DelExp: A Relational Container Abstraction With Applications to Compositional Analysis (Artifact)

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

This artifact accompanies the submission entitled "DelExp: a Relational Container Abstraction with Applications to Compositional Analysis. It enables to reproduce experimental claims made in the paper. The artifact requires a computer with Docker installed.

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/DARTS.12.1.15

Related Article Milla Valnet, Raphaël Monat, and Antoine Miné, "DelExp: A Relational Container Abstraction: with Applications to Compositional Analysis", in 40th European Conference on Object-Oriented Programming (ECOOP 2026), LIPIcs, Vol. 372, pp. 29:1–29:27, 2026.
`https://doi.org/10.4230/LIPIcs.ECOOP.2026.29`

Related Conference 40th European Conference on Object-Oriented Programming (ECOOP 2026), June 29–July 3, 2026, Brussels, Belgium

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2026 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

The artifact allows verifying the following claims:

(a) The analysis time and overhead of the benchmarks from Fig. 13 and 14 (Sec. 8.2) from the paper can be reproduced.
(b) The analysis automatically infer the correct summaries for the OCaml and Python functions of Fig. 13 and 14.

The steps needed to verify those claims are detailled in Annex A, B.1 and B.2.

2 Content

The artifact package includes:

- `ocaml` contains OCaml programs from Fig. 13.
- `python` contains Python programs from Fig. 14.
- `mopsa-analyzer` contains the source code of the analyzer. It is a fork of `https://gitlab.com/mopsa/mopsa-analyzer`.
- `generate_table.jl` is the Julia script used to reproduce experimental results.

We apply for the Available badge. This artifact is publically available on Zenodo, with a DOI.

We apply for the Reusable badge. The source code is provided and publically available on Zenodo; it is documented on this document on Sec. 2.3. A more detailed user manual for the Mopsa platform can be found at `https://mopsa.gitlab.io/mopsa-analyzer/user-manual/`. The sources can be recompiled following the instructions from Sec. 2.1. The analyzer can be re-used to analyze OCaml and Python programs other than our benchmarks. This document provides documentation to do so.

Note that the OCaml analysis, since based on [2], is limited to a pure, monomorphic subset of OCaml, with recursivity, higher-order and ADTs but without complex features such as modules or functors.

2.1 Build from sources

Dependencies are already installed in the Docker and the analyzer is already built. This part gives instruction to recompile the analyzer if you wish to.

The artifact is publically available on the long-term platform Zenodo at `https://zenodo.org/records/18498584`. Sources are available in file `/home/mopsa/mopsa-analyzer`. Prerequisites are detailed in `README.md`. They can be recompiled using:

```
cd mopsa-analyzer && make clean && ./configure && make && make install
```

OCaml programs

After that, the OCaml analyzer can be used on a file `[name].ml` with the following command:

```
mopsa-ocaml name.ml -config=ocaml/delay_exp_univ_pack.json -debug=print_summaries
```

To get help with the command, you can type:

```
mopsa-ocaml --help
```

All our OCaml tests can be found in the `ocaml` directory.

Python programs

The Python analyzer can be used on a file `[name].py` with the following command:

```
mopsa-python -py-set-alloc-pol=range -no-warning -gc -numeric_heterogeneous=polyhedra
-config=python/values-relational-pack-het-delexp.json name.py
```

To get help with the command, you can type:

```
mopsa-python --help
```

All our Python tests can be found in the `python` directory.

Note that the analysis of those examples requires you to **be in the `python` directory** to be able import the dependency `random_containers.py`.

2.2 Detailed analysis

To see the final state, you can run the command with the option `-lflow`.

OCaml

In OCaml, an abstract state should look like this:

```
Analysis terminated successfully
Last flow =
            cur :
                | Partition 1 :
                  ADTs constructs : #Start variant for each ADT variables
                     x -> {  Cons  },
                  numeric-relations(pack-[constraint]) : #DelExp
                     {  constraint(Cons_0(y)) >= constraint(Cons_0(x))}
                  numeric-relations : #Polyhedra domain
                    {  y=0  }
                | Partition 2 :
                  ...
V No alarm
Analysis time: 0.014s
Checks summary: 0 total
```

Since the OCaml analysis on which we built upon is disjunctive, pattern-matching are analyzed by creating one partition by pattern. To see the generated summaries in a program, you can run the command with the option `-debug=print_summaries`. The relations inferred by the DelExp domain are visible in the `numeric-relations(pack-[constraint])` section.

