





Monarch: A Modular Framework for Abstract Definitional Interpreters in Haskell

Bram Vandenbogaerde , Sarah Verbelen ✉, Noah Van Es , and
Coen De Roover 
{bram.vandenbogaerde, sarah.verbelen, noah.van.es,
coen.de.roover}@vub.be

Vrije Universiteit Brussel, Brussels, Belgium

Abstract. Abstract definitional interpreters are an approach to developing abstract interpretation-based static analyses in which language semantics are expressed through monadic recursive interpreters. These interpreters are then instantiated with an abstract value domain and executed in a suitable monadic context that carries abstract program state. Unfortunately, correctly implementing these definitional interpreters remains a difficult task. Moreover, instantiating analyses requires configuring many components. In this tool paper, we present the design of a framework called MONARCH that provides reusable components to programmers for implementing abstract definitional interpreters. Our design consists of the following components: abstract domains, a framework for expressing program semantics, and analysis instantiation techniques. Finally, we present an implementation in Haskell and give example instantiations in Scheme and Python to show how these components are used.

Keywords: Abstract Definitional Interpreters · Abstract Interpretation · Static Analysis.



1 Introduction

Static analyses aim to decide behavioral program properties without actually running the program. Their use is widespread in compilers, integrated development environments, software verification, and so on. Abstract interpretation is a principled approach to static analysis design where program properties are computed through an abstraction of concrete program semantics. For example, whereas a concrete program semantics calculates the value of `fac(5)` to be 120, an abstract interpreter might only compute its sign (i.e., `+`). Abstract definitional

interpreters [5] (ADI) offer a recipe for constructing such abstract program semantics by deriving them from a recursive evaluation function parametrized by a computational context (i.e., a monad) and an abstract value domain (e.g., the sign). This computational context and abstract value domain are then instantiated to obtain a functioning abstract interpretation based static analysis.

Unfortunately, developing such abstract definitional interpreters remains a non-trivial task. For this reason, we present our design of MONARCH, a framework for constructing static analyses based on the abstract definitional interpreter approach. We highlight how we use advanced programming language concepts to create reusable building blocks that can be used by analysis developers.

To summarise, we make the following contributions in this tool paper:

- We present a modular design of a framework for static analysis using abstract interpretation called MONARCH. Our design consists of foundational building blocks and combinators for abstract domains, a monadic framework to specify program semantics, and techniques based on monad transformers to enable flexible and layered instantiations of program analyses. Our design focuses on modularity and composability.
- We present two case studies to show how several practical challenges are addressed, such as non-determinism and control flow (i.e., escaping control flow using `MonadEscape`), reusing semantics using effect polymorphism, and providing an efficient way to represent abstract values (i.e., sparse labeled products introduced in Section 4.1).

2 Motivation

In this section, we first introduce the key components of abstract definitional interpreters through the implementation of a toy language derived from the λ -calculus. Moreover, we will discuss the difficulties in implementing such abstract definitional interpreters. The toy language is depicted in Listing 1 and consists of lambda expressions, applications, numbers, `if` expressions, assignments, sequencing, and exceptions. The language lacks booleans for simplicity. Instead, the number 0 represents falsehood and any other number represents truthiness.

```
data Exp = Lam String Exp | Num Int | Var String | App Exp Exp
        | If Exp Exp Exp | Set String Exp | Seq Exp Exp
        | Throw Exp | Catch String Exp Exp

type Env = Map String Val
data Val = Clo Exp Env | NumV Int
```

Listing 1: A toy language based on the λ -calculus. It supports lambda abstraction, application, if, assignments, sequencing, exceptions, variables, and number literals.

Abstract definitional interpreters derive their semantics starting from a concrete recursive interpreter. The listing below depicts an interpreter that consists of an evaluation function `eval` and a closure application function called `apply`. The evaluation function is executed in a monadic context `m` which supports effects for tracking the store and the environment, and for catching and throwing exceptions. The effects to track the environment and the store are generated through the `MonadEnvironment` and the `MonadStore` type class, respectively. These type classes contain operations to get the current store and environment, as well as to change them. The effects to catch and throw exceptions are generated through the `catchError` and `throwError` operations respectively, specified in the `MonadError` type class.

```

eval :: (MonadEnvironment m, MonadStore m, MonadError m)
      => Exp -> m Val
eval (Lam x e)      = bind getEnv (\env -> unit (Clo (Lam x e) env))
eval (Num n)        = unit (NumV n)
eval (Var x)        = lookupEnv x >>= lookupSto
eval (App e1 e2)    = do
  { v1 <- eval e1 ; v2 <- eval e2 ; apply v1 v2 }
eval (If e1 e2 e3) = ifM (eval e1) (eval e2) (eval e3)
eval (Set x e)      = do
  { v <- eval e ; a <- lookupEnv x ; setSto a v }
eval (Seq e1 e2)    = eval e1 >> eval e2
eval (Throw e)      = eval e >>= throwError
eval (Catch x e1 e2) =
  eval e1 `catchError` (\v -> do env <- getEnv
                                let clo = (Clo (Lam x e2) env)
                                apply clo v)

apply :: (MonadEnvironment m, MonadStore m, MonadError m)
       => Val -> Val -> m Val
apply (Clo (Lam x e) lenv) v2 = do
  a <- alloc x
  updateSto a v2
  withEnv (extendEnv a lenv) (eval e)

```

From this concrete interpreter, an abstract definitional interpreter can be derived [5,25]. To this end, the concrete interpreter needs to undergo a number of transformations. First, the concrete values (represented by `Val`) of the language need to be abstracted. As an example, closures are abstracted to sets of abstract closures and numbers to their sign. The goal of this abstraction is to render the state space finite such that the evaluation terminates for any program input.

Next, several operations within the semantics need to be adapted according to the change in the value domain. For example, the `apply` function uses pattern matching to determine which closure needs to be applied to value `v2`. When using sets as an abstraction, this pattern matching needs to be replaced by a traversal

over all elements of the set in such a way that each closure can be applied to `v2` separately and their results can be joined together.

The evaluation of `if` expressions is another example. As numbers are abstracted to their sign, it is possible that the condition is simultaneously true and false. To overapproximate the program behavior in this case, operations such as `ifM` need to be adapted so that the consequent and alternative of the `if` expression are both executed and the results of both branches are joined afterwards. To support these changes, the evaluation function can be executed in a different monadic context that supports non-deterministic computations. An example of such monadic context is the `MonadPlus` type class. This type class specifies an `mplus` function, which executes two computations non-deterministically, and an `mzero` function, which represents the empty computation.

Overapproximating exception handling is challenging, as the exception handler (i.e., `Catch`) must account for both execution paths in which an exception is thrown and those in which no exception occurs. In this example, we make use of the `MonadEscape` type class to accomplish this. This type class is explained in more detail in Section 4.2.

