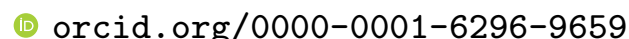
Université Libre de Bruxelles
Computer Science Department

MEMO-F524 Masters thesis

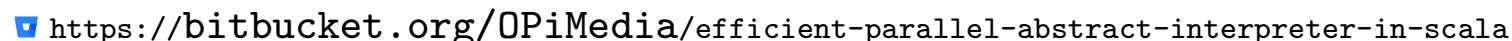
An Efficient and Parallel Abstract Interpreter in Scala

— First Algorithm —

Olivier Pirson — opi@opimedia.be


orcid.org/0000-0001-6296-9659

November 12, 2018


https://bitbucket.org/OPiMedia/efficient-parallel-abstract-interpreter-in-scala

Vrije Universiteit Brussel

Promotors: Coen De Roover, Wolfgang De Meuter
Advisor: Quentin Stievenart

The context: Abstract Interpretation for Static Analysis
The problematic: Too heavy for real programs
First parallel algorithm
Implementation
First results
Done and todo
References

1 The context: Abstract Interpretation for Static Analysis

2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 Implementation

5 First results

6 Done and todo

7 References

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

We need **tools** to help us to **build correct programs**.

Testing is not enough.

Figure: First "flight" of Ariane 5 in 1996.

Static Analysis:
study the **behaviour of programs without executing** them.

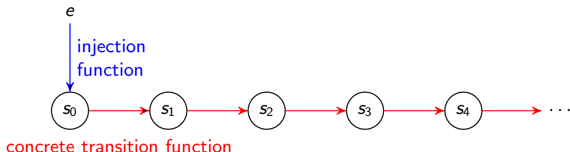
Not trivial questions about the behaviour of a program
are **undecidable** (RICE's theorem).

Abstract Interpretation:
approximation technique to perform static analysis.

Trade-off:
approximation must be **enough precise** to have an useful analysis,
and enough imprecise to make the problem **decidable**.

From Concrete Interpretation...

Trace: concrete interpretation with **small-step semantics**, for **one** instance.



Program is **executed** by interpreter,
described by an **Abstract Machine** (AM).

- One execution is for one instance on this program.
- e is for one expression, i.e. a program.
- s_i are the successive states during this execution.

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

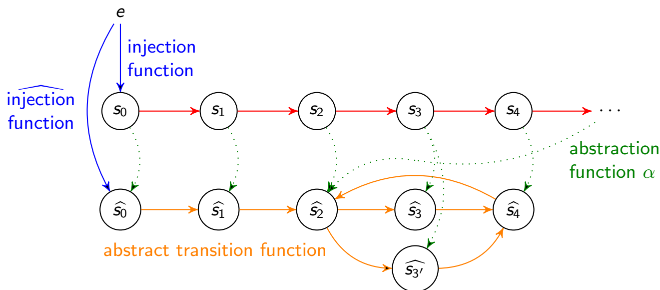
Implementation

First results

Done and todo

References

Trace: concrete interpretation with **small-step semantics**, for **one** instance.



Abstracting Abstract Machine (**A**AM).
2 over-approximations:

- on addresses: by modulo; give a finite state space
- on values: abstraction

Abstract transition function returns **all** directly reachable states.

State graph: abstract interpretation, for **all** instances.
"*The abstract simulates the concrete*" (Might).

Approximation of values: abstraction

Based on mathematical notion of **lattice** (**partially ordered sets** with additional properties).

Examples of abstraction for the set of integer values:

- the type integer
- intervals
- sign: $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ abstracted by $\{\bot, -, 0, +, (- \text{ and } 0), (0 \text{ and } +), \top\}$

Figure: René MAGRITTE, **Le Calcul Mental**. 1940.

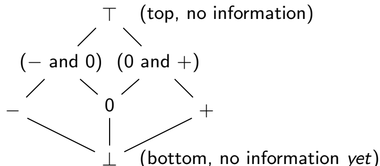


Figure: HASSE diagram of the complete lattice of signs.

