Scala-AM: A Modular Static Analysis Framework

Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
{qstieven,mvdcamme,wdmeuter,cderoove}@vub.ac.be

***Abstract*—We present Scala-AM, a modular framework for static analysis based on systematic abstraction of abstract machines. Scala-AM achieves modularity by separating operational semantics, abstract values and machine abstraction concerns via *actions*, thus enabling language designers, static analysis developers and machine abstraction experts to combine their efforts. This modularity enables us to support semantics for multiple languages, and to include multiple machine abstractions in our framework. Different operational semantics and machine abstractions can then be combined with minimal effort.**

We evaluate our framework by demonstrating how it enables implementing machine abstractions independently of the operational semantics, and by instantiating the framework for a static taint analysis of Scheme.

I. Introduction

The approach of abstract interpretation using systematic abstraction of abstract machines [17] starts from an interpreter in small-step operational semantics implemented as an abstract machine that transitions between states, which is systematically abstracted into an *abstract abstract machine* (AAM).

At the cost of precision, the domain of the abstracted machine is designed to be finite: instead of interpreting a single run of the program resulting in a possibly infinite sequence of states, the machine over-approximates all possible runs resulting in a finite graph. This graph might contain spurious paths that may not be reachable during concrete program execution, but can be explored by subsequent analyses.

It is hard to perform static analysis of highly dynamic languages with higher-order functions, where control- and data-flow are intertwined: when analyzing e.g., a function call `(f x)`, it may be impossible to precisely determine the value of the operator `f` statically. The AAM approach is a technique that can deal with such languages. Its main advantage is that the resulting description of the abstracted machine is close to how a concrete interpreter would be described. With this technique, precision can be configured in various way, with the most precise configuration resulting in concrete interpretation. This enables static analysis developers to experimentally verify soundness of their tools by checking that the abstract interpretation results correctly over-approximate the concrete interpretation results.

However, extending AAM beyond simple calculi becomes tedious, as the same systematic abstraction must always be applied to the abstract machine of each newly supported language. This forces related works using this approach to simplify their description, e.g., by limiting their language to the λ-calculus, mixing language semantics with machinery [6].

We believe that a better separation can be achieved between the language semantics and the abstraction mechanism. To this end, we developed Scala-AM[1], a static analysis framework in Scala. Scala-AM enables language designers to describe language semantics independently of the abstraction concern. Instantiating a machine abstraction with a language definition then leads to an abstract abstract machine that can perform both static analysis and concrete interpretation. We provide machine abstractions and language definitions that can be reused to minimize implementation work. Languages have to be described in small-step operational semantics, and should account for abstract values. For example, consider how the following code snippet might be analyzed:

```
(if (>= x 0) ; x originates from user input
  (sqrt x)
  (error "negative number detected"))
```

A concrete run of this code either results in the computed square root of `x`, or in a user error. When executed abstractly, `x` is bound to an over-approximation of its concrete value, such as the abstract value `Int`. Comparing `x` with `0` results in the abstract value `Bool`, requiring the machine to explore both branches of the conditional. Abstract values therefore also affect the implementation of the operational semantics in the abstract interpreter.

We developed Scala-AM with modularity and reusability of existing components in mind. Beyond the separation between machine abstraction and operational semantics, we also introduce modularity with respect to other components. Most importantly, we modularize the domain of abstract values used during the interpretation in a separate component that can be tuned independently of the semantics and the machine abstraction. Likewise, addresses and timestamps, which influence the precision of the abstracted abstract machine, are independent entities. In the end, static analyses can be performed on the fly during the computation of the state graph.

In this paper, we make the following contributions:

- We identify how to separate semantics from machinery in the AAM approach to abstract interpretation (Section II).
- We describe Scala-AM, a modular framework that implements this idea, where static analysis developers and language designers can develop components independently, each focusing on their own concerns (Section III).
- We evaluate our approach by implementing an existing machine abstraction (Section IV), and by building a static taint analysis in Scala-AM (Section V).