The final step is to render the number of calls to `eval` finite. To ensure termination, the number of inputs need to be finite and the outputs should converge to a stable value. This is usually accomplished by caching the results of `eval` for any given input, and returning the cached result if `eval` is executed another time with input that has been evaluated before.

```
eval :: (MonadPlus m, MonadEscape m,
        MonadEnvironment m, MonadStore m)
      => Exp -> m AVAl
eval (Lam x e) = bind getEnv (\env ->
  unit (Set.singleton (Clo (Lam x e) env)))
eval (Num n)   = unit (NumV (sign n))
eval (Var x)   = lookupEnv x >>= lookup
eval (App e1 e2) = do
  { v1 <- eval e1 ; v2 <- eval e2 ; apply v1 v2 }
eval (If e1 e2 e3) = mplus (eval e2) (eval e3)
eval (Set x e)     = do
  { v <- eval e ; a <- lookupEnv x ; update a v }
eval (Seq e1 e2) = eval e1 >> eval e2
eval (Throw e)   = eval e >>= throw
eval (Catch x e1 e2) =
  eval e1 `catch` (\v -> do
    env <- getEnv
    let clo = (Clo (Lam x e2) env)
        apply (Set.singleton clo) v)
```

```
apply :: (MonadPlus m, MonadEscape m,
         MonadEnvironment m, MonadStore m)
```

```

=> AVa1 -> AVa1 -> m AVa1
apply clos v2 =
  foldr mplus mzero (Set.map
    (\case (Clo (Lam x e) lenv) -> do
      a <- alloc x
      updateSto a v2
      withEnv (extendEnv a lenv) (eval e)
    (NumV _) -> mzero)
  clos )

```

The code listing shown above depicts the changes necessary for rendering our concrete interpreter abstract. Highlighted in orange are changes made to render the value domain abstract and finite, highlighted in blue are changes concerning the semantics. This example illustrates the intersection of various analysis concerns, as well as the need for suitable abstractions for implementing these analysis concerns. In the rest of this paper, we propose a set of abstractions formulated as type classes to address these concerns.

3 Background

In this section we review the concepts underpinning the design of abstract definitional interpreters. We start by recalling the concept of a *monad* which enables expressing computational effects in a definitional interpreter. Next, we illustrate how functions that are polymorphic in their computational effects can be expressed, and show how they are used to configure an abstract definitional interpreter for a specific analysis.

3.1 Monads

Monads are a key ingredient of our conceptual framework. Monads enable separating computational effects from the actual results of a computation [18, 19]. They are characterised by two functions (depicted below): a **unit** and a **bind**. The **unit** function embeds an effect-free computation **a** into an effectful computation **m a**, while the **bind** function extracts the value from an effectful computation and applies an effectful function to it, essentially providing the ability for two computations to be executed in *sequence*.

```

class Monad m where
  unit :: a -> m a
  bind :: m a -> (a -> m b) -> m b

```

In Haskell, to avoid having to nest **bind** expressions, the so-called **do**-notation can be used. The code listing below demonstrates its usage and its **bind**.

```

example = do
  a <- m1
  b <- m2
  return (a + b)
example =
  bind m1 (\a ->
    bind m2 (\b ->
      return (a+b))

```

A classic example of monadic computations is computations that carry *state*. Such computations can be characterised by the interface described in `MonadState`.

```
class (Monad m) => MonadState s m | m -> s where
  put :: s -> m ()
  get :: m s
```

The `MonadState` type class states that for some monad `m` and state `s`, the interface can be used if functions `put` and `get` are implemented. A suitable candidate for `m` in this case is a function `forall a . s -> (a, s)` for some fixed `s`. We will refer to such functions with the type `State s`. The implementation of these instances for the `Monad` and `MonadState` type classes are shown below.

```
type State s = forall a . s -> (a, s)
instance Monad (State s) where
  unit v = \s -> (v, s)
  bind m f = \s -> let (a, s') = (m s) in (f a) s'
instance MonadState s (State s) where
  put s = \_ -> ((), s)
  get = \s -> (s, s)
```

3.2 Effect Polymorphism

The previous section introduced the `MonadState` type class. One might wonder why such a type class is needed, as computations carrying state can be readily expressed using `State s`. Encoding `MonadState` as a type class enables *effect polymorphism*. Consider, for example, a function `inc` (depicted below) that implements incrementing an integer statefully. The function is not concerned with the internal representation of this state in the monadic structure. Thus, its type reflects that it can be executed in any monadic context `m`, given that this context implements the `MonadState` type class and provides functions `put` and `get`.

```
inc :: MonadState Int m => m ()
inc = bind get (\n -> put (n+1))
```

In Section 4.2, we discuss how this effect polymorphism enables reusable semantics for different analysis instantiations.

4 Architecture

Figure 1 depicts the architecture of MONARCH. The framework consists of three major parts. The `DOMAIN` part depicted in the bottom left provides building blocks for abstract domains. It is detailed in Section 4.1. Section 4.2 presents the `SEMANTICS` part depicted in the bottom right, which provides monadic interfaces for specifying abstract semantics. In Section 4.3, the final part `ANALYSIS` (depicted at the top) is discussed. This part provides building blocks for instantiating an abstract semantics into a *static program analysis*. It does so by providing *monad transformers* that implement the monadic interfaces of the semantics.

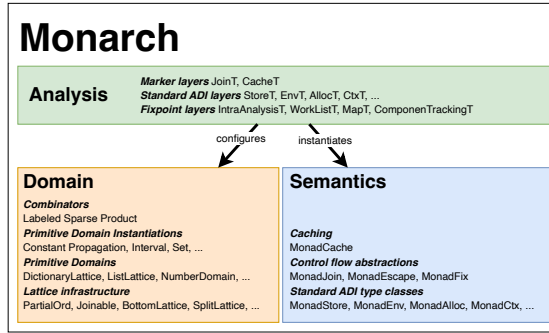


Fig. 1. Architecture of MONARCH.

4.1 Domain

In this section we first introduce the primitive building blocks for creating representations of abstracted primitive values. Next, we propose *combinators* to combine these primitive building blocks into representations of abstractions of more complex values. Finally, we discuss efficiency and performance implications of such representations and we propose a memory-efficient and type-safe one.

Primitive Building Blocks: Lattices MONARCH follows a similar design as [3], using lattices. Lattices are a mathematical structure formed using partially ordered sets for which each finite subset has a supremum and an infimum. Mathematically, it is sufficient to define the elements of the set and its partial ordering relation. However, in practice, it is often more efficient to formulate the operations for computing the supremum and infimum explicitly. These operations are called the *join* and *meet* of a lattice respectively. For the purposes of this paper, we discuss *join* only. Thus we arrive at a structure with two operations and two elements: a partial order, a join, a top element, and a bottom element.

```
class PartialOrder v where
  leq    :: v -> v -> v
class (PartialOrder v) => BottomLattice v where
  bottom :: v
class (PartialOrder v) => TopLattice v where
  top    :: v
class (PartialOrder v) => Joinable v where
  join   :: v -> v -> v
```

All abstract domains are an instance of a *join semi-lattice* which is described in MONARCH by four type classes: `PartialOrder`, `Joinable`, `BottomLattice` and `TopLattice` (depicted above). Splitting the operations of a join semi-lattice in this way provides more flexibility for creating instances of these type classes

when a type does not implement all of them. For example, the top element of a *powerset lattice* of closures cannot be defined without access to the analyzed program, and thus lacks an implementation for `TopLattice`.

Primitive Building Blocks: Domain Lattices form *domains* by extending them with domain-specific operations. For example, lattices for representing *numbers* can form a *number domain* by implementing abstract versions of operations such as `+`, `*`, etc. These operations are combined into type classes.

A minimal domain needs an operation for converting concrete values to values belonging to that domain and an operation to check whether a concrete value is covered by an abstract value. We combine these operations into a `Domain` type class (depicted below), which serves as the superclass for more specific domains.

```
class (PartialOrder v) => Domain v c where
  alpha  :: c -> v
  gamma  :: c -> v -> Bool
  gamma c v = leq (alpha c) v
```

The `alpha` function converts concrete values `c` to abstract values `v`, while the `gamma` function checks whether a concrete value is covered by the abstract value. These functions borrow their names from *Galois connections*, which are often used to describe abstract semantics in abstract interpreters.

