Building a Modular Static Analysis Framework in Scala (Tool Paper)

Quentin Stiévenart   Jens Nicolay   Wolfgang De Meuter   Coen De Roover

Vrije Universiteit Brussel, Belgium

{qstieven,jnicolay,wdmeuter,cderoove}@vub.ac.be

Abstract

We present SCALA-AM, a framework for implementing static analyses as systematically abstracted abstract machines. Analyses implemented on top of SCALA-AM separate operational semantics from machine abstraction concerns. This modularity facilitates varying the analyzed language and the applied abstraction method in an analysis. We describe the design of our framework and demonstrate its use in a static analysis for the DOT calculus. We conclude with a tour of the features of Scala through which SCALA-AM achieves its modularity.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Theory, Languages, Verification

Keywords Static Analysis, Abstract Interpretation, Abstract Machines

1. Introduction

Static analysis is a useful technique for assessing the correctness of programs, to detect bugs, or prove their absence. However, supporting modern programming language features like higher-order functions and concurrency in static analyzers is far from trivial. The *Abstracting Abstract Machines* (AAM) approach [5] to static analysis enables languages and analyses to be implemented in a natural and systematic way, by taking a concrete interpreter and systematically abstracting it into an abstract interpreter. However, most formalizations of AAM and its derivatives tend to mix the language semantics with the machine abstraction, leading to poor modularity.

In SCALA-AM[1], we separate the definition of the language semantics from the machine abstraction in order to obtain high modularity. This allows both language designers and abstract machine specialists to work together on the same artefact and focus on their domain of expertise. We believe our framework is a useful foundation for supporting new features of languages, while serving as a platform to help abstract machine researchers develop new techniques and test them on languages larger than simple calculi.

SCALA-AM has been developed with modularity in mind, and this goal has been achieved thanks to the features of Scala. In this paper, we present the design of SCALA-AM, we demonstrate how languages can be supported by implementing semantics for DOT [1], and we describe some of the features of Scala for achieving modularity. This paper makes the following contributions:

- We describe a modular static analysis framework that supports multiple languages and abstraction mechanisms by separating machine abstraction from semantics.
- We describe the use of the framework in a static analysis for the DOT language.
- We report on the Scala features through which SCALA-AM achieves its high modularity.

2. Background: Abstracting Abstract Machines

The small-step operational semantics of a language can be described as an abstract machine. Injecting the program into the machine's initial state causes the machine to start applying its state transition function repeatedly until a final state is reached. A *trace* of the machine states arising during the program's execution is computed.

The *Abstracting Abstract Machines* [5] approach is a technique where such an abstract machine is systematically abstracted into an *abstract abstract machine*, or an abstract interpreter. Defined over a finite state space with a non-deterministic state transition function, it now produces a *graph* instead of a trace — also for non-terminating programs.

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

This graph can be queried for the results of the static analysis corresponding to the applied machine abstraction.

The main advantage of the AAM approach is that the resulting abstract interpreter closely resembles the concrete one from which it is systematically derived through abstraction. Various abstraction patterns arise, which can be applied to support other languages or new language features. For example, potential sources of infiniteness can be threaded through the machine's *store* to ensure its state space remains finite. This is the case for closures: a closure is composed of an expression and an environment, and environments map variables to values, which themselves can be closures. This recursive knot in the state space is cut by threading closures through the store, and having environments map variables to addresses in the store. Restricting addresses to a finite number then renders the state space finite.

Related work on AAM has produced different techniques to abstract the concrete abstract machine, with varying degrees of precision. A current summary of the state of the art can be found in [3]. The goal of SCALA-AM is to enable experimenting with these techniques on different languages with minimal implementation effort: changing the language analyzed should not impact the machine abstraction, and vice-versa.

3. The SCALA-AM Framework

SCALA-AM performs static analysis on programs in languages described as small-step operational semantics. A semantics description forms a component of SCALA-AM, and another component, the machine abstraction, drives the interpretation of the program.

