DelExp: a Relational Container Abstraction

with Applications to Compositional Analysis

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

Data containers, such as lists, arrays, trees, etc, raise challenges for program verification. In static analysis by abstract interpretation, one popular approach is summarization: multiple elements of a data structure are abstracted into a single one, favoring performance over precision. This technique is at the core of most container abstractions – from smashing to segmentation – of arrays, lists or algebraic data types.

However, summarization approaches are unable to express relations between containers, even when relational numerical abstract domains are used. Our work introduces DelExp, a new domain able to express relations between summarized variables. DelExp can state that the content of a data structure is included in the content of another data structure, up to a given transformation. DelExp is language-agnostic, modular in the abstraction chosen for any other types (integers, strings, functions, etc.), and can be seamlessly combined with existing container abstractions. We show how DelExp allows us to infer precise summaries for compositional analyses of container-manipulating functions in a pure functional language. We present extensions to DelExp supporting polymorphism and higher-order transformations.

Our implementation of DelExp within the MOPSA static analysis platform confirms that DelExp works out of the box with pre-existing container abstractions. Our evaluation targets both Python programs manipulating lists and relational summary generation for OCaml functions handling algebraic data types.

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, Abstract Interpretation, Functional Programming

Digital Object Identifier 10.4230/LIPIcs...12

1 Introduction

Most programs use various kinds of containers such as lists, maps, dictionaries or sets, i.e. data structures of possibly unbounded size storing a collection of objects. Collecting information on those collections can be crucial to verify some properties on the program behaviours, e.g. assertions over elements inside the containers. In this work, we aim at discovering *relational properties* over containers in order to improve the precision of static value analysis. Our approach is rooted in the abstract interpretation framework: we compute over-approximations of the possible behaviour of a program. The resulting analysis is fully automated and does not require annotations.

Value analysis on containers raises specific challenges: standard abstract domains manipulate a fixed number of objects whereas a container can be of unbounded size. A standard approach from Blanchet et al. [3] is to fold, or "smash", together a set of objects into one *summarized* object. A single abstract object then represents multiple concrete objects. Gopan et al. [13] proposed a systematic approach to transform standard numerical

```python
# assumes l:List[int]
r=[]
for x in l:
  r.append(2*x)
```

Figure 1 Motivating list-manipulating function, written in Python.

domains, including relational domains, into summarizing domains, manipulating summarized variables – that we will call weak variables. However, this approach cannot express relations between the set of values inside containers – e.g. inclusion between one set and another.

Cox et al. [11] defined a domain called QUIC (Quantified Union/Intersection Constraint) graphs. It can express constraints over numerical sets, under the form $\bigcap X_i \subseteq \{\nu \in \bigcup Y_j \mid B[\nu]\}$, B being an abstract domain acting as a predicate on the elements ν of the collection. When abstracting containers as sets, this domain can express some relations between containers. However, this domain is limited to inclusions between union/intersection of sets refined by a predicate. As is, this domain cannot represent an element-wise transformation on a given set. By contrast, the domain presented in our article can handle transformations similar to map.

Consider the Python program in Fig. 1. Given a list l of integer, it builds the list r by multiplying every element in l by 2. Neither standard relational domains on integers, such as the polyhedra domain, nor the QUIC graphs domain can express that the elements of the list r are elements of the list l multiplied by two. We develop here a family of domains, called DelExp, which, given a transformation θ, can state relations of the form $x \prec \theta(y)$, meaning that x abstracts a set of elements included in $\theta(y) = \{\theta(e) \mid e \in y\}$. In particular, by introducing the weak variables l_c and r_c to represent all elements of l and r, our derived analysis can infer that r contains only elements that are the double of elements from l, i.e. $l_c \prec 2 \times r_c$. This family of domains is *language-agnostic*: it only operates on weak variables (here, l_c and r_c), and therefore, can be used in product with any analysis manipulating weak variables. Indeed, our implementation has been exploited within both a Python and an OCaml analysis.

An important use case of our relational domains is the definition of compositional (or function-modular) analyses. These analyses consist in analyzing functions once, at their definition site, to infer a summary, i.e. a contract, valid for every input, that over-approximates the function behaviour. The summary is then applied to analyze each function call. One difficulty faced by compositional analyses is precision loss: analyses need to produce a unique summary, correct for every possible input. A key ingredient to achieve interesting precision levels is to rely on relational abstract domains which are able to express relations between variables. Relational domains have been used extensively in previous works to create input-output relational summaries [8, 12, 20, 21].

Recently, a compositional analysis was developed for higher-order languages such as Haskell or OCaml [31]. However, since this analysis is based on summarization [13] to abstract user-defined Algebraic Data Types (ADTs), it fails to express input-output relations between two ADTs. Consider the OCaml program in Fig. 2. The previous analysis [31] creates two weak variables, namely `l.Cons.1` and `r.Cons.1`, abstracting the content of the input list l and the output list r (contained in the 1st field of `Cons`). It only infers $\top$ as summary for `mult2`, as it is unable to express a relation between `l.Cons.1` and `r.Cons.1`. However, with our DelExp domains, the analysis is able to infer: $\forall l, r. r = \texttt{mult2}(l) \implies \texttt{r.Cons.1} \prec 2 \times \texttt{l.Cons.1}$.

```
type list = Cons of int * list | Nil
let rec mult2 l =
  match l with
  | Cons(h,q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
```

Figure 2 Motivating list-manipulating function, written in OCaml

It means that the integer `Cons` fields of variable r (represented by `r.Cons.1`) are included in the values of the integer `Cons` fields of variable l (represented by `l.Cons.1`) multiplied by two.

Note that the variables that are used in the generated summary for `mult2` are not only weak, i.e., they represent an unbounded set of concrete memory locations, but are also *optional*, i.e., this set of location may be empty. Indeed, if `l` is `Nil`, then `l.Cons.1` is not defined, neither is `r.Cons.1`. Environments with optional variables, called *heterogeneous* environments [19], require careful handling, as the standard join and meet operators are unsound and need to be adapted. Our domains support them.

The OCaml list type is a user-defined Algebraic Data Type whose abstraction was delegated to the weak variable `l.Cons.1`. Our DelExp domain then manipulates those variables. This works similarly with tree data-structures. Consider the following ADT:

```
type tree = Node of tree * int * tree | Leaf
```

The ADT domain delegates the abstraction of all node labels of a variable x to `x.Node.2`, on which our DelExp domain can operate. Consequently, it can infer a similar summary for a function multiplying by 2 every label of a tree.

Finally, a relation $x \prec y$, meaning that the elements of x form a subset of the elements of y, does not need hypotheses on the type and underlying abstract domain for x and y. They can abstract containers of integers, strings, etc. In fact, they can even abstract polymorphic containers – e.g. a polymorphic list `'a list`.

Contributions. This paper presents the following contributions:

- We define the *DelExp* abstract domain, stating simple constraints of the form $x \prec y$. This formalization supports optional weak variables – i.e. possibly empty containers. It is language-agnostic and domain-modular – i.e. no assumption is made on the underlying domain and type of weak variables. Therefore, it can be used on polymorphic lists, algebraic data types or array segments.
- We generalize it to a family of abstract domains stating constraints "up to transformation", of the form $x \prec \theta(y)$. Depending on the chosen transformation, those domains can also express relations between polymorphic variables.
- As a case study, we develop a compositional analysis for a pure polymorphic functional language leveraging those domains. It can generate summaries for higher-order polymorphic recursive functions manipulating user-defined algebraic data types.
- We have implemented the resulting analysis in the MOPSA platform [18] and assessed its efficiency and precision both on list-manipulating Python programs and on ADTs-manipulating functions from the OCaml standard library. DelExp was integrated seamlessly into MOPSA: it did not require any change to the other abstract domains for the Python and OCaml analyses to work.

$$
\begin{array}{rll}
e_1, e_2, \cdots \in \mathcal{E} ::= & x \in \mathbb{V} \mid n \in \mathbb{Z} \mid rand(1,n) \mid e_1 \oplus e_2 \mid t[n] \mid [e_1; \ldots; e_n] \\
s_1, s_2, \cdots \in \mathcal{S} ::= & s_1; s_2 & \\
& \mid x \leftarrow e & \mid \text{while } e \text{ do } s \\
& \mid t[e] \leftarrow e & \mid \text{if } (e)\{s_1\} \text{ else } \{s_2\}
\end{array}
$$

Figure 3 Syntax of the Universal toy language

Set of program variables	$\mathbb{V}$
Values	$\mathcal{V} ::= \mathbb{Z} \cup \{[v_1; \ldots; v_n] \mid n \in \mathbb{N}, v_i \in \mathcal{V}\}$
Concrete environments	$\Sigma = \mathbb{V} \to \mathcal{V}$
Concrete semantics of expr. e	$\mathbb{E}[\![e]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\mathcal{V})$
Concrete semantics of statement. s	$\mathbb{S}[\![s]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma^\perp)$

Figure 4 Formalization of the concrete semantics

Limitations. As it is based on weak variables, our container analysis, as a first step, over-approximates containers as sets. The abstraction is content-based: it expresses content inclusion between containers – possibly up to transformation. Therefore, it cannot express order-dependent specification – e.g. list sortedness – nor multiplicity-sensitive properties – e.g. equality between two lists. It cannot express relations between specific elements of the container – e.g. the n-th element of a container relates to the m-the element of another. Inferring such fine-grained relational properties on generic containers would require developing new domains and is considered future work.

Outline. Section 2 presents a relational analysis on a simple toy language with weak variables. Section 3 proposes a simple abstract domain expressing relations on summarization variables. Section 4 shows how we can derive more expressive, "up to transformation" relations from this domain. Section 5 defines a compositional analysis on a pure functional language and motivates the use of the DelExp domain in this context. Section 6 adds the support of heterogeneous environment to the DelExp domain, to support possibly empty containers. Section 7 derives a sound and automatic analysis for our polymorphic functional language. Section 8 presents our implementation and experimental results for both OCaml and Python analysis. Section 9 presents related works and Sec. 10 concludes.

2 Relational Analysis with Weak Variables

This section introduces backgrounds on relational analyses for containers in a toy imperative language. Section 2.1 introduces a toy imperative language. Section 2.2 motivates and defines relational analyses. Section 2.3 introduces weak variables and how they can be abstracted in relational domains. These weak variables are leveraged in Sec. 2.4 to showcase different container abstractions, on Python lists and on OCaml Algebraic Data Types.

