one. We can analyze `to_fun` once and for all, supposing that we do not know anything about `f`'s behavior. We get $\texttt{r1} = \top, \texttt{r2} = \top$. This loss of information is due to a join of both possible cases from `to_fun`'s behavior. However, this is not inherent to compositional analysis, since the specification of `to_fun` is:

$$\texttt{to\_fun}(c, x) = \begin{cases} n & \text{if } c = \texttt{Cst}(n) \\ f(x) & \text{if } c = \texttt{Fun}(f) \end{cases}$$

It is thus possible to describe `to_fun`'s behavior without a priori information on possible values for c, as long as *disjunctions* on those possible values are possible. The precision loss mentioned above was indeed due to the impossibility to express disjunctions when abstracting higher-order functions. This is why we focus in this section on lifting partitioning methods to higher-order.

5.1 Partitioning Function Summaries

Even at first-order, a similar precision loss occurs on very simple programs:

```
let binary x = if x > 0 then 10 else -10
```

By conducting a compositional non-relational analysis with the interval domain, we are only able to infer that the output is in $[-10, 10]$, a quite weak result. This comes from the fact that `binary` has two very distinct behaviors depending on the input value. Joining them induces a precision loss. The issue can be overcome by partitioning on the input [7], then keeping separate those behaviors: $\lambda x.(x > 0 \wedge res = 10) \vee (x \leq 0 \wedge res = -10)$. Note that relationality cannot, on its own, overcome this cause of imprecision. The polyhedra domain gives the same result as intervals on `binary`. However, Boutonnet and Halbwachs [8] successfully use relationality and partitioning together. Consider the following example:

```
let max x y = if x >= y then x else y
```

With the interval domain, even with partitioning, we cannot express anything interesting about `max`, whereas the polyhedra domain alone can only infer: $\lambda xy.x \leq res \wedge y \leq res$. But with *disjunctive relational summaries*, we can separate disjunct behaviors, then summarizing the maximum function with: $\lambda xy.(x < y \wedge res = y) \vee (x \geq y \wedge res = x)$. Maintaining multiple partitions is costly, so a key point of partitioning is to decide when to partition and when to merge different behaviors. Various heuristics were developed on that topic. Those methods are compatible with recursion. Indeed, Bourdoncle [7] and Boutonnet and Halbwachs [8] are able to give precise summaries for McCarthy 91 function.

We want to benefit from this precision improvement at higher-order. In a higher-order setting however, functions are first class values: they can be input or output of other functions, etc. As first class objects, they should be represented as abstract values. Compared to previous works at first-order, we then need to define the *abstract domain of functions*. To the best of our knowledge, this is the first presentation of disjunctive summaries as a domain abstracting functions complete with all its lattice operators, including join and widening.

5.2 Disjunctive Relational Summaries as a Domain on Functions

As in Section 3.2, we choose $(\mathcal{R}(V), \sqsubseteq_r^\sharp, \sqcup_r^\sharp, \sqcap_r^\sharp, \nabla_r^\sharp, \gamma_r, \mathrm{proj}_r, \lambda_r)$ a relational domain on V. This way, the formalization is independent of the chosen domain.

▶ **Definition 5.1** (Function domain). *Given a function type $\tau = \tau_1 \to \cdots \tau_n \to \tau_r$, we define $\mathcal{F}_\tau^\sharp$ the set of elements abstracting functions of type τ. Let n_i be the number of variables necessary to abstract an object of type τ_i.*

$$\mathcal{F}_\tau^\sharp = \{(V, r, (p_i)_{[\![1,m]\!]}) \mid V = (x_1, \ldots, x_n), \forall i, x_i \in \mathbb{V}^{n_i}, \forall i, j, i \neq j \Rightarrow x_i \neq x_j, r \in \mathbb{V}, \\ \forall i \in [\![1, m]\!], p_i \in \mathcal{R}(V \cup \{r\})\}$$

A function is thus abstracted as a triplet of: a n-tuple of input variables, an output variable, and a collection of m relations abstracting different cases in the function's behavior. We denote $(x_1, \ldots, x_n), r, (p_1, \ldots, p_m)$ as $\lambda x_1 \ldots x_n. \bigvee_{i=1}^{m} p_i(x_1, \ldots, x_n, r)$. For example, `let add x y = x + y` is abstracted as $((x, y), r, (r = x + y))$, written $\lambda xy.r = x + y$.

▶ **Definition 5.2** (Function concretization). *For* $\tau = \tau_1 \to \cdots \tau_n \to \tau_r$, $f = ((x_i)_{i\in[\![1,n]\!]}, r, (p_i)_{i\in[\![1,m]\!]}) \in \mathcal{F}^\sharp_\tau$, *the concretization is:*

$$\gamma_\tau(f, \_) = \{f : \tau \mid \forall (a_1, \ldots, a_n) : \tau_1 \times \cdots \times \tau_n, \\ [x_1 = a_1] \cdots [x_n = a_n][r = f(a_1, \ldots, a_n)] \in \bigvee_{i=1}^{m} \gamma_r(p_i)\}$$

This concretization is simply an update of Definition 3.4, with a disjunction of relations instead of a unique one.