To analyze a program, SCALA-AM first *injects* the program into an initial machine state. A state represents the current state of the evaluation, and contains information such as the expression to evaluate, the binding environment, the store, and an abstraction of the stack. It is the responsibility of the machine abstraction to manipulate the stack. The abstract machine then performs *step*s on the current state by relying on the language semantics described as a small-step operational semantics. If the current state needs to evaluate an expression, it will call the semantics' `stepEval` function. If the current state needs to continue with the topmost continuation, the machine abstraction pops the continuation and calls the semantics' `stepKont` function. Both of these functions return a set of *actions* that describe the successor states to generate. Actions, returned by `stepEval` and `stepKont`, can be *eval* actions when an expression has to be evaluated, *value* actions when a value has been reached, requiring the next step to inspect the topmost continuation frame, and *push* actions when a continuation has to be pushed on the stack before evaluating a given expression.

This process is repeated until all reachable states have been explored. The state space of the abstract machine has to be finite to ensure the convergence of this process.

$$
\begin{aligned}
t, u \in \mathit{Term} &::= \lambda x.t \;\mid\; \nu x.d \;\mid\; \texttt{let}\; x \;\texttt{=}\; t \;\texttt{in}\; u \\
&\;\mid\; x \;\mid\; xy \;\mid\; x.a \\
d \in \mathit{Definition} &::= \texttt{\{}x = t\texttt{\}} \;\mid\; d_1 \wedge d_2
\end{aligned}
$$

$$
\begin{array}{r|l}
\varsigma \longmapsto \varsigma' & a = \mathit{alloc}(\varsigma) \\
\hline
\langle x, \rho, \sigma, \kappa \rangle & \langle \sigma(\rho(x)), \sigma, \kappa \rangle \\
\langle \lambda x.t, \rho, \sigma, \kappa \rangle & \langle \mathbf{clo}(\lambda x.t, \rho), \sigma, \kappa \rangle \\
\langle \nu x.d, \rho, \sigma, \kappa \rangle & \langle \mathbf{obj}(\nu x.d, \rho), \sigma, \kappa \rangle \\
\langle \texttt{let}\; x \;\texttt{=}\; t \;\texttt{in}\; u, \rho, \sigma, \kappa \rangle & \langle t, \rho, \sigma, \mathbf{letk}(x, u, \rho) : \kappa \rangle \\
\langle x\, y, \rho, \sigma, \kappa \rangle & \langle t, \rho[x \mapsto a], \sigma[a \mapsto v_{arg}], \kappa \rangle \\
 & \text{where } \mathbf{clo}(\lambda x.t, \rho) = \sigma(\rho(x)) \\
 & \qquad v_{arg} = \sigma(\rho(y)) \\
\langle x.a, \rho, \sigma, \kappa \rangle & \langle t, \rho[y \mapsto \rho(x)], \sigma, \kappa \rangle \\
 & \text{where } \mathbf{obj}(\nu y.d, \rho) = \sigma(\rho(x)) \\
 & \qquad t = \mathit{member}(d, a) \\
\langle v, \sigma, \mathbf{letk}(x, u, \rho) : \kappa \rangle & \langle u, \rho[x \mapsto a], \sigma[a \mapsto v], \kappa \rangle
\end{array}
$$

Figure 1. DOT small-step semantics as an abstract machine, where x, y are variable names and a is a field name.

In SCALA-AM, actions enable formulating machine abstractions and language semantics independently of each other. Other components can be implemented independently as well. The most important of these components is the value lattice, which forms the abstraction of the values of the language analyzed, and which enables tuning the precision of the resulting analysis. Addresses and timestamps are other components that influence context sensitivity and precision of the machine abstraction.

4. Example: Analyzing DOT Programs

We now show how support for a language can be added to SCALA-AM. We take the DOT language [1] as an example[2].