2.1 An Imperative Toy Language

We define in Fig. 3 a simple imperative toy language with integers and arrays. $\mathbb{V}$ is the set of variables, $\oplus$ any operator of the language ($+$, $-$, $\times$, $/$, $==$, $<$, $>$, etc.). Figure 4 describes

```
1 x <- rand(1,10);
2 y <- 2*x ;
3 if {x=10} z <- y else z <- 20 ;
```

Figure 5 Example program motivating relational analysis.

Abstract environments	$\Sigma^\sharp = (\mathbb{V} \to \mathcal{D}) \times \mathcal{D}_r$
Abstract semantics of expr. e	$\mathbb{E}^\sharp[\![e]\!] : \Sigma^\sharp \to \mathbb{V} \times \Sigma^\sharp$
Abstract semantics of stmt. s	$\mathbb{S}^\sharp[\![s]\!] : \Sigma^\sharp \to \Sigma^\sharp$

Figure 6 Formalization of the abstract semantics.

the semantic domain of the language. The semantic values of the language $\mathcal{V}$ are integers and arrays. Σ is the set of concrete environments, associating a value to each variable. The concrete semantics of an expression e is $\mathbb{E}[\![e]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\mathcal{V})$. It computes all possible semantic values of an expression in a set of possible environments (e.g. $rand(1, n)$ can yield n different values). The concrete semantics of a statement s is $\mathbb{S}[\![s]\!] \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma^\perp)$, where $\perp$ denotes non-termination. It updates input environments by executing the statement. Its semantics is standard, and not detailed here.

2.2 Relational Analysis

Given a language, we want to 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, ADTs) to the relevant abstract domain $\mathcal{D}_\tau$. 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. However, this formulation cannot easily express a relation between the value of the expression and those of the variables. Indeed, let us consider the program in Fig. 5. A numerical domain such as the interval domain can abstract `x` and `y` independently, as $\mathtt{x} \in [1, 10]$ and $\mathtt{y} \in [2, 20]$, but cannot infer the relation `y = 2*x`. It then fails to infer that `z`=20. To overcome this limitation, we need an abstract numerical domain able to store relations between variables – e.g. the polyhedra domain [10].

In our solution to this problem (see Fig. 6), we will separate non-relational domains $\mathcal{D}$ and relational domains $\mathcal{D}_r$. We will maintain as abstract environments a pair, containing a map $m \in \mathbb{V} \to \mathcal{D}$ from variables to abstract values (i.e. elements of non-relational domains) and a tuple $d \in \mathcal{D}_r$ of relations from the relational domains. Thus, $\sigma^\sharp = (m, d) \in \Sigma^\sharp$. Consequently, the abstract semantics of an expression e, $\mathbb{E}^\sharp[\![e]\!]$, returns a variable aside with an abstract environment storing the relations over this variable. The abstract semantics of a statement $\mathbb{S}^\sharp[\![s]\!]$ updates an abstract environment with statement s. For example, let p be the program in Fig. 5, and [.] a starting environment with no information on program variables. The analysis of p is:

$$\mathbb{S}^\sharp[\![p]\!][.] = (\mathtt{x} \to [1, 10], \mathtt{y} \to [2, 20], \mathtt{z} \to [20, 20]; \mathtt{y} = 2\mathtt{x})$$

2.3 Content Summarization with Weak Variables

```
1  x <- [a-1;a;a+1];  (* array definition  *)
2  y <- x[2];         (* get array cell    *)
3  x[0] <- a+2;       (* modify array cell *)
```

Figure 7 Small example to illustrate weak variables

Abstracting containers implies representing an unknown number of objects. However, standard abstract domains such as the interval domain or the polyhedra domain can only manipulate a known, finite number of variables. To re-use those abstractions, Blanchet et al. [3] proposed to *summarize* a (possibly infinite) set of objects into one unique variable. Therefore, we can delegate the analysis of containers' content to standard domains. This approach was formalized by Gopan et al. [13] and extended to relational domains. Those summarization variables, called *weak variables*, represent multiple concrete elements and consequently concretize to a set of values. Standard variables are then called *strong variables*. We define $\mathbb{V}^{\text{weak}}$ (resp. $\mathbb{V}^{\text{strong}}$) the sets of weak (resp. strong) variables. We add those variables to our language from Fig. 3.

Consider the array-manipulating program in Fig. 7. The array `x` is composed of 3 concrete cells (i.e. 3 objects). Those concrete elements can be summarized together into one single weak variable $x_c \in \mathbb{V}^{\text{weak}}$ abstracting the content of `x`. Weak variables then concretize to a *set* of values. We update our definition of concrete environments:

▶ **Definition 2.1** (Environments). *The set of concrete environments is defined as* $\Sigma = (\mathbb{V}^{weak} \to \mathcal{P}(\mathcal{V})) \times (\mathbb{V}^{strong} \to \mathcal{V})$.

We define the lift $\uparrow \cdot : \mathcal{P}(\mathcal{V}) \cup \mathcal{V} \to \mathcal{P}(\mathcal{V})$ as $\uparrow X = X$ if $x \in \mathcal{P}(\mathcal{V})$, or $\{X\}$ if $x \in \mathcal{V}$: it extracts the set of possible values of a variable, whether it is weak or strong. A sound abstraction for $x_c \in \mathbb{V}^{\text{weak}}$ in the polyhedra domain is $a - 1 \leq x_c \leq a + 1$: all concrete cells represented by x_c are between $a - 1$ and $a + 1$. Note that those variables must be handled with care: e.g. modifying any of the concrete cells a summary variable represents must affect its corresponding weak variable. Their manipulation then requires specific operators.

▶ **Definition 2.2** (Expand). *Given two variables* v *and* v', $\texttt{expand}(v, v')$ *adds to the environment a variable* v' *with the same potential values as* v*:*

$$\mathbb{S}[\![\texttt{expand}(v, v')]\!]S = \{\sigma[v' \to Z] \mid \sigma \in S \wedge (v' \in \mathbb{V}^{weak} \implies Z \subseteq \uparrow \sigma(v)) \\ \wedge (v' \in \mathbb{V}^{strong} \implies Z \in \uparrow \sigma(v))\}$$

It corresponds to copying every constraint verified by v to v'. This is the operation used when reading a cell of a container (cf. line 2 from Fig. 7): if the summarization variable respects a constraint, so does the value contained in the cell. Knowing $a - 1 \leq x_c \leq a + 1$, we cannot derive that $x_c = \texttt{y}$. Indeed, if we discover that $\texttt{y}{=}a + 1$ it doesn't mean that $x_c = a + 1$, i.e. that every element in `x` is equal to $a + 1$. However, we can safely derive that `y` verifies the same constraints as x_c, i.e. $a - 1 \leq \texttt{y} \leq a + 1$.

▶ **Definition 2.3** (fold). *Given two variables* v *and* v', $\texttt{fold}(v, v')$ *adds the values of* v' *as possible values for* v*:*

$$\mathbb{S}[\![\texttt{fold}(v, v')]\!]S = \{\sigma[v \to Z] \mid \sigma \in S \ \wedge \ Z \subseteq \uparrow \sigma(v) \cup \uparrow \sigma(v')\}$$

This is the operation used when adding or modifying the cell of a container (cf. line 3): given a cell from the modified container, it is either a cell from the former container or a

new cell. In the abstract, it either respects constraints from the former container or from the new one. Consequently, after line 3, $a-1 \leq x_c \leq a+1 \cup x_c = a+2$, i.e. $a-1 \leq x_c \leq a+2$.

Abstraction of those operators are provided by each domain implementing the support for weak variables. Gopan et al. [13] defines computable abstractions for classic numerical domains (e.g. polyhedra, octagons). Once those operators are defined, given an abstract semantics $\mathbb{S}^\sharp[\![.]\!]$ on a language without weak variables, we can derive an abstract semantics of assignments for weak variables:

▶ **Definition 2.4** (Assign by weak update)**.**

$$\mathbb{S}^\sharp[\![x^{strong} \leftarrow e]\!]\sigma^\sharp = \mathbb{S}^\sharp[\![x \leftarrow \texttt{expanded}(e)]\!]$$
$$\mathbb{S}^\sharp[\![x^{weak} \leftarrow e]\!]\sigma^\sharp = \mathbb{S}^\sharp[\![\texttt{fold}(x, tmp)]\!] \circ \mathbb{S}^\sharp[\![tmp \leftarrow \texttt{expanded}(e)]\!]\sigma^\sharp$$

$\mathbb{S}^\sharp[\![x^{\text{strong}} \leftarrow e]\!]$ means that the value of x is in the set of values that expression e evaluates into. When the right-hand side e contains weak variables, the $\texttt{expanded}(e)$ models reading from a collection by replacing every occurrence of a weak variable y in expression e with a temporary strong variable tmp, initialized with $\texttt{expand}(y, tmp)$. Given $x \in \mathbb{V}^{\text{strong}}$, $\sigma^\sharp \in \Sigma^\sharp$, $\mathbb{S}^\sharp[\![\texttt{remove}(x)]\!]\sigma^\sharp$ removes the variable x from the abstract states $\Sigma^\sharp$. This statement is used to remove temporary variables after the assignment. This prevents inferring unsound relations: the assignment $x^{\text{strong}} = y^{\text{weak}}$ means that the value of x is the value of one of the concrete elements represented by y (i.e. $\sigma(x) \subseteq \sigma(y)$) whereas storing $x^{\text{strong}} = y^{\text{weak}}$ in a relational domain (e.g. the polyhedra domain) would mean that x and y have the same concretization (i.e. $\sigma(x) = \sigma(y)$).

Finally, if the left-hand side is a weak variable x^{weak}, then the assignment means that we add a possible value for x: we fold the right-hand side value into x.

▶ **Example 2.5.** Consider the abstract environment $\sigma^\sharp = (y^{\text{weak}} : [1, 12])$. Then:

$$\mathbb{S}^\sharp[\![x^{\text{strong}} \leftarrow 2 * y^{\text{weak}}]\!]\sigma^\sharp = \mathbb{S}^\sharp[\![x \leftarrow \texttt{expanded}(2 * y^{\text{weak}})]\!]\sigma$$

$\texttt{expanded}(2 * y^{\text{weak}})$ creates a strong temporary variable tmp, initialized with the same values as y^{weak}, which is removed after the assignment, i.e.:

$$\begin{aligned}\mathbb{S}^\sharp[\![x^{\text{strong}} \leftarrow y^{\text{weak}}]\!]\sigma^\sharp &= \mathbb{S}^\sharp[\![\texttt{remove}(tmp)]\!] \circ \mathbb{S}^\sharp[\![x^{\text{strong}} \leftarrow 2 * tmp]\!](\sigma^\sharp[tmp : [1, 12]]) \\ &= (x^{\text{strong}} : [2, 24], y^{\text{weak}} : [1, 12])\end{aligned}$$

Those summary variables are used in the literature, to abstract strings [26], numerical trees [19] or algebraic data types [31]. They can be used in combination with a relational domain. For example, the polyhedra domain can express that a weak variable x is lower than some strong variable y, meaning that any value represented by x is lower than y (e.g. $a-1 \leq x_c \leq a+2$ in the example). However, it cannot infer relations between the content of two weak variables. Indeed, consider the semantics of the assign: the weak variables from the right-hand side are expanded to temporary fresh strong variables before performing an assign. Ultimately, as a limitation, it cannot express how the content of a container relates to the content of another container. The difficulty to express relations between weak variables was underlined by Siegel and Simon [30]. This very problem will be overcome by our DelExp domain (Section 3).

