MAF: A Framework for Modular Static Analysis of Higher-Order Languages

Noah Van Es, Jens Van der Plas, Quentin Stiévenart, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{noah.van.es,jens.van.der.plas,quentin.stievenart,coen.de.roover}@vub.be

***Abstract*—A modular static analysis decomposes a program's analysis into analyses of its parts, or *components*. An *inter-component* analysis instructs an *intra-component* analysis to analyse each component independently of the others. Additional analyses are scheduled for newly discovered components, and for dependent components that need to account for newly discovered component information. Modular static analyses are scalable, can be tuned to a high precision, and support the analysis of programs that are highly dynamic, featuring e.g., higher-order functions or dynamically allocated processes.**

In this paper, we present the engineering aspects of MAF, a static analysis framework for implementing modular analyses for higher-order languages. For any such modular analysis, the framework provides a reusable *inter-component* analysis and it suffices to implement its *intra-component* analysis. The *intra-component* analysis can be composed from several interdependent and reusable Scala traits. This design facilitates changing the analysed language, as well as the analysis precision with minimal effort. We illustrate the use of MAF through its instantiation for several different analyses of Scheme programs.

***Index Terms*—static program analysis, modular analysis**

I. Introduction

Since the introduction of *modular static program analysis* [1], several program analyses have featured a modular design [2]–[4]. Such analyses divide the analysis of a program into the analysis of the program's parts, which are called *components*. Examples of such components include compilation units, function calls, and processes. Each component is analysed in isolation from every other. However, components are not always completely independent, and information derived during the analysis of one component may need to be taken into account in the analysis of other components. Therefore, a component may need to be reanalysed multiple times. In recent approaches [2], [5], this is made explicit in the design through *dependencies*: a component may *depend* on parts of the analysis state, and is reanalysed when that state is updated during the analysis of other components.

Modular analyses have been shown to scale well [2], [3], [6]. Each component is analysed in isolation, and each component is a fraction of the size of the program to analyse. As a result, even for an expensive analysis, the complexity of the analysis is bounded by the size of the components. As long as the number of components itself does not suffer from an explosion, the analysis remains scalable.

Another advantage of modular analyses is their ability to support programs where the call graph is not known statically and is generated on the fly during the analysis. This is of particular importance when designing analyses for dynamic languages that support higher-order functions. While there are industry-ready analysis frameworks for languages such as C [7], [8], analysis frameworks that target dynamic languages with higher-order functions remain research-oriented [9]–[11], and do not follow a modular approach to static analysis.

We present an open-source framework for modular analyses[1], called Modular Analysis Framework, or MAF in short. For any modular analysis, the framework provides a generic *inter-component* analysis. It suffices to implement the *intra-component* analysis, which can be composed from reusable Scala traits provided by the framework. These traits structure and provide alternative implementations for analysis concerns such as semantics, abstract domain, or context sensitivity. Developers can therefore tune the precision of an analysis by mixing in alternative traits into its implementation.

This paper makes the following contributions:

- We present the design of MAF, a framework for implementing modular analyses of programs with highly dynamic features such as higher-order functions or dynamically allocated processes.
- We illustrate the use of MAF through its instantiation for various analyses for Scheme programs.
- We reflect on our experiences gained by building MAF, which we deem sufficiently general to provide insights for the design of other static analysis frameworks.

II. Background: Modular Analysis

Before describing the architecture of MAF, we explain and illustrate the concept of modular static analysis. A modular analysis divides a program into components and analyses these in isolation. Components can be function calls, processes or compilation units, for example. The analysis of a single component is referred to as the *intra-component* analysis. Although components are analysed separately, they may not be independent of one another. For example, threads can spawn one another, or read and write to shared mutable state.