▶ **Example 5.3.** For $f^\sharp = \lambda x_1 x_2.(x_1 \leq x_2 \wedge r > 0) \vee (x_1 > x_2 \wedge r < 0)$, $\tau =$ `int` $\to$ `int` $\to$ `int`:

$$\gamma_\tau(f^\sharp, \_) = \{f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \mid \forall x_1, x_2 \in \mathbb{N}, \\ x_1 \leq x_2 \implies f(x_1, x_2) > 0 \wedge x_1 > x_2 \implies f(x_1, x_2) < 0\}$$

That is, $f^\sharp$ represents all functions whose result is lower than 0 when the first input is the smallest, and greater than 0 otherwise.

We need to define comparison, join, meet, and widening operators on functions. Since we consider only well-typed programs, those operators will be defined for functions of identical type: in particular, they will have the same number of parameters. These definitions build upon relational domain operators.

Note that renaming the inputs or the output of the function does not change its concretization. Without loss of generality, we assume when comparing two functions that they use the same names for inputs or outputs (this can always be ensured by renaming). We then define abstract inclusion, meet, join, and widening operators on the functional domain:

▶ **Definition 5.4** (Inclusion). *For* $f = V, r, (f_i)_{[\![1,n_1]\!]}$, $g = V, r, (g_i)_{[\![1,n_2]\!]}$, *the inclusion* $\subseteq^\sharp$ *on* $\mathcal{P}^\sharp_n$ *is:* $f \subseteq^\sharp g \iff \forall i \in [\![1, n_1]\!], \exists j \in [\![1, n_2]\!], f_i \sqsubseteq^\sharp_r g_j$

f is included in g if every disjunct of f is included in a disjunct of g. This order is the classic order for disjunctive completion domains [1].

▶ **Example 5.5.** With this definition, given two functions of type `int -> int`:

$$x, r, \begin{cases} x > 1 \wedge r = 10 \\ x = 0 \wedge r = 10 \\ x < 0 \wedge r = \top \end{cases} \quad \subseteq^\sharp \quad x, r, \begin{cases} x \geq 1 \wedge r = 10 \\ x < 1 \wedge r = \top \end{cases}$$

▶ **Definition 5.6** (Meet). *Given* $f = V, r, (f_i)_{i\in[\![1,m_1]\!]}$ *and* $g = V, r, (g_i)_{i\in[\![1,m_2]\!]}$, *the abstract intersection of functions is defined as* $\sqcap^\sharp_f$*:*

$$f \sqcap^\sharp_f g = V, r, \left(\bigsqcup_{j=1}^{m_2} (f_i \sqcap^\sharp_r g_j) \right)_{i\in[\![1,n_1]\!]}$$

That is, given a disjunct f_i, we compute its meet with every disjunct g_j and then join them together, keeping n_1 partitions. Indeed, for performance reasons, we do not want to maintain all (quadratic) combinations of pairwise intersections. This method ensures that we keep the number of partitions used in the right argument. This abstraction of the meet is then not commutative. An alternative definition would partition with regard to the abstraction with the highest number of disjuncts, which would likely be more precise. An experimental study, left for future work, could be led to determine the best version.

▶ **Example 5.7.** Given two functions of type `int -> int` with different input partitioning:

$$n, r, \begin{cases} n \geq 0 \wedge 1 \leq r \leq 3 \\ n < 0 \wedge -3 \leq r \leq -1 \end{cases} \quad \sqcap_f^\sharp \quad n, r, \begin{cases} n \geq 1 \wedge 2 \leq r \leq 4 \\ n = 0 \wedge r = 3 \\ n < 0 \wedge -4 \leq r \leq -2 \end{cases}$$

$$= n, r, \begin{cases} (n \geq 1 \wedge 2 \leq r \leq 3) \sqcup_p^\sharp (n = 0, r = 3) \sqcup_p^\sharp \bot_p \\ \bot_p \sqcup_p^\sharp \bot_p \sqcup_p^\sharp n < 0 \wedge -3 \leq r \leq -2 \end{cases}$$

$$= n, r, \begin{cases} n \geq 0 \wedge 2 \leq r \leq 3 \\ n < 0 \wedge -3 \leq r \leq -2 \end{cases}$$

▶ **Definition 5.8** (Join). *Given $f = V, r, (f_i)_{i \in [\![1, n_1]\!]}$ and $g = V, r, (g_i)_{i \in [\![1, n_2]\!]}$, the abstract union of functions is defined as $\sqcup_f^\sharp$. We write $G_i = \{g_j \mid 1 \leq j \leq n_2 \wedge proj_r(g_j) \sqsubseteq proj_r(f_i)\}$ the set of g_j whose constraints on the inputs are included in the constraints of f_i, and $G^\neg = \{g_j \mid 1 \leq j \leq n_2 \wedge \forall 1 \leq i \leq n_1, g_j \notin G_i\}$ the set of g_j not included in any G_i.*

$$f \sqcup_f^\sharp g = V, r, (f_i \bigsqcup_{g_j \in G_i} g_j)_{i \in [\![1, n_1]\!]} \vee (g_j)_{j \in [\![1, n_2]\!] \wedge g_j \in G^\neg}$$

That is, we keep every disjunct f_i, and join them with disjuncts of g respecting their conditions on the input. We add disjuncts g_j that are compatible with no f_i. Similarly to the meet operator, we do not want to simply keep every disjunct, so as to limit their number.