Properties of a lattice are such that it contains the **join** of 2 elements and the succession of operation give a **fixed point**.

Example of a state graph on a very simple program

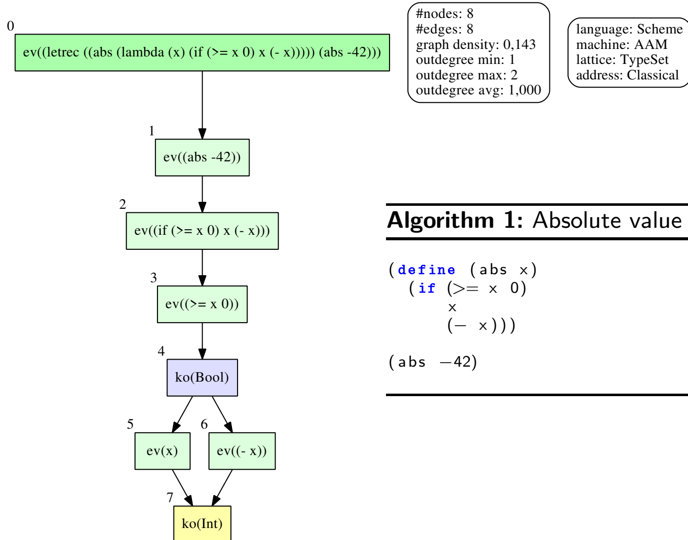


Figure: State graph of the abs program with lattice type.

Algorithm 1: Absolute value of −42

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))

(abs -42)
```

1 The context: Abstract Interpretation for Static Analysis

2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 Implementation

5 First results

6 Done and todo

7 References

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

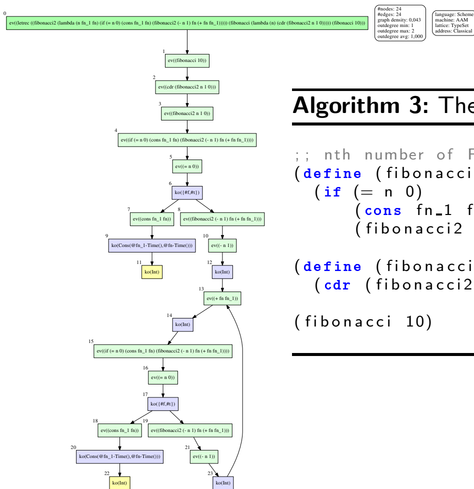
Algorithm 2: The 10th FIBONACCI number.

```
;; nth number of Fibonacci (by recursive process)
(define (fibonacci n)
  (if (<= n 1)
      n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))

(fibonacci 10)
```

Figure: State graph of the FIBONACCI program with lattice type.

Example of a state graph on a other very simple program (bis)



Algorithm 3: The 10th FIBONACCI number.

```
;; nth number of Fibonacci (by iterative process)
(define (fibonacci2 n fn_1 fn)
  (if (= n 0)
      (cons fn_1 fn)
      (fibonacci2 (- n 1) fn (+ fn fn_1))))

(define (fibonacci n)
  (cdr (fibonacci2 n 1 0)))