4.1 Adding Support for DOT

Input languages have to be described as small-step operational semantics that perform operations over an environment and store. This is done for DOT in Fig. 1. DOT expressions are naturally represented as case classes in Scala:

```
trait Term { val pos: Pos }
case class Lam(x: String, t: Term, pos: Pos)
case class Obj(x: String, d: Def, pos: Pos) extends Term
case class Let(x: String, t: Term, u: Term, pos: Pos) extends Term
case class Var(x: String, pos: Pos) extends Term
case class App(x: String, y: String, pos: Pos)
case class Sel(x: String, a: Member, pos: Pos) extends Term

trait Def
case class Field(a: Member, t: Term, pos: Pos) extends Def
case class Aggregate(d1: Def, d2: Def, pos: Pos) extends Def
```

These expressions have to implement the `Expression` type class, requiring the presence of a component `Pos` which serves to map expressions to their source code position.

The definition of the semantics corresponding to the formal definition of Fig. 1 only accesses values through the

[2] SCALA-AM, the full implementation of this example and others can be found at `https://github.com/acieroid/scala-am/`.

DotLattice type class. The functions stepEval and stepKont are defined as follows:

```
val dabs = implicitly[DotLattice[Abs]]
val addr = implicitly[Address[Addr]]

def stepEval(t: Term, env: Env, ...) = t match {
  case Var(x, _) => for {
    v <- evalVar(x, env, store)
  } yield Action.value(v, store)
  case Lam(x, t, _) =>
    Action.value(dabs.lambda(x, t, env), store)
  case Obj(x, d, _) =>
    Action.value(dabs.obj(x, d, env), store)
  case Let(x, t, u, _) =>
    Action.push(FrameLet(x, u, env), t, env, store)
  case App(x, y, _) => for {
    fun <- evalVar(x, env, store)
    arg <- evalVar(y, env, store)
  } yield dabs.getClosures(fun).map({
    case (x, t, env) =>
      val a = addr.variable(x, arg, time)
      Action.eval(t, env.extend(x, a), store.extend(a, arg)) })
  case Sel(x, a, _) => for {
    obj <- evalVar(x, env, store)
  } yield dabs.getObjects(obj).map({
    case (x, defs, env) =>
      findTermMember(defs, a) match {
        case Some(t) =>
          val ad = addr.variable(x, obj, time)
          Action.eval(t, env.extend(x, ad), store.extend(ad, obj))
        case None =>
          Action.error(NoTermMember(a, obj.toString, t.pos))
      } })}
def stepKont(v: Abs, frame: Frame, ...) = frame match {
  case FrameLet(x, u, env) =>
    val a = addr.variable(x, v, time)
    Action.eval(u, env.extend(x, a), store.extend(a, v)) }
```

The stack component of the abstract machine (κ in Fig. 1) is *not present* in the semantics definition. Pushing on the stack is done through Action.push, and popping from the stack when a value is reached is handled automatically by the machine abstraction, which then calls stepKont. The evalVar: (String, Env, Store) => MayFail[Abs] function looks up the value of a variable, and may result in an error (see Section 6.4 for a description of the MayFail monad). This enables describing the semantics independently of the machine abstraction. State-of-the-art machine abstractions are provided (AAM, AAC and P4F [3]), and each abstracts the stack differently. Other machine abstractions can be added as well.

Values in the language have to be represented as a join semi-lattice. The definition of the semantics is independent of the implementation of this lattice through type class DotLattice. This enables describing the semantics independently of the actual values. Different abstraction levels (e.g., concrete values, types of values, . . .) can be used with a single definition of the semantics.

```
trait DotLattice[L] extends JoinLattice[L] {
  def lambda(v: String, body: Term, env: Env): L
  def obj(v: String, defs: Definition, env: Env): L
  def getClosures(x: L): Set[(String, Term, Env)]
  def getObjects(x: L): Set[(String, Definition, Env)] }
```

A straightforward implementation of this lattice is to have sets of values as lattice elements, where values can be either a closure or an object. We omit this implementation for brevity.

