MONARCH: A MONadic ARCHitecture for Static Analyses through Abstract Definitional Interpreters

Monarch is a static analysis framework for developing static analyses based on the abstract definitional interpreters paradigm. This artifact contains the source code of the framework and reflects the state of the framework at 2024-09-12.

The latest version of the framework is available at https://github.com/softwarelanguageslab/monarch.

Setting up

Using the Docker image

This artifact is also provided as a Docker image (in `image.tar.gz`) that includes all the tools with their correct versions in addition to a compiled executable that can be used to test the Python analysis provided by the framework.

The Docker image can be used as follows:

```
$ docker load < image.tar.gz
$ docker run -it monarch
Missing: COMMAND

Usage: maf-exe COMMAND

  MAF: Monadic Analysis Framework
```

It exposes the `maf-exe` binary which expects a subcommand (ie., the analysis to run) and some optional arguments (e.g.., the file to analyze). For instance to analyze a Python file, the following command may be used:

```
$ docker run -it monarch python -f maf2-analysis/programs/python/Counter.py
```

Building from source

The framework can be built from source but requires a few dependencies:

- cabal: the framework was tested with cabal version 3.10
- ghc: version 9.4.8 of GHC is required for the framework to build successfully.

Both of these dependencies can be easily installed using `ghcup`.

Using the Framework

In this section we detail how the framework is structure and how an analysis can be constructed from its building blocks.

Framework Structure

The framework is structured into multiple packages according to their responsibilities:

- `maf2-syntax` provides parsers and data structures (such as AST definitions) for the languages supported by our framework. Currently it includes support for Scheme, Python and Erlang.
- `maf2-domains` provides the basic building blocks and their combinators for constructing abstract domains to be used by a static analysis.
- `maf2-analysis` provides the building blocks for expressing program semantics and instantiating a static analysis from this program semantics.

Domains Package This package mainly consists of two modules: `Lattice` and `Domain`. The `Lattice` module provides the basic lattice type classes (in `Lattice.Class`) such as `PartialOrder` and `Joinable`. Moreover, it contains instances of these type classes for basic Haskell data types such as `Maybe` and `Set`.

The `Domain` module refers to specific abstract domain and is split similarly to the `Lattice` module. It contains type classes (in `Domain.Core.*.Class`) that express abstract domain-specific operations such as the `NumberDomain` which specifies operations such as `plus` and `minus` and also instances for these type classes (for example `Domain.Core.NumberDomain.ConstantPropagation`).

For implementing combinations of domains the `HMap` structure may be used (which correspond to the `SparseLabeledProduct` from the paper). To use them to their fullest extent both `Data.TypeLevel.HMap` and `Lattice.HMapLattice` must be imported. The paper contains more details on its usage.

Finally, the `maf2-domains` package also includes abstract domains specific to our supported programming languages. For example, the `Domain.Python` module contains the abstractions used by the Python analysis.

Analysis Package The analysis package contains building blocks for expressing program language semantics as well as for instantiating an analysis. Both building blocks are provided in the `Analysis.Monad.*` modules. These modules define the type classes used for expressing language semantics as well as the monad transformer that are instances of these type classes.

Similar to the `maf2-domains` package, language specific semantics and analysis instantiations are in their own modules. For instance, the semantics for the Scheme analysis is in `Analysis.Scheme.Semantics`, while the semantics for the Python semantics are in the `Analysis.Python.Semantics`.

The structure for each language-specific analysis is similar. Language semantics are in `Analysis.LANG.Semantics`, while its monad type class constraints are in `Analysis.LANG.Monad`. This module sometimes also defines language-specific monadic type classes and their transformers. Static analysis instantiations are usually in the `Analysis.LANG` module, except for Python which uses the `Analysis.LANG.Fixpoint` module.

Usage as a library

New analyses can be built on top of Monarch by embedding them into their own folder in the appropriate `maf2-*` package. An alternative, more recommended, way is to use the framework as a library by adding it as a regular dependency to your Cabal or Stack project.