To this end, a modular analysis tracks *dependencies* for components: whenever the analysis of a component reads some part of the (shared) analysis state, this component has a dependency on that part of the state; whenever the analysis updates some part of the analysis state, all components with a dependency on that part of the state need to be reanalysed,

[1] Available here: https://github.com/softwarelanguageslab/maf

2470-6892/20/$31.00 ©2020 IEEE

and we say this dependency is *triggered*. This is taken care of by the *inter-component* analysis, which orchestrates the analysis of components by keeping track of all components and dependencies, and repeatedly scheduling components that need to be (re)analysed using the intra-component analysis (i.e., newly discovered components, and components that have a dependency on an updated part of the analysis state).

In what follows, we illustrate a function-modular analysis of a Scheme program, where components align with function calls, i.e., calls to user-defined functions are analysed in isolation [5]. Consider the following program, where a function `map` is defined at line 1, and called at line 7.

```
1  (define (map f l)
2    (if (null? l)
3        '()
4        (cons (f (car l)) (map f (cdr l)))))
5  (define (main)
6    (define (inc n) (+ n 1))
7    (map inc '(1 2 3)))
```

We describe how this program is analysed by the function-modular analysis. In each step, we denote the worklist of the inter-component analysis as a set $\mathbb{W}$. This worklist contains all components that need to be analysed; initially, it only contains a single component that corresponds to the entry point of the program. In this case, this is the component $C_{\texttt{main}}$, corresponding to the `main` function in the program[2].

1) $\mathbb{W} = \{C_{\texttt{main}}\}$ – The first intra-component analysis analyses $C_{\texttt{main}}$. A call to `map` is found (line 7), for which a new component $C_{\texttt{map}}$ is created and added to $\mathbb{W}$. A dependency is registered from $C_{\texttt{main}}$ to the return value of $C_{\texttt{map}}$.
2) $\mathbb{W} = \{C_{\texttt{map}}\}$ – The analysis continues by analysing $C_{\texttt{map}}$, during which it encounters two function calls: a call to `map`, and a call to `f` (both on line 4)[3]. The analysis tracks the values to which the parameters of a function are bound, and in this case `f` is bound to the function `inc` created at line 6. A new component $C_{\texttt{inc}}$ for the call to `f` is created and added to the worklist. For the call to `map`, a component $C_{\texttt{map}}$ already exists, which is reused. As no return values are present for either $C_{\texttt{map}}$ or $C_{\texttt{inc}}$ yet, a placeholder value $\perp$ is returned for each call, and two dependencies are registered from $C_{\texttt{map}}$: one dependency to the return value of each component. When the analysis of $C_{\texttt{map}}$ finishes, the empty list is stored as the return value of $C_{\texttt{map}}$, since it returns the empty list in one of its branches (and we do not yet know the return value of the other branch). The dependency on this return value is triggered, and the dependent components, $C_{\texttt{map}}$ and $C_{\texttt{main}}$, are therefore added to the worklist.
3) $\mathbb{W} = \{C_{\texttt{inc}},\ C_{\texttt{map}},\ C_{\texttt{main}}\}$ – The first component in $\mathbb{W}$, $C_{\texttt{inc}}$, is analysed next[4]. After analysing $C_{\texttt{inc}}$, the updated return value of $C_{\texttt{inc}}$ causes $C_{\texttt{map}}$ to be added to the worklist (however, it is already present in $\mathbb{W}$).
4) $\mathbb{W} = \{C_{\texttt{map}},\ C_{\texttt{main}}\}$ – $C_{\texttt{map}}$ is now reanalysed, using the updated return value of $C_{\texttt{inc}}$. If the return value of $C_{\texttt{map}}$ is updated, $C_{\texttt{map}}$ and $C_{\texttt{main}}$ are again added to the worklist.
5) The analysis continues until the worklist $\mathbb{W}$ is empty. When this is the case, the analysis state has converged.

Note that the analysis must be constructed in such a way that the analysis state is guaranteed to converge eventually. To this end, static analyses typically use *abstract values* that approximate all possible values a variable can have during concrete executions of the program.