2.4 Container Abstraction

Weak variables can be leveraged to abstract containers in various contexts.

Arrays. Weak variables can be used to abstract arrays or lists in any language. To any array a, we associate a weak variable a^{weak} which summarizes every object contained in a. As seen in the previous section, the operations of array manipulation derive from `fold` and `expand` operators. When updating the cell of an array a with the expression e, we compute $a^{\text{weak}} \leftarrow e$, which folds inside t^{weak} the value of e. When reading the n-th cell $a[n]$ of a inside a variable x, we perform $x \leftarrow t^{\text{weak}}$, which expands the values of t^{weak} in x. This works for static arrays of fixed size (as in C), as well as dynamic arrays with changing, unbounded size, such as Python lists.

In particular, weak variables can be used to abstract Python lists, as highlighted by this example (abstraction in comment):

```
a = [12,3,7] #t_c:[3,12]
a.append(15) #t_c:[3,15]
h = a[0]     #h:[3,15]
```

OCaml Algebraic Data Types. Valnet et al. [31] defines an abstraction for algebraic data types based on weak variables. Given any *user-defined* algebraic data type, they associate one weak variable per non-recursive construct field:

```
type t =  C1 of t1,1 * ... t1,n | ... | Cm of tm,1 * ... * tm,n
```

The field (i,j) of type `ti,j` of a variable x is then abstracted by a weak variable called `x.Ci.j`. Therefore, all the elements of the field (i,j) recursively nested inside the variable x are smashed (i.e. folded) together inside the variable `x.Ci.j`. They also record the possible variants `Ci` used by each variable x, which thus limits the set of weak variables `x.Ci.j` used to abstract x. Consider the following tree:

```
type tree = Node of tree * int * tree | Leaf
let t0 = Node(Node(Leaf,12,Leaf)3,Node(Leaf,7,Leaf))  (* t0.Node.1:[3,12], t0:{Node} *)
let t = Node(Leaf,15,t0)                              (* t.Node.1:[3,15], t:{Node} *)
let h = match t with Node(_,h,_) -> h                 (* h:[3,15] *)
```

All integer fields of `Node` recursively defined in `t0` are represented by `t0.Node.1`. The abstraction of this weak variable in the interval domain is [3,12]. `t0:{Node}` means that `t0` starts with the variant `Node`.

Note that OCaml and Python examples share a similar precision loss. Indeed, when h is defined by `expand` of the weak variable, the abstract constraints on variable t are propagated to h, but we forget that h relates to t. With the polyhedra domain, we cannot store how h and t relate together, i.e. that h belongs to the set of elements that t represents (or, equivalently, that $\sigma(h) \subseteq \sigma(t)$). If we discover information about t afterwards, we will not be able to propagate them to h.

To overcome this limitation, the next section will define a domain able to store such a relation. The domain we will present is language-agnostic and manipulates weak variables independently of which set of objects they abstract. In the following, we will show examples both on Python lists and on OCaml ADTs.

3 The DelExp Domain

In this section, we define a domain able to express *relations* between weak variables. As a start, we want to express that a variable x verifies the same constraints as another variable y. Our domain is defined in order to be combined with existing analysis to improve their precision. Therefore, we suppose that an abstract semantics computing abstract environments in $\Sigma^\sharp$ is already defined, of concretization $\gamma^\sharp_\Sigma : \Sigma^\sharp \to \mathcal{P}(\Sigma)$.

3.1 DelExp Domain

We define $\Sigma^\sharp_{\mathbb{C}} = \mathbb{V} \to \mathcal{P}(\mathbb{V}^{\text{weak}})$, a map from variables to a set of weak variables. Given $\mathbb{C} \in \Sigma^\sharp_{\mathbb{C}}$, if $x \in \mathbb{C}(v)$, it means that it concretizes to an environment σ where $\uparrow \sigma(v) \subseteq \uparrow \sigma(x)$, i.e. that v verifies all the constraints that x does, or, equivalently, that the elements represented by v form a subset of the elements represented by x. This is expressed by the following concretization:

▶ **Definition 3.1** (Concretization). *We define the concretization* $\gamma : \Sigma^\sharp \times \Sigma^\sharp_{\mathbb{C}} \to \mathcal{P}(\Sigma)$. *For* $(\sigma^\sharp, \mathbb{C}) \in \Sigma^\sharp \times \Sigma^\sharp_{\mathbb{C}}$*:*

$$\gamma\left((\sigma^\sharp, \mathbb{C})\right) = \{\sigma \in \gamma^\sharp_\Sigma(\sigma^\sharp) \mid \forall v \in \mathbb{V}, \forall x \in \mathbb{C}(v), \uparrow \sigma(v) \subseteq \uparrow \sigma(x)\}$$

We will write $x \in \mathbb{C}(v)$ as $v \prec x$. Considering the definition of `expand`, $\uparrow \sigma(v) \subseteq \uparrow \sigma(x)$ also means that $\sigma = \mathbb{S}[\![\texttt{expand}(v, x)]\!]\sigma$. This domain states the possibility to perform some expansions on the environment: therefore, we will call it the DelExp (Delay Expand) domain.

▶ **Example 3.2.** This domain is language agnostic: it operates directly on weak variables, independently of what they abstract. Suppose l is non-empty. Consider those OCaml and Python programs, both computing the tail of the list l:

```
let x = match l with | Cons(h, q) -> q | Nil -> Nil
```

```
x=l[1:]
```

In OCaml, we can express `x.Cons.1` $\prec$ `q.Cons.1`. Similarly, in Python, we can express $x_0 \prec l_0$. Those are sound over-approximations of the final environments: the elements of the tail x of the list l satisfy every constraint that the elements of the list l do.

▶ **Definition 3.3** (Partial order). *We define the abstract partial order on* $\mathbb{C}$*:*

$$\mathbb{C} \subseteq_c \mathbb{C}' \iff \forall v \in \mathbb{V}, \mathbb{C}'(v) \subseteq \mathbb{C}(v)$$

That is, $\mathbb{C}$ is more precise than $\mathbb{C}'$ if it contains more constraints.

▶ **Example 3.4.** Given $\mathbb{V}^{\text{weak}} = \{x, y\}$ and $\mathbb{V}^{\text{strong}} = \{a\}$, we define $\mathbb{C} : \mathbb{V} \to \mathcal{P}(\mathbb{V}^{\text{weak}}) = x \to \{y\}; y \to \emptyset; a \to \{y\}$, meaning that x and a verify all the constraints that y does, and $\mathbb{C}' =: \mathbb{V} \to \mathcal{P}(\mathbb{V}^{\text{weak}}) = x \to \{y\}; y \to \emptyset; a \to \emptyset$, meaning that x verifies all constraintsthat y does. Then $\mathbb{C} \subseteq_c \mathbb{C}'$: $\mathbb{C}$ expresses strictly more constraints on the environments than $\mathbb{C}'$, therefore it represents a smaller set of environments.

3.2 Operators

We now define the operators of our abstract domain. Note that before performing those operations, as is standard in relational domains (e.g. polyhedra domain), we perform the transitive closure of constraints under the rule: $x \prec y \wedge y \prec z \implies x \prec z$.

▶ **Definition 3.5** (Join). *We define the join* $\sqcup_c$ *on* $\Sigma^\sharp_{\mathbb{C}}$*:* $\mathbb{C} \sqcup_c \mathbb{C}' = v \mapsto \mathbb{C}(v) \cap \mathbb{C}'(v)$

When performing a join, we need to keep constraints present in both environments.

▶ **Property 3.1.** $\sqcup_c$ *is a sound over-approximation of* $\cup$ *and is a widening (because* $\mathbb{V}$ *and* $\mathcal{P}(\mathbb{V}^{weak})$ *are finite).*

▶ **Definition 3.6** (Meet). *We define the meet* $\sqcap_c$ *on* $\Sigma^\sharp_{\mathbb{C}}$*:* $\mathbb{C} \sqcap_c \mathbb{C}' = v \mapsto \mathbb{C}(v) \cup \mathbb{C}'(v)$

Similarly to the join, when performing the meet, we keep all the constraints from both environments: they are satisfied in the result environment.

▶ **Property 3.2.** $\sqcap_c$ *is a sound over-approximation of* $\cap$.

3.3 Transfer Functions

We update our abstract semantics with the transfer functions for this domain.

Discovering constraints. First, we define the `add_expand` function (Algorithm 1) which, given a variable x, an expression e, and a DelExp environment $\mathbb{C}$, returns an environment in which the constraint $x \prec y$ is added if e is a variable y.

Algorithm 1 `add_expand`

Input : $x, e, \mathbb{C} \in \mathbb{V} \times \mathcal{E} \times \mathbb{C}$
if $e = y \in \mathbb{V}$ **then**
  if $x \in \mathbb{V}^{strong}$ **then**
    return $\mathbb{C}[x \to \{y\}]$;
  else
    return $\mathbb{C}[x \to \mathbb{C}(x) \cap \{y\}]$;
  end
else
  return $\mathbb{C}$;
end

We then update the semantics of `expand`:

$$\mathbb{S}^\sharp[\![\texttt{expand}(x, e)]\!](\sigma^\sharp, \mathbb{C}) = \mathbb{S}^\sharp[\![\texttt{expand}(x, e)]\!]\sigma^\sharp, \texttt{add\_expand}(x, e, \mathbb{C})$$

When we expand a variable x with an expression $e = y \in \mathbb{V}$, then x verifies the same constraints as y (Definition 2.2). Consequently, we add $x \prec y$ in the DelExp domain.