[1]https://github.com/acieroid/scala-am

II. Separating Semantics from Machinery

The original formulation of abstract interpretation using abstracted abstract machines [17], given in Fig. 1, mixes the description of the semantics of the language under analysis with abstractions that render it an abstract abstract machine. Most of the fundamental work on abstracted abstract machines suffer from the same problem [17, 4, 6, 5]. In this section, we list some observations about the formalizations of current abstract abstract machines in literature. These observations lead us to the following insight: an abstract abstract machine is an instantiation of operational semantics and an abstraction mechanism. The operational semantics specifies how interpretation should proceed at every program point, while the machine abstraction is responsible for keeping its machine state up-to-date. In Scala-AM, the semantics' decision on how interpretation should proceed is abstracted through the concept of *actions*, such as *evaluate expression e* or *push a frame on the stack*. The machine abstraction handles these actions by updating its machine state to reflect this decision.

$$
\begin{aligned}
e \in \mathit{Exp} &::= (\lambda x.e) \mid (e_0 e_1) \mid x\\
v \in \mathit{Val} &::= (\lambda x.e)\\
\rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Addr}\\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathcal{P}(\mathit{Val} \times \mathit{Env} + \mathit{Kont})\\
\kappa \in \mathit{Kont} &::= \mathbf{mt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(v, \rho, a)\\
\varsigma \in \mathit{State} &::= \langle e, \rho, \sigma, a \rangle
\end{aligned}
$$

$$
\begin{array}{r|l}
\multicolumn{2}{c}{\varsigma \longmapsto \varsigma', \text{ where } \kappa \in \sigma(a), b = \mathit{alloc}(\varsigma)}\\
\hline
\langle x, \rho, \sigma, a \rangle & \langle v, \rho', \sigma, a \rangle \text{ where } (v, \rho') \in \sigma(\rho(x))\\
\langle (e_0 e_1), \rho, \sigma, a \rangle & \langle e_0, \rho, \sigma \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b \rangle\\
\langle v, \rho, \sigma, a \rangle & \\
\text{if } \kappa = \mathbf{ar}(e, \rho', c) & \langle e, \rho', \sigma \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b \rangle\\
\text{if } \kappa = \mathbf{fn}((\lambda x.e), \rho', c) & \langle e, \rho'[x \mapsto b], \sigma \sqcup [b \mapsto (v, \rho)], c \rangle
\end{array}
$$

Fig. 1: Original formulation of AAM, where $x \in \mathit{Var}$ are variable names and $a \in \mathit{Addr}$ are addresses.

Current formalizations of abstract machines lead us to the following observations, which we enforce in Scala-AM:

1) **Semantics can act based on the current expression in the machine state.** The behavior of the semantics is dependent on the current expression of the abstract machine state. In Scala-AM, the semantics provides a `stepEval` function to the machine abstraction which returns actions that the machine abstraction needs to perform to make an evaluation step on this expression. This function does not rely on components specific to the machine such as the continuation store (see Item 5). This is made explicit in Scala-AM, whereas it is only implicit in related work.
2) **Semantics can peek at the top continuation frame when a value is reached.** The original description of AAM for λ-calculus does not make explicit the concept of *value*[2], but formalizations that use an eval-continuation machine (e.g., Johnson and Van Horn [6]) explicitly have two types of states: states where an expression has to be evaluated, and states where a value has been reached. In Scala-AM, abstract machines are explicitly in eval-continuation style (i.e., states can either contain an expression to evaluate, or a computed value). The Scala-AM framework expects the semantics to provide a `stepKont` function, which is called with the top frame on the stack when a value is reached by the abstract machine. Similarly to `stepEval`, this function returns a set of actions.
3) **Manipulation of the stack by the semantics is limited.** The semantics only pushes continuation frames on the stack, or inspects the topmost frame when a value has been reached. In Scala-AM, the stack is manipulated via actions, produced by `stepEval` and `stepKont`. The topmost frame is popped by the machine abstraction before `stepKont` is called, and the semantics can rely on the machine abstraction to pass it this topmost frame.
4) **The store maps a finite domain of addresses to sets of values.** Though the store appears in the abstract machine states, the values contained in the store are only used by the semantics. Instead of mapping addresses to sets of values, as in the original formulation of AAM, the store in Scala-AM generalizes this to any lattice. When different values must be stored at the same address, the values are joined within the lattice. The original description can be simulated via a powerset lattice. As both the semantics and the machine abstraction only interact with the lattice via an interface in Scala-AM, the value domain is independent of both components.
5) **The abstraction of the stack is specific to the abstract machine** While Van Horn and Might [17] use the same store for values and continuations, only the abstract machine accesses the continuations and only the semantics accesses the values. In Scala-AM, values and continuations are mapped in separate stores. As the continuation store is only accessible by the machine abstraction, the machine can improve precision by selecting another continuation store mechanism, such as the one described by Johnson and Van Horn [6], without affecting other components. As the abstraction of the stack is opaque to the semantics, it is up to the machine to pass the topmost continuation frame to the `stepKont` function of the semantics.