Combinators: Products We conclude this section by discussing how to combine multiple domains. Program analyses typically require combinations of domains to reason about program behavior. These combinations are usually composed of multiple primitive domains and can be expressed as their product. The main problem of product values is their memory consumption when implemented naively. A naive implementation could represent this product as a structure with a field for each possible value. Some of these fields would then be set to \perp to indicate their absence in the abstract value. However, this means that most product values will contain only a few distinct values (as most values are absent and thus \perp), resulting into inefficient memory utilisation.

We demonstrate this using empirical evidence gathered by running static analyses on a total of 129 Scheme benchmark programs, grouped into six benchmark suites. Table 1 shows the origin of these benchmark suites, as well as their number of lines of code excluding comments and blank lines.

We show the inefficiency by measuring the number of distinct values within a product on a number of Scheme programs. The benchmark analysis is based on the modular analysis of Nicolay et al. [20, 24] and computes an abstraction of the program’s memory. The analysis is configured as follows:

- **Domain:** we configured the analysis to compute its analysis results in a *constant propagation domain* (i.e., for computing the set of constant variables), except for pointers and closures which are stored as sets of abstract addresses and pairs of expressions with their environment, respectively.

Table 1. LoC for programs in each benchmark.

benchmark	count	mean	min	50%	max
ad	20	105.55	23	52	604
gabriel	11	109.00	19	46	570
gambit	25	154.68	1	33	632
scpl	57	42.60	4	26	318
toplas98	3	317.67	188	188	577
wcr2019	13	119.46	1	51	437

- **Context sensitivity:** we configured the analysis to be context insensitive meaning that calls are only differentiated based on the closure being called and not, for example, based on the call-site.

These parameters affect the precision of the resulting analysis. We chose these parameters since, in practice, they offer a good trade-off between precision and performance. However, we argue that our results present a best-case estimation of the real memory usage when using a naive implementation of a product.

Table 2. Percentage of values of a certain size, computed for 119 benchmark programs.

	1	2	3	4	5	6	7	8
min	32.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00
25%	86.10	1.72	0.00	0.00	0.00	0.00	0.00	0.00
50%	92.63	5.58	0.88	0.00	0.00	0.00	0.00	0.00
75%	98.05	9.34	2.04	0.21	0.00	0.00	0.00	0.00
max	100.00	65.25	20.64	25.46	12.04	1.58	56.57	22.33

Table 2 depicts the results of our study. We instrumented the analysis to keep track of the number of constituents for each product value being created. The Scheme domain implemented in the analysis consists of *18 constituents*. However, in our benchmarks, only 8 distinct constituents were inhabited. The table shows the minimum, maximum, and median values for the number of values in each category.

Our results demonstrate that in most cases, abstract product values contain only one constituent. In more precise analyses with more precise domains and choice for sensitivity, the proportion of values with only one constituent is likely to increase. This is because a value with more constituents is less precise than values with fewer constituents. A more precise domain gives rise to more precise analysis results therefore increasing the likelihood of fewer constituents in the product. Note that we measure the number of values created during an analysis in order to show the potential impact on memory consumption. Our results do not show the precision of the final analysis result.

Based on the results of our empirical study, we recommend implementing product values as sparse labeled products. To this end, MONARCH provides a `SparseLabeledProduct` data structure that is internally implemented as a heterogeneous map. Its function is to express the product’s constituents at a type-level, while providing an efficient run-time memory representation. An excerpt of the sparse labeled product for representing Scheme values is depicted below.

```
data SchemeType = IntType | BoolType | PaiType |
                CloType | PrimType | ...
type Values      = '[ IntType  ::-> CP Integer,
                    BoolType  ::-> CP Bool,
                    PaiType   ::-> Set Addr,
                    CloType   ::-> Set (Exp, Env),
                    PrimType  ::-> Set String, ...]

type SchemeVal  = LabeledSparseProduct Values
```

The constituents of the sparse labeled product are expressed as a type-level list of key-value pairs (denoted by `key ::-> value`). At run time `key` types are demoted to data values so that they can be used as keys in a map datastructure. This enables the efficient memory representation of the labeled product, since absent constituents do not need to be stored in the map datastructure. Moreover, expressing the product as a type-level list enables reasoning about its constituents through typeclass constraints.

This representation renders the combination of various abstract domains trivial. Moreover, it provides both type-safety and a memory-efficient run-time representation that only stores the present subdomains.

4.2 Semantics

In MONARCH, programming language semantics are expressed as recursive definitional interpreters. These interpreters are rendered compatible with the abstract interpretation framework through a library of monadic type classes that allow expressing programming language semantics. Most of our monadic type classes are standard for abstract definitional interpreters [1, 5, 13]. MONARCH provides type classes for interacting with abstract environments (`MonadEnvironment`), abstract stores (`MonadStore`) and abstract memory allocation (`MonadAlloc`).

In addition to these standard type classes, MONARCH defines interfaces for expressing conditional control flow and escaping control flow, as well as for expressing and automatically deriving fixpoint combinators.

Environments Most languages provide variables for keeping track of arbitrary state or for naming complex program terms. These variables also need to be modeled by the abstract interpreter. To do so, a mapping from variables to their memory locations (address) needs to be provided. This can also be seen in the toy example from Section 2. This mapping can be summarized through three functions: `lookupEnv`, `withEnv` and `getEnv`, shown below.

```
class (Monad m) => MonadEnvironment adr m where
  lookupEnv  :: String -> m adr
  withEnv   :: [(String, adr)] -> m a -> m a
  getEnv    :: m [(String, adr)]
```

The `lookupEnv` function returns a computation that results in the address of the given variable name. The `withEnv` function runs the given computation `m a` (e.g., a call to `eval`) in the environment described by the list of mappings in the first argument. Finally, the `getEnv` function captures the current environment, which is needed for implementing language features such as closures.

Stores A store provides a mapping from addresses to values and models the program’s memory. In a concrete semantics, functions `lookupSto` and `updateSto` (depicted below) suffice to express the interactions with the store. However, abstract interpretation necessitates an additional function which we call `extendSto`.

```
class (Monad m) => MonadStore adr v m where
  lookupSto  :: adr -> m v
  extendSto  :: adr -> v -> m ()
  updateSto  :: adr -> v -> m ()
```

The difference between `updateSto` and `extendSto` is that `updateSto` changes the value of an *existing* address, while `extendSto` adds a new address to the store and associates a value with it. This distinction is required for enabling support for *strong updates* since they need to differentiate between an existing binding being updated or a new binding being introduced in the store. Note that depending on the analysis, both `extendSto` and `updateSto` could join the new value with the previously stored value. They merely signal the intention of the semantics to introduce a new address (i.e., `extendSto`) or to update an existing one (i.e., `updateSto`).

Nondeterminism In contrast to a single execution path in a concrete interpretation of the program, an abstract interpretation might explore multiple execution paths. This is because the interpreter has to reason with approximate values. For instance, evaluating the condition of an `if` expression can yield `true`, `false` or both meaning that both of its branches have to be approximated.