To this end, your `cabal.project` file needs to contain the following:

```
with-compiler: ghc-9.4.8

source-repository-package
   type: git
   location: https://github.com/softwarelanguageslab/maf-hs.git
   tag: 87ccbf160e0a1b0f579e808723fd4d50fcc848e3
   subdir: maf2-syntax maf2-analysis maf2-domains
           thirdparty/language-python/language-python


source-repository-package
   type: git
   location: https://github.com/bramvdbogaerde/layers-mtl.git
   tag: 5201fd1814053630f762c382a79e49b7bf14df8f
```

```
packages: .
```

The first repository package makes the Monarch framework available to your package since it is not published on Hackage. It ensures that a correct version of the `language-python` dependency is used that is compatible with Monarch. The second repository package makes the `layers-mtl` package available, which is used internally in Monarch but not available on Hackage. **Note that only GHC 9.4.8 is currently supported.**

The commit hashes used above point to the version of the framework bundled with this artifact (excluding the commit adding this README).

Next, you need to add the Monarch packages (i.e., `maf2-syntax`, `maf2-domains`, `maf2-analysis`) as dependencies to your `packagename.cabal` file. Not all Monarch packages are required as dependencies however; you can add or remove packages depending on the desired functionality.

An Analysis for a simple lambda-calculus

To get started with implementing static analyses in Monarch, we suggest the `examples/lambda-calculus` package as an example. This package contains an analysis for a simple lambda-calculus inspired programming language.

Reproducing examples of the paper

The paper contains a case study of implementing a static analysis in Python. The code listings presented in the paper can be found at the following locations:

- **3.1: Abstract Domain**: the implementation of the `PyObj` can be found in the `maf2-domains` package inside the `Domain.Python.Objects.PyObjHMap` module. Here, it is defined as the `PyObjHMap` on line 60. The contents of the sparse labeled product are defined as `PyPrm` just above the definition of the `PyObjHMap`.
- **3.2: Semantics**: the semantics presented in the paper is spread across a set of modules, all in the `maf2-analysis` package.
  - As opposed to the presentation in the paper, the `PyM` monad is defined as type class-instance pair in the `Analysis.Python.Monad` module. Lines 81-96 define an equivalent set of constraints as those presented in the paper.
  - The definition of the primitives is available in the `Analysis.Python.Primitives` module. In contrast to the paper, which focusses on the implementation of `plus`, the artifact includes a number of abstractions to reduce code duplication for similar operations (e.g., other arithmetic operations).
  - The semantics for Python is defined in the `Analysis.Python.Semnatics` module. The paper presents the semantics for escaping control flow. This semantics can be found at line 107 and line 39. The implementation of `pyReturnable`, which catches the escaping value and returns it, is inlined in the paper for clarity but is implemented in the `PyM` instance in the artifact. Those implementations are equivalent otherwise and use `MonadEscape` internally.
  - The implementation of `lookupAttr`is exactly as in the paper and can be found on line 97 in the `Analysis.Python.Objects` module.
- **3.3: Analysis**: the instantiation of the Python semantics into a static analysis for Python can be found in `Analysis.Python.Fixpoint` module. Its implementation is identical to the presentation in the paper.

Executing the Python Analysis

The Python analysis in `Analyis.Python.Fixpoint` computes an abstraction of the program's memory. This analysis can be executed by running the artifact with the appropriate set of

arguments.

The command depicted below illustrates how the artifact can be invoked for executing the Python analysis (assuming the `maf2-analysis` package was installed using `cabal install`):

```
maf-exe python -f maf2-analysis/programs/python/Counter.py
```

This command executes the Python analysis for file `Counter.py`. This file has the following contents:

```
class Counter:
    def __init__(self, initial_value):
        self.value = initial_value
    def increment(self):
        self.value += 1
    def decrement(self):
        self.value -= 1
    def status(self):
        return self.value

class DoubleCounter(Counter):
    def increment(self):
        self.value += 2

def f():
    ctr = DoubleCounter(100)
    ctr.increment()
    ctr.increment()
    ctr.decrement()
    return ctr.status()

f()
```

It essentially implements a counter that supports operations `increment`, `decrement` and `status`. When this program is executed in a concrete Python interpreter, the result of its execution is `103`.