The concept of actions enables Scala-AM to separate semantics and machine abstractions in Scala-AM. We describe them in detail in Section III-B.

III. Design of the Framework

The Scala-AM architecture is based on the observation that operational semantics can be described separately from machine abstractions. We explored this further and designed a tool in which one can specify the following components independently and compose them to implement multiple variants of static analyses:

[2]This is because a value in λ-calculus is a closure, formed by a λ expression with its environment, i.e., values are members of $\mathit{Val} \times \mathit{Env}$. A state that reaches such a value therefore has the same expression component as a state that has to evaluate a λ expression in a given environment.

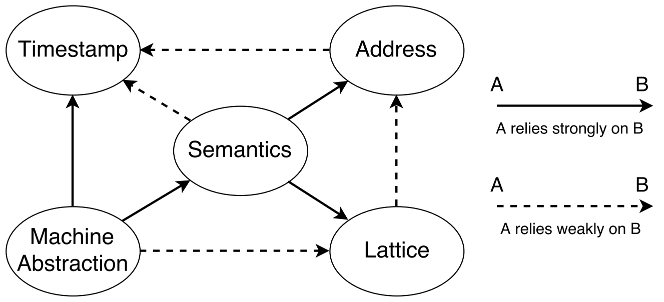

Fig. 2: Design of SCALA-AM.

- **Semantics**: language semantics describe how to evaluate a program in a given language. Making these semantics modular enables supporting multiple programming languages. We have built semantics for Scheme and λ-calculus, as well some variants (Scheme with concurrency primitives [15], by-call and by-need λ-calculus). Section V demonstrates how users of our tool can build upon these semantics to develop new analyses.
- **Lattice**: the value domain of the analyzed language is described in a lattice, and is separated from both the semantics and the abstract machine. The choice of lattice influences the precision of the analysis.
- **Abstraction mechanism**: multiple variants of the original AAM machine, with varying degrees of precision, exist. Separating the machine abstraction from the semantics of the analyzed language enables experimenting with various abstractions. It also enables designers of abstract machines to assess how their proposed machine fares in a variety of languages with minimal effort. SCALA-AM includes some abstraction mechanisms encountered in the literature [17, 6, 5, 11]. Section IV demonstrates how new machine abstractions can be added.
- **Timestamp**: timestamps introduce context sensitivity in the abstract machine, thus improving precision.
- **Address**: addresses influence the precision of the analysis, and are parameterized by the timestamps.

Fig. 2 illustrates the separation between components. Timestamps do not use any other components. Addresses may rely on timestamps, but do not perform any operations on them. Lattices may rely on addresses to represent values such as closures or cons cells. The semantics employs timestamps to allocate new addresses. The semantics also heavily relies on the lattice component for performing semantic operations on the values used in the language. The machine abstraction interacts with the semantics, via the `stepEval` and `stepKont` functions, and with the timestamp, as it increments it at every machine step. To a lesser extent, the machine abstraction may also depend on the lattice for performing subsumption checks.

In the remainder of this section, we describe the interface that must be implemented by each component.