(fibonacci 10)
```

Figure: State graph of the FIBONACCI program with lattice type.

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

With an AAM the problem is became decidable.
But in general an AAM is **too slow** to analyse real programs.

One way to have a faster AAM is to
parallelize generation of the state graph.

Goal of this master thesis:
implement and compare several parallelizations of AAM in the framework Scala-AM.

Parallel model for the implementation: **actors** with **Akka**.

Language analysed: **Scheme**.

Figure: Agents Smith from the *Matrix* trilogy.

Actor model

Actor, like an object, isolated entity with its own encapsulated data and behaviour. With some fundamental differences.

- Entirely private.
- **No shared** mutable state (so no data race).
- Communication by immutable **asynchronous messages** (sent and received sequentially).
- Each actor has a mailbox (a queue).
- Capability to create other actors.

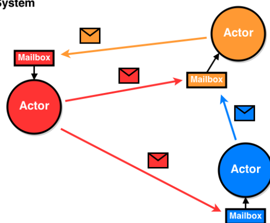


Figure: Richard Doyle. *Using Akka and Scala to Render a Mandelbrot Set*. 2014.
http://blog.scottlogic.com/2014/08/15/using-akka-and-scala-to-render-a-mandelbrot-set.html

The context: Abstract Interpretation for Static Analysis
The problematic: Too heavy for real programs
First parallel algorithm
Implementation
First results
Done and todo
References

1 The context: Abstract Interpretation for Static Analysis
2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 Implementation
5 First results
6 Done and todo
7 References

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

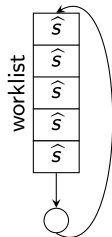
The context: Abstract Interpretation for Static Analysis
The problematic: Too heavy for real programs
First parallel algorithm
Implementation
First results
Done and todo
References

Factorized version from K. Dewey, V. Kashyap, B. Hardekopf. *A parallel abstract interpreter for JavaScript*. 2015.

Algorithm 4: The sequential worklist algorithm

```
procedure MAIN_PROCESS(worklist, memo, ς̂old, ς̂new, ς̂)
    if memo does not contain partition(ς̂) then
        memo[partition(ς̂)] := ς̂
        put ς̂ on worklist
    else
        ς̂old := memo[partition(ς̂)]
        ς̂new := ς̂old ⊔ ς̂
        if ς̂new ≠ ς̂old then
            memo[partition(ς̂)] := ς̂new
            put ς̂new on worklist
        end_if
    end_if
end_procedure

procedure SEQUENTIAL
    put the initial abstract state ς̂0 on the worklist
    initialize map memo : Partition → Σ♯ to empty
    repeat
        remove an abstract state ς̂ from the worklist
        for_all abstract states ς̂′ in next_states(ς̂) do
            MAIN_PROCESS(worklist, memo, ς̂old, ς̂new, ς̂′)
        end_for
    until worklist is empty
end_procedure
```



Worklist parallel strategy

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

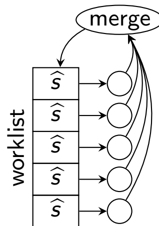
Done and todo

References

Factorized version from K. Dewey, V. Kashyap, B. Hardekopf. *A parallel abstract interpreter for JavaScript*. 2015.

Algorithm 5: The worklist parallel algorithm

```
put the initial abstract state ς̂0 on the worklist
initialize templist to empty
initialize map memo : Partition → Σ♯ to empty
repeat
    for_all abstract states ς̂ in the worklist do_in_parallel
        for_all abstract states ς̂′ in next_states(ς̂) do
            begin_thread_safe
                MAIN_PROCESS(templist, memo, ς̂old, ς̂new, ς̂′)
            end_thread_safe
        end_for
    end_parallel_for
    swap worklist and templist
until worklist is empty
```



Worklist parallel strategy

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Redundant computations.

Synchronization at the merge step.

Article test on few real JavaScript programs.
Results show that this adaptation of the sequential algorithm is not optimal.

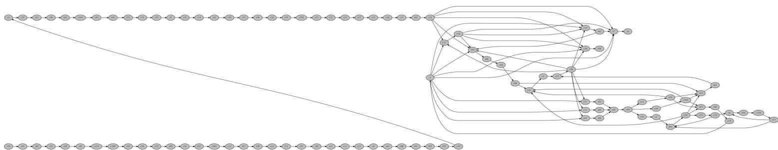


Figure: L. Andersen, M. Might. *Multi-core Parallelization of Abstracted*. 2013.

Improvements with producer/consumer model.

Better, per-context parallel strategy

An Efficient and Parallel Abstract Interpreter in Scala — First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Authors of *A parallel abstract interpreter for JavaScript* introduce a per-context parallel strategy.
The main idea is to separate these two parts:

- state exploration
- control of state space by some merging operations.

The intuitive idea is to parallelize “functions” instead basic “blocks”.

1 The context: Abstract Interpretation for Static Analysis

2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 **Implementation**

5 First results

6 Done and todo

7 References

Actor logic

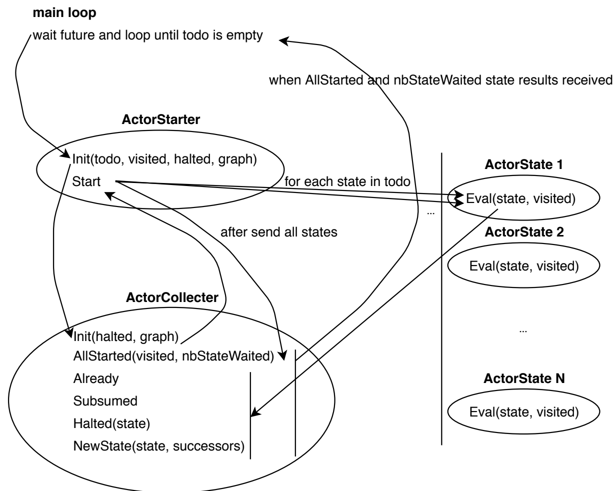


Figure: Actors logic used to implemented the worklist parallel strategy.

Scala implementation – main loop

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Algorithm 6: The main loop, until the worklist todo is empty