The result of our static analysis, however, is as follows:

```
PROGRAM:

Counter@gbl = type@gbl("Counter",(),{})
lambda ():
   Counter@gbl.__init__ = lambda (self@1,initial_value@2):
      self@1.value = initial_value@2
   Counter@gbl.increment = lambda (self@3):
      self@3.value = self@3.value.__add__(1)
   Counter@gbl.decrement = lambda (self@4):
      self@4.value = self@4.value.__sub__(1)
   Counter@gbl.status = lambda (self@5):
      return self@5.value()
DoubleCounter@gbl = type@gbl("DoubleCounter",(Counter@gbl),{})
lambda ():
   DoubleCounter@gbl.increment = lambda (self@6):
      self@6.value = self@6.value.__add__(2)()
f@gbl = lambda ():
   ctr@gbl = DoubleCounter@gbl(100)
```

```
   ctr@gbl.increment()
   ctr@gbl.increment()
   ctr@gbl.decrement()
   return ctr@gbl.status()
f@gbl()

RESULTS PER COMPONENT:

          <main> | [[None]]
      <func 1:1> | [[None]]
      <func 2:5> | [[None]]
      <func 4:5> | [[None]]
      <func 6:5> | [[None]]
      <func 8:5> | [[5:9],[7:9],[13:9],[16:25]]
     <func 11:1> | [[None]]
     <func 12:5> | [[None]]
     <func 15:1> | [[5:9],[7:9],[13:9],[16:25]]

OBJECT STORE RESULTS:

         [5:9] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([5:9],[[PrimObject IntAdd]]),
                                      ([7:9],[[PrimObject IntAdd]]),
                                      ([13:9],[[PrimObject IntAdd]]),
                                      ([16:25],[[PrimObject IntAdd]])]}
         [5:9] | {__class__ → [[TypeObject IntType]], @IntPrm → Top}
         [7:9] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([5:9],[[PrimObject IntSub]]),
                                      ([7:9],[[PrimObject IntSub]]),
                                      ([13:9],[[PrimObject IntSub]]),
                                      ([16:25],[[PrimObject IntSub]])]}
         [7:9] | {__class__ → [[TypeObject IntType]], @IntPrm → Top}
        [13:9] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([5:9],[[PrimObject IntAdd]]),
                                      ([7:9],[[PrimObject IntAdd]]),
                                      ([13:9],[[PrimObject IntAdd]]),
                                      ([16:25],[[PrimObject IntAdd]])]}
        [13:9] | {__class__ → [[TypeObject IntType]], @IntPrm → Top}
       [16:11] | {__class__ → [[11:1:ClsNew]], ?value → [[5:9],[7:9],[13:9],[16:25]]}
[16:11:IniBnd] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([16:11],[[2:5],[PrimObject ObjectInit]])]}
       [16:25] | {__class__ → [[TypeObject IntType]], @IntPrm → Constant 100}
        [17:5] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([16:11],[[4:5],[12:5]])]}
        [18:5] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([16:11],[[4:5],[12:5]])]}
        [19:5] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([16:11],[[6:5]])]}
       [20:12] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([16:11],[[8:5]])]}
         [1:1] | {__class__ → [[TypeObject CloType]], @CloPrm → fromList [<func@1:1>]}
  [1:1:FrmTag] | {__class__ → [[TypeObject FrameType]]}
  [1:1:ClsStr] | {__class__ → [[TypeObject StringType]], @StrPrm → Constant "Counter"}
  [1:1:ClsTup] | {__class__ → [[TypeObject TupleType]], @TupPrm → CPList [] 0 []}
```

```
[1:1:ClsDct] | {__class__ → [[TypeObject DictionaryType]],
                @DctPrm → CPDict (fromList []) (fromList []) []}
[1:1:ClsNew] | {__class__ → [[TypeObject TypeType]],
               ?__init__ → [[2:5]], ?__mro__ → [[1:1:IniCll]],
               ?__name__ → [[1:1:ClsStr]], ?decrement → [[6:5]],