A. Semantics

Semantics are parameterized by abstract values (`Abs`), addresses (`Addr`) and timestamps (`Time`). The semantics are responsible for evaluating and parsing programs, and for providing an initial environment and store:

- `stepEval: (Exp, Env, Store, Time) => Set[Action]`: returns the set of actions that the abstract machine must perform when evaluating an expression in a given environment.
- `stepKont: (Abs, Frame, Store, Time) => Set[Action]`: returns the set of actions to perform when a value has been computed and the given frame is at the top of the stack.
- `parse: String => Exp`: parses a program from a string. SCALA-AM provides parsing utilities for s-expression based languages.
- `initialBindings: Set[(String, Addr, Abs)]`: returns a collection of bindings that must be present in the initial environment (mapping `String` to `Addr`) and store (mapping `Addr` to `Abs`). This function can be used to implement builtins and primitives of a language, such as Scheme's + or `car` functions.

B. Action

Actions are the key element in the interaction between the machine abstraction and the semantics. At any point during the evaluation, the machine may call the semantics' `stepEval` and `stepKont` functions. It then receives a set of actions specifying what must be performed next in the small-step interpretation of the program. Possible actions are the following:

- `ActionReachedValue(Abs, Store)`: a value has been computed. Updates to the store are part of the action.
- `ActionEval(Exp, Env, Store)`: an expression has to be evaluated in the given environment. Updates to the store are also part of the action.
- `ActionPush(Frame, Exp, Env, Store)`: a frame has to be pushed on the stack and evaluation will proceed with the given expression, environment and store.
- `ActionError(SemErr)`: an error, e.g., an unbound variable, has been reached by the semantics. The current trace will not be explored further.

C. Machine Abstraction

A machine abstraction is parameterized by abstract values, addresses and timestamps, as all these components influence the behavior of the machine. A machine abstraction has to implement the following functions, both of which require an implementation of semantics for expressions of type `Exp`:

- `eval: (Exp, Semantics) => Output` computes the abstract state graph output as the transitive closure of the transition relation of the abstract machine, using the given semantics.
- `analyze[L]: (Exp, Semantics, Analysis) => Option[L]` is similar to `eval`, but performs calls to the analysis in order to propagate the analysis information through the visited states.

D. Lattice

The abstract value domain is described as a lattice, and must implement the `JoinLattice` type class which specifies the following functions:

- `bottom: Abs` specifies a bottom ($\bot$) element that represents the absence of information.
- `join: (Abs, Abs) => Abs` joins the information of the two values: `join(v1,v2)` implements $v1 \sqcup v2$.
- `subsumes: (Abs, Abs) => Boolean` defines the partial order between lattice values. If `subsumes(v1, v2)` is true, then we have $v1 \sqsupseteq v2$, i.e., `v1` contains at least as much information as `v2`.

Lattices must minimally satisfy this interface. However, the values that make up a lattice depend on the language analyzed. Lattices can implement refined type classes that enables the semantics to inject values into the lattice domain. For example, we provide lattices for Scheme-like languages, with injection functions for values commonly used in these languages: numbers, first-class closures, vectors, cons-cells etc. These lattices also provide basic operations on values, from which complex primitives can be built independently of the actual lattice implementation.

To verify our lattice implementations, we use ScalaCheck to perform quickchecking of general lattice properties, inspired by Midtgaard and Møller [10]. We also include handwritten test cases. The specification of these tests is implementation independent, and new lattice implementations can be tested independently of other components with minimal implementation overhead.

E. Timestamp and Address

SCALA-AM also enables the user to redefine addresses and timestamps through the type classes `Address` and `Timestamp`. Implementations are provided for some existing strategies: context insensitive timestamps, k-CFA timestamps [13], concrete timestamps to model concrete execution [17], and classical addresses using allocation-site information [1].

F. Analysis

All previous components enable performing abstract interpretation. However, to perform useful static analyses, we must not only compute the possible states reachable by a program, but we must also derive and collect information from these states. To this end, we define an `Analysis` trait that, given a type of information `L` to compute over the program, describes how this information evolves during interpretation. This component will be used by the machine abstraction and the following functions will be called throughout the computation:

- `stepEval: (Exp, Env, Store, Time, L) => L` specifies how to update the information when the semantics' `stepEval` function is called.
- `stepKont: (Frame, Abs, Store, Time, L) => L` specifies how to update the information when the semantics' `stepKont` function is called.
- `error: (SemErr, L) => L` specifies how to update the information when an error state with the given error message is reached.
- `join: (L, L) => L` specifies how to join the information computed by the analysis.
- `init: L` is the initial value for the information computed.