4.2 Running the Analysis

Having defined the semantics of DOT, one can use one of the provided machine abstractions to compute the flow graph of a DOT program, which can then be used to verify properties of the analyzed program:

```
1 val dot = new DotLanguage[ClassicalAddress.A]
2 val lattice = dot.DotLatticeImpl
3 implicit val isDotLattice = lattice.isDotLattice
4 val sem = new dot.DotSemantics[lattice.L, ZeroCFA.T]
5 val machine = new AAM[Term, lattice.L,
6                       ClassicalAddress.A, ZeroCFA.T]
7 val res = machine.eval(sem.parse(args(0)), sem, true, None)
8 res.toDotFile("graph.dot")
```

The DotLanguage class is parameterized with the addresses used, and contains the definition introduced above. To analyze a program, a machine abstraction has to be instantiated (line 5), and its eval method takes as argument the input program (parsed by the semantics), the semantics, a boolean indicating if the computed state graph has to be kept in memory, and an optional timeout. The resulting graph can be extracted to a file (line 8)[3]. We highlight here what the modularity of the framework allows us to perform:

- To use a different lattice, one would only change line 2.
- To analyze a different language, one would change line 4 to provide the language semantics, and the occurence of Term on line 5 (as a different language would use different expressions).
- To use a different machine abstraction, one would change line 5.
- Context sensitivity can also be tuned by using different type arguments in line 5: here we use program locations as addresses and perform a context-insensitive analysis (0-CFA).

5. Performance

Lacking a corpus of DOT programs to benchmark our example analysis on, we evaluate the performance of SCALA-AM on an extended version of the Gabriel benchmarks for Scheme instead[4]. We computed the flow graph for each program in the corpus using a) the Scheme semantics provided by the framework, b) the AAM machine abstraction with a global store, c) a lattice representing values by their types, and d) no context sensitivity. Most of the benchmarks complete in under a second. Table 1 includes an excerpt of the results, including the slowest benchmarks.

While performance has not been our main concern, these results show that our implementation can analyze small programs in a decent amount of time. Note that the running time does not depend on the size of the program in LOCs,

[3] Unfortunately, the format used to represent graphs in SCALA-AM is also called DOT. This graph language has no link whatsoever wih the DOT language analyzed in this section.

[4] The benchmarks are included with the SCALA-AM distribution. The benchmarks were run on a 2014 Mac Book Pro, with a Intel Core i7 2.8GHz processor and 16 GB of DDR3 memory, with Scala version 2.11.8.

Benchmark	LOC	Time (s)
`church`	28	191.72
`mceval`	239	177.36
`dderiv`	82	4.63
`regex`	81	1.15
`scm2java`	268	< 0.01

Table 1. Running time of SCALA-AM on Scheme benchmarks.

but rather on the complexity of its behavior. The `church` benchmark, for instance, is short but contains many higher-order functions. The `scm2java` compiler benchmark, in contrast, is 10 times as long but written in a procedural manner, and takes less than one millisecond to analyze.

6. Scala as a Driver for Modularity

In this section, we describe a few features of Scala that allow us to obtain high modularity in SCALA-AM.

6.1 Type Classes

The use of type classes [2] is essential in the SCALA-AM framework to abstract over the implementation of its components. For example, components requiring a lattice take as type parameter a type that implements the `JoinLattice` type class. The `JoinLattice` type class specifies a join semi-lattice that has to provide a bottom element ($\bot$), a way to join elements ($\sqcup$), and an ordering ($\sqsubseteq$). A join semi-lattice is a monoid (`mzero` is $\bot$, `mappend` is $\sqcup$) with a partial ordering. In fact, we automatically derive both monoid and partial ordering instances from join lattices in SCALA-AM.

```
trait JoinLattice[L] {
  def bottom: L
  def join(x: L, y: L): L
  def subsumes(x: L, y: L): Boolean }
```

We also have type classes for expressions, addresses, and timestamps, with implementations that model common techniques such as 0-CFA, k-CFA, and concrete execution. These are the components that make our framework modular: changing the implementation of any of these components changes the kind of analysis that is performed. Addresses and timestamps influence precision of the abstract machine, and the lattice influences precision over the values.