In this example, a component corresponds to a set of calls to a given function. For example $C_{\texttt{inc}}$ corresponds to all calls of `inc`. In order to improve the precision of a modular analysis, a *component context* can be attached to each component. It can be used to create different components for different calls to the same function, so that only calls to the same function and with the same calling context are represented by the same component. Component contexts can for example be an approximation of the call stack.

III. Design of MAF

MAF is a framework to implement modular static program analyses in Scala. In this framework, a modular analysis is a class that implements the `ModAnalysis` abstract class. This class provides the basic infrastructure for a modular analysis, and declares a number of abstract methods that need to be supplied by different analysis instantiations.

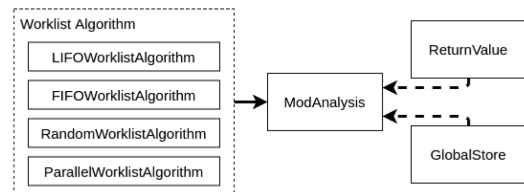

Fig. 1. Overview of the architecture of MAF. Nodes represent traits that can be mixed in to instantiate new analyses. Plain edges indicate a *necessary* requirement, and dotted edges indicate an *optional* extension.

Creating such an instance of a modular analysis requires defining the parts of the analysis that were left abstract in the `ModAnalysis` class. To this end, a number of other traits can be mixed in together with the `ModAnalysis` class to define (parts of) these necessary constructs, relieving the analysis developer of the burden of implementing them manually. This allows for reuse within the framework, as the analysis developers can mix in the traits they need, while still being able to add custom behaviour by defining new traits. A subset of the traits currently present in the framework are represented in Figure 1. As we illustrate later on, mixing in traits is the primary way to build analyses in MAF.

In this section, we describe the core traits to be used with `ModAnalysis`, which provide generic modular analysis infrastructure and behaviour. These are considered generic in the sense that they can be used to build modular analyses for various languages. In the next section, we build upon these traits to define modular analyses for Scheme.

[2]We assume that the `main` function is the entry point of the program.

[3]No components are created for calls to built-in functions (such as `cons`).

[4]Note that the order of the work list does not impact the termination, nor the result of the analysis [5].

A. The `ModAnalysis` Abstract Class

The `ModAnalysis` class has a central position in MAF, as it is the root class that all modular analyses inherit from. This class, of which a fraction of the code is shown below, is instantiated with the program under analysis (`prog`) (line 1).

```
1  abstract class ModAnalysis(prog: Expr) {
2    trait Dependency
3    var deps = Map[Dependency,Set[Component]]()
4    type Component
5    def initialComponent: Component
6    abstract class IntraAnalysis(cmp: Component) {
7      def register(dep: Dependency): Unit = ...
8      def trigger(dep: Dependency): Unit = ...
9      def spawn(cmp: Component): Unit = ...
10     def analyze(): Unit
11   }
12   def analyze(): Unit
13   def addToWorkList(cmp: Component): Unit
14 }
```

`ModAnalysis` provides the core infrastructure regarding the management of dependencies. It defines an extensible trait to represent dependencies (line 2), and tracks for each dependency the set of components that need to be reanalysed if the dependency is triggered (line 3), i.e., when the analysis state corresponding to the dependency has been modified. The analysis is parameterised in two important aspects.

Intra-Component Analysis. Different modular analyses employ a different definition of components and their corresponding intra-component analysis. A modular analysis must define a type `Component` (line 4) that represents components, and must provide the `initialComponent` corresponding to the first component of the program `prog` to analyse (line 5). As in our example of Section II, components can represent function calls, in which case the initial component corresponds to the initial call to the `main` function of the program.