An analysis is typically dependent on the language analyzed and the lattice used, as will be illustrated in Section V where we build a static taint analysis for Scheme programs.

IV. Example: AAM Abstract Machine

To define a new machine abstraction, the `AbstractMachine` trait, with an `eval` and an `analyze` function, must be implemented. In this section, we implement the AAM approach [17] where each state has a *control* component, which is either:

- `ControlEval(Exp, Env)` if an expression has to be evaluated in the given environment.
- `ControlKont(Abs)` if a value has been reached and a frame will have to be popped from the stack.
- `ControlError(SemErr)` if an error has been reached.

States of this machine are represented by class `State` composed of a control component, a store binding addresses to abstract values, a continuation store binding continuation addresses to continuations, a continuation address `a` representing the current continuation, and a timestamp `t`.

The `step` method of `State` computes the set of its successor states. `step` checks the control component, dispatches to the appropriate function in the semantics and then *integrates* the actions to generate the set of resulting states:

```scala
def step(sem: Semantics): Set[State] = control match {
  case ControlEval(e, env) =>
    integrate(a, sem.stepEval(e, env, store, t))
  case ControlKont(v) => kstore.lookup(a).flatMap({
    case Kont(frame, next) =>
      integrate(next, sem.stepKont(v, frame, store, t)) })
  case ControlError(_) => Set() }

def integrate(a: KontAddr, as: Set[Action]): Set[State] = {
  val t2 = time.tick(t)
  as.map({
    case ActionReachedValue(v, store) =>
      State(ControlKont(v), store, kstore, a, t2)
    case ActionPush(frame, e, env, store) =>
      val next = NormalKontAddress(e, t)
      val kstore2 = kstore.extend(next, Kont(frame, a))
      State(ControlEval(e, env), store, kstore2, next, t2)
    case ActionEval(e, env, store) =>
      State(ControlEval(e, env), store, kstore, a, t2)
    case ActionError(err) =>
    State(ControlError(err), store, kstore, a, t2) }) }
```

The `inject` helper function takes the expression to analyze, and returns the initial state by adding the necessary extra information: the initial environment and store, the initial continuation store (empty), the initial continuation address (`HaltKontAddress`, a specific address indicating that the stack is empty), and the initial timestamp.

We use the `halted` method to check whether a state has reached the end of the execution. This function returns true if the state is an error state or a value state with no current continuation.

The `eval` function of the machine abstraction first generates the successors of the initial state by calling `step` and then loops over these states. It keeps track of its visited states to avoid re-exploring states already explored.

```
def eval(exp: Exp, sem: Semantics): Output = {
  def loop(todo: Set[State], visited: Set[State]):
    AAMOutput = todo.headOption match {
    case Some(s) if visited.contains(s) =>
      loop(todo.tail, visited, graph)
    case Some(s) if (s.halted) =>
      loop(todo.tail, visited + s, graph)
    case Some(s) =>
      loop(todo.tail ++ s.step(sem), visited + s)
    case None => AAMOutput(...) }
  loop(Set(State.inject(exp, sem.initialBindings)), Set()) }
```

Although not shown here, a state graph can also be incrementally generated and returned in the output.

Static analysis can be either performed on the graph resulting from `eval`, or on the fly via the `analyze` function, which works very similarly to `eval`, but propagates analysis information while exploring the graph.

V. Case Study: Taint Analysis of Scheme Programs

We now demonstrate how a static taint analysis on higher-order side-effecting programs can be constructed in Scala-AM. We only give a general outline of how the analysis is constructed, but the full implementation is publicly available[3].