Reduction. We now define a reduction function to propagate information from our DelExp domain to underlying domains. First, we define the `apply_constraint` function (Algorithm 2) which, given a constraint $x \prec y$, applies it to the environment: to do so, it enforces that x is the expansion of y in the current environment. Then, the `reduce` function (Algorithm 3) takes as input an abstract environment $\sigma^\sharp$ and a DelExp "to-expand" environment $\mathbb{C}$. It returns a refined version of $\sigma^\sharp$. To do so, it applies sequentially each constraint in $\mathbb{C}$ to update $\sigma^\sharp$. Thanks to this reduction, other domains can benefit from information stored in the DelExp domain. Note that this reduction is agnostic with respect to those underlying domains.

Algorithm 2 `apply_constraint`

Input : $\sigma^\sharp, x \prec y$
$\sigma^\sharp \leftarrow \mathbb{E}^\sharp[\![\texttt{expand}(x, y)]\!]\sigma^\sharp$;
return $\sigma^\sharp$

Algorithm 3 `reduce`

Input : $\sigma^\sharp, \mathbb{C}$
for $x \prec y \in \mathbb{C}$ **do**
  $\sigma^\sharp \leftarrow$ `apply_constraint`$(\sigma^\sharp, x \prec y)$
end
return $\sigma^\sharp$

A possible interpretation for this analysis is that it delays the application of the `expand` operator. When we uncover information about variable y, knowing that $x \prec y$, we can propagate them with the `expand` operator.

▶ **Example 3.7.** Given programs from Example 3.2 computing the tail of a non-empty list l, the OCaml analysis will perform `q.Cons.1` $\leftarrow$ `l.Cons.1` then `x.Cons.1` $\leftarrow$ `q.Cons.1` using `expand`, then remove intermediate variable q, therefore yielding `x.Cons.1`$\prec$`l.Cons.1`. Similarly, the Python analysis will perform $x_0 \leftarrow l_0$, therefore yielding $x_0 \prec l_0$. The inference of those constraints works at the level of weak variables, independently of the underlying containers.

4 Inclusions up to Transformations

The DelExp domain presented so far yields precise results when a variable verifies the same constraints as another variable (e.g. when a list is a sub-list of another). Therefore, the DelExp domain can then be seen as a simplification of the QUIC graphs domain [11], which can express constraints over numerical sets, under the form $\bigcap X_i \subseteq \{\nu \in \bigcup Y_j \mid B[\nu]\}$, B being an abstract domain acting as a predicate on ν. However, many functions on containers apply a *transformation* to its elements. As is, our DelExp domain, as well as QUIC graphs, fail to capture a relation "up to transformation" between two variables. This section proposes a systematic methodology to extend the expressivity of the DelExp domain in this direction.

4.1 AffineDelExp Domain

We recall the Python program from the introduction:

```
# assumes l:List[int]
for x in l:
  r.append(2*x)
```

Given a non-empty list l, it builds r as the list l where every element was multiplied by 2. The standard polyhedra domain on weak variables, the QUIC graphs domain [11] or our current DelExp domain from Section 3 all fail to capture a relation between l and r the input and the output lists. We would like to express constraints under the form $x \prec 2y$ to express that x respects every constraint that $2y$ does, i.e. that the set of values represented by x is included in the set of values in y multiplied by 2. Given a set X, we denote as $P_f(X)$ the set of finite subsets of X. We define an *affine* DelExp domain able to express such constraints:

▶ **Definition 4.1** (Abstract domain). *We define* $\Sigma^\sharp_A = \mathbb{V} \to \mathcal{P}_f(\{aX + b \mid a, b \in \mathbb{Z}, X \in \mathbb{V}\})$.

This domain, that we will name AffineDelExp, stores constraints under the form $x \prec ay + b$. Note that, like the standard polyhedra domain, this domain ignores possible overflows (there are none in Python).

▶ **Definition 4.2** (Concretization). *We define the concretization* $\gamma : \Sigma^\sharp \times \Sigma^\sharp_A \to \Sigma$. *For* $(\sigma^\sharp, \mathbb{C}) \in \Sigma^\sharp \times \Sigma^\sharp_A$*:*

$$\gamma\left((\sigma^\sharp, \mathbb{C})\right) = \{\sigma \in \gamma^\sharp_\Sigma(\sigma^\sharp) \mid \forall x \in \mathbb{V}, x \prec ay + b \in \mathbb{C} \implies \uparrow \sigma(x) \subseteq a \times \uparrow \sigma(y) + b\}$$

Consequently, if $x \prec 2y \in \mathbb{C}$, it means that in each environment, the set of elements in the container summarized by x is a subset of the set of elements in the container summarized by y multiplied by 2. The order relation, join, meet and widening defined in Section 3 can be left unchanged, up to the update of the transitive closure. In this new domain, the transitive closure, needed for the precision of the operators, is computed with the rule: $x \prec ay + b \wedge y \prec cz + d \implies x \prec acz + bc + d$.

We then need to update how we discover constraints and how we perform the reduction applying them. This is done by trivial modifications on `add_expand` and `apply_constraint`: we store $x \prec ay+b$ when we encounter $\mathbb{E}^\sharp[\![\texttt{expand}(x, ay+b)]\!]\sigma^\sharp$ and we apply $\mathbb{E}^\sharp[\![\texttt{expand}(x, ay+b)]\!]\sigma^\sharp$ to perform a reduction. Consequently, our analysis is able to automatically discover as final state for the Python programs $r_0 \prec 2 \times l_0$, i.e. the list r is included in a transformation of the list l where each element is multiplied by 2.

4.2 General Methodology

Section 4.1 demonstrates that only a few modifications need to be applied to DelExp to express relations up to affine transformations. This section aims at generalizing this methodology. Given a certain set of transformations Θ (e.g. affine transformations, polynomial transformation, etc.), we want to be able to infer constraints under the form $x \prec \theta(y)$ for θ in Θ, meaning that x verifies the same constraints that the set $\theta(y)$, i.e. that the elements of y to which we applied the transformation θ. To do so, we build an abstract domain of the form $\Sigma^\sharp_\Theta = \mathbb{V} \to \mathcal{P}_f(\{\theta(X) \mid X : \tau_1 \in \mathbb{V}, \theta : \tau_1 \to \tau_2 \in \Theta\})$.

▶ **Definition 4.3** (General concretization). *The concretization is of the form:*

$$\gamma\left((\sigma^\sharp, \mathbb{C})\right) = \{\sigma \in \gamma^\sharp_\Sigma(\sigma^\sharp) \mid \forall \theta : \tau_1 \to \tau_2 \in \Theta, \forall x : \tau_2, y : \tau_1 \in \mathbb{V},$$
$$x \prec \theta(y) \in \mathbb{C} \implies \sigma(x) \subseteq \theta(\sigma(y))\}$$

As for affine transformations, we need to update how we discover constraints and how we perform the reduction applying them. We only need to perform those three steps:

- Soundly update `add_expand` definition to decide when to add a constraint $x \prec \theta(y)$.
- Soundly update `reduce` definition to decide how to enforce this constraint in the abstract environment.
- Soundly modify how the transitive closure is computed (depending on Θ).

Defining a new domain on the set of transformations Θ (e.g. polynomial transformations) then permits to precisely express an element-wise transformation from this set. For instance, Θ can be the set of polynomial transformation. This methodology is at the heart of our domain expressivity, and will be further developed in the context of OCaml compositional analysis (see Section 7.1). Θ will then be the set of pure OCaml functions from the analyzed code, enabling us to express higher-order properties on programs.

5 Compositional Analysis for a Pure Functional Language

This section provides background on a compositional analysis for a pure monomorphic functional language introduced by Valnet et al. [31]. Then, it shows that the precision

$$
\begin{aligned}
e_1, e_2, \cdots \in \mathcal{E} ::= \;& x \in \mathbb{V} \\
& \mid n \in \mathbb{Z} && \mid e_1 \oplus e_2 \\
& \mid e_0\ e_1\ \ldots\ e_n && \mid \text{fun } x_1 \ldots x_n \to e \\
& \mid \text{let } x = e_1 \text{ in } e_2 && \mid \text{let rec } x = e_1 \text{ in } e_2 \\
& \mid C(e_1, \ldots, e_n) && \mid \text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m \\
& \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
p_1, p_2, \cdots \in \Pi ::= \;& \_ \mid\ x \in \mathbb{V} \mid n \in \mathbb{Z} \mid C(p_1, \ldots, p_n) \\
& \text{where } \forall i \neq j, fv(p_i) \cap fv(p_j) = \emptyset
\end{aligned}
$$

Figure 8 Syntax of the functional language.

$$
\begin{aligned}
& \mathcal{V} ::= \mathbb{Z} \cup \{\ C(v_1, \ldots, v_n) \mid C \in \mathbb{C}, v_i \in \mathcal{V}\ \} \cup \bigcup_{n \in \mathbb{N}} [\mathcal{V}^n \to \mathcal{V}] \\
& \Sigma = \mathbb{V} \to \mathcal{V} \\
& \mathbb{E}[\![\,\cdot\,]\!] : \mathcal{E} \times \Sigma \to \mathcal{V}^\perp
\end{aligned}
$$

Figure 9 Semantic domain of the functional language.

issue of relational domains on weak variables prevents this analysis from generating precise summaries. We will extend the DelExp domain in Sec. 6 to combine it with this analysis and generate precise summaries for this language in Sec. 7

5.1 Syntax and Semantics

Figure 8 details a pure monomorphic functional language without side-effects. Its key features are recursion, algebraic data types and higher-order. $\mathcal{E}$ 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, and type variants. 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 9 describes the semantic domain of the language. $\mathbb{C}$ is the set of all possible variants. Semantic values of the language $\mathcal{V}$ are integers, type variants containing values, and continuous functions from values to values. Finally, Σ is the set of concrete environments, associating a value to each variable. The concrete semantics of an expression e is $\mathbb{E}[\![e]\!] \in \Sigma \to \mathcal{V}^\perp$, where $\perp$ denotes non-termination. It associates a semantic value to an expression in an environment. This semantics follows the standard for eagerly evaluated functional languages (such as OCaml). We suppose that programs are well-typed according to usual type inference on functional languages [16, 24]. Note that this language is monomorphic so far. This limitation will be overcome later on thanks to our DelExp domain.

5.2 Compositional Analysis

Compositional analysis aims at over-approximating at definition site all the possible behaviours of a function. The function is analyzed once and for all: we then store the result of the analysis, its *summary*. Compositional analyses significantly improve scalability when a function is called multiple times, as summary applications are in general computationally

```
let a = random 5 10 (* environment before *)
let mult2 x = 2*x   (* function definition *)
let b = mult2 a     (* function application *)
```

Figure 10 Small example to illustrate compositional analysis.

cheaper than reanalyzing a function at every calling context [5, 8]. Therefore, it can help the analysis to scale up. Besides, compared to non-compositional analyses, they can be used to infer functions specifications.