▶ **Example 5.9.** Given two functions of type `int -> int`:

$$n, r, \begin{cases} n \geq 0 \wedge 1 \leq r \leq 3 \\ n < 0 \wedge -3 \leq r \leq -1 \end{cases} \quad \sqcup_f^\sharp \quad n, r, \begin{cases} n \geq 1 \wedge 2 \leq r \leq 4 \\ n = 0 \wedge r = 3 \\ n < 0 \wedge -4 \leq r \leq -2 \end{cases}$$

$$= n, r, \begin{cases} (n \geq 0 \wedge 1 \leq r \leq 3) \sqcup_p^\sharp (n \geq 1 \wedge 2 \leq r \leq 4) \sqcup_p^\sharp (n = 0 \wedge r = 3) \\ (n < 0 \wedge -3 \leq r \leq -1) \sqcup_p^\sharp (n < 0 \wedge -4 \leq r \leq -2) \end{cases}$$

$$= n, r, \begin{cases} n \geq 0 \wedge 1 \leq r \leq 4 \\ n < 0 \wedge -4 \leq r \leq -1 \end{cases}$$

Widening is defined the same way as join, replacing $\sqcup_p^\sharp$ with $\nabla_p^\sharp$. However, to ensure convergence, we also limit the number of disjuncts to a user-controlable constant. When performing widening, we join disjuncts when necessary to respect this limit, before delegating widening to the underlying domain.

In conclusion, we defined the set $\mathcal{F}_\tau^\sharp$ alongside its concretization, inclusion, join, meet, and widening operators, as an abstract domain for functions of type τ. We now explain how to apply a computed summary in $\mathcal{F}_\tau^\sharp$ at a call site, to evaluate the effect of a function call.

5.3 Function Application with Partitioning

Function summaries being now disjunctive, we need to update the semantics from Section 3.2:

$$\mathbb{E}^\sharp[\![e_0 \dots e_k]\!]\sigma^\sharp = \text{let } (x_1, \dots, x_n, r, (p_i)_{i\in[\![1,m]\!]}), \sigma_0^\sharp = \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp \text{ in} \tag{5}$$

$$\text{let } \forall i \in [\![0, k-1]\!]\ e_{i+1}^\sharp, \sigma_{i+1}^\sharp = \mathbb{E}^\sharp[\![e_{i+1}]\!]\sigma_i^\sharp \text{ in} \tag{6}$$

$$\text{let } \forall i \in [\![1, m]\!]\ \sigma_{k,i}^\sharp = \texttt{add\_vars}(\sigma_k^\sharp, \texttt{dom}(p_i)) \text{ in} \tag{7}$$

$$\text{let } \forall i \in [\![1, m]\!]\ p_i = \texttt{add\_vars}(\lambda_r(p_i), \texttt{dom}(\sigma_n^\sharp)) \text{ in} \tag{8}$$

$$\text{let } \forall i \in [\![1, m]\!]\ p_i' = (p_i \sqcap^\sharp \sigma_{k,i}^\sharp)[x_j = e_j^\sharp]_{j\in[\![1,k]\!]} \text{ in} \tag{9}$$

$$\text{let } X = \{x_{k+1}, \dots, x_n\} \text{ in} \tag{10}$$

$$\begin{cases} \left(X, r, (\text{proj}_r(p_i'))_{i\in[\![1,m]\!]\wedge p_i'\neq\bot}\right), \sigma^\sharp & \text{if } k < n \\ r, (p_i')_{i\in[\![1,m]\!]\wedge p_i'\neq\bot} & \text{otherwise} \end{cases} \tag{11}$$

First, we compute the semantics of the function e_0 (5). Then we compute sequentially the semantics of its argument expressions e_1, …, e_k (6). As in Section 3.2, we extend p_i and $\sigma_k^\sharp$ to the same definition domain (7,8). We combine them and add to each function disjunct p_i the equality constraint between formal argument x_j and value $e_j^\sharp$ (9). X is the set of free formal arguments (not bound to an actual argument) in case of partial application (10).

Partial application comes naturally (case $k < n$ in (11)). When partially applying a function, we project the result on the set of remaining variables. The result is then a set of relations consisting in a disjunctive summary. It abstracts the function resulting from the partial application. When the application is total (otherwise in (11)), there is no remaining parameters: we simply change the format of the result. In both cases, we remove empty disjuncts $p_i' = \bot$, which correspond to unsatisfiable constraints on the inputs.

▶ **Example 5.10.** The summary of the function `max` is $(x, y), r, (x > y \wedge r = x) \vee (x \leq y \wedge r = y)$. Then partially applying it with one argument a, we get the abstraction:

$$\mathbb{E}^\sharp[\![\texttt{max}\ a]\!]\{a \geq 5\} = y, r, \{(a > y \wedge r = a \wedge a \geq 5) \vee (a \leq y \wedge r = y \wedge a \geq 5)\}$$

This way, we deduce that the function returns a result which is always greater than 5, and is y if it is greater than a and a otherwise.