A. Lattice

The analysis uses a product lattice that combines a standard constant propagation lattice with a taint lattice (Fig. 3). The standard lattice models the possible values obtained during the evaluation of the program, while the taint lattice models the *taint status* of the values. A value starts as untainted, until it goes through a *source*. Due to imprecision in the analysis, tainted and untainted values may need to be joined, in which case the result is the top element of the lattice, i.e., a value that *may be tainted*. In practice, tainted values are annotated with the program location of the source of taint, and locations are joined in a set when taint status has to be joined.

$$\begin{tikzcd}
 & \top & \\
\text{UNTAINTED} \arrow[ur, no head] & & \text{TAINTED} \arrow[ul, no head] \\
 & \bot \arrow[ul, no head] \arrow[ur, no head] &
\end{tikzcd}$$

Fig. 3: The taint lattice used in the static taint analysis.

B. Language and Semantics

As Scheme semantics are already implemented in Scala-AM, we just extend the semantics with three primitives that model sources that taint values, sanitizers, and sinks.

1) `(taint x)` returns the value of its argument with taint status set to Tainted.
2) `(sanitize x)` returns the value of its argument with the taint status set to Untainted.
3) `(sink x)` results in an error if a tainted value may flow to `x`. If the taint status of `x` is $\top$, it results in both an error and a non-error successor state.

In practice, sources, sinks, and sanitizers are modeled by annotating parts of a program instead of relying on explicit primitives. However, program transformations can automatically convert such annotations to these three primitives without having to modify the semantics.

C. Analysis

We design an analysis that locates potential flow of tainted values from sources to sinks in Scheme programs. When a tainted value flows to a sink, it results in an error, as we have detected a potential leak in the program. The information computed by the analysis is a set of leak errors that can be reached during exploration of the program state space. The join operation is the union over sets of errors, and the initial value of the analysis is the empty set. The precision of the resulting analysis depends on components such as the machine abstraction, the addresses and timestamps. One can achieve different levels of flow-sensitivity and context-sensitivity depending on the configuration chosen.

D. Usage

The following code depicts the steps needed to perform the analysis on a program stored in the form of a string.

```
val prims = new TSchemePrimitives // instantiate primitives
val sem = new SchemeSemantics(prims) // instantiate semantics
// parse program to an expression
val pgm = sem.parse("(let* ((x (taint 1)) (y x)) (sink y))")
val m = new AAM // instantiate an abstracted abstract machine
val res = m.analyze(pgm, sem, TaintAnalysis) // run analysis
```

VI. Related work

Most of the currently available static analysis frameworks have little support for extensions to multiple programming languages, especially for higher-order languages. Notable exceptions are Facebook's tools pfff[4] and Infer[5]. Pfff supports many languages, including languages with higher-order functions such as Lisp, Python and Haskell. However, each new language must implement its own analyses, as pfff requires specifying a function that generates the control flow graph for each language. We rely on the abstract machine to perform the work of building the control flow graph, based on the actions returned by the semantics. Infer only supports intra-procedural analyses of object-oriented imperative languages, by relying on the fact that such languages share similar concepts. Our framework can be easily extended to support new concepts, and performs inter-procedural analysis.

The K [12] framework and Kiama [14] both allow to express abstract machines with the same programming effort as in Scala-AM, but have no built-in support for abstracting components of the machine to perform static analysis.

The Rascal [8] framework provides a domain-specific language to develop analysis and manipulation tools. On the other hand, we provide a framework focused on abstract interpretation, written in an expressible general purpose language, which shares some features with Rascal (e.g., data-type definitions, strong static typing, pattern matching, comprehension).

The Clang static analyzer [9] supports LLVM bytecode, which can be generated for multiple languages. However, some

[3]https://github.com/acieroid/scala-am/wiki/Example:-static-taint-analysis

[4]http://github.com/facebook/pfff

[5]http://fbinfer.com

languages may not have a natural compilation to LLVM bytecode and useful information can be lost during the compilation process. The IKOS tool [3] also uses a specific representation in which supported languages have to compile. This low-level representation can be generated from LLVM bytecode. However, just like the Clang static analyzer, potential information may be lost between the original source code and the analyzed intermediate representation. Likewise, the Soot [16, 2] and the Wala frameworks[6] perform intra- and inter-procedural dataflow analysis on Java intermediate bytecode. Though the Wala framework has recently been extended with support for Javascript, neither framework provides mechanisms for developers to add support for other languages.