Since our language (Figure 8) is pure (no side effects), the function does not modify the environment: its behavior can be represented by a relation between its inputs and its output. Bautista et al. [2] formalize a so-called folk technique to generalize this relation with side effects at first-order.

We recall the notations of Valnet et al. [31], which we will extend later on. Given a function type $\tau = \tau_1 \to \cdots \to \tau_{n+1}$, we denote as $\mathcal{F}^\tau$ the set of functions of type τ. We can then *choose* an abstract domain $\mathcal{R}(V)$ of concretization $\gamma_\mathcal{R} : \mathcal{R}(V) \to \Sigma|_V$ to represent relations between input and output variables V: it abstracts $\Sigma|_V$, i.e. concrete environments restricted to V. The domain for functions of type τ is then $\mathcal{D}_\tau = \mathbb{V}^n \times \mathbb{V} \times \mathcal{R}(V)$. 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. A key advantage of this formalization is that it is *domain-modular*. This framework can then be re-used as is in our analysis with our DelExp abstract domain as relational domain.

▶ **Definition 5.1** (Functions concretization). *The concretization* $\gamma_\tau : \mathcal{D}_\tau \to \mathcal{F}^\tau$ *is:*

$$\gamma_\tau(\langle(x_1, \ldots, x_n), r, p\rangle) = \{f : \tau \mid \forall(v_1, \ldots, v_n) : \tau_1 \times \cdots \times \tau_n, \\ [x_1 = v_1] \cdots [x_n = v_n][r = f(v_1, \ldots, v_n)] \in \gamma_\mathcal{R}(p)\}$$

A function f is abstracted as $\langle(x_1, \ldots, x_n), r, p\rangle$ if the relation between its inputs and its output can be described by p. We call this abstraction the *function summary* and denote it as $\lambda x_1 \ldots x_n.p(x_1, \ldots, x_n, r)$.

▶ **Example 5.2** (Concretization). Given `mult2` of type $\tau =$ `int` $\to$ `int` and $V = \{x, r\}$, we choose $\mathcal{R}(V)$ as the polyhedra domain on V. A sound over-approximation for `mult2` is then $\langle x, r, r = 2x\rangle$, i.e. $\lambda x.r = 2x$.

Sound operations on this domain are defined by Valnet et al. [31], together with a domain for *disjunctive* summaries (allowing partitioning on the inputs) and transfer functions to analyze (possibly recursive) functions and apply those summaries.

▶ **Example 5.3** (Analysis of `mult2`). As an example of function analysis, we analyze `mult2` :

$\mathbb{S}^\sharp[\![$ let $mult2 =$ fun $x \to 2 * x$ $]\!][.]$

- $\mathbb{E}^\sharp[\![$ fun $x \to 2 * x$ $]\!][.]$
  - $\mathbb{E}^\sharp[\![2 * x]\!]\sigma^\sharp[x = \top]$
    - $\leftarrow 2 * x, [x = \top]$
  - $\leftarrow \langle x, r, r = 2 * x\rangle, [.]$
- $\leftarrow [mult2 \to \langle x, r, r = 2 * x\rangle]$

This analysis trace shows the analysis steps. The right arrow means that we analyze a sub-expression. The left arrow means that a result is returned. `mult2` is analyzed once and for all at *definition site*. Consequently, we compute the abstract semantics of the body with no hypothesis on input values: x is initialized at $\top$. The generated relation is then valid for all inputs. Note that relationality is critical there. It enables us to discover information on the function behavior despite making no hypothesis on the values of the input x.

▶ Remark 5.4 (Recursive functions). Note that the analysis of recursive functions is done via fixpoint iteration: starting from a function with bottom input-output relation, we iteratively complete the function summary by renalysing the function, until reaching a fixpoint, and use a widening to converge in finite time.

▶ **Example 5.5** (Application of `mult2`)**.** As an example of function application, we analyze:

$\mathbb{E}^\sharp[\![mult2\ (7+1)]\!]\sigma^\sharp$
- $\longrightarrow \mathbb{E}^\sharp[\![mult2]\!]\sigma^\sharp$
- $\longleftarrow \langle x, r, r = 2 * x\rangle, \sigma^\sharp$
- $\longrightarrow \mathbb{S}^\sharp[\![\ x = 7+1\ ]\!]\sigma^\sharp$
- $\longleftarrow \sigma^\sharp[x \to 8]$
- $\longleftarrow r, \sigma^\sharp[x \to 8, r \to \top] \sqcap^\sharp \sigma^\sharp[x \to \top, r = 2x] =$ $r, \sigma^\sharp[x \to 8, r \to 16]$

First, we compute the summary of the applied function. Since $mult2$ was already analyzed, its summary is stored in the environment. We only need to instantiate this summary with input values: we assign formal parameters to the input values in the current environment and perform its intersection with the summary to get the result $r = 16$.

6 DelExp with Heterogeneous Environments

Our analysis manipulates summarization variables to abstract contents from containers, e.g. to abstract the content of a list. However, those containers may be empty: as a consequence, the summarization variables may correspond to no concrete objects. Such variables are called *optional* and must be handled with extra care. In particular, they require reformulating standard concretizations.

6.1 Heterogeneous Environments

We introduce here the formalization from Journault et al. [19]. Together with a standard abstract environment $\sigma^\sharp$, they define two sets of variables $l, u \in \mathcal{P}(\mathbb{V})$, $l \subseteq u$. l represents the set of non-optional variables, i.e. variables existing for sure, in every concretize environments. u represents the set of all possibly existing variables in the program. Therefore, variables in $u \setminus l$ are optional: they may not exist in some concretized environments. Those variable sets are soundly inferred throughout the analysis of the programs. We denote as $dom(\sigma)$ the definition domain of the environment σ. Given $\sigma' \in \Sigma$, $\sigma'_{|dom(\sigma)}$ restricts the definition domain of σ' to $dom(\sigma)$. Given a set of environments S, $S_{|\mathrm{dom}(\sigma)}$ lifts this restriction to every environment of the set. The new concretization of abstract environments is:

▶ **Definition 6.1** (Heterogeneous concretization)**.** *We define the concretization:*

$$\gamma_{\mathcal{H}}\left((\sigma^\sharp, l, u)\right) = \{\sigma \mid l \subseteq dom(\sigma) \subseteq u \wedge \sigma \in \gamma_{\mathbb{Z}}(\sigma^\sharp)_{|dom(\sigma)}\}$$

We denote as $\sqcup_{\mathcal{H}}$ (resp. $\sqcap_{\mathcal{H}}$) the join (resp. the meet) on heterogeneous environments, defined in [19]. As shown in the introduction, summarization variables of ADTs fields may not exist in some executions. Therefore, their abstraction, from [31], is based on careful manipulation of those optional variables.

▶ **Example 6.2** (Heterogeneous summaries)**.** We illustrate the difficulty of handling heterogeneous environments:

```
type either = Left of int | Right of int
let to_int a =
  match a with
  | Left n -> n
  | Right n -> n
let b = if rand () then Left 5 else Right 6
let x = to_int b
```

A sound heterogeneous summary is $\langle a, r, a : \{\texttt{Left}, \texttt{Right}\}, r = \texttt{a.Left} = \texttt{a.Right}\rangle$ with $l = \{r\}$ and $u = \{r, \texttt{a.Left}, \texttt{a.Right}\}$. The heterogeneous relational constraint $r = \texttt{a.Left} = \texttt{a.Right}$ where r is always defined and `a.Left` and `a.Right` are optional variables represents multiple cases of homogeneous relational constraints. It includes the case where only r and `a.Left` exists and $r = \texttt{a.Left}$, and the case where only r and `a.Right` exists and $r = \texttt{a.Right}$. Note that however, it loses the information that exactly one variable among `a.Left` and `a.Right` exists: it also abstracts a concrete environment where `a.Left` and `a.Right` both exist and $r = \texttt{a.Left} = \texttt{a.Right}$. This can be recovered by the fact that `Either` is a non-recursive ADT.

A sound abstract value for b is $\{\texttt{Left}, \texttt{Right}\}, \texttt{b.Left} = 5, \texttt{b.Right} = 6$ with $l = \emptyset$ and $u = \{\texttt{b.Left}, \texttt{b.Right}\}$. If we naively compute the intersection when applying the function, we end up with $x = \texttt{b.Left} = 5 = \texttt{b.Right} = 6$, i.e. $\bot$. This is unsound: we propagate information from values that may not exist as if their existence was certain. A sound intersection was proposed in Journault et al. [19].

6.2 Heterogeneous Version of DelExp

The DelExp domains (Secs. 3 and 4) presented so far in this section are only defined on *homogeneous* environments: the operators (join, meet, widening) are defined between environments on the same set of non-optional variables. To use it in combination with the ADT domain from Valnet et al. [31], we need to provide support for heterogeneous environments. To express relations between optional variables, we have to adapt its concretization by providing it with the lower and upper sets of defined variables l and u.

▶ **Definition 6.3** (Heterogeneous concretization of DelExp)**.** *We define the concretization* $\gamma : \Sigma^\sharp \times \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}) \times \mathbb{C} \to \Sigma$ *on the standard DelExp domain (Sec. 3):*

$$\gamma\left((\sigma^\sharp, l, u, \mathbb{C})\right) = \{\sigma \in \Sigma \mid \sigma \in \gamma_{\mathcal{H}}(\sigma^\sharp, l, u) \tag{1}$$

$$\wedge\, \forall v \prec x \in \mathbb{C}, v \in dom(\sigma) \implies x \in dom(\sigma) \tag{2}$$

$$\wedge\, v \in dom(\sigma) \implies \sigma(v) \subseteq \sigma(x)\} \tag{3}$$

The concretization requires that if a constraint $v \prec x$ exists then it only represents environments σ where $v \in \mathrm{dom}(\sigma) \implies x \in \mathrm{dom}(\sigma)$. Adding this precision is a *choice* that modifies the semantics of the domain. Without this restriction, we could only propagate

information from the constraint $v \prec x$ if we are sure that v *and* x exists. With it, we only need to make sure that x exist. Note that to be sound, an analysis may only add a constraint $v \prec x$ in the environment if the existence of v indeed guarantees that x exists.