6.2 Extending Type Classes

The values in languages described in SCALA-AM must implement the `JoinLattice` type class. However, this type class is very limited: it only requires the definition of generic join semi-lattice value $\bot$ and operations $\sqcup$ and $\sqsubseteq$. These operations are the only ones needed by the machine abstraction. However, when defining language semantics, it should be possible to inject values into or extract values from the lattice and perform operations on these values. Instead of hardcoding a specific lattice in the implementation of the language semantics, we opt for a modular approach and create an extended type class that can be plugged into the semantics, enabling analysis with different lattice implementations. The type class extends `JoinLattice` and provides the necessary additional lattice operations. This is illustrated in the example of Section 4.

As another example, consider the Scheme language, for which we can define a `SchemeLattice` type class as follows.

```
trait SchemeLattice[L] extends Joinlattice[L] {
  def inject(x: Int): L
  def inject(x: Boolean): L
  def unaryOp(op: UnaryOperator)(x: L): MayFail[L]
  def binaryOp(op: BinaryOperator)(x: L, y: L): MayFail[L]
  // ...and others
}
```

The `MayFail` monad represents computations that may fail and/or succeed, and is described in Section 6.4.

The implementation of a lattice can represent values concretely (e.g., `1`, `#t`), by their types (e.g., `Int`, `Bool`), or by some other abstraction (e.g., intervals for integers). Low-level operations on these values are performed through the `unaryOp` and `binaryOp` functions. For example, to add two values, one would call `lattice.binaryOp(Plus)(x, y)`. These low-level operations are the responsibility of the lattice implementation.

To implement primitives of the language, one can define a set of primitive operations that are defined in terms of low-level operations, and are therefore independent of the lattice implementation. Consider for example the Scheme primitive +, which takes a variable amount of arguments. All the following calls to + are valid: `(+)`, `(+ 1)`, `(+ 1 2)`. This primitive can be defined as follows.

```
object Plus extends NoStoreOperation("+") {
  override def call(args: List[Abs]) = args match {
    // (+) is 0
    case Nil => abs.inject(0).point[MayFail]
    // (+ x rest) is x added to (+ rest)
    case x :: rest => call(rest) >>= (plus(x, _)) }}
```

In this fashion, a large subset of the Scheme language can be supported with minimal overhead for adding new lattice implementations. The same technique can be applied to other languages as well. SCALA-AM provides implementations of common lattices over simple domains (integers, floats, booleans, strings) that can be used as building blocks to implement more complex lattices, such as the Scheme lattice.

6.3 Testing through Quickchecking with ScalaCheck

The join operation of a join semi-lattice $\langle L, \sqcup\rangle$ must be associative ($x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$), commutative ($x \sqcup y = y \sqcup x$), and idempotent ($x \sqcup x = x$). Moreover, when extending a join semi-lattice with other operations, some properties have to be satisfied: $\bot$ is preserved through operations, operations are monotone, and operations are a sound over-approximation of their concrete counterpart.

Take for example a lattice that has support for integers, injected through `inject: Int => L`, and has a `plus: (L, L) => L` operation. We need the following properties to hold:

- Bottom is preserved: $\texttt{plus}(\bot, x) = \bot$, $\texttt{plus}(x, \bot) = \bot$.
- The operation is monotone: $b \sqsubseteq c \implies \texttt{plus}(a, b) \sqsubseteq \texttt{plus}(a, c) \wedge \texttt{plus}(b, a) \sqsubseteq \texttt{plus}(c, a)$.

- The operation is a sound over-approximation of its concrete counterpart: $\texttt{inject}(a + b) \sqsubseteq \texttt{plus}(\texttt{inject}(a), \texttt{inject}(b))$.