The intra-component analysis is itself defined by the `IntraAnalysis` class (line 6). This class needs to implement a single method `analyze` (line 10) to perform the intra-component analysis of the given component. To implement `analyze`, one can make use of methods that are provided in the `IntraAnalysis` class to manage dependencies. Specifically, it provides a method `register` to register a dependency (line 7) corresponding to some analysis state that was read when analysing the current component, and a method `trigger` to trigger a dependency (line 8) after updating the analysis state. Method `spawn` (line 9) should be called whenever another component is discovered (e.g., upon a function call); the `ModAnalysis` class then ensures the component is scheduled for analysis if needed.

Worklist Algorithm. A modular analysis maintains a worklist of components that still need to be analysed; a worklist algorithm decides *how* (e.g., in what order) the components in the worklist are analysed. In general, different worklist algorithms result in different exploration orders in the analysis, which can influence the performance of the analysis but not its final result. To parameterise the worklist algorithm in `ModAnalysis`, two methods need to be provided. First, the `analyze` method (line 12) should implement the actual worklist algorithm, repeatedly picking and analysing components in the worklist. Second, a method `addToWorkList` (line 13) to add components to the worklist needs to be implemented. This method is called by `ModAnalysis` whenever a component is newly discovered or needs to be reanalysed.

B. The `Worklist` Traits

MAF comes with several traits that implement common worklist algorithms and that can directly be mixed in with the `ModAnalysis` class. The example below shows `SequentialWorklistAlgorithm`, a simple worklist algorithm that relies on a generic worklist of class `WorkList`. The `analyze` method loops until the analysis is finished, calling the `step` method, which performs the analysis of one component and updates the worklist accordingly.

```
1  trait SequentialWorklistAlgorithm extends ModAnalysis {
2    def emptyWorkList: WorkList[Component]
3    var wl = emptyWorkList.add(initialComponent)
4    def step() = { ... /* performs one analysis step */ }
5    def addToWorkList(cmp: Component) = { wl = wl.add(cmp) }
6    def finished() = wl.isEmpty
7    def analyze() = while (!finished()) { step() }
8  }
```

The definition of `emptyWorkList` is abstract and needs to be provided by the actual worklist implementation. For example, the `LIFOWorklistAlgorithm` trait relies on a LIFO-ordered worklist. MAF also includes other sequential worklist algorithms, e.g., using a FIFO or priority-based ordering. It also includes a parallel worklist algorithm, where multiple threads analyse components in the worklist in parallel [12].

C. The `GlobalStore` Trait

The `GlobalStore` trait is an example of a core trait that extends the `ModAnalysis` class. It can be used for modular analyses that require a *store*, which approximates the run-time heap of the program under analysis.

```
1  trait GlobalStore extends ModAnalysis with ... {
2    var store: Map[Addr, Value]
3    case class AddrDependency(addr: Addr) extends Dependency
4    trait GlobalStoreIntra extends IntraAnalysis {
5      def readAddr(addr: Addr): Value = {...}
6      def writeAddr(addr: Addr): Boolean = {...}
7    }
8  }
```

Most importantly, it provides two new methods `readAddr` (line 5) and `writeAddr` (line 6) which can be used in the intra-component analysis to read and write values at addresses in the store, respectively. The advantage of using these methods is that they provide a high-level interface to manipulate the store for the developer of the intra-component analysis, abstracting away the management of dependencies. Indeed, the store is part of the shared analysis state, and components can depend on the value at a certain address in the store, represented by an `AddrDependency` (line 3). The `readAddr` method automatically registers such dependencies (using the `register` method) upon read operations from the store, and the `writeAddr` method triggers such dependencies (using the `trigger` method) upon write operations.