5.4 Function Analysis with Partitioning

We update function analysis semantics from Section 3.2 to allow a disjunctive result:

$$\mathbb{E}^\sharp[\![\text{fun } x_1 \cdots x_n \to body]\!]\sigma^\sharp = \text{let } c_1, \dots, c_p = \texttt{get\_preconditions}(body) \text{ in} \tag{12}$$

$$\text{let } \forall i \in [\![1, p]\!],\ e_i^\sharp, \sigma_i^\sharp = \mathbb{E}^\sharp[\![body]\!](\sigma^\sharp \wedge c_i)[x_j \to \top] \tag{13}$$

$$\text{and } s_i = \text{proj}_r(\sigma_i^\sharp[r \to e_i^\sharp]) \text{ in} \tag{14}$$

$$\left((x_1, \dots, x_n), r, (s_i)_{i\in[\![1,p]\!]}\right), \sigma^\sharp \tag{15}$$

The `get_preconditions` helper function scans the body to get a set of predicates $(c_1, \dots, c_p)$ suitable for partitioning (12). Then, we analyze the body of the function with each precondition c_i (13) and get a set of possible behaviors s_i (14). The function is then abstracted as the disjunction of those behaviors for the function (15). Note that we do not keep partitioned states *inside* the analysis of a recursive function, to avoid the risk of diverging

sets of partitions. Consequently, when applying a function inside the body of the recursive function, we immediately merge cases. also note that this definition is independent of the chosen `get_preconditions` partitioning heuristics.

We chose to use a heuristics, so called *local reachability*, introduced by Boutonnet and Halbwachs [8]. We try to partition regarding conditions controlling reachability to certain key points – branches in conditional branching or pattern matching. Note that finding those preconditions is a pre-analysis, performed just before analyzing the function itself. Consequently, when a function body we are either analyzing or pre-analyzing performs a function call, the abstract summary of the called function is already available (or it is set to bottom if this is the first iteration of a recursive call).

To sum up the method, the body of the function is pre-analyzed to decide a partitioning on the inputs. It starts with a precondition $I^\sharp$, which can be $\top$, and follows those steps:

1. Analyze the function body with an interprocedural analysis and note $r_i^\sharp$ the result at each reachable control point;
2. Choose a control point for which $r_i^\sharp \neq \bot$;
3. Choose an abstract value $s^\sharp$ which is complementable, i.e. its complementary can be expressed in the domain (note that $x \leq y$ as well as $x > y$ tests are always complementable for polyhedra on integers) such that $\text{proj}_r(r_i^\sharp) \sqsubseteq^\sharp s^\sharp$. We get disjuncts $I^\sharp \sqcap s^\sharp$ and $I^\sharp \sqcap \bar{s}^\sharp$.

We may iterate this algorithm to get more disjuncts and gain more precision.

6 Combining both Domains into an Analysis

Sections 4 and 5 presented, respectively, independent extensions of a first-order numeric analysis to algebraic data types and to higher-order functions. Thanks to their parametricity, we can combine these two extensions to analyze our target language from Section 2:

- Algebraic data types can contain higher-order values (i.e., functions). Indeed, the parametricity of the ADT abstraction supports functions as leaf data types, themselves represented by the higher-order part of our work.
- Function summaries from the higher-order section can express precise properties on ADTs they manipulate (including the case of recursive functions manipulating recursive ADT). Indeed, the abstract domain representing functions is parametric in the domain chosen to express relations between inputs and outputs (e.g., polyhedra).

We show how the resulting analysis performs on our example function `to_fun` and state its soundness theorem.

6.1 Analysis Cooperation Example

We choose as relational domain the polyhedra domain, combined with a domain stating the equality of functions. We analyze our example from the introduction. We recall its code:

```
1  let to_fun a =
2    match a with
3    | Cst n -> (fun x -> n) (* p1 *)
4    | Fun f -> f (* p2 *)
```

Pre-analysis by local reachability. We analyze the function with a classic relational analysis. We denote the control point after the first branch of the match as p_1 and the one after the second branch of the match as p_2. We denote possible constructors of a as a_c.

The analysis in p_1 gives us $r_1^\sharp = (x, r, r = n), [a_c = \{\texttt{Cst}\} \wedge a.1.1 = n] \neq \bot$. We see that $proj(\{a_c, a.1.1, a.2.1\}, r_1^\sharp) = [a_c = \{\texttt{Cst}\}]$ is complementable, of complementary $[a_c = \{\texttt{Fun}\}]$. Consequently, we get two disjuncts: $a_c = \{\texttt{Cst}\}$ and $a_c = \{\texttt{Fun}\}$.