To this end, MONARCH exposes the `MonadJoin` type class, which executes two monadic computations and combines their results together. Its definition, depicted below, consists of two functions: `mjoin` and `mbottom`. This type class is similar to `MonadPlus` where `mjoin` corresponds to `mplus` and `mbottom` corresponds to `mzero`. In contrast to `MonadPlus`, however, `MonadJoin` adds lattice constraints on the output of its computations `m a`. This is because implementations of `MonadJoin` can either choose to combine all results into a set, akin to the non-determinism monad, or to join results of two computations together using `join`. Therefore, a `Joinable` constraint is required for `mjoin`.

```
class (Monad m) => MonadJoin m where
  mjoin    :: Joinable a      => m a -> m a -> m a
  mbottom  :: BottomLattice a => m a
```

This `MonadJoin` interface can be used to implement over-approximating conditional control flow. Below, we present the `cond` function for expressing this conditional control flow. It uses the `MonadJoin` interface to combine computations of the consequent and alternative branches of `if` expressions. It uses the `mbottom` function to represent *empty computations*. This is the case when the condition for one of the branches becomes infeasible.

```
cond :: (BoolDomain b, MonadJoin m, Joinable v, BottomLattice v)
      => m b -> m v -> m v -> m v
cond cnd csq alt = mjoin t f
  where t = cnd >>= (\b -> if isTrue b then csq else mbottom)
        f = cnd >>= (\b -> if isFalse b then alt else mbottom)
```

As an example, we implement semantics for evaluating `if` expressions. This implementation is straightforward because `if` expressions almost directly translate to `cond` actions. The semantics essentially states that the condition (`cnd`) is evaluated first, and based on its result the consequent (`csq`), alternative (`alt`), or both are evaluated. This implementation is nearly identical to a concrete semantics, but our framework enables the use of `cond` for expressing non-deterministic `if` expressions (when the condition could be both true or false).

```
eval (If cnd csq alt) = cond (eval cnd) (eval csq) (eval alt)
```

Escaping Control Flow We define *escaping control flow* as control flow that interrupts normal sequential program execution. Examples of escaping control flow are not only exceptions or program errors, but also more complex language features such as *early function returns*, *loop breaking*, and so on. MONARCH abstracts from these language features through a single monadic type class called *MonadEscape* which is depicted in the code listing below.

```
class MonadEscape m where
  type Esc m :: Type
  throw  :: JoinLattice a => Esc m -> m a
  catch  :: JoinLattice a => m a -> (Esc m -> m a) -> m a
```

`MonadEscape` resembles the `MonadError` type class from the well-known `mtl` library¹. Both type classes contain methods for signalling an escaping or error condition through `escape` and `throwError` respectively. Moreover, both feature a method for catching potential errors that occur in a given computation through `catch` (`catchError` resp.). They differ, however, in their constraints.

¹ From: <https://hackage.haskell.org/package/mtl>

Whereas `MonadError` does not constrain the output of its monadic computations, `MonadEscape` constrains the output to a `JoinLattice` which is an alias for a combination of `Joinable`, `PartialOrder` and `BottomLattice`. This is because an implementation of `catch` needs to potentially account for both an escaping program path and a normal program path by joining the results of its handler with the results from the normal program path.

Error `Esc m` is typically set to an abstract representation of potential program errors. For example, to represent errors of type `Error` abstractly, a `Set Error` can be used. In general, developers can use a `Domain` constraint to provide this mapping from a concrete error to its abstract representation:

```
type EscapeConstraints m = (Domain (Esc m) Error, MonadEscape m)
```

Fixpoint Infrastructure The key idea of abstract definitional interpreters is that they express abstract program semantics through a recursive evaluation function. Unfortunately, naively applying this evaluation function might lead to undesired non-termination issues. Instead, abstract definitional interpreter frameworks typically require some additional bookkeeping. For instance, to make sure that the analysis terminates, an in-out caching [5] mechanism can be employed through open recursion. This in-out caching mechanism keeps track of earlier inputs to the evaluation function and their outputs, and returns the associated output when applied more than once to the same input.

We support these mechanisms through two type classes. The first, `MonadFix`, provides a fixpoint function that transforms Kleisli arrows [16] (monadic computations that still require an input) into cached Kleisli arrows.

```
type Kleisli m b c = b -> m c -- Kleisli arrow
class MonadFix b c m where
  fix :: (Kleisli m b c -> Kleisli m b c) -> Kleisli m b c
```

The first argument of `fix` corresponds to a version of an evaluation function using open recursion. This can be seen by replacing `b` with `Exp` and `c` with `v`. This results in the following function signature:

```
eval :: (Exp -> m v) -> Exp -> m v
eval recur e = _
-- or equivalently with Kleisli arrows
eval :: Kleisli Exp v -> Kleisli Exp v
```

Then, when a cached version of `eval` is required, the `recur` function can be used. This enables polymorphism over the caching and fixpoint mechanism.

The second type class to support in-out caching mechanisms is `MonadCache`. The methods in this type class compute the complete inputs and outputs to the evaluation function. These inputs and outputs also include values encapsulated by the monad in addition to the value specified by the return type of the evaluation function. The `MonadCache` type class, along with its associated methods and types, is depicted in the code listing below.

```

class MonadCache m where
  type Key m k :: Type
  type Val m v :: Type
  type Base m  :: Type -> Type
  key  :: k -> m (Key m k)
  val  :: Val m v -> m v
  run  :: (k -> m v) -> Key m k -> Base m (Val m v)

```

Associated type families `Key` and `Val` compute the *type* of input and output respectively. Note that `Key` is also indexed by a type `k` which represents the input to the evaluation function—or in general to the Kleisli arrow—being cached. `Val` is indexed similarly by the output type of the cached Kleisli arrow. Functions `key` and `val` allow the current input to be extracted and the given output to be restored respectively. The `run` function takes the cached Kleisli arrow and *runs* it by supplying it with the expected input. `Base` computes the type of the first layer in a monadic stack that does not require caching, which can be the identity monad or some other monad. This enables expressing effects that are global to the fixpoint iteration, such as a global store [9].

To illustrate this, we depict an instance of `MonadCache` for the `StateT` monad transformer below. For its `Key` it adds the input state `s` to input `k`, and does so similarly for its output state for `Val`. The implementation of `key` and `val` are straightforward since they simply extract the input state and restore the output state respectively. MONARCH provides instances of `MonadCache` for all major `mt1` monad transformers in a similar manner.

```

instance MonadCache m => MonadCache (StateT s m) where
  type Key (StateT s m) k = Key m (k, s)
  type Val (StateT s m) v = Val m (v, s)
  type Base (StateT s m)  = Base m
  key k = StateT $ \s -> (,s) <$> key (k, s)
  val  = StateT . const . val
  run f = run (\(k,s) -> runStateT (f k) s)

```

4.3 Analysis

A semantics can be instantiated into an analysis by instantiating all type parameters used in the evaluation function. These type parameters correspond to different static analysis concerns. In MONARCH, these type parameters correspond to abstract domains and monad transformer stacks.

We discuss several areas of interest. The first is the instantiation of the domain. Next, we discuss *effect layering*, which enables combining effects into a single monad that supports all expected effects. Third, we explore some instantiations of monad transformers. Finally, we discuss integrating fixpoint algorithms.

Domain Instantiation In addition to descriptions of abstract domains through a set of interfaces represented as type classes, MONARCH also provides a number

of type class instances for these domains. For example, the `CP a` type provides a constant propagation lattice for any type `a`, while `CP Integer` implements a number domain. Similarly, MONARCH provides instances for other frequently used domains such as strings, lists, dictionaries, booleans, and so on. Moreover, MONARCH enables composing these instantiations into powerful domains using domain combinators such as the sparse labeled product described in Section 4.1.