```
@scala.annotation.tailrec
def loop(todo: List[State], visited: VS[State], halted: Set[State], graph: Graph):
    ParAAMOutput = todo match {
  case Nil => {             // finished, the worklist todo is empty
    actorSystem.terminate
    ParAAMOutput(halted, VisitedSet[VS].size(visited), timeout.time, graph, false, stats)
  }

  case _ =>
    if (timeout.reached)  // exceeded the maximal time allowed, stop
      ParAAMOutput(halted, VisitedSet[VS].size(visited), timeout.time, graph, true, stats)
    else {                  // will send each state from worklist todo to ActorState
      val future = actorStarter ? ActorStarter.Init(todo, visited, halted, graph)

      val (newTodo, newVisited, newHalted, newGraph) =
        Await.result(future, Timeout("some" seconds).duration)
          .asInstanceOf[(List[State], VS[State], Set[State], Graph)]

      loop(newTodo, newVisited, newHalted, newGraph)
    }
}
```

Scala implementation – ActorStarter 1/2

Algorithm 7: Init other actors, send states to all ActorState

```
final class ActorStarter extends Actor {
  import ActorStarter._

  var todo: List[State] = null
  var newVisited: VS[State] = VisitedSet[VS].empty
  var actorI: Int = -1
  var nb: Int = -1

  override def receive = {
    case Init(todo: List[State], visited, halted: Set[State], graph: Graph) =>
      this.todo = todo
      newVisited = visited
      actorI = 0
      nb = 0
      actorCollecter ! ActorCollecter.Init(halted, graph, sender)

    case Start =>
      for (state <- todo) {
        if (actors(actorI) == null)  // create new actor
          actors(actorI) = actorSystem.actorOf(Props(new ActorState))

        actors(actorI) ! ActorState.Eval(state, newVisited, actorCollecter)
        newVisited = VisitedSet[VS].add(newVisited, state)

        actorI += 1
        if (actorI == maxActorStates) actorI = 0
        nb += 1
      }

      actorCollecter ! ActorCollecter.AllStarted(newVisited, nb)
  }
}
...
```

Scala implementation – ActorStarter 2/2

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Algorithm 8: Init other actors, send states to all ActorState

```
...
object ActorStarter {
  val Start = 1
  case class Init(todo: List[State], visited: VS[State], halted: Set[State], graph: Graph)
}

val actorStarter: ActorRef = actorSystem.actorOf(Props(new ActorStarter))
```

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Algorithm 9: Starts ActorStarter, collect results from all ActorState