Full analysis. Let *body* be the body of `to_fun`.

$$\mathbb{E}^\sharp[\![\text{fun } a \to body]\!]\sigma^\sharp = (a_c, a.1.1, a.2.1), r, \begin{cases} \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Cst}\}] \\ \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Fun}\}] \end{cases}$$

We analyze the first case. We have $\texttt{match}^\sharp(\sigma^\sharp[a_c = \{\texttt{Cst}\}], a, p_1) = \sigma^\sharp[a_c = \{\texttt{Cst}\}], \bot$ so:

$$\begin{aligned} \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Cst}\}] &= \mathbb{E}^\sharp[\![\text{fun } x \to n]\!]\sigma^\sharp[a_c = \{\texttt{Cst}\}] \cup^\sharp \bot^\sharp \\ &= (x, r', r' = n), [a_c = \{\texttt{Cst}\} \wedge a.1.1 = n] \end{aligned}$$

Similarly, for the second case:

$$\begin{aligned} \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Fun}\}] &= \mathbb{E}^\sharp[\![\text{fun } x \to n]\!]\sigma^\sharp[a_c = \{\texttt{Fun}\}] \cup^\sharp \bot^\sharp \\ &= f, [a_c = \{\texttt{Fun}\} \wedge a.2.1 = f] \end{aligned}$$

In the end, the semantics of `to_fun` is:

$$\mathbb{E}^\sharp[\![\text{fun } a \to body]\!]\sigma^\sharp = (a_c, a.1.1, a.2.1), r, \begin{cases} [a_c = \{\texttt{Cst}\} \wedge r = (x, r', r' = a.1.1)] \\ [a_c = \{\texttt{Fun}\} \wedge a.2.1 = f] \end{cases}$$

We denote this summary as s and bind `to_fun` to s in the environment. Note that this summary is as precise as in the concrete. We can evaluate `f_1 = to_fun Cst`(5) and `f_2 = to_fun Fun`$(\text{fun } x \to n)$:

$$\begin{aligned} \mathbb{E}^\sharp[\![\texttt{to\_fun Cst}(5)]\!]\sigma^\sharp &= r, \sigma^\sharp[a_c = \{\texttt{Cst}\} \wedge r = (x, r', r' = a.1.1)] \\ &\quad [a_c = \{\texttt{Cst}\} \wedge a.1.1 = 5] \\ &= r, \sigma^\sharp[a_c = \{\texttt{Cst}\} \wedge a.1.1 = 5 \\ &\quad \wedge r = (x, r', r' = a.1.1)] \\ \mathbb{E}^\sharp[\![\texttt{to\_fun Fun}(\text{fun } x \to n)]\!]\sigma^\sharp &= r, \sigma^\sharp[a_c = \{\texttt{Fun}\} \wedge a.2.1 = f] \\ &\quad [a_c = \{\texttt{Fun}\} \wedge a.2.1 = (x, r', r' = x + 1)] \\ &= r, \sigma^\sharp[a_c = \{\texttt{Fun}\} \wedge a.2.1 = (x, r', r' = x + 1) \\ &\quad \wedge r = a.2.1] \end{aligned}$$

In both cases, one of the summary disjuncts has an empty intersection with the environment, so it does not appear in the final state. Applying the summaries, we are able to infer $r_1 : 5, r_2 : 9$. In the end, we are able to recover the same precision as a non-compositional method (Section 5) while staying compositional. This way, we do not have to re-analyze `to_fun`'s body when computing r_1 and r_2, instead using the summaries inferred at call site. Additionally, we get a precise contract for `to_fun`, which is valid for every input.

6.2 Analysis Soundness

The analysis defined in this article is sound, i.e. the analysis of a program P over-approximates the reachable states of P. Section 3.1 defined the concretization of abstract environments (Definition 3.2) and the concretization of the abstract semantics of an expression (Remark 3.3).

▶ **Theorem 6.1.** *The abstract semantics* $\mathbb{E}^\sharp$ *is sound, i.e.:*

$$\forall \sigma \in \Sigma, \forall \sigma^\sharp \in \Sigma^\sharp, \forall e : \tau \in \mathcal{E}, \sigma \in \gamma_{\Sigma^\sharp}(\sigma^\sharp) \implies \mathbb{E}[\![e]\!]\sigma \in \gamma_\tau(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$$

Proof. The proof is by induction over the syntax of expression e. ◀

7 Experimental Evaluation

7.1 Implementation

The methods described in the article were implemented in MOPSA [22], an open source and multi-language platform to ease the development of abstract analyzers. Similarly to other static analyzers by abstract interpretation, such as Astrée [6], Frama-C [9], Infer [16], or Julia [40], MOPSA is based on a composition of several abstract domains. Compared to them, it goes further in terms of modularity, enforcing a finer level of granularity of abstractions. It encourages relational abstractions for all types and features powerful domain communication mechanisms (such as reduced products, cartesian products, expression rewriting) and supports the reuse of abstractions in the analysis of widely different languages (such as C [37, 23] and Python [33, 34, 35]). The core of the platform is composed of 22 000 LoC of OCaml.

We added 3 000 LoC to support OCaml analysis and to implement the algebraic data types and higher-order domains described in the article. Our analysis is fully automatic: it leverages typing information from the frontend – handled by the OCaml compiler – to automatically associate abstract domains to values, based on their types. The user sets a global precision level by selecting more or less expressive domains for base types (e.g., polyhedra or intervals for integers). Choosing the best precision/performance trade-off is a long-standing problem for the analysis of any language. It is not addressed by this paper.