Effect Layering Monads do not compose [10], meaning that their operations (i.e., `unit` and `bind`) cannot be readably expressed as the composition of their individual operations. Instead, *monad transformers* [17] have been proposed, which are best described as monads with “a hole”. Consider the *state monad transformer*, depicted in the code listing below, as an example.

```
newtype StateT s m a = StateT (s -> m (a, s))
```

The monad is parametrized by the type of state `s`, its result type `a`, but also by a monad `m`. It essentially allows for additional effects to be added around the resulting value and the output state. The idea is that computational effects can be composed using a set of smaller monads that are stacked on top of each-other by **lifting** operations from monads lower in the stack. Typically, monad transformers are presented as a library of transformers and their type classes. For example, `StateT` implements the type class `MonadState`. The idea is that all other transformers also implement this type class, and delegate state operations down to a state monad lower in the stack. Therefore, the entire stack now implements the required type class and becomes a suitable instantiation.

The example shown below declares the type of a stack that introduces effects for tracking an environment and store. This stack can subsequently be used for running semantics that require environment and store effects.

```
type EvalM =
  StoreT Address Value (EnvT String Address Identity)
```

The main problem with monad transformers is that for each new type class and transformer combination a corresponding instance needs to be implemented, which leads to poor scalability. This is because not all operations can be expressed simply as a **lifting** of lower operations.

Therefore, monad transformers do not seem a suitable candidate for instantiating our semantics. Instead, inspired by the Haskell `layers` library², we propose a class called `MonadLayer`. This type class consists of two functions: `upperM` and `lowerM`. The former fulfils the role of `lift`. The latter is somewhat more complex, and fulfils the role of “lowering” a monadic computation into the stack.

```
class MonadLayer l where
  upperM :: Monad m => m a -> l m a
  lowerM :: (forall b . m b -> m b) -> l m a -> l m a
```

² From: <https://hackage.haskell.org/package/layers>

`lowerM` is a function that accepts two arguments. The first is another function that executes a monadic action on the lower layer. The second argument is a monadic computation `l m a` where `l` is a monad transformer, `m` is the monad one level below the transformer, and `a` is the result from the monadic computation.

It is important to note that type variable `b` is universally quantified inside the function in the first argument. This is because implementations of `lowerM` need to “push” information about their effects into the lower layer so that they are not lost in composition. For example, to remove a layer from `StateT` it needs to be executed with some initial state `s`. The output of this execution is a *pair* that consists of a value `a` and a new state `s`. Therefore, to execute an action on the lower layer, this pair needs to be part of the monadic computation. To illustrate, we provide the instance of `MonadLayer` for `StateT` below.

```
instance MonadLayer (StateT s) where
  upperM = lift
  lowerM f (StateT run) = StateT (f . run)
```

Below, we demonstrate how a `MonadLayer` can be used to delegate monadic actions to a lower monad. The code listing depicts an instance of `MonadEnvironment` for any monad layer `l`. The `lookup` and `getEnv` function can be implemented using `upperM`, while `withEnv` needs to be implemented using `lowerM`. Essentially, to delegate `withEnv` computations, the second argument of `withEnv` needs to be *lowered* into the layer below, so that the lower `withEnv` can be executed.

```
instance (MonadLayer l, MonadEnvironment m)
  => MonadEnvironment (l m) where
  getEnv      = upperM getEnv
  lookup var  = upperM (lookup var)
  withEnv bds ma = lowerM (withEnv bds) ma
```

Effect Locality Monad transformers and layers do not necessarily commute. The combination of a `Maybe` with a `State` monad for instance yields very different results depending on the order in which they are combined. For instance, composing `Maybe` after `State` results in a computation that can fail but retains its state up to the failure, while reversing this order results in a computation that can fail but loses its final state. This becomes more apparent when unfolding the type signature of the different composition orderings:

```
newtype MaybeT m a = MaybeT (m (Maybe a))
MaybeT (StateT Identity) a = s -> (Maybe a, s)
StateT (MaybeT Identity) a = s -> Maybe (a, s)
```

Thus, the depth of the monad transformer stack provides a *locality* dimension. The lower a monad is in the stack, the more global its effects become. In the earlier example, `MaybeT` is located lower in the stack, thus its failures are more global than the state so that the state is discarded when failure occurs.

Monad Transformers For instantiating the monadic computation, the analysis developer is expected to construct a stack of monad transformers that provide implementations for each type class constraint of the evaluation function. To this end, MONARCH provides a library of monad transformers that have instances for these type classes. For example, to satisfy the `MonadStore` constraint, the framework provides a `StoreT` transformer that supports weak updates.

Interestingly, some monadic type classes have multiple implementations. One example is non-determinism, for which we describe two transformers that satisfy the `MonadJoin` constraint: `JoinT` and `NonDetT`. Recall that after one or more paths have been explored, their results need to be merged. The `JoinT` monad layer enables this by *joining* the results together so that only one path remains with a joined value (as is typically done when using traditional dataflow analyses). Alternatively, `NonDetT` can be used to accumulate each result into a list so that the semantics continues with the result of each path separately (as is typically done when using the AAM approach [25]).

```
newtype JoinT m a = JoinT (m a) deriving (Monad)
instance MonadJoin (JoinT m) where
  mzero = return bottom
  mjoin (JoinT ma) (JoinT mb) = join <$> ma <*> mb
runJoinT :: JoinT m a -> m a
runJoinT (JoinT m) = m
```

Essentially, `mjoin` runs both paths sequentially and then joins their results together. `NonDetT` works similarly but uses a list monad to capture all results.

```
newtype NondetT m a = NondetT (ListT m a) deriving (Monad)
```

The operations of `MonadJoin` can be trivially implemented as the empty list for `mzero` and as list concatenation for `mjoin`.

Constructing Cached Fixpoint Computations Finally, we briefly discuss instantiating an analysis with a fixpoint algorithm. To this end, we consider a simple evaluation function with the following constraints:

```
type EvalM v m = (MonadFix m Exp v, MonadJoin m, MonadStore m)
eval :: EvalM v m => Exp -> m v
```

The `Key` type (from `MonadCache`) gives rise to a *component*. A component typically represents a part of a program that is to be analyzed until its end and whose analysis does recurse infinitely. More concretely, it is the input to the `evaluation` function. Thus, most analyses can be structured as an *intra-component* analysis (i.e., `eval`) and an *inter-component* analysis. The inter-component analysis drives the intra-component analysis until it reaches a fixpoint.