D. The `ReturnValue` Trait

Similar to how the `GlobalStore` trait provides a high-level interface to manipulate a store, the `ReturnValue` trait provides a high-level interface to support components for which a return value needs to be stored. For example, if components represent function calls, then the return value of these function calls, computed by the intra-component analyses, need to be stored. To this end, the `GlobalStore` trait is used to store the return value of a component at a dedicated address, defined on line 1. In doing so, we do not need to worry about dependencies on return values, as these are managed by `GlobalStore`.

```
1  case class ReturnAddr(cmp: Component) extends Addr
2  trait ReturnValue extends GlobalStore {
3    trait ReturnValueIntra extends GlobalStoreIntra {
4      def writeResult(res: Value, cmp: Component) =
5        writeAddr(returnAddr(cmp), res)
6      def readResult(cmp: Component): Value =
7        readAddr(returnAddr(cmp))
8      def call(cmp: Component): Value =
9        { spawn(cmp) ; readResult(cmp) }
10   }
11 }
```

The `ReturnValue` class provides two methods to manipulate return values: `writeResult` writes the return value of a given component (line 4), whereas `readResult` reads its return value (line 6). Furthermore, it provides a convenience method `call` (line 8), which implements the common pattern where we read the return value of a component after creating it (e.g., when reading the return value of a called function).

IV. Instantiating an Analysis for Scheme

In this section, we illustrate the usage of MAF by instantiating several analyses for Scheme. In particular, we show how one can build upon the core `ModAnalysis` class and its traits discussed in the previous section to develop a function-modular analysis (similar to the example discussed in Section II) for Scheme, which is referred to as ModF [5]. Such a ModF analysis should implement the `SchemeModAnalysis` abstract class, which is a subclass of the `ModAnalysis` class. Similar to the `ModAnalysis` class, several traits can be mixed in to configure parameters specific to a ModF analysis (such as its context-sensitivity) or of a modular analysis in general (such as its worklist algorithm, using the core traits discussed earlier). Figure 2 shows an overview for a subset of these traits.

A. The `SchemeModAnalysis` Class

The main purpose of the `SchemeModAnalysis` class is to fill in the definition of components and the corresponding intra-component analysis for a ModF analysis for Scheme. We show some of its code below.

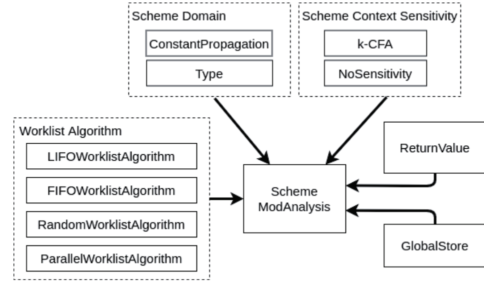

Fig. 2. Overview of the traits that can be used to create analyses for Scheme. Arrows denote a necessary requirement, and dotted boxes indicate a choice between multiple traits.

```
1  abstract class SchemeModAnalysis(prg: SchemeExp)
2       extends ModAnalysis(prg) with GlobalStore
3                                with ReturnValue {
4    trait Component
5    case object Main extends Component
6    case class Call(clo: Closure, ctx: Context)
7                    extends Component
8    def initialComponent = Main
9    class SchemeIntraAnalysis(cmp: Component)
10       extends IntraAnalysis(cmp) with GlobalStoreIntra
11                                  with ReturnValueIntra {
12     def eval(exp: SchemeExp, env: Env): Value =
13       exp match {
14         case SchemeVar(v) => readAddr(env(v))
15         case l: SchemeLam => lattice.clo(l,env)
16         case SchemeApl(fun,args) =>
17           apply(eval(fun,env),
18                 args.map(arg => eval(arg,env)))
19         ...
20       }
21     def analyze() =
22       writeResult(eval(body(cmp), env(cmp)), cmp)
23   }
24   type Context
25   def allocCtx(p: Position, as: List[Value], ...): Context
26
27   type Value
28   val lattice: SchemeLattice[Value, ...]
29 }
```

A component for ModF is defined on lines 4–7. It can be either the `Main` object, representing the initial call to the `main` function of the program, or a `Call` object, representing a call to some function in the program. The latter includes not only the closure that is called, but also some context of type `Context` to distinguish between different calls to the same function. On line 8, `Main` is given as the initial component.