In MOPSA, a configuration file describes how domains are combined to define an analysis. A configuration for our analysis is shown in Figure 7. Sequences delegate the analysis of an expression to the right-hand side domain when relevant. Reduced products perform reductions between both domain results. Cartesian products implement collaborations between domains. `ML.program` is the frontend, handling the import from the OCaml compiler. `ML.intraproc` handles conditionals and assertions. `ML.let_rec` computes an abstract fixpoint for recursive definitions of variables. Domains `ML.functions` (defined in Section 5.2) and `ML.adts` (defined in Section 4.1) depend on each other because of their parametricity. They share underlying domains for ground types, here strings and integers. MOPSA includes a Universal language, implementing ready-to-use abstractions for basic constructs. Here, integers are abstracted by the reduced product of the polyhedra domain `U.polyhedra` and the congruence domain `U.congruences`. `U.string` abstracts strings. The configuration could be defined using other domains for ground types as well. Our implementation is parametric in those domains.

On a technical note, our ADT domain required support for heterogeneous operations, that is, operations on environments defined on different sets of variables [24], which is the case when joining two match cases. Our functions' domain required adapting the already existing trace partitioning [30] for C-like languages on the platform. Sound operations on summarization variables from Gopan et al. [18] were already part of the platform and were leveraged when handling summarization variables of recursive algebraic data types.

7.2 Experiments

We tested our implementation on the examples from this article as well as 40 handwritten programs highlighting the precise handling of recursive functions manipulating algebraic data types, functions partitioning on algebraic constructors, partial application, algebraic data

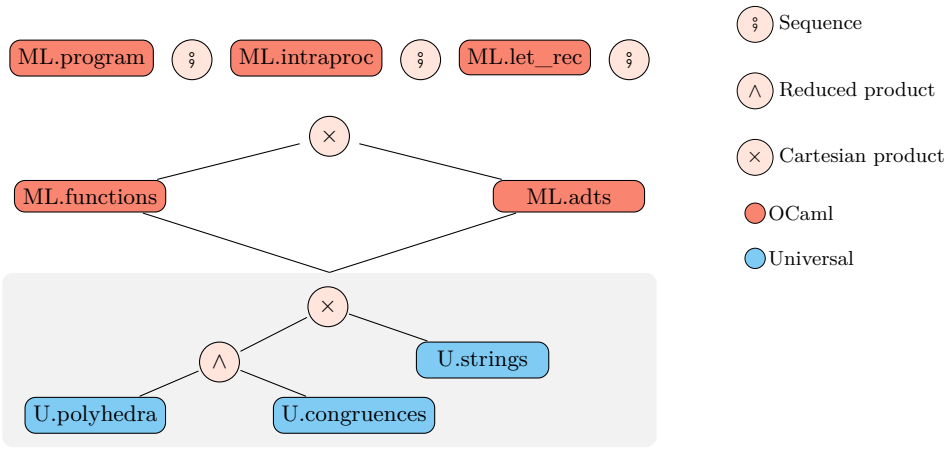

Figure 7 OCaml configuration in MOPSA.

Program name	Analysis (ms)	Program name	Analysis (ms)
`numeric_loop3b.ml`	12	`filter_le.ml`	131
`binomial.ml`	400	`make_list.ml`	131
`mc91.ml`	30	`match_when.ml`	9
`tak.ml`	923	`is_exhaustive.ml`	10
`abs.ml`	15	`partial_app.ml`	15
`xor.ml`	36	`embed_fun.ml`	9
`rec_add.ml`	44	`to_fun.ml`	11
`non_terminate.ml`	6	`f_from_g.ml`	11

Figure 8 Benchmarks for the OCaml analysis.

types containing functions, etc. In particular, it inferred correct summaries for `filter_le` and `to_fun` functions. Additionally, we selected 20 programs from Salto's benchmarks [27] that are compatible with the current limitations of our approach, i.e. they do not contain modules, nor imperative features, nor polymorphism. Figure 8 displays the analysis time for an extract of our 60 programs. The first five programs are from Salto's benchmarks. The programs we consider are small: they consist in around a dozen lines each.

Precision. For most numerical recursive functions (e.g. `binomial`, `ackerman`, `mc91`), our relational compositional analysis gives results at least as precise as Salto's non-compositional and non-relational approach, but with only one analysis of the body. This illustrates how relationality and disjunctions can recover precision lost by compositionality, thus enhancing performance when the analyzed function is analyzed multiple times. Besides, for some functions, no precise invariant can be discovered without relationality – e.g. `rec_add`, or `numeric_loop3b` from Salto's benchmark. Our analysis was therefore of the same precision or better than Salto on 14 out of those 16 examples. Finally, as explained in Section 4.2 our analysis is the first to support `when` clauses, and to detect non-exhaustive pattern-matching.

Performance. All our tests were performed on a Intel(R) Core(TM) i7-8565U CPU 1.80GHz with 16 GB of RAM. The analysis for those small programs is lower than a second. The slowest analysis was for the `tak` function, making three recursive calls in its body. Those

runtimes are comparable to the analysis performed by Salto – less than a second. The runtime difference on the (small) benchmarks available is not significant enough to deduce that one analysis is faster, and to evaluate which part of our method results in a speed-up and which part in a slowdown. Moreover, Salto and MOPSA feature different abstractions beside compositionality that may impact the analysis time as well. A more thorough evaluation of runtime trade-offs is thus left for future work. Previous work by Boutonnet and Halbwachs [8] at first-order concluded that summary construction time is often negligible with regard to total analysis time. Besides, in our implementation, maintaining a summary is equivalent to storing it (they are immutable, as functions are currently pure). We optimistically hope that future work would draw a similar conclusion on higher-order programs.