The code listing below illustrates how to express this in our framework. We first define the monadic stack for the intra-analysis and separate out the remaining type class constraints. Then, we run the *inter-component* analysis. While doing so, we satisfy the remaining type class constraints by adding the appropriate monad transformer to the transformer stack.

```

type IntraT m = MonadStack '[ JoinT, CacheT ] m
type Cmp      = Key (IntraT Identity) Exp
type Res      = Val (IntraT Identity) Val
type InterM   = (MonadStore m, MapM Cmp Res)
intra :: InterM m => Cmp -> m ()
intra e = runFixT @(IntraT (AnalysisT Cmp m) (eval e)
                  & runAnalysisT e
-- computing a fixpoint over the intra analysis
inter = lfp intra
-- adding global effects and running the analysis
analyze :: Exp -> (Map Adr Val, Map Cmp Res)
analyze e = inter e & runStoreT emptyStore
                  & runWithMapping @Cmp @Res & runIdentity

```

The result of this analysis is an abstraction of the heap (through `MonadStore` and `runStoreT`) and a mapping from each component in the program to its evaluated value (i.e., `Res`). MONARCH allows developers to instantiate the analysis with different fixpoint strategies. Some fixpoint strategies require additional bookkeeping that needs to be added to the set of global effects at the bottom of the monad transformer stack. Such bookkeeping can be added to the `runAnalysisT` monad transformer but is omitted for brevity. This design also allows reordering monad transformers to obtain different semantics [5], similar to Section 4.3. For example, moving `StoreT` to the intra-component analysis results in an analysis with *local stores* rather than a *global store* analysis [25].

Interestingly, this design also allows for effects that are neither global nor local, but sit somewhere in between. For instance, a *flow-sensitive* store is typically implemented by widening the store at the component level, keeping track of the store at the start (in-store) and at the end (out-store) of the analysis for that component. When components are function calls, on each function call the “in-store” is joined with the contents of the store at the call-site of the function. Then, the function body gets analyzed, after which the “out-store” is changed to the contents of the store at the end of the function body. This “out-store” is finally used to continue evaluation after calling the analysed function at each call-site.

This design can be implemented by moving the `StoreT` layer between the caching transformer and join transformers. The change causes the store to be no longer cached, as it is below the caching layer. Keeping the store above the join transformer ensures that stores are joined rather than threaded along multiple nondeterministic program paths. Additional manual plumbing is required to update the store contents as described above. This is depicted below.

```

1 type IntraT = MonadStack '[ CacheT, StoreT, JoinT ]
2 intra :: forall obj m . AnalysisM m obj => Cmp -> m ()
3 intra e = runIntraAnalysisT e m
4     where m = do s <- fromJust <$> get (StoreIn e)
5           r <- cache e (runCallT (uncurry callFix) . eval)
6           & runStoreT s

```

```

7           & runJoinT
8           put (StoreOut e) s'
9           return r
10        callFix :: Exp -> IntraT m Res
11        callFix bdy = do e' <- key bdy
12                   spawn e'
13                   changed <- joinWith (StoreIn e') =<< currentStore
14                   if changed then mbottom
15                   else do rv <- cached e'
16                          rs <- get (StoreOut e')
17                          v <- maybe mbottom return rv
18                          s <- maybe mbottom return rs
19                   putStore s
20                   return v

```

In this version, the transformers in the stack are rearranged to place the store transformer in between the caching and the joining layer. As mentioned before, this causes the store to be omitted from the keys and values of the cache, therefore becoming part of the global analysis state instead of the components themselves. The open recursion of the evaluation function through `runCallT` ensures that function calls are treated differently in order to change the contents of the store. The function calling semantics is depicted in the `callFix` function which takes the body of the function, checks whether the function has already been analyzed, and if so reads its result from the cache. Then, the function’s “in-store” is updated to include the contents of the current store (i.e., the one at the call-site of the function). Then, after the function has been analyzed, the store at the call-site is updated through `putStore` (line 19).

Since the join and store transformers are no longer part of the cache, the `cache` (line 5) and `runIntraAnalysisT` (line 3) functions will no longer automatically run these layers. Hence, `runJoinT` and `runStoreT` need to be manually called after the cached layers have been executed through `cache`.

5 Instantiating the Framework

To illustrate the framework’s use in practice, we discuss two case studies in which we instantiate two different types of analyses for two programming languages. The first analysis is for the Scheme language. The second analysis is for a subset of the Python language. Both analyses compute an abstraction of the program’s memory, as well as its read, write and call effects. The languages differ significantly in their syntax, semantics, and types of run-time values. Table 3 summarizes the similarities and differences of the corresponding analysis implementations and connects them to the architecture of our framework (cf. Fig. 1). For each implementation, we present the abstract domain, relevant semantics, and show how the analysis is instantiated using layers of monad transformers. **The source code for our case studies is available in the artifact.**

Table 3. Overview of ADI configuration per language

	Python	Scheme
Domain		
<i>Primitive</i>	Integer, Real, String	Integer, Real, Symbol
<i>Combinator</i>	sparse product in primitive fields, Dictionary, Tuple	coproduct abstracted as sparse product, Vector, Pair
Semantics		
<i>Store</i>	environment frames, objects	vector, pairs, strings, variables
<i>Escape</i>	return, break, errors	errors
<i>Join</i>	yes	yes

5.1 Domain

We use the labelled sparse product lattice proposed in Section 4.1 to represent Scheme’s abstract domain. An excerpt of its implementation is depicted below. Labels and values are separated by a `::->`. Its definition is parametrized by a configuration `m` (represented by a type-level association list) which configures the domain with specific subdomains for each type of value.

```

type Values m = '[ IntKey  ::-> Assoc IntConf m,
                  BoolKey ::-> Assoc BoolConf m,
                  PaiKey  ::-> Set (Assoc PaiConf m),
                  CloKey  ::-> Set (Assoc ExpConf m, Assoc EnvConf m),
                  PrimKey ::-> Set String, ...]
type SchemeValue m = LabeledSparseProduct (Values m)

```

For instance, abstract closures are allocated at `CloKey` and are represented as sets of pairs consisting of an expression and an environment. Another example is the abstraction of Scheme numbers, which are allocated at the `IntKey` and map to a primitive integer sub-domain as configured by the `IntConf` key. The entire implementation of this Scheme domain is done in roughly 500 lines of code.

The implementation of the Python analysis takes a different approach. In Python all values are modelled as objects that have their own methods and associated class. Thus, a sparse product representation is no longer suitable to represent those values. Instead, we use an abstraction of dictionary values (similar to [7]) which stores method identifiers and field names as keys, and their abstracted value as values. An instantiation of this domain is depicted below. Again, parameter `m` is a *configuration* which configures the Python domain with specific primitive subdomains for each type of value.

```

data PyObj m = PyObj
  { dct :: CPDictionary String (Assoc ValueConf m),
    prm :: LabeledSparseProduct (PyPrm m) }

```

Our `LabeledSparseProduct` appears again here, to implement so-called *primitive fields*. These primitive fields are used to represent primitive values such as

integers or *lists* which cannot be represented as an object and class, but can still be extended from in a class hierarchy. Thus we represent them as fields in every object that might be set if the object represents a primitive value such as integers. The definition of `PyPrm` is similar to `Values` and is left out for brevity. The implementation of the domain for Python objects is done in approximately 250 lines of code in total.

Conclusion. Real-world abstract domains often consist of many subdomains. We provide modular, reusable primitive building blocks to represent these different subdomains and combinators to combine them. This saves analysis developers from performing the tedious and error-prone tasks of implementing abstract domains themselves.