The intra-component analysis for ModF is defined in the class `SchemeModFIntraAnalysis`. In general, an intra-component analysis is relatively easy to implement; in ModF, this boils down to an intra-procedural analysis. We implement such an intra-procedural analysis using a simple recursive evaluator (lines 12–20). For example, to evaluate a variable, we just look it up in the environment, and read the value at the resulting address in the store. To evaluate a function call, we recursively evaluate the operator and operands before applying the function. For brevity, the definition of the `apply` method is omitted. Method `apply` does not actually step into the called function, but just reads the return value of the component corresponding to the function that is called (using method `call` of `ReturnValue`). Method `analyze`, the entry-point of the intra-component analysis, can then be implemented by simply writing the value resulting from the evaluation of the function's body as the return value of the component under analysis (line 21).

Both the context-sensitivity and abstract domain used in the analysis are parameterised in `SchemeModAnalysis`. To specify context-sensitivity, one must define what contexts are

used (line 24), and how they are allocated (line 25) given the information that is available for some function call (such as the call site and the arguments of the function call). To specify the abstract domain, one must define what abstract values are used (line 27), and provide a corresponding instantiation of the `SchemeLattice` type class (line 28), which implements common Scheme operations on these abstract values.

B. The `SchemeContextSensitivity` Traits

We have included several well-known context-sensitivity policies for MODF in MAF. These policies are again made available as traits that can directly be mixed into `SchemeModAnalysis` instances. In addition, one can easily define custom context-sensitivity policies. As an example, we show how k-call-site sensitivity is implemented.

```
1  trait KCallSiteSensitivity extends SchemeModAnalysis {
2    val k: Int
3    type Context = List[Position]
4    def allocCtx(pos: Position, ..., caller: Component) =
5      (pos :: context(caller)).take(k)
6  }
```

Similarly, we have included other simple context-sensitivity policies, such as argument sensitivity, as well as more complex compound sensitivities, which allocate different contexts for different closures. The trait `NoSensitivity` can be used for a context-insensitive analysis, i.e., an analysis where no additional context is used for components.

C. The `SchemeDomain` Traits

MAF also comes with several traits to configure the abstract domain of a `SchemeModAnalysis`. These determine how values during the execution of the program are approximated by abstract values in the analysis. For instance, when using the `Type` abstract domain, values at a given program location are abstracted by the set of all possible types at that location.

D. Example Instantiations

We now put everything together, illustrating how these traits can be combined to instantiate a modular analysis.

Context-Insensitive Analysis. The first example we consider is a context-insensitive analysis for Scheme, i.e., an analysis that does not distinguish between multiple calls to the same function. To this end, several of the traits discussed previously are mixed into the `SchemeModAnalysis` class. Below, we show how we can create a context-insensitive analysis using a constant propagation domain and a worklist algorithm with a LIFO exploration strategy. We assume `prg` is the Scheme program to analyse.

```
1  new SchemeModAnalysis(prg) with NoSensitivity
2                             with ConstantPropagationDomain
3                             with LIFOWorklistAlgorithm
```

Context-Sensitive Analysis. In contrast, the code shown below creates a context-sensitive analysis (specifically, using 1-CFA), given a Scheme program `prg`. Notice that we mix in the `KCallSiteSensitivity` trait mentioned earlier, and specify k to be 1 for a 1-CFA analysis. For the sake of this example, we use a type domain and a worklist algorithm with a random exploration order.

```
1  new SchemeModAnalysis(prg) with KCallSiteSensitivity
2                             with TypeDomain
3                             with RandomWorklistAlgorithm {
4    val k = 1 /* k of k-CFA */
5  }
```

Other Instantiations. There are numerous variations possible to instantiate an analysis for Scheme. Most notably, MAF also includes the ability to instantiate analyses for a multi-threaded variation of Scheme, where components correspond to processes instead of function calls, and each process is analysed in isolation [2]. Since analysing a single process is in general as challenging as the analysis of any other sequential program, we have in turn implemented the intra-component analysis using a regular MODF analysis. As a result, one obtains a modular analysis (for concurrent programs) on top of a modular analysis (for sequential programs). This demonstrates the flexibility of both MAF and of modular analyses in general.