▶ **Property 6.1.** *The unchanged meet $\sqcap_c^{\mathcal{H}} = \sqcap_c$ is sound.*

▶ **Definition 6.4** (Heterogeneous join)**.** *We define the join $\sqcup_c^{\mathcal{H}}$ on $\mathbb{C}$:*

$$\begin{aligned}(f_1, l_1, u_1, \mathbb{C}_1) \sqcup_c^{\mathcal{H}} (f_2, l_2, u_2, \mathbb{C}_2) = ((f_1, l_1, u_1) \sqcup_{\mathcal{H}} (f_2, l_2, u_2),\\ \{x \prec y \in \mathbb{C}_1 \mid x \notin u_2\} \cup\\ \{x \prec y \in \mathbb{C}_2 \mid x \notin u_1\} \cup (\mathbb{C}_1 \cap \mathbb{C}_2))\end{aligned}$$

▶ **Property 6.2.** *$\sqcup_c^{\mathcal{H}}$ is sound.*

▶ **Example 6.1.** *We compute $(f_1, \{a, x_w, y_w\}, \{a, x_w, y_w\}, \{x_w \prec y_w\}) \sqcup_c^{\mathcal{H}} (f_2, \{a, y_w\}, \{a, y_w\}, \top)$. Since x_w only exists in the left component, we can keep $x_w \prec y_w$ in the final result. This is due to the semantics of this constraint: it only applies to environments where x_w exists. We can safely keep it: it will not propagate information to environments where x_w does not exist. Therefore, the formula yields $(f_1 \sqcup_{\mathcal{H}} f_2, \{a, y_w\}, \{a, x_w, y_w\}, \{x_w \prec y_w\})$.*

Algorithm 4 $\texttt{apply\_constraint}_{\mathcal{H}}$

Input : $(f, l, u, \mathbb{C}), x \prec y$
$\sigma^\sharp \leftarrow (f, l, u, \mathbb{C})$
if $x \in l$ **then**
  $\sigma^\sharp \leftarrow \mathbb{S}^\sharp[\![\texttt{expand}(x, y)]\!](f, l, u)(f, l, u), \mathbb{C}$
end
else
  $\sigma^\sharp \leftarrow \mathbb{S}^\sharp[\![\texttt{remove}(x)]\!](f, l, u) \sqcup \mathbb{S}^\sharp[\![\texttt{expand}(x, y)]\!](f, l \cup \{x, y\}, u), \mathbb{C}$
end
return $\sigma^\sharp$

The heterogeneous reduction, $\texttt{reduce}_{\mathcal{H}}$, is an update of `reduce` (Algorithm 3) where `apply_constraint` is replaced with $\texttt{apply\_constraint}_{\mathcal{H}}$. $\texttt{apply\_constraint}_{\mathcal{H}}$ behaves like `apply_constraint` when the variables are non-optional. Otherwise, if x is optional, we perform a join between environments where x does not exist (in which case we perform $\texttt{remove}(x)$) and environments where it is non-optional (in which case we perform $\texttt{expand}(x, y)$).

▶ **Property 6.3.** *$\texttt{reduce}_{\mathcal{H}}$ is sound.*

Those operators are proven sound with regard to the heterogeneous concretization. This work on heterogeneous environments allows the DelExp domain to work with optional variables, i.e. variables that may not exist. A similar formalization can adapt AffineDelExp to heterogeneous environments. In particular, it makes this domain compatible with the state-of-the-art relational abstractions for numerical trees [19] and for Algebraic Data Types [31].

▶ **Example 6.5** (OCaml Algebraic Data Types)**.** Given the OCaml function:

```
let rec filter_le l inf =
  match l with
  | Cons(h,q) ->
    if h > inf then filter_le q inf
```

```
    else Cons(h,filter_le q inf)
| Nil -> Nil
```

The analysis can infer the summary:

$$\lambda l\ \text{inf}.r : \{\texttt{Nil}, \texttt{Cons}\} \wedge l = \{\texttt{Nil}, \texttt{Cons}\} \wedge (r_0 < inf, \{\}, \{r_0, l_0\}) \wedge r_0 \prec l_0$$

The concretization of the summary means: "the result of `filter_le` may be empty; if it is not, then the input list is not empty either, and the content of the output list is lower than inf and included in the input list". This is more precise than the summary for this function in Valnet et al. [31], i.e. this summary without the constraint $r_0 \prec l_0$.

7 Case Study: Functional Compositional Analysis Over ADTs

This section shows how the domains introduced in Secs. 3 and 4 can be used to perform a precise analysis in the presence of containers. We extend the analysis from Valnet et al. [31] on pure monomorphic ADT-manipulating OCaml programs described in Sec. 5.1. We highlight key design choices that make this combination possible.

Heterogeneity. The reduction of DelExp domains is sound when in product with a domain that implements sound operations for heterogeneous environments. Non-relational domains can be lifted without efforts to heterogeneous environments [19]. Defining heterogeneous operations for relational domains requires more care. The OCaml analysis we built upon only supports two relational domains: the polyhedra domain and the equality domain (equalities are only stated between variables that are sure to exist together). Both of them implement sound heterogeneous operators.

Modularity. Second, note that those domains are agnostic in the domain chosen to abstract underlying variables: consequently, they can work in combination with any domain. Valnet et al. [31] use summarization variables to represent Algebraic Data Types fields in a similar modular fashion: the ADT domain is agnostic in the domain chosen to abstract them. Thanks to those parametric definitions, they can be combined, and used in a higher-order settings.

Polymorphism. In fact, the default DelExp domain do not make any assumption on the very type of the variables they manipulate. In particular, those variables can be *polymorphic*. We store the constraints between polymorphic variables during the analysis of a polymorphic function. When the function is applied to a known ground type τ, we can then perform the reduction with the domain $\mathcal{D}_\tau$. We therefore extend the existing OCaml analysis to support polymorphism (e.g. the `map` function from Fig. 11).

7.1 ApplyDelExp: a Higher-Order Transformation

As a showcase of the methodology from Sec. 4.2, we will expand our DelExp domain with pure higher-order transformation (without side-effect). We want to express constraints under the form $x \prec f(y)$ with x, y, and f being variables of the language. We define as abstract domain $\Sigma^\sharp_{\text{fun}} = \mathbb{V} \to \mathcal{P}_f(\{f(X) \mid \exists \tau_0, \tau_1, f \in \mathbb{V}_{\tau_0 \to \tau_1}, X \in \mathbb{V}_{\tau_1}\})$.

- We update `add_expand`: if the abstract expression $e^\sharp$ is (syntactically) $f(y)$ then we add the constraint $x \prec f(y)$.

```
let rec map f l =
  match l with
  | Cons(h,q) -> Cons(f h, map f q)
  | Nil -> Nil
```

Figure 11 Motivating example for our higher-order polymorphic transformation

- We update `reduce`: if the environment contains $x \prec f(y)$, we execute `expand`$(x, f(y))$ in the environment.
- We compute the transitive closure under the rule that $x \prec f(y)$ and $y \prec z$ implies that $x \prec f(z)$.

Note that the reduction is sound in a *pure* fragment of the language, i.e. where functions have no side-effects. We name this domain ApplyDelExp. It enables us to derive a precise summary for `map`.

7.2 Analysis of `map`

The trace in Fig. 12 details the analysis of `map` from Fig. 11. Note that some details are skipped for pedagogical purposes: not all steps are described and the sets l and u from heterogeneous environments are not represented.

Given an OCaml list `l`, `l.Cons.1` summarizes the elements of the integer fields of the variant `Cons` nested inside `l` (see Sec. 2.4). For readability purposes, `l.Cons.1` (resp. `r.Cons.1` and `q.Cons.1`) will be denoted as l_0 (resp. r_0 and q_0). `c` is a temporary variable for `Cons(f h, map f q)` and `c'` a temporary variable for its queue `map f q`. Constraints from DelExp domains are highlighted in gray. The variable r corresponds to the result of `map`. Depending on the expression e, $\mathbb{E}^\sharp[\![e]\!]$ delegates the abstraction of e to the relevant abstract domain (e.g. $\mathbb{E}^\sharp_{ADT}[\![\, e \,]\!]$).

We start with an empty environment [.]. As map is recursive, we need to perform fixpoint iterations to infer its summary, as described in Remark 5.4. We first start by assuming that map's summary is $\bot$ and analyze the body of map to get a new summary s. We then reanalyze the body of map by assuming that s is the new summary of `map`, etc. until a fixpoint is reached:

1. After the first iteration ①, the inferred behavior for map is that the input can only be `Nil` and the output is then `Nil`.
2. We then start the second iteration with this hypothesis on `map`'s behaviour (②). `match`$^\sharp$ computes in $\sigma_1^\sharp$ an over-approximation of the environments in which l matches the first pattern, whereas `match`$^\sharp_\neg$ computes in $\sigma_2^\sharp$ an over-approximation of the environments in which they do not. We then analyze `Cons(f h, map f q)` in $\sigma_1^\sharp$ (③): the head is `f h` (④), which yields $\mathtt{c}_0 \prec fh$. The tail (⑤) is empty (by application of the summary inferred at the first iteration), so we do not fold any other constraint inside $\mathtt{c}_0$. When removing h in the final summary and assigning $r = \mathtt{c}$, we deduce $r_0 \prec fl_0$.
3. We start the third iteration with this new hypothesis on `map`'s behaviour (⑥). We focus on the analysis of `Cons(f h, map f q)`. First, the analysis of the head derives $\mathtt{c}_0 \prec fh$ (⑦). Then the queue `c'` (summarized by $\mathtt{c}'_0$) of the list is $map\ f\ q$, which yields $\mathtt{c}'_0 \prec fq_0$ (⑧). We then fold $\mathtt{c}'_0$ inside $\mathtt{c}_0$ (⑨): we join $\mathtt{c}_0 \prec fh$ with $\mathtt{c}_0 \prec fq_0$ knowing that $h \prec l_0$ and $h \prec q_0$, i.e. we only keep the constraints that are true in both DelExp environments. We deduce $r_0 \prec fl_0$.