5.2 Semantics

Depicted below are the type class constraints on the evaluation function for implementing the Python semantics (`PyM`, left) and the Scheme semantics (`SchemeM`, right) respectively. Both sets of constraints are similar: they both have a type class that describes their abstract domain (`PyObj` for Python and `SchemeValue` for Scheme), they both need `MonadJoin` and `MonadEscape`, and they need a representation of environment and store. Furthermore, we also use a monad called `AllocM` which enables the allocation of memory addresses.

```

type PyM m obj = (
    PyObj' obj, MonadJoin m,
    MonadEscape m,
    MonadEnvironment m ObjAdr PyEnv,
    MonadStore m ObjAdr obj,

    AllocM m PyLoc ObjAdr,
    AllocM m FrmLoc ObjAdr
)

type SchemeM m v = (
    SchemeValue v, MonadJoin m,
    MonadEscape m,
    MonadEnvironment m (Adr v) (Env v),
    MonadStore m (PAdr v) (PaiDom v),
    MonadStore m (Adr v) (VarDom v),
    MonadStore m (VAdr v) (VecDom v),
    MonadStore m (SAdr v) (StrDom v),

    AllocM m Ide (Adr v),
    AllocM m Exp (PAdr v),
    AllocM m Exp (VAdr v),
    AllocM m Exp (SAdr v)
)

```

The language semantics differ more. We focus on the representation of primitive operations (e.g., arithmetic operations) since they show the interaction of language semantics with the abstract domains.

In the interpreter for Scheme, primitives are implemented as ordinary Haskell functions that accept a list of argument values as input. The interpreter applies an operation from the abstract domain and returns the result. In the abstract domain, primitives are then represented as sets of strings which are used as keys to find the associated Haskell function in a table of primitives. Below, we depict a number of simple arithmetic functions from the Scheme analysis. The full implementation of all Scheme primitives is approximately 150 lines of code.

```

fix2 :: String
      -> (forall m . PrimM m v => v -> v -> m v) -> Prim v

```


Another example of how our analysis framework facilitates implementing complex language features is depicted below. It shows the implementation of attribute lookup for Python objects. First, it checks whether the object contains the attribute, and looks it up in the fields of its class otherwise. Our framework enables expressing abstract semantics close to the concrete, as can be seen in `hasAttr`, `getAttr`, etc. These return abstract values which lead to approximation in the semantics, but the way they are expressed is close to a concrete semantics.

```
lookupAttr loc attr =
  pyDeref $ \adr obj ->
    cond (return $ hasAttr attr obj)
        (return $ getAttr attr obj)
        -- if not found locally => look in the class
        (do cls <- atAttr "__class__" obj
            lookupAttrInClass loc attr adr cls)
```

Conclusion. Our type classes form a *framework* for defining abstract language semantics. Our case studies have shown that the framework is applicable to two very different semantics which demonstrates its power and expressiveness.

5.3 Analysis

Below we present the analysis instantiations for Scheme and Python. Both are implemented as a pair of functions `intra` and `analyze`. The former defines the set of layers which are cached by the caching mechanism and whose input and output will be used by the fixpoint iterator. The latter defines all the layers *outside* of the cache. This boundary is set by the `runBaseT` layer at the bottom of the stack defined in `intra`. Both instantiations place the store monad transformer below the cache layer. This is because the analyses use a *global store* which requires the store to remain outside of the cached state.

```
-- Scheme
intra :: (SchemeDomain v, AnalysisM m v) => Exp -> m v
intra e = eval e & runAlloc (PaiAdr @ctx)
          & runAlloc (VecAdr @ctx)
          & runAlloc (StrAdr @ctx)
          & runAlloc (EnvAdr @ctx)
          & runCtx ctx & runJoinT & runBaseT
analyze e = lfp (intra e)
            & runStoreT initialSto
            & runWithStore @(Map StrAdr (StrDom v))
            & runWithStore @(Map VecAdr (VecDom v))
            & runWithStore @(Map PaiAdr (PaiDom v))
            & runIdentity
-- Python
```

```

intra :: AnalysisM m obj => PyBdy -> m ()
intra bdy = evalBdy bdy & runMayEscape
              & runEnvT initialEnv
              & runAlloc (const . allocPtr)
              & runAlloc (const . allocFrm)
              & runCtxT () & runJoinT & runBaseT
analyze prg = lfp (intra prg)
              & runWithStore @(Map ObjAdr obj) @ObjAdr
              & runIdentity

```

The instantiations for Python and Scheme follow the same structure: both use the same fixpoint iterator function called `lfp`, need an environment and context, and use the `JoinT` layer for merging results from different paths in the analyzed program. Both instantiations are implemented in approximately 150 lines of code each. The major difference between the Scheme and Python analyses is in their use of *stores*. To run a Python analysis only a single store is required while a Scheme analysis requires four. This is because the Scheme analysis uses four memory segments: variable segment, vector segment, string segment and pair segment (in contrast, in Python everything is an object, and hence only an object segment is required). We chose this design because our memory allocation strategy never allocates values from different segments on the same address. Physically segmenting the stores allows enforcing this design at the type level. This usage also highlights how the `MonadStore` interface provides the correct `lookup` function by traversing the monad transformer stack, and uses the store that is applicable for the type of address used.

Conclusion. We instantiated an analysis for Python and for Scheme subsets. These instantiations are similar, demonstrating that our monad transformer stack facilitates expressing effects for different semantics.

6 Related Work

In this section, we present other static analysis tools based on (definitional) abstract interpreters and discuss how our framework differs.

Monadic Abstract Interpreters. Sergey et al. [23] propose monadic abstractions for abstracting a CESK machine. To this end, they start from the AAM approach and derive monadic components. Although their work is the first to introduce monads in the context of abstracting abstract machines, it does not deal with the definitional aspect of the interpreter, nor does it integrate monad transformer stacks for composing different aspects of the analysis together.

Definitional Interpreters through Arrow Combinators. Keidel et al. [15] propose a library of *arrow combinators* [8] for expressing sound and composable

abstract program semantics. Similar to their work, we also present a collection of type classes to describe the semantics of the analyzed language. However, we encode the expected abstractions as *monads* instead of *arrows*. We argue that arrow transformers are quite new and less supported by the Haskell ecosystem. Moreover, their abstraction of `ArrowPlus` (`MonadJoin` in our implementation) lacks a `Joinable` and `BottomLattice` constraint. This makes the implementation of this type class as a join of the values across program branches more difficult, as a layer within the monad transformer stack cannot assume the lattice structure of the values captured within the monadic computations.

In a follow-up work, Keidel et al. [14] propose fixpoint combinators for expressing the fixpoint algorithms for abstract definitional interpreters. Our work follows similar strategies but is specifically tailored to monadic abstract interpreters, rather than arrow transformers. The `MonadCache` type class essentially implements a caching mechanism specifically for Kleisli arrows. However, next to caching, the `MonadCache` type class also serves a more practical purpose: it allows to run a (part of) a monad transformer stack.

Abstract Definitional Interpreters without Monads. Brandl et al. [1] propose a Scala framework for implementing abstract definitional interpreters. In contrast to our work, their framework proposes to eliminate the monad transformer stack by representing computational effects through an *imperative* effect stack. This reduces compilation times as the compiler no longer needs to aggressively inline each monadic operation across the transformer stack, but requires careful implementations of effect handlers to store and restore global state when appropriate. For instance, combining a store effect handler with a non-determinism handler requires that the store handler restores its state imperatively after the first branch is evaluated and before the execution moves on to the next branch. Monad transformers do not have an imperative state and do not require separate restore logic.

Other Abstract Interpretation Frameworks. MAF [24] is a framework for implementing modular analyses for higher-order programming languages. This work focuses on the *ModX* [20] approach to modular analyses and does not present abstractions for expressing abstract program semantics. Moreover, support for combining abstract domains is limited.

The MOPSA analyser [11] is a modular OCaml platform used to build sound semantic static analysers based on abstract interpretation. In contrast to our work, MOPSA does not follow the abstract definitional interpreter design.