V. LESSONS LEARNT

We now describe some best practices we adopted during the development of MAF and of analyses with it.

A. Collect Benchmarks During Development

In order to evaluate the soundness, precision and performance of analyses constructed with MAF, we have collected a suite of benchmark programs to analyse. Currently, all analyses that we implemented in MAF target Scheme programs, although the framework could support other languages as well. These benchmark programs are taken from various sources, including well-known benchmark suites for Scheme and programs used in related work on program analysis. We have collected 590 Scheme programs, totalling 125 kLOC.

B. Automate Precision Measurement

We have automated the process of evaluating the precision of the analyses developed with MAF, to quickly assess the impact of changes to the precision of analyses. MAF provides built-in support for comparing the precision of analyses by comparing the precision of computed abstract values in the store. If the store resulting from an analysis A maps an address to a value v_A, and another analysis B maps it to value v_B, then A is more precise for that address than B if $v_A \sqsubset v_B$, according to the partial ordering of the abstract domain.

However, different analyses may be configured to use different abstract domains; to make abstract values comparable in precision, MAF converts abstract values to a common domain of abstract values, enabling their comparison in that domain.

C. Automate Soundness Tests

To assess the soundness of the analyses developed with the framework, MAF contains several instruments.

Abstract Domain Tests. These check the implementation of the abstract domain by quickchecking several mathematical properties of abstract domains [13] using ScalaCheck.

Primitive Tests. Primitive tests check the implementation of the built-in language functions against their specification, by checking that the abstract values resulting from calls to these primitives are correctly approximated by the analysis.

Soundness Tests. Soundness tests assess the correctness of an entire analysis by inspecting its results. Essentially, these are a specialisation of the precision measurement. Instead of comparing the abstract values computed by two analyses, the store of an analysis is compared to the store obtained by running a concrete interpreter. For a sound analysis, the values in the analysis store must always subsume the values produced by a concrete interpreter. For non-deterministic programs, e.g., due to parallelism, the concrete interpreter is run multiple times, accumulating the concrete values.

D. Automate Performance Evaluation

MAF contains benchmarking code to evaluate the performance of analyses. To this end, only the analysis to be used, warm-up time, number of repetitions and analysis timeout need to be specified. This enables a quick setup to evaluate the performance improvements of changes to analyses.

E. Visualise the Analysis to Ease Debugging

Static analyses are known to be hard to debug, and visual means of debugging help this process [14]. To this end, we have equipped MAF with an interactive visual debugger that shows how components are created in each step of the analysis. Currently, the debugger only shows how components create one another, though this can be easily extended to support visualising other kinds of dependencies and analysis results.

F. Run All Automated Tasks in CI Infrastructure

Static analyses are complex pieces of code, where a small change can have an unexpected impact the analysis' performance, its soundness or its precision. In order to be notified of such changes as early as possible, we have integrated our automated soundness tests in a continuous integration infrastructure, and plan on doing the same for performance and precision tests. As running all such tests upon every commit would take too much time, only a fraction of the tests are run upon every commit. The full test suite is run on a daily basis. We found that this continuous monitoring, especially for correctness, is vital throughout the development of analyses.

VI. Related Work

Numerous static analysers are described in the literature. Currently, analysers targeting dynamic higher-order languages such as Scheme [15], JavaScript [9]–[11], or Python [16] are mostly research-oriented. Although some of these can analyse huge code bases, none of these follow the design of a modular static analyser. MAF is based on the modular design of Scala-AM [15], but instead of performing whole-program analyses, MAF is focused on modular analyses.