Note that our method's interest goes beyond performance: it automatically generates *contracts* for higher-order functions. Therefore, it plays a role in proving not only the absence of runtime errors, but also refined properties on the program behavior, being able to automatically discover functions specifications.

8 Related Work

Type systems. To prove properties on functional programs, type systems are widely used. The simplest ones already prevent some errors, such as adding a function to an integer. More expressive type systems, used for program verification, include dependent types, and in particular refinement types. They have enjoyed a steady popularity over the years [44, 43], relying under-the-hood on SMT solvers to reason on properties, mainly numeric ones. Although formulated within the framework of abstract interpretation, we believe our work shares similarities, as it provides a compositional analysis and infers numeric invariants.

Deductive methods. Deductive methods, such as Cameleer [38] or F* [41], can prove precise properties, such as correctness with regard to a specification, but need often both user annotations and SMT solvers. It differs from our goal, a fully-automated and solver-free method, to infer semantic properties.

Non-value abstract interpretation analysis. Whereas many works approach static analysis of functional languages, they often focus on control flow analysis, which, as precised in Liang and Might [28], does not suffice to keep track of values through flow. Cousot and Cousot [13] define higher-order functions abstractions, e.g. as relations or as sets. They also formulate refinements, e.g. disjunctive completion. They are however mainly interested in comportment analysis such as strictness or termination. We use similar abstractions and refinements, but to define a value analysis. Besides, we support algebraic data types with an abstraction fully parametric in possibly relational domains. Montagu and Jensen [36] develop methods to infer a form of frame condition for pure functional higher-order languages, i.e. identifying equality relations between parts of algebraic values. They suffer from the same lack of information when applying an input function as we do. They are however limited to non-recursive types, and cannot handle complex numeric relations.

Abstract interpretation-based value analysis. In our experimental evaluation, we compared to Salto [27], a static value analyzer verifying OCaml programs. It supports a wider subset of the language, such as side-effects and modules. Their method differs from ours, being non-relational and non-compositional. Our compositional approach could be more scalable. Journault et al. [24] propose a relational abstract domain for trees but are limited to this

specific data structure. Bautista et al. [3], Bautista et al. [4] describe a relational abstraction for numeric algebraic values and infer structural equalities between non-numeric fields, but are limited to non-recursive objects. Valnet et al. [42] support recursion, but not higher-order.

Compositional analysis. Input-output relational analysis has been long studied, first applied to `while` programs [26]. Our method can be seen as an extension of symbolic relational separate analysis from Cousot and Cousot [14], extended to support ADTs and higher-order. This is a long-known method [14] to improve analysis scalability. Compositionality has been used in multiple settings: Farzan and Kincaid [17] use them to analyze independently decomposed parts of a program and Kincaid et al. [25] for inter-procedural analysis, analyzing procedures independently of their calling context, once and for all. Codish et al. [11] develop a compositional analysis for logic programs, therefore proving it useful in a declarative context. Bautista et al. [2, 5] define an ADT domain in an input-output analysis for non-recursive imperative programs. In the MOPSA platform [22], a prototype modular analysis was implemented for C programs manipulating strings [23]. To improve precision, Bourdoncle [7] expresses non-relational disjunctive summaries. Boutonnet and Halbwachs [8], on which we built upon, permit relationality. All those methods are however limited to first-order.

9 Conclusion and Future Work

This article presents two abstract domains, one tailored to handle recursive ADTs, and the other one to represent functions used as first-class values, as well as higher-order functions. Thanks to the parametricity of both domains, each domain can leverage the other, e.g. to abstract functions manipulating ADTs (and conversely). The combination of these two domains yields a compositional and relational analysis for a pure functional programming language. This analysis has been implemented into the MOPSA framework to analyze a pure subset of the OCaml language. Our preliminary evaluation shows the precision of our approach on 60 programs, including 20 benchmarks from Salto [27].

We now review the limitations of our approach and discuss future work. The language we have studied here is pure; extending our analysis to support references will be challenging. Our analysis focuses on a monomorphic functional programming language; we could rely, in future work, on the polymorphic equality domain from Montagu and Jensen [36] to handle polymorphism. Our current analysis does not detect arithmetic overflows: we believe this is an important, yet orthogonal concern. Our analysis is technically able to infer ranges of numerical variables and can thus detect overflows. However, our OCaml analysis does not support the wraparound semantics of overflows for now. To the best of our knowledge, there are no compositional value analysis currently tackling this issue, even in the case of first-order imperative languages. Previous work on compositional static analysis for first-order languages (e.g. Boutonnet and Halbwachs [8]) proved that compositionality is useful for scaling up. We postulate that similar speed-ups could be achieved in a higher-order setting with compositional analyses. While this article provides a contribution towards this goal by proposing a precise and compositional analysis, we have not evaluated its scalability in our preliminary experiments. We believe this work is a first step towards automatically proving functional properties (e.g., sorting [19]) in a functional setting with higher-order functions.