LISA [6] is a library for building analyses based on abstract interpretation. LISA uses a control flow graph representation, allowing analysis implementers to reuse the library’s existing analyses by only writing a parser and control flow graph builder for the language they want to analyse. In MONARCH this can be achieved by implementing an evaluation function for the language of interest. Other static analysis tools such as Soot [12], Phasar [22], ... also rely on control flow graphs but are tailored to specific programming languages such Java or

LLVM IR. Although MONARCH currently focusses on Scheme and Python, it is designed in a language-agnostic manner.

CIAOPP [2] is an abstract interpretation-based preprocessor and analyser for Ciao Prolog programs. Similar to our work, the tool provides an interface to instantiate client analyses and the ability to configure the analysis in several ways. However, the tool is specialised to the Ciao Prolog programming language, while MONARCH provides many language-agnostic building blocks to allow analysis developers to create new analyses for other languages (as demonstrated by our case studies in Section 5).

Staged Abstract Interpreters. [26] Shows how an abstract interpreter can be specialised to a program to optimise it. Although this paper does not present any novel ideas specifically for abstract definitional interpreters, their optimisation steps could be of interest to integrate into the MONARCH framework.

Overall, our work follows a long tradition in abstract interpretation [4] and (abstract) definitional interpreters [14, 21, 23]. MONARCH combines this tradition into a single framework and proposes abstractions in the form of `MonadJoin`, `MonadEscape` and `MonadCache`. Moreover, our framework provides a strong foundation for building abstract domains through its rich set of primitive domains and its powerful domain combinators, such as sparse labeled products.

7 Conclusion

We have presented our framework called MONARCH for implementing abstract definitional interpreters in Haskell. Our design consists of three parts: abstract domains, abstract program semantics, and analysis instantiation. We have presented the key features of each part and how they integrate with one another. For its domain component, we found that a rich library of primitive domains and combinations thereof are the most suitable for representing abstract domains. We also found that type-level data structures are paramount for implementing complex abstract domains succinctly and efficiently by providing abstractions that reason about all subdomains automatically. For defining programming language semantics, we found that, similar to other abstract definitional interpreters, implementing language semantics is best performed in a polymorphic context expressed using a set of monadic type class constraints. Moreover, although nearly identical to their standard `mtl` counterparts, monadic type classes for abstract definitional interpreters require additional type class constraints relating to lattices and strong updates. We have also shown that caching infrastructure can be expressed in a generic manner through a `MonadCache` type class constraint without the need for less common representations of computations such as arrows. Finally, we discussed two different instantiations of analyses for Scheme and Python and showed how our design facilitated implementing these analyses.

Acknowledgments. This work was partially supported by an Innoviris grant for the ECOPIPE project (Joint R&D 2022, grant number 2022-PS-JRDIC-8), by the Research Foundation Flanders (FWO) (grant number 1187122N), and by the Cybersecurity Research Program Flanders.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Brandl, K., Erdweg, S., Keidel, S., Hansen, N.: Modular abstract definitional interpreters for webassembly. In: 37th European Conference on Object-Oriented Programming. pp. 5:1–5:28 (2023). <https://doi.org/10.4230/LIPICS.EC00P.2023.5>
2. Bueno, F., Hermenegildo, M., López, P., Morales, J.F., Puebla, G.: The ciaopp program processor (1996)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Proceedings of the 4th ACM Symposium on Principles of Programming Languages. pp. 238–252 (1977)
4. Cousot, P., Cousot, R.: Abstract interpretation: past, present and future. In: Proceedings of the Joint Meeting of the 23rd Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 2:1–2:10 (2014)
5. Darais, D., Labich, N., Nguyen, P.C., Van Horn, D.: Abstracting definitional interpreters (functional pearl). Proceedings of the ACM on Programming Languages 1(ICFP), 1–25 (2017)
6. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 1–6 (2021). <https://doi.org/10.1145/3460946.3464316>
7. Fulara, J.: Generic abstraction of dictionaries and arrays. In: Proceedings of the 4th International Workshop on Numerical and Symbolic Abstract Domains. Electronic Notes in Theoretical Computer Science, vol. 287, pp. 53–64 (2012). <https://doi.org/10.1016/J.ENTCS.2012.09.006>
8. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37(1-3), 67–111 (2000). [https://doi.org/10.1016/S0167-6423\(99\)00023-4](https://doi.org/10.1016/S0167-6423(99)00023-4)
9. Johnson, J.I., Labich, N., Might, M., Van Horn, D.: Optimizing abstract abstract machines. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 443–454 (2013)
10. Jones, M.P., Duponcheel, L.: Composing monads. Tech. rep., Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale (1993)
11. 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. pp. 1–18 (2020)
12. Karakaya, K., Schott, S., Klauke, J., Bodden, E., Schmidt, M., Luo, L., He, D.: Sootup: A redesign of the soot static analysis framework. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 229–247. Springer (2024)

13. Keidel, S., Erdweg, S.: Sound and reusable components for abstract interpretation. *Proceedings of the ACM on Programming Languages* **3**(OOPSLA), 176:1–176:28 (2019). <https://doi.org/10.1145/3360602>
14. Keidel, S., Erdweg, S., Hombücher, T.: Combinator-based fixpoint algorithms for big-step abstract interpreters. *Proceedings of the ACM on Programming Languages* **7**(ICFP), 955–981 (2023). <https://doi.org/10.1145/3607863>
15. Keidel, S., Poulsen, C.B., Erdweg, S.: Compositional soundness proofs of abstract interpreters. *Proceedings of the ACM on Programming Languages* **2**(ICFP), 72:1–72:26 (2018). <https://doi.org/10.1145/3236767>
16. Kleisli, H.: Every standard construction is induced by a pair of adjoint functors. *Proceedings of the American Mathematical Society* **16**(3), 544–546 (1965)
17. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. p. 333–343 (1995). <https://doi.org/10.1145/199448.199528>
18. Moggi, E.: *An abstract view of programming languages*. University of Edinburgh. Department of Computer Science. Laboratory for Foundations of Computer Science (1990)
19. Moggi, E.: Notions of computation and monads. *Information and Computation* **93**(1), 55–92 (1991). [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4)
20. Nicolay, J., Stiévenart, Q., De Meuter, W., De Roover, C.: Effect-driven flow analysis. In: *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation*. *Lecture Notes in Computer Science*, vol. 11388, pp. 247–274 (2019)
21. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: *Proceedings of the ACM Annual Conference*. vol. 2, pp. 717–740 (1972). <https://doi.org/10.1145/800194.805852>
22. Schubert, P.D., Hermann, B., Bodden, E.: Phasar: An inter-procedural static analysis framework for c/c++. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 393–410. Springer (2019)
23. Sergey, I., Devriese, D., Might, M., Midtgaard, J., Darais, D., Clarke, D., Piessens, F.: Monadic abstract interpreters. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 399–410 (2013). <https://doi.org/10.1145/2491956.2491979>
24. Van Es, N., Van der Plas, J., Stiévenart, Q., De Roover, C.: MAF: A framework for modular static analysis of higher-order languages. In: *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*. pp. 37–42 (2020)
25. Van Horn, D., Might, M.: Abstracting abstract machines. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. pp. 51–62 (2010)
26. Wei, G., Chen, Y., Rompf, T.: Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proceedings of the ACM on Programming Languages* **3**, 1–32 (10 2019). <https://doi.org/10.1145/3360552>