Few tools or techniques have been developed to help static analysis developers. The survey of Nguyen et al. [14] reveals what static analysis developers wish would exist. The need for clear visualisations of the analysis is established, which motivated us to integrate a visual debugger in MAF. The work of Andreasen et al. [17] provides multiple techniques to increase soundness and precision of static analysers. Our automated soundness and precision tests are in line with these techniques, although Andreasen et al. propose multiple interesting techniques that could be integrated within MAF.

VII. Conclusion

In this paper, we presented MAF, a framework for developing modular static analyses. MAF supports the analysis of dynamic, higher-order languages. The framework provides a generic *inter-component analysis* that can be instantiated by defining an *intra-component analysis*. It features a highly composable and flexible design, as analyses can be implemented by mixing in pre-existing analysis traits, or by specialising required behaviour in new traits. This allows analysis developers to focus on the core aspects of their analysis, either to tune precision or to support new languages and constructs. We have demonstrated this by instantiating an analysis in MAF for Scheme programs. We concluded with a summary of best practices we adopted during the development of MAF, which includes test automation and visual debugging.

Acknowledgements

This work was partially supported by the *"Cybersecurity Initiative Flanders"* and by the Research Foundation – Flanders (FWO) (grant numbers 11D5718N and 11F4820N).

References

[1] P. Cousot and R. Cousot, "Modular static program analysis," in *CC*, ser. LNCS, vol. 2304. Springer, 2002, pp. 159–178.
[2] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "A general method for rendering static analyses for diverse concurrency models modular," *J. Syst. Softw.*, vol. 147, pp. 17–45, 2019.
[3] A. Miné, "Relational thread-modular static value analysis by abstract interpretation," in *VMCAI*, 2014, pp. 39–58.
[4] M. Journault, A. Miné, and A. Ouadjaout, "Modular static analysis of string manipulations in C programs," in *SAS*, 2018, pp. 243–262.
[5] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Effect-driven flow analysis," in *VMCAI 2019*.
[6] E. Goubault, S. Putot, and F. Védrine, "Modular static analysis with zonotopes," in *SAS*, 2012, pp. 24–40.
[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTRÉE analyzer," in *ESOP*, 2005, pp. 21–30.
[8] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NFM*, 2011, pp. 459–465.
[9] S. H. Jensen, A. Møller, and P. Thiemann, "Type Analysis for JavaScript," in *SAS*, ser. Lecture Notes in Computer Science, J. Palsberg and Z. Su, Eds., vol. 5673. Springer, 2009, pp. 238–255.
[10] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Purity analysis for javascript through abstract interpretation," *Journal of Software: Evolution and Process*, vol. 29, no. 12, 12 2017.
[11] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A Static Analysis Platform for JavaScript," in *FSE*, 2014, pp. 121–132.
[12] N. Van Es, Q. Stiévenart, J. Van der Plas, and C. De Roover, "A parallel worklist algorithm for modular analyses," in *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, September 27-28, 2020*. IEEE Computer Society, 2020.
[13] J. Midtgaard and A. Møller, "Quickchecking static analysis properties," *Softw. Test. Verification Reliab.*, vol. 27, no. 6, 2017.
[14] L. Nguyen Quang Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Trans. Software Eng.*, vol. 46, no. 7, 2020.
[15] Q. Stiévenart, M. Vandercammen, W. De Meuter, and C. De Roover, "Scala-AM: A modular static analysis framework," in *SCAM*, 2016.
[16] A. Fromherz, A. Ouadjaout, and A. Miné, "Static value analysis of python programs by abstract interpretation," in *NFM*, 2018.
[17] E. S. Andreasen, A. Møller, and B. B. Nielsen, "Systematic approaches for increasing soundness and precision of static analyzers," in *SOAP@PLDI*, 2017, pp. 31–36.