```
final class ActorCollecter extends Actor {
  import ActorCollecter._

  var mainSender: ActorRef = null
  var newTodo: List[State] = null
  var newHalted: Set[State] = null
  var newGraph: Graph = null
  var nb: Int = -1  // count AllStarted received + the number of state results receive
  var nbStateWaited: Int = -1
  var newVisited: VS[State] = VisitedSet[VS].empty

  def increment() = {
    nb += 1
    if (nb > nbStateWaited)  // all started and finished all states, then send results
      mainSender ! (newTodo, newVisited, newHalted, newGraph)  // to actorStarter
  }

  override def receive = {
    case NewState(state: State, successors: Set[State]) =>
      newGraph = newGraph.map(_.addEdges(state, successors))
      newTodo ++= successors
      increment
    case Already => increment
    case Subsumed => increment
    case Halted(state: State) =>
      newHalted += state
      increment
...
```

An Efficient and Parallel Abstract Interpreter in Scala
—
First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Algorithm 10: Starts ActorStarter, collect results from all ActorState

```
...
    case Init(halted: Set[State], graph: Graph, mainSender: ActorRef) =>
      this.mainSender = mainSender
      newTodo = Nil
      newHalted = halted
      newGraph = graph
      nb = 0
      nbStateWaited = Int.MaxValue  // biggest value to avoid end before init par AllStarted
      newVisited = VisitedSet[VS].empty
      sender ! ActorStarter.Start  // to actorStarter
    case AllStarted(visited, nbStateWaited: Int) =>
      this.nbStateWaited = nbStateWaited
      newVisited = visited
      increment
  }
}
object ActorCollecter {
  val Already = 2
  val Subsumed = 3
  case class NewState(state: State, successors: Set[State])
  case class Halted(state: State)
  case class Init(halted: Set[State], graph: Graph, mainSender: ActorRef)
  case class AllStarted(visited: VS[State], nbStateWaited: Int)
}

val actorCollecter: ActorRef = actorSystem.actorOf(Props(new ActorCollecter))
```

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

First parallel algorithm

Implementation

First results

Done and todo

References

Algorithm 11: evaluate states