Inspired by [4], we include quickcheck-style tests in our framework using the ScalaCheck[5] library to test these properties. To support tests for a new lattice implementation, generators for elements of the lattice have to be developed. A generator has to be able to generate arbitrary elements of the lattice, and arbitrary elements that are subsumed by a given element.

```
trait LatticeGenerator[L] {
  def any: Gen[L]
  def le(l: L): Gen[L] }
```

Given a generator for elements of a lattice, we can then generate lattice tests. Below is a fragment of our test suite to test integer lattices[6].

```
case class IntSpec[I : IsInteger](gen: LatticeGenerator[I]) {
  val int = implicitly[IsInteger[I]]
  val bot = int.bottom
  property("plus on bottom is bottom") {
    forAll { (a: I) =>
      assert(int.plus(bot, a) == bot && int.plus(a, bot) == bot)
    }}
  property("plus is monotone") {
    forAll { (a: I, c: I) => forAll(gen.le(c)) { (b: I) =>
      assert(int.subsumes(c, b) &&
             int.subsumes(int.plus(a, c), int.plus(a, b)) &&
             int.subsumes(int.plus(c, a), int.plus(b, a)))
    }}}
  property("plus is a sound over-approximation") {
    forAll { (a: Int, b: Int) =>
      assert(int.subsumes(int.plus(int.inject(a), int.inject(b)),
                          int.inject(a + b)))
    }} }
```

Combined with manual test cases, this approach allows us to have high confidence in the correctness of the lattice implementations with relatively low effort.

6.4 The Need for a May Fail Monad

Interpreters that work with abstract values must be able to represent computations that may fail *and* produce a result at the same time. For example, in a lattice representing values by sets of types, if a is `{Int}` and b is `{Int, Bool}`, then computing $a + b$ produces an integer and a failure result. SCALA-AM represents failures by one or more semantic errors (`SemErr`). A computation that may fail has to return a potential resulting value and a list of errors (`(Option[L], List[SemErr])`). Representing all computations that may fail by such a type leads to a lot of boilerplate code. We therefore introduce the `MayFail` monad. This monad has three kinds of values: success, error, or both.

```
trait MayFail[L]
case class MFSuccess[L](l: L) extends MayFail[L]
case class MFError[L](errs: List[SemErr]) extends MayFail[L]
case class MFBoth[L](l: L, errs: List[SemErr]) extends MayFail[L]
```

The bind function on this monad is defined as follows.

[5] https://www.scalacheck.org

[6] This test code could be further parameterized by the operations to test: the code for testing `plus` and `minus` is almost identical.

```
def bind[A, B](fa: MayFail[A])(f: (A) => MayFail[B])
  : MayFail[B] = fa match {
  case MFSuccess(l) => f(l)
  case MFError(errs) => MFError(errs)
  case MFBoth(l, errs) => f(l) match {
    case MFSuccess(l) => MFBoth(l, errs)
    case MFError(errs2) => MFError(errs ++ errs2)
    case MFBoth(l, errs2) => MFBoth(l, errs ++ errs2) }}
```

Thanks to Scalaz'[7] support for monads, to implicits, and Scala's `for` notation, we are able to obtain readable descriptions of language semantics, as demonstrated in Section 4.

7. Conclusion and Future Work

In this paper, we presented SCALA-AM, a modular static analysis framework that can be extended to support multiple languages and machine abstractions. A central concept in the framework are *actions* returned by the semantics to drive the machine abstraction. We demonstrated how languages, described as small-step operational semantics, can be added to the framework, and how Scala features allowed us to achieve high modularity. We believe SCALA-AM can be used as a foundation to build and experiment with static analyses without having to implement everything from scratch.

The framework currently provides support for a large subset of Scheme, as well as multiple machine abstractions. It also supports modeling languages with shared memory concurrency. As future work, we plan on extending it with other languages and features such as non-local control flow. This would require extending the set of actions that can be returned by the semantics. We also plan on working on optimizations of abstract machines in order to construct efficient static analyses that can support a wide variety of languages without polluting the language semantics.

Acknowledgments

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

References

[1] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In *WadlerFest 2016*, volume 9600 of *Lecture Notes in Computer Science*.

[2] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA'10*.

[3] 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.

[4] J. Midtgaard and A. Møller. Quickchecking static analysis properties. In *ICST'15*.

[5] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP'10*.

[7] http://typelevel.org/projects/scalaz