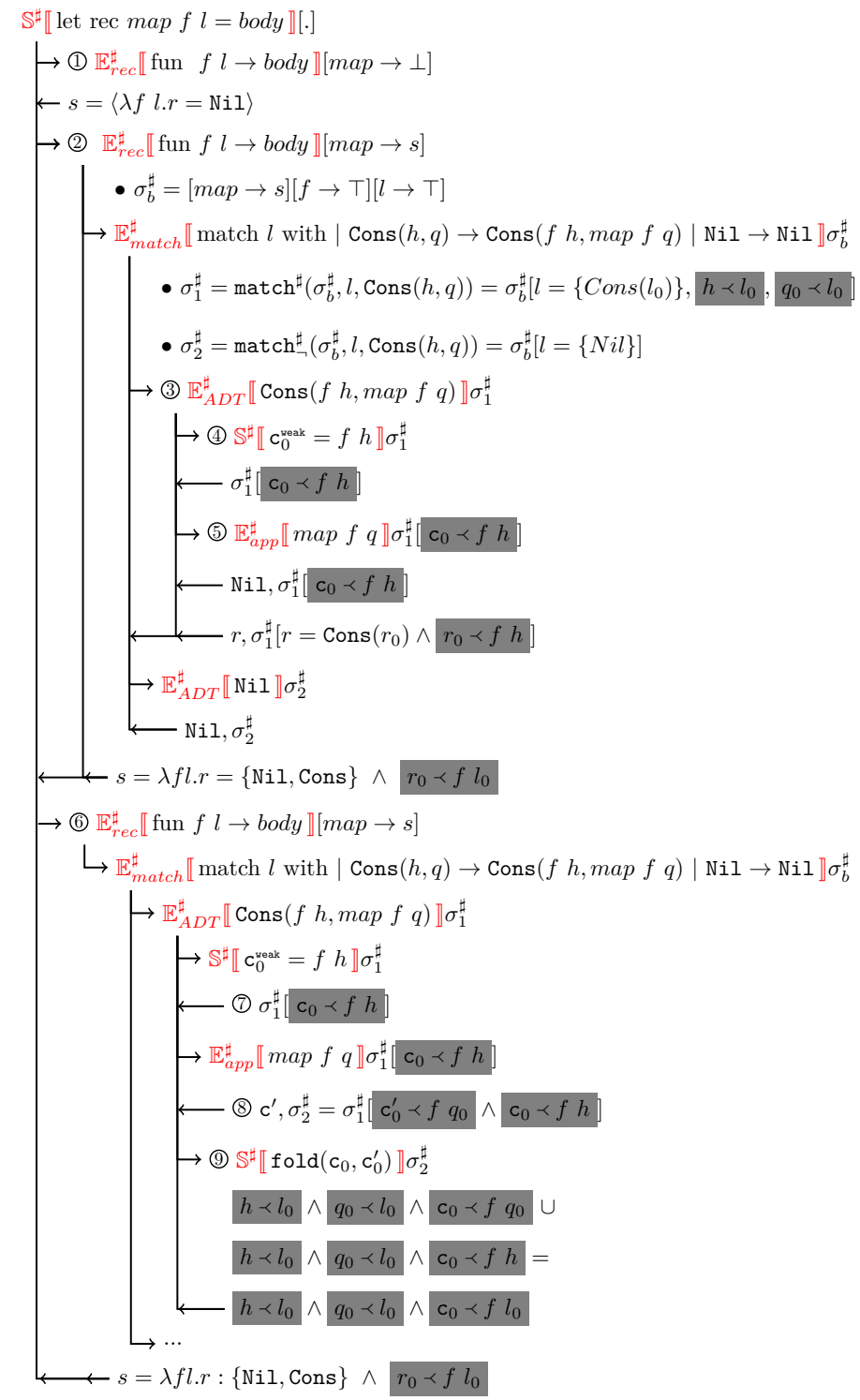

Figure 12 Analysis of the `map` function with the DelExp domain.
DelExp constraints are highlighted in gray. l_0 (resp. $r_0, q_0, \texttt{c}_0, \texttt{c}'_0$) summarizes the elements of l (resp. $r, q, \texttt{c}, \texttt{c}'$), r is the result of map and s its summary.

4. Iterations 2 and 3 ended up with the same summary: we reached a fixpoint.

As a conclusion, the generated summary states: if the input list is empty, the output list is empty. Otherwise, the output list content is included in the input list content to which we applied the (pure) input function. The ApplyDelExp domain we created following the methodology from Sec. 4.2 enhances the precision of the inferred summary. The ability to derive "up-to-transformation" DelExp domains can help significantly improve compositional analysis in the presence of containers. Note however that we did not prove the equality between those two sets, nor that the order of elements is maintained in the resulting list. To this end, we would need to express that h and q cover all elements in l: this cannot be done in our DelExp domains.

8 Experimental Evaluation

8.1 Mopsa

The analysis described in this article is implemented in MOPSA [18]. MOPSA (Modular Open Platform for Static Analysis) is a platform aiming at easing the development of new static analyzers [25, 29]. This platform, written in OCaml, is open-source and multi-language. It features language-agnostic abstractions such as integer domains (polyhedra, octagons, intervals) and implements the transfer functions for standard constructs such as conditional branching, `while` loops or assignments. The core of the platform is composed of 22 000 LoC of OCaml. Existing analyzers for C, Python, and OCaml base their implementation on those components by adding additional syntactic structures, iterators, and language-specific domains that complete the language-agnostic abstractions.

MOPSA traditionally encourages both relational abstractions and information sharing between domains through powerful communication mechanisms (such as reduced products or expression rewriting). Those design choices perfectly align with our domain-agnostic relational domain for weak variables: DelExp was integrated seamlessly into MOPSA. We implemented the basic DelExp domain from Sec. 3, the AffineDelExp from Sec. 4 and the ApplyDelExp domain from Sec. 7.1 as a separate module of around 1000 LoC of OCaml. We simply added it both to the OCaml compositional analysis from Valnet et al. [31] (restricted to a pure and monomorphic fragment) and to the Python analysis from Monat et al. [27], which uses weak variables to summarize lists and sets. It did not require any change to the existing abstract domains and transfer function. Section 4.2 mentioned that `add_expand` and `reduce` functions need to be redefined when formalizing a new transformation: the same applies for the implementation. Once the basic DelExp domain was implemented, we simply extended the `add_expand` and `reduce` functions to handle those affine and apply constraints.

The resulting analyses are fully automatic. They compute over-approximations of the values of the program variables. In particular, the OCaml analysis infers functions summaries.

8.2 Benchmarks

All our tests were performed on a Intel(R) Core(TM) Ultra 7 165H CPU with 32 GB of RAM. Execution time were computed as an average over 10 runs of the analysis. All programs are between 6 and 20 lines and consist of a unique container-manipulating function. The summaries have been visually checked in the execution trace of the analysis.

OCaml. As our OCaml analysis is compositional, we focused on assessing the precision of the generated summaries on ADT-manipulating functions. Figure 13 presents our results.

Function	Delexp (ms)	Overhead	Summary
`hd.ml`	8	2.6	$r \prec$ `l.Cons.1`
`tl.ml` (Ex. 3.2)	8	2.5	`r.Cons.1` $\prec$ `l.Cons.1`
`filter.ml`	46	2.3	`r.Node.1` $\prec$ `l.Node.1`
`map.ml` (Fig. 11)	48	3.4	`r.Cons.1` $\prec f($`l.Cons.1`$)$
`copy.ml`	56	3.0	`r.Cons.1` $\prec$ `l.Cons.1`
`filter_le.ml` (Ex. 6.5)	160	3.7	`r.Cons.1` $\prec$ `l.Cons.1` $\wedge$ `r.Cons.1` $< inf$
`mult2.ml` (Fig. 2)	65	3.4	`r.Cons.1` $\prec 2 \times$ `l.Cons.1`
`mult2tree.ml`	99	3.2	`r.Node.1` $\prec 2 \times$ `l.Node.1`
`mult3plus4.ml`	67	3.6	`r.Cons.1` $\prec 3 \times$ `l.Cons.1` $+ 4$
`maptree.ml`	61	2.8	`r.Node.1` $\prec f($`l.Node.1`$)$
`listtotree.ml`	62	2.9	`r.Cons.1` $\prec$ `l.Node.1`

Figure 13 Analysis of ADT-manipulating OCaml functions

Function	Delexp (ms)	Overhead	Summary
`copy.list.py`	69	1.3	$l2 \prec l1$
`copy.list.comprehension.py`	69	1.4	$l2 \prec l1$
`copy.set.py`	56	1.03	$s2 \prec s1$
`copy.set.comprehension.py`	59	1.03	$s2 \prec s1$
`filter.list.py`	92	1.5	$l2 \prec l1$
`filter.list.comprehension.py`	91	1.5	$l2 \prec l1$
`filter.set.py`	57	1.04	$s2 \prec s1$
`filter.set.comprehension.py`	65	1.08	$s2 \prec s1$
`mult2.list.py` (Fig. 1)	70	1.4	$l2 \prec 2 \times l1$
`mult2.list.comprehension.py`	50	2	$l2 \prec 2 \times l1$
`mult2.set.py`	57	1.1	$s2 \prec 2 \times s1$
`mult2.set.comprehension.py`	62	1.06	$s2 \prec 2 \times s1$
`partition.list.py`	124	1.7	$l_2 \prec l_1 \wedge l_3 \prec l_1$
`partition.set.py`	76	1.1	$s_2 \prec s_1 \wedge s_3 \prec s_1$
`partition.set-list.py`	104	1.4	$l_2 \prec s_1 \wedge l_3 \prec s_1$
`partition.list-set.py`	96	1.2	$s_2 \prec l_1 \wedge s_3 \prec l_1$
`tail.list.py` (Ex. 3.2)	76	1.3	$l_2 \prec s_1$

Figure 14 Analysis of list-manipulating Python programs

The first four programs are inspired from the OCaml standard library over lists. For `filter_le.ml`, the OCaml analysis without DelExp could infer that `l.Cons.1` $< inf$, but for every other functions, the generated summary without DelExp is $\top$, i.e. we have no information about the function's behaviour. The DelExp domains computed the intended summaries for the ADT-manipulating functions. In particular, notice that it also inferred precise summaries for the versions of `mult2` and `map` that operate on the labels of a tree rather than the elements of a list. Note that given the `filter` function, which, given a functional predicate as input, filters the list with the elements respecting this predicate, we can compute that the output list is included in the input list. We can also infer a specification for `listtotree.ml`, that translates a list to a tree: we can infer that the elements of the output tree are included in the elements of the input list. It highlights that our domain is agnostic in the underlying structure and type of the objects abstracted by the weak variables. We also computed the overhead of our DelExp domain by comparing the execution time with and without our domain. Our overhead is between 2 and 4. Part of this cost is due to the computation of our transitive closure. As an optimization, our transitive closure could be computed more efficiently using labeled union-find data structure from Lesbre et al. [22].