Python

A Python abstract state looks similar to an OCaml one but doesn't feature partitions. It also features various other domains, as the Python analysis supports a wide subset of the language:

```
Analysis terminated successfully
Last flow =
            cur :
                  random_containers : ...
                  TypeVar annotations : ...
                  addrs : # Map variable's names to related objects
                    l1 -> { @list:name.py:range1.list }
                    l2 -> { @list:name.py:range2.list }
                    ...
                  attributes : ...
                  heap : ...
                  numeric-relations(pack-[constraint]) : #DelExp
                     {  constraint(@list:name.py:range1.list) >=
                         constraint(@list:name.py:range2.list)}
                  numeric-relations : #Polyhedra domain
                    {  y=0  }
V No alarm
Analysis time: 0.014s
Checks summary: 0 total
```

Note that since Python lists are objects, the `addrs` domain maps the variable names to the related objects. Here, the variable `l1` is mapped to `list:name.py:range1.list`. Constraints on the content of `l1` will then be expressed on `list:name.py:range1.list`.

2.3 Structure of the source code

The source code can be found in `/home/mopsa/mopsa-analyzer`. It is organized as follow:

- The core of the analyzer is defined in `framework`.
- The `languages/universal` folder describes analysis components common to the C, Python and OCaml analyzers, such as interprocedural inlining, loop invariant inference, and the analysis of some intraprocedural constructs.
- The `languages/ocaml` folder of the OCaml analysis consists in different files and directories:
  - The entry point of the analysis is `iterators/program.ml`.
  - `common` defines the frontend and the utility functions.
  - `desugar` performs dynamic rewriting of OCaml expression into Universal expressions to delegate their analysis.
  - `iterators` defines behaviour for control-flow operators.
  - `lang` defines the abstract syntax tree and its pretty printer.
  - `objects` defines domains for standard OCaml values. In particular, `objects/delay_expand.ml` implements the DelExp (Sec 3.1), AffineDelExp (Sec 4.1) and ApplyDelExp (Sec 7.1) domains.
- The `languages/python` folder of the Python analysis is similarly structured. Note that the DelExp domain used by Python is the one stored in `languages/ocaml/objects`
- `share/mopsa/` contains `json` configurations for different analyses. Those configurations described which domains are used by the analysis. The configurations used there are:
  - For OCaml: `ocaml/delay_exp_univ_pack.json`
  - For Python: `python/values-relational-pack-het-delexp.json`

  They both rely on `languages/ocaml/objects/delay_expand.ml`.

More details on the Mopsa platform are available at `https://mopsa.lip6.fr/`. In particular, its user manual is available at `https://mopsa.gitlab.io/mopsa-analyzer/user-manual/`. This manual describes the general analyzer usage as well as options common to all analyses (C, Python, etc.), while we mention here only the options and usage specific to Python and OCaml analyses.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). The artifact is available on Zenodo, at: `https://zenodo.org/records/18498584`. This artifact consists in a docker image, in which the experimental evaluation from the article can be reproduced via two scripts. The sources of the analyzer are available inside the image. They are already built, but can be re-built following instructions of Sec. 2.1. To run the compressed docker image, run:

```
docker load < ecoop26-artefact.tar.gz
```

It should output:

```
Loaded image: mopsa-ocaml:mopsa-ocaml
```

You can then run:

```
docker run -it mopsa-ocaml:mopsa-ocaml
```

Once inside the docker, you can check that it works correctly by running:

```
julia/bin/julia wrapper_generate_tables.jl
```

This command should print progress bars for the analyses of all Python and OCaml programs with and without the DelExp domain and display the analysis times and overheads from Fig. 13 and 14.

4 Tested platforms

The experimental evaluation was performed on a Intel(R) Core(TM) Ultra 7 165H CPU with 16 Core and 32 GB of RAM, on Ubuntu 24.04.3. It was only tested on a x86-64 architecture.

5 License

The artifact is available under license LGPL.

6 MD5 sum of the artifact

2be58bceeab0fd0bab85dedfacca3a4c

7 Size of the artifact

1.89 GB

A Generating analysis time and overhead from Fig. 13 and 14. (5 min.)