```
final class ActorState extends Actor {
  import ActorState.Eval

  override def receive = {
    case Eval(state: State, visited, actorCollecter: ActorRef) =>
      if (VisitedSet[VS].contains(visited, state))  // already
        actorCollecter ! ActorCollecter.Already
      else if (subsumption &&                        // subsumed
        VisitedSet[VS].exists(visited, state, (s2: State) => s2.subsumes(state)))
        actorCollecter ! ActorCollecter.Subsumed
      else if (state.halted)                         // halted stated
        actorCollecter ! ActorCollecter.Halted(state)
      else                                           // new state
        actorCollecter ! ActorCollecter.NewState(state, state.step(sem))
  }
}
object ActorState {
  case class Eval(state: State, visited: VS[State], actorCollecter: ActorRef)
}
```

1 The context: Abstract Interpretation for Static Analysis

2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 Implementation

5 First results

6 Done and todo

7 References

First (very rough) results

An Efficient and Parallel Abstract Interpreter in Scala — First Algorithm

The context: Abstract Interpretation for Static Analysis

The problematic: Too heavy for real programs

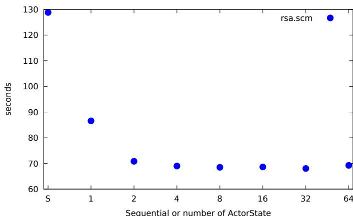
First parallel algorithm

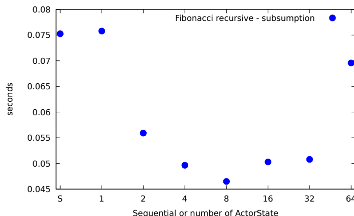
Implementation

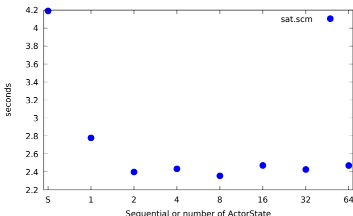
First results

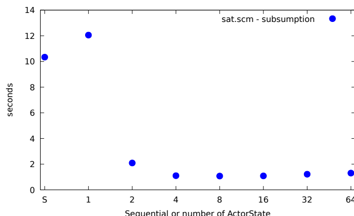
Done and todo

References









Quick average on 5 repetitions after 3 repetitions skipped (3 and 1 for rsa.scm) with Scala 2.12.7 and Akka 2.5.18 (Java 1.8.0 – GraalVM 1.0.0) on Intel Xeon Gold 6148 2.40GHz, 16 “cores” available (Hydra).

1 The context: Abstract Interpretation for Static Analysis

2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 Implementation

5 First results

6 Done and todo

7 References

Work done

- Reading of theoretical background on lattices, AAM and parallelism.
- Redaction of an introduction to the subject (preparatory work, previous year).
- Learning actors, Akka.
- Learning use of some tools: sbt, Scala environments. . .
- Learning functioning of Scala-AM.
- Removed some parts of Scala-AM (other machines or languages), implementation of little features.
- Implemented the worklist parallel algorithm.
- Reimplemented the `Graph` data structure.
- Some thoughts about a kind of metric to characterize "ideal" parallelization of given Scheme programs.

The context: Abstract Interpretation for Static Analysis
The problematic: Too heavy for real programs
First parallel algorithm
Implementation
First results
Done and todo
References

Work to be done

- Maybe revise the implementation of the worklist parallel algorithm.
- Think about how and to what to benchmark.
- Implement better parallel algorithm(s).
- Try impact of metrics considered.
- Evaluate all of them,
  identify advantages and disadvantages for each of them.
- Reading of other articles on parallelization.
- Final redaction.

1 The context: Abstract Interpretation for Static Analysis

2 The problematic: Too heavy for real programs

3 First parallel algorithm

4 Implementation

5 First results

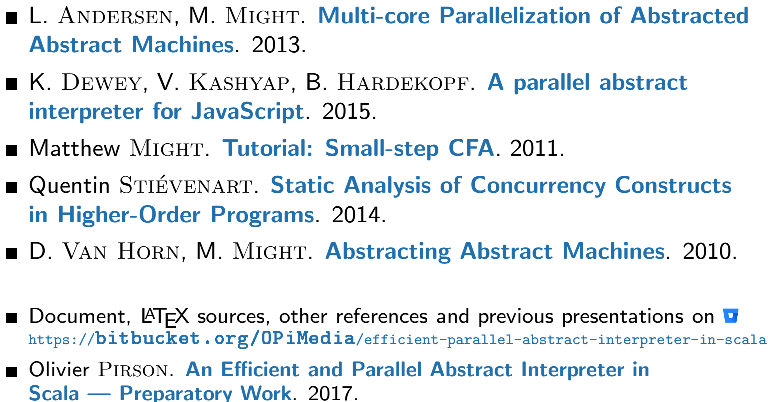
6 Done and todo

7 References

The context: Abstract Interpretation for Static Analysis
The problematic: Too heavy for real programs
First parallel algorithm
Implementation
First results
Done and todo
References

References

Thank you! Questions time...



- L. Andersen, M. Might. **Multi-core Parallelization of Abstracted Abstract Machines**. 2013.
- K. Dewey, V. Kashyap, B. Hardekopf. **A parallel abstract interpreter for JavaScript**. 2015.
- Matthew Might. **Tutorial: Small-step CFA**. 2011.
- Quentin Stiévenart. **Static Analysis of Concurrency Constructs in Higher-Order Programs**. 2014.
- D. Van Horn, M. Might. **Abstracting Abstract Machines**. 2010.

- Document, LaTeX sources, other references and previous presentations on https://bitbucket.org/OPiMedia/efficient-parallel-abstract-interpreter-in-scala
- Olivier Pirson. **An Efficient and Parallel Abstract Interpreter in Scala — Preparatory Work**. 2017.