VII. Future Work

Our framework has three purposes: offer tool support for language designers that describe languages with operational semantics, enable static analysis developers to implement static analyses with minimal implementation effort, and enable researchers in abstract interpretation to prototype different machine abstractions formulations in a variety of languages. We envision several directions for the future work towards each of these goals.

a) Semantics: While Section V demonstrates that Scala-AM can already be used as a base for building interesting static analyses for Scheme, we wish to investigate how to facilitate supporting more language features. We believe that while some features (e.g., objects and records) can be implemented solely by modifying the lattice domain, others (e.g., exceptional control flow) might require extending the set of actions. These additions would require some, but minimal changes in the abstract machine.

b) Machine Abstractions: The machine abstractions provided with the framework are currently implemented in a relatively straightforward manner, with no focus on performance. We are interested in implementing existing optimizations [7] to assess their impact on performance. We already investigated the use of *global store widening*, which has a great impact on performance. We also envision our framework being used to experiment with new machine abstractions and optimizations.

c) Static Analysis: Work on the framework mostly focused on the language definition and machine abstraction components, but we plan on experimenting with more complex static client analyses. We also plan on building some analyses that can work across multiple languages to enable tool support for the same analysis on languages supported by Scala-AM.

VIII. Conclusion

We presented the design and implementation of Scala-AM, a highly modular static analysis framework. Scala-AM enables language designers to add support for new languages and new language constructs, without affecting the analysis aspect. Static analysis developers can rely on existing language semantics and machine abstractions to describe static analyses in a high-level implementation. Abstract interpretation researchers can experiment with abstract machine abstractions in order to optimize and improve the precision of the analyses. We believe this is a step forward towards a framework that can be used as a foundation for building static analyses that span across multiple languages, as well as an experimentation platform that relieves implementers from the need to implement the same parts of an analysis over and over again.

[6]http://wala.sourceforge.net

Acknowledgments

Quentin Stiévenart is funded by the *GRAVE* project of the Research Foundation - Flanders (FWO).

References

[1] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[2] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *SOAP'12*. ACM, 2012.

[3] G. Brat, J. A. Navas, N. Shi, and A. Venet. Ikos: A framework for static analysis based on abstract interpretation. In *SEFM'14*. Springer, 2014.

[4] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. *ACM SIGPLAN Notices*, 47(9), 2012.

[5] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. Van Horn. Pushdown control-flow analysis for free. *ACM SIGPLAN Notices*, 51(1), 2016.

[6] J. I. Johnson and D. Van Horn. Abstracting abstract control. *ACM SIGPLAN Notices*, 50(2), 2015.

[7] J. I. Johnson, N. Labich, M. Might, and D. Van Horn. Optimizing abstract abstract machines. *ACM SIGPLAN Notices*, 48(9), 2013.

[8] P. Klint, T. Van Der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM'09*, pages 168–177. IEEE, 2009.

[9] T. Kremenek. Finding software bugs with the clang static analyzer. *Apple Inc*, 2008.

[10] J. Midtgaard and A. Møller. Quickchecking static analysis properties. In *ICST'15*. IEEE, 2015.

[11] M. Might and D. Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. In *Static Analysis*. Springer, 2011.

[12] G. Rosu and T. Serbanuta. An overview of the K semantic framework. *JLAP*, 79(6), 2010.

[13] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie-Mellon University, 1991.

[14] A. M. Sloane. Lightweight language processing in kiama. In *GTTSE'09*, pages 408–425. Springer, 2009.

[15] Q. Stievenart, J. Nicolay, W. De Meuter, and C. De Roover. Detecting concurrency bugs in higher-order programs through abstract interpretation. In *PPDP'15*. ACM, 2015.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *CASCON'99*. IBM Press, 1999.

[17] D. Van Horn and M. Might. Abstracting abstract machines. *ACM SIGPLAN Notices*, 45(9), 2010.