To generate the analysis time and the overhead from Fig. 13 and 14., you simply need to run:

```
julia/bin/julia wrapper_generate_tables.jl
```

The command will run a Julia script. First, it will precompiled the libraries needed for its execution, then it will display the tables. It should last around 5 minutes. This script will run the analysis of each program with and without the family of DelExp domains and compare the execution time (averaged over 10 iterations). The analysis time should be less than a second, and the overhead of the same order of magnitude than in the paper (roughly between 2 and 4 for OCaml and 1 and 2 for Python).

B Generating summaries

To generate the summaries of the functions displayed in Fig. 13 and 14., the summaries must be visually checked in the execution trace of the analysis.

B.1 OCaml summaries (10 min.)

The Ocaml analysis is compositional: at the definition site of a function, it will generate a summary, i.e. an over-approximation of the behaviour of a function. We start with an example. To visualize the summary of `mult2.ml`, you can go in `ocaml` and run:

```
mopsa-ocaml -config=ocaml/delay_exp_univ_pack.json mult2.ml -debug=print_summaries
```

The `debug` option prints the inferred summaries. You should get as a result:

```
[print_summaries 0.061] SUMMARY OF mult2 :
  cur :
    | Cons (l) :
       ADTs constructs :
          Cons_1(l:271) -> {  Cons, Nil  },
          Cons_1(res(mult2:270)) -> {  Cons, Nil  },
          l:271 -> {  Cons  },
          res(mult2:270) -> {  Cons  } ,
       numeric-relations(pack-[constraint]) :
         {  constraint(2 * Cons_0(l:271) ) >= constraint(Cons_0(res(mult2:270)))},
       numeric-relations(pack-[globals]) : T,
    | Nil (l) :
       ADTs constructs :
          l:271 -> {  Nil  },
          res(mult2:270) -> {  Nil  } ,
       numeric-relations(pack-[globals]) : T
```

The abstract state separates cases where the input list `l` is an empty list (`| Nil (l)`) and where it is not (`| Cons (l)`). The result variable of a given function `f` is `res(f)`. You can visually check that in the empty case, the result list is empty, i.e. `res(mult2:270) ->  Nil` , whereas in the non-empty case, the content of the output list is included in the content of the input list multiplied by 2: `constraint(2 * Cons_0(l:271) ) >= constraint(Cons_0(res(mult2:270)))`.

Other cases are similar. When in file `tests/ocaml`, to get the summary of the function defined in `[name].ml`, you only need to run:

```
mopsa-ocaml -config=ocaml/delay_exp_univ_pack.json name.ml -debug=print_summaries
```

B.2 Python summaries (10 min.)

Note that the Python analysis on which we built upon [1] does not support compositionality, as this is made very difficult by Python's dynamic typing and flexible semantics (e.g., overloading of common operators such as + with ___add___, and ___radd___ methods). Instead of printing function summaries, we check that the correct relations are inferred between the function's inputs and outputs.

The analysis of our examples requires you to **be in the `python` directory** to be able import the dependency `random_containers.py`. To visualize the result of `copy.list.py`, you can go in `python` and run:

```
mopsa-python -py-set-alloc-pol=range -no-warning -gc -numeric_heterogeneous=polyhedra
-config=python/values-relational-pack-het-delexp.json copy.list.py
```

Our Python files end up with print instructions over inputs and outputs variables (e.g. `print(l1,l2)`). This will print the final abstract state of the analysis projected over those two variables. You should get as a result:

```
addrs : ...
   l1:5 -> {  @list:random_containers.py:16.6-8  },
   l2:6 -> {  @list:copy.list.py:4.6-8  }
numeric-relations(pack-[constraint]) :
   { constraint(@list:random_containers.py:16.6-8.list) >=
      constraint(@list:copy.list.py:4.6-8.list)  },
```

In the `numeric-relations(pack-[constraint])` part, you should be able to spot: `constraint(list:random_containers.py:16.6-8.list) >= constraint(list:copy.list.py:4.6-8.list)`.

To get the complete final abstract state, you should run the command with `-lflow` option: the abstract state is then quite big since the Python analysis implements domains for numerous features. Other cases are similar. When in file `tests/python`, to get the final state of the function defined in `[name].py`, you only need to run:

```
mopsa-python -py-set-alloc-pol=range -no-warning -gc -numeric_heterogeneous=polyhedra
-config=python/values-relational-pack-het-delexp.json name.py
```

References

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

2 Milla Valnet, Raphaël Monat, and Antoine Miné. Compositional static value analysis for higher-order numerical programs. In *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 333, page 15. Dagstuhl Publishing, 2025. doi:10.4230/LIPIcs.ECOOP.2025.32.