References

1 Samson Abramsky and Achim Jung. *Domain theory.* Oxford University Press, 1994.
2 Santiago Bautista. *Static Analysis of Algebraic Data Types and Arrays.* PhD thesis, ENS Rennes, 2023.

3 Santiago Bautista, Thomas Jensen, and Benoît Montagu. Numeric domains meet algebraic data types. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains*, pages 12–16, 2020. doi:10.1145/3427762.3430178.

4 Santiago Bautista, Thomas Jensen, and Benoît Montagu. Lifting numeric relational domains to algebraic data types. In *International Static Analysis Symposium*, pages 104–134. Springer, 2022. doi:10.1007/978-3-031-22308-2_6.

5 Santiago Bautista, Thomas Jensen, and Benoît Montagu. An input–output relational domain for algebraic data types and functional arrays. *Formal Methods in System Design*, pages 1–74, 2024.

6 Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival, et al. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends® in Programming Languages*, 2(2-3):71–190, 2015. doi:10.1561/2500000002.

7 François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

8 Rémy Boutonnet and Nicolas Halbwachs. Disjunctive relational abstract interpretation for interprocedural program analysis. In *Verification, Model Checking, and Abstract Interpretation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings 20*, pages 136–159. Springer, 2019. doi:10.1007/978-3-030-11245-5_7.

9 David Bühler. *Structuring an abstract interpreter through value and state abstractions: eva, an evolved value analysis for Frama-C*. PhD thesis, Université de Rennes 1, 2017.

10 Marc Chevalier and Jérôme Feret. Sharing ghost variables in a collection of abstract domains. In *VMCAI*, volume 11990 of *Lecture Notes in Computer Science*, pages 158–179. Springer, 2020. doi:10.1007/978-3-030-39322-9_8.

11 Michael Codish, Saumya K Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 451–464, 1993. doi:10.1145/158511.158703.

12 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977. doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*, pages 95–112. IEEE, 1994.

14 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, pages 159–179. Springer, 2002. doi:10.1007/3-540-45937-5_13.

15 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978. doi:10.1145/512760.512770.

16 Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8):62–70, 2019. doi:10.1145/3338112.

17 Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64. IEEE, 2015. doi:10.1109/FMCAD.2015.7542253.

18 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*, pages 512–529. Springer, 2004. doi:10.1007/978-3-540-24730-2_38.

19 Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 339–348. ACM, 2008. doi:10.1145/1375581.1375623.

20 Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

21 Matthieu Journault. *Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference*. PhD thesis, Sorbonne Université, 2019.

22 Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, pages 1–18. Springer, 2020. doi:10.1007/978-3-030-41600-3_1.

23 Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In *SAS*, volume 11002 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2018. doi:10.1007/978-3-319-99725-4_16.

24 Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. An abstract domain for trees with numeric relations. In *European Symposium on Programming*, pages 724–751. Springer, 2019. doi:10.1007/978-3-030-17184-1_26.

25 Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices*, 52(6):248–262, 2017. doi:10.1145/3062341.3062373.

26 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997. doi:10.1145/256167.256195.

27 Pierre Lermusiaux and Benoît Montagu. Detection of uncaught exceptions in functional programs by abstract interpretation. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14577 of *LNCS*, pages 391–420. Springer, 2024. doi:10.1007/978-3-031-57267-8_15.

28 Shuying Liang and Matthew Might. Entangled abstract domains for higher-order programs. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, DC*, 2013.

29 Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press, 2 edition, 2022.

30 Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*, pages 5–20. Springer, 2005. doi:10.1007/978-3-540-31987-0_2.

31 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

32 Raphaël Monat. *Static type and value analysis by abstract interpretation of Python programs with native C libraries*. PhD thesis, Sorbonne Université, 2021.

33 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of Python programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, pages 17–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

34 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Value and allocation sensitivity in static python analyses. In *SOAP@PLDI*, pages 8–13. ACM, 2020. doi:10.1145/3394451.3397205.

35 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of python programs with native C extensions. In *SAS*, volume 12913 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2021. doi:10.1007/978-3-030-88806-0_16.

36 Benoît Montagu and Thomas Jensen. Stable relations and abstract interpretation of higher-order programs. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020. doi:10.1145/3409001.

37 Abdelraouf Ouadjaout and Antoine Miné. A library modeling language for the static analysis of C programs. In *SAS*, volume 12389 of *Lecture Notes in Computer Science*, pages 223–247. Springer, 2020. doi:10.1007/978-3-030-65474-0_11.

38 Mário Pereira and António Ravara. Cameleer: A deductive verification tool for OCaml. In *International Conference on Computer Aided Verification*, pages 677–689. Springer, 2021. doi:10.1007/978-3-030-81688-9_31.

39 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, 2011. doi:10.1145/1926385.1926390.

40 Fausto Spoto. The Julia static analyzer for Java. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 39–57. Springer, 2016. doi:10.1007/978-3-662-53413-7_3.

41 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

42 Milla Valnet, Raphaël Monat, and Antoine Miné. Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs. In *JFLA 2023-34èmes Journées Francophones des Langages Applicatifs*, pages 211–242, 2023.

43 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In *ICFP*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.

44 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257. ACM, 1998. doi:10.1145/277650.277732.