Python. The Python static analysis we build upon [27] is not compositional, as this is made very difficult by Python's dynamic typing and flexible semantics. However, our Python programs define lists and sets (l_1 and s_1) initialized with a random size and content and give them as input to functions, which compute new lists or sets (l_2,l_3 or s_2,s_3) by applying a certain transformation. Operations on lists and sets can be defined with `for` loops or with comprehensions. Figure 14 shows our results. `partition.x.py` splits a container into two containers with respect to an integer pivot. This is the intermediate operation of quicksort. Note that `partition.set-list.py` builds two lists from a set and `partition.list-set.py` two sets from a list. The overhead of the DelExp domains in Python is between 1.1 and 1.7. Since the `reduce` function is called at every function application in our implementation, the OCaml compositional analysis of recursive functions performs more reduction than a similar program in Python. This may explain the overhead difference between the two languages.

Comparison with QUIC graphs. Compared to QUIC graphs, we have a greater expressivity on map-like functions, i.e. functions that apply element-wise transformations. `copy.set.py`, `partition.set.py` and `filter.set.py` from Fig. 14 are examples from QUIC's Table 1 [11], chosen for their interest and translated to Python from their toy language. Note that we cannot experimentally compare both approaches: no artifact was archived at the time, and we could not compile the available source code.[1]. However, Cox et al. [11] reported an overhead of a factor of 9 on similar standard set-manipulating functions. Note that they operate on a toy set language, whereas DelExp was run on top of existing analyses, for real languages of different paradigms (statically typed and functional for OCaml, dynamically typed and object-oriented for Python).

An artifact reproduces those experimental claims[2].

[1] https://plv.colorado.edu/projects/quicgraphs/
[2] https://zenodo.org/records/18498584

9 Related Work

Weak variables. Programs often manipulate an unbounded number of objects (e.g. heap-allocated data or containers) whereas most abstract domains manipulate fixed and finite numbers of objects. Gopan et al. [13] overcame this issue by grouping a possibly unbounded collection of concrete objects into one abstract *summarization* variable. They defined operators to soundly manipulate them. This generic framework was used to abstract different kind of containers. For numerical trees, Journault et al. [19] use tree automata (a generalization of regular expressions) to represent a set of paths from root to node – or equivalently, a set of nodes. To each abstract path, they associate a weak variable collecting the integer label of those nodes. Monat [26] used them to summarize all possible characters, represented as their ASCII code, whereas Valnet et al. [31] used them to summarize ADTs' fields.

Arrays. The abstraction of fixed-size containers such as arrays also leverages those summarization variables [3]. Gopan et al. [14] partitioned arrays with respect to the value of an integer variable i (provided by the user or by a pre-analysis) into three index-bounded segments, $a_{<i}$ (resp. $a_{>i}$) for array slots of index lower (resp. greater) than 1 and a_i for slot i. For both segments $a_{<i}$ and $a_{>i}$, a weak variable abstracts the indices of the segment and another abstracts its content. With a similar approach, Halbwachs and Péron [15] define for each segment *slice variables*, ranging over arrays indices. They allow expressing relations between segments from two different arrays ($a = b$ on a set of indices p means that $\forall i \in p, a[i] = b[i]$). *Segmentation* is introduced by Cousot et al. [9] as a functor both deciding how to partition arrays and how to summarize their contents. Partitioning is then semantically-guided and can dynamically change while analyzing the program. However, it cannot express content relations between summarized segments. Those methods were reviewed by Bautista et al [1] and extended [2] so that segment summaries can contain non-recursive algebraic data types and refer to arbitrary program variables. Non-contiguous partitions are supported by Liu and Rival [23]. Segmentation allows very precise abstraction. However, since this approach is index-based, it is not suited for general containers (sets, maps, linked lists or user-defined algebraic data types). As segmentation defines when to create weak variables whereas DelExp increases precision in the presence of those variables, they could be used in combination. Braine et al. [6], Monniaux and Gonnord [28] handle array analysis by translation into Horn Clauses, then delegated to solvers: they show the approach can prove array sortedness.

Compositional analysis. Compositional analysis is a long-known technique to improve analysis scalability [8]. First applied to `while` programs [21], it has been used in various settings, to analyze independently decomposed parts of a program [12] or for inter-procedural analysis [20]. Bautista et al. [1, 2] define an ADT domain in an input-output analysis for non-recursive imperative programs. The precision of function summaries can be improved with partitioning [4, 5, 31].

Relational analysis on containers. A key ingredient to achieve interesting precision levels with compositionality is to rely on relational abstract domains. However, retaining relational information between weak variables is a non-trivial concern [30]. Cox et al. [11] defined the domain QUIC graphs, presented for sets of scalars. Our domain differs from QUIC graphs in three main aspects: whereas QUIC graphs can express union and intersection of sets, our domain can instead express transformations (Sec. 4) on weak variables. Besides, our domain

supports polymorphic variables (Sec. 7) and heterogeneous environments (Sec. 6). Another relevant line of work is shape analysis, which aims at abstracting memory regions (e.g. heap allocated data structures), partitioning them according to high-level predicates . They can express relations with sophisticated folding and unfolding operators [7]. For example, Illous et al. [17] can express in-place modification of data structures ("transform-into" relation). Techniques based on separation logic sometimes use fold and expand operators to abstract memory regions: they may leverage the DelExp domains to improve their precision in this context.

10 Conclusion

This article presents a new family of relational domains for containers and shows how to derive precise analyses for container-manipulating programs. Built upon the concept of weak variables [3, 13], those domains can express relations between them. Compared to the state of the art, our domains yield a higher expressivity for containers-transforming programs, e.g. it can precisely abstract map-like functions.

Those domains are highly parametric. First, they are language-agnostic: they are formalized on a very generic language manipulating weak variables. They are also modular in the domain chosen to represent weak variables: those variables can abstract a set of objects of any type, even a polymorphic one. Since they support heterogeneous environments, they can manipulate *optional* weak variables: therefore, they can express properties on the content of possibly empty containers. Finally, new domains expressing element-wise content transformation can be derived at low cost from the DelExp domain – e.g. the ApplyDelExp domain can express higher-order transformations. As a consequence, they can be used in product with any existing analysis manipulating weak variable, for applications as different as Python lists and user-defined OCaml Algebraic Data Types.

Since they are *relational*, those domains make it possible to define precise compositional analyses for container-manipulating functions. In particular, we used it to extend the OCaml compositional analysis from Valnet et al. [31] on algebraic data types, so far less expressive and restricted to monomorphism. Those domains were implemented into the MOPSA platform in product with the existing OCaml analysis[31]. Experimental results show that they enable to automatically discover precise invariants on ADT-manipulating, possibly higher-order, functions from the OCaml standard library. The same domains were used in product with an existing Python analysis [26], proving the genericity of the approach. The overhead of the DelExp domain was around 1.5 for Python and 3 for OCaml, lower than the overhead of the QUIC graphs approach [11] (around 9).

For future work, we consider using the DelExp domain in combination with segmentation techniques or shape analysis abstractions to evaluate its use in different contexts. We also aim at defining more sophisticated domains on containers able to express more fine-grained properties. As a foreseen next step, we want to design a relational domain able to express properties on balanced binary search trees to automatically prove correctness of functional algorithms manipulating them.

Acknowledgements

This work was supported by project ANR-22-PECY0005 "Secureval" managed by the French National Research Agency for France 2030. This work was supported by France's Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001.

References

1 Bautista, S.: Static Analysis of Algebraic Data Types and Arrays. Ph.D. thesis, ENS Rennes (2023)

2 Bautista, S., Jensen, T., Montagu, B.: An input–output relational domain for algebraic data types and functional arrays. Formal Methods in System Design pp. 1–74 (2024)

3 Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The essence of computation: complexity, analysis, transformation, pp. 85–108, Springer (2002)

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

5 Boutonnet, R., Halbwachs, N.: 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, pp. 136–159, Springer (2019)

6 Braine, J., Gonnord, L., Monniaux, D.: Data Abstraction: A General Framework to Handle Program Verification of Data Structures. In: Lecture notes in computer science, Lecture notes in computer science, vol. 12913, pp. 215–235, Chicago, United States (Oct 2021), doi:10.1007/978-3-030-88806-0_11, URL https://inria.hal.science/hal-03321868

7 Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. ACM SIGPLAN Notices **43**(1), 247–260 (2008)

8 Cousot, P., Cousot, R.: Modular static program analysis. In: International Conference on Compiler Construction, pp. 159–179, Springer (2002)

9 Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. ACM SIGPLAN Notices **46**(1), 105–118 (2011)

10 Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 84–96 (1978)

11 Cox, A., Chang, B.Y.E., Sankaranarayanan, S.: Quic graphs: Relational invariant generation for containers. In: European Conference on Object-Oriented Programming, pp. 401–425, Springer (2013)

12 Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: 2015 Formal Methods in Computer-Aided Design (FMCAD), pp. 57–64, IEEE (2015)

13 Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: 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, pp. 512–529, Springer (2004)

14 Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 338–350 (2005)

15 Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 339–348, PLDI '08, ACM (2008)

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

17 Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. Formal methods in system design **57**(3), 343–400 (2021)

18 Journault, M., Miné, A., Monat, R., Ouadjaout, A.: 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, pp. 1–18, Springer (2020)

19 Journault, M., Miné, A., Ouadjaout, A.: An abstract domain for trees with numeric relations. In: European Symposium on Programming, pp. 724–751, Springer (2019)

20 Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.: Compositional recurrence analysis revisited. ACM SIGPLAN Notices **52**(6), 248–262 (2017)

21 Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems (TOPLAS) **19**(3), 427–443 (1997)

22 Lesbre, D., Lemerre, M., Ait-El-Hara, H.R., Bobot, F.: Relational abstractions based on labeled union-find. Proceedings of the ACM on Programming Languages **9**(PLDI), 1194–1219 (2025)

23 Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 282–299, Springer (2015)

24 Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3), 348–375 (1978)

25 Miné, A.: Mopsa (2025), URL https://mopsa.lip6.fr

26 Monat, R.: Static type and value analysis by abstract interpretation of Python programs with native C libraries. Ph.D. thesis, Sorbonne Université (2021)

27 Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of Python programs. In: 34th European Conference on Object-Oriented Programming (ECOOP 2020), pp. 17–1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2020)

28 Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free horn clauses. In: International Static Analysis Symposium, pp. 361–382, Springer (2016)

29 Ouadjaout, A., Monat, R., Miné, A., Journault, M., Parolini, F., Milanese, M., Boillot, J.: Mopsa (2025), URL https://gitlab.com/mopsa/mopsa-analyzer

30 Siegel, H., Simon, A.: Summarized dimensions revisited. Electronic Notes in Theoretical Computer Science **288**, 75–86 (2012)

31 Valnet, M., Monat, R., Miné, A.: Compositional static value analysis for higher-order numerical programs. In: 39th European Conference on Object-Oriented Programming (ECOOP 2025), vol. 333, p. 15, Dagstuhl Publishing (2025)