               ?increment → [[4:5]], ?status → [[8:5]]}
[1:1:IniBnd] | {__class__ → [[TypeObject BoundType]],
               @BndPrm → fromList [([1:1:ClsNew],[[PrimObject TypeInit]])]}
[1:1:IniCll] | {__class__ → [[TypeObject TupleType]],
               @TupPrm → CPList [[[1:1:ClsNew]],
                                 [[TypeObject ObjectType]]] 2
                                 [[1:1:ClsNew],[TypeObject ObjectType]]}
       [2:5] | {__class__ → [[TypeObject CloType]],
               @CloPrm → fromList [<func@2:5>]}
[2:5:FrmTag] | {__class__ → [[TypeObject FrameType]],
               ?initial_value → [[16:25]],
               ?self → [[16:11]]}
       [4:5] | {__class__ → [[TypeObject CloType]],
               @CloPrm → fromList [<func@4:5>]}
[4:5:FrmTag] | {__class__ → [[TypeObject FrameType]],
               ?self → [[16:11]]}
       [6:5] | {__class__ → [[TypeObject CloType]],
               @CloPrm → fromList [<func@6:5>]}
[6:5:FrmTag] | {__class__ → [[TypeObject FrameType]],
               ?self → [[16:11]]}
         [8:5] | {__class__ → [[TypeObject CloType]],
                  @CloPrm → fromList [<func@8:5>]}
  [8:5:FrmTag] | {__class__ → [[TypeObject FrameType]],
                  ?self → [[16:11]]}
        [11:1] | {__class__ → [[TypeObject CloType]],
                  @CloPrm → fromList [<func@11:1>]}
 [11:1:FrmTag] | {__class__ → [[TypeObject FrameType]]}
 [11:1:ClsStr] | {__class__ → [[TypeObject StringType]],
                  @StrPrm → Constant "DoubleCounter"}
 [11:1:ClsTup] | {__class__ → [[TypeObject TupleType]],
                  @TupPrm → CPList [[[1:1:ClsNew]]] 1 [[1:1:ClsNew]]}
 [11:1:ClsDct] | {__class__ → [[TypeObject DictionaryType]],
                  @DctPrm → CPDict (fromList []) (fromList []) []}
 [11:1:ClsNew] | {__class__ → [[TypeObject TypeType]],
                  ?__mro__ → [[11:1:IniCll]], ?__name__ → [[11:1:ClsStr]],
                  ?increment → [[12:5]]}
 [11:1:IniBnd] | {__class__ → [[TypeObject BoundType]],
                  @BndPrm → fromList [([11:1:ClsNew],[[PrimObject TypeInit]])]}
 [11:1:IniCll] | {__class__ → [[TypeObject TupleType]],
                  @TupPrm → CPList [[[11:1:ClsNew]],
                                    [[1:1:ClsNew]],
                                    [[TypeObject ObjectType]]] 3
                                    [[1:1:ClsNew],
                                    [11:1:ClsNew],[TypeObject ObjectType]]}
       [12:5] | {__class__ → [[TypeObject CloType]], @CloPrm → fromList [<func@12:5>]}
 [12:5:FrmTag] | {__class__ → [[TypeObject FrameType]], ?self → [[16:11]]}
       [15:1] | {__class__ → [[TypeObject CloType]], @CloPrm → fromList [<func@15:1>]}
 [15:1:FrmTag] | {__class__ → [[TypeObject FrameType]]}
       [5:23] | {__class__ → [[TypeObject IntType]], @IntPrm → Constant 1}
```

```
 [7:23] | {__class__ → [[TypeObject IntType]], @IntPrm → Constant 1}
[13:23] | {__class__ → [[TypeObject IntType]], @IntPrm → Constant 2}
```

The analysis first prints a simplification of the input program. Then, it prints the result of each component, and finally it prints all memory addresses with their associated values.

In the example program above, each memory address is represented by its allocation site. For example, the instance of `DoubleCounter` created at line 16, has the address of `[16:11]`.

The result of the program is stored at component `<func 15:1>`, which is a set of pointers pointing to the different values of the counter. This result has multiple values because of imprecision introduced by the flow-insensitive configuration of the analysis.