Toward a Verified Software Toolchain for Java

David Pichardie - INRIA Rennes
How do you trust your software?
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques.
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques

• Manual verifications
  – do not scale
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale

- Automatic bug finders
  - may miss some bugs
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques.

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms
  - ex: the Astrée static analyzer

http://www.astree.ens.fr/

~1M loc of a critical control-command software analyzed

0 false alarms
How do you trust the tool that verifies your software?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms
  ex: the Astrée static analyzer

http://www.astree.ens.fr/

~1M loc of a critical control-command software analyzed

0 false alarms

<table>
<thead>
<tr>
<th>manual verification</th>
<th>bug finders</th>
<th>sound verifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>yesterday</td>
<td>today</td>
<td>tomorrow</td>
</tr>
</tbody>
</table>
How do you trust the tool that verifies your software?

The increasing complexity of safety critical systems requires efficient validation techniques

• Manual verifications
  - do not scale

• Automatic bug finders
  - may miss some bugs

• Automatic, sound verifiers
  - find all bugs, may raise false alarms
    ex: the Astrée static analyzer

• Formally-verified verifiers
  - the verifier comes with a soundness proof
  - that is machine checked

Manual verification    Bug finders    Sound verifiers    Formally verified verifiers

yesterday    today    tomorrow    after tomorrow
How do we verify the verifier?
How do we verify the verifier?

A simple idea:
How do we verify the verifier?

A simple idea:

Program and prove your verifier in the same language!
How do we verify the verifier?

A simple idea:

Program and prove your verifier in the same language!

Which language?
How do we verify the verifier?

A simple idea:

Program and prove your verifier in the same language!

Which language?

Coq
Coq: an animal with two faces
Coq: an animal with two faces

First face:
First face:

- a proof assistant that allows to interactively build proof in constructive logic
Coq: an animal with two faces

First face:
• a proof assistant that allows to interactively build proof in constructive logic

Second face:
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic

Second face:
- a functional programming language with a very rich type system
Coq: an animal with two faces

First face:
• a proof assistant that allows to interactively build proof in constructive logic

Second face:
• a functional programming language with a very rich type system
  example:

  sort: ∀ l: list int, { l’: list int | 
  Sorted l’ ∧ PermutationOf l l’ }
Coq: an animal with two faces

First face:
• a proof assistant that allows to interactively build proof in constructive logic

Second face:
• a functional programming language with a very rich type system
  example:
  \[
  \text{sort: } \forall l : \text{list int}, \{ \ l' : \text{list int} \mid \text{Sorted } l' \land \text{PermutationOf } l \ l' \} \\
  \text{• with an extraction mechanism to Ocaml}
  \[
  \text{sort: int list } \rightarrow \text{ int list}
  \]
Our Methodology
Our Methodology

We program the static analyzer inside Coq

\begin{verbatim}
Definition analyzer (p:program) := ...
\end{verbatim}
Our Methodology

We program the static analyzer inside Coq

**Definition** analyzer (p:program) := ...

We state its correctness wrt. a formal specification of the language semantics

**Theorem** analyser_is_sound :
\[ \forall p, \text{analyser } p = \text{Yes } \rightarrow \text{Sound}(p) \]
Our Methodology

We program the static analyzer inside Coq

\textbf{Definition} \text{ analyzer (p:program) := ...}

We state its correctness wrt. a formal specification of the language semantics

\textbf{Theorem} \text{ analyser_is_sound} :
\forall p, \text{ analyser} p = \text{Yes} \rightarrow \text{Sound}(p)

We interactively and mechanically prove this theorem

\textbf{Proof}. ... (* few days later *) ... \text{Qed.}
Our Methodology

We program the static analyzer inside Coq

\textbf{Definition} analyzer (p:program) := ...

We state its correctness wrt. a formal specification of the language semantics

\textbf{Theorem} analyser_is_sound : \\
\forall p, analyser p = Yes \rightarrow \text{Sound}(p)

We interactively and mechanically prove this theorem

\textbf{Proof}. ... (* few days later *) ... Qed.

We extract an OCaml implementation of the analyzer

\textbf{Extraction} analyzer.
A Posteriori Validation
An important tool in our toolbox

We program the full static analyzer inside Coq

```coq
Definition analyzer (p:program) :=
  ...;
  let x := complex_computation p in
  ...;
```

... or ...

6
**A Posteriori Validation**

An important tool in our toolbox

We program the *full* static analyzer inside Coq

```coq
Definition analyzer (p:program) :=
    ...
    let x := complex_computation p in
    ...
```

... or we program some parts in Coq, other parts in OCaml and use a verified validator

```coq
Definition analyzer (p:program) :=
    ...
    let x := ocaml_external_complex_computation p in
    match validator x p with
    | OK ⇒ ...
    | Error ⇒ abort
    end.
```

Ideally we also prove (on paper) that if the external implementation implements correctly a well-known algorithm then the validator will always succeed (completeness)
Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   • (informally) shared by any end-user programmer, compiler, static analyzer
   • less specialized than static analyzer’s abstract semantics

2. Logical Framework
   • only the proof checker needs to be trusted
   • we don’t trust sophisticated decision procedures
Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   • (informally) shared by any end-user programmer, compiler, static analyzer
   • less specialized than static analyzer’s abstract semantics

2. Logical Framework
   • only the proof checker needs to be trusted
   • we don’t trust sophisticated decision procedures
1. Formal specification of the programming language semantics
   • (informally) shared by any end-user programmer, compiler, static analyzer
   • less specialized than static analyzer’s abstract semantics

2. Logical Framework
   • only the proof checker needs to be trusted
   • we don’t trust sophisticated decision procedures

Still a large code base but at least a *foundational* code base: logic & semantics
Verified Static Analysis: Challenges

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»
Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»
Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed
Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

• You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
- Not all proofs share the same vocabulary. We seek for a proof at the level of the language semantics.
Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
- Not all proofs share the same vocabulary. We seek for a proof at the level of the language semantics.
- You rarely have a full proof. You reason on a core subset of a language but interactions between all features may invalidate this proof.
Verified Static Analysis: Challenges

Common belief

«Once you have a good proof of your tool on paper, mechanizing it is just a matter of time!»

I think such a reasoning is severely flawed

- You never have a proof of a tool. You have a proof of an algorithm and implementation details sometimes matter.
- Not all proofs share the same vocabulary. We seek for a proof at the level of the language semantics.
- You rarely have a full proof. You reason on a core subset of a language but interactions between all features may invalidate this proof.
- «matter of time»: do not confound decidability with tractability. Without a good methodology a mechanized proof can overwhelm human capacities.
Verified PL Stacks: Achievements

Some major achievements have changed our expectations about programming language mechanized proofs

- M6: JVM bytecode interpreter in ACL2 (Liu)
- Jinja: source & bytecode Java, compiler, BCV in Isabelle/HOL (Klein & Nipkow, and extensions by Lochbihler)
- CompCert: realistic C compiler in Coq (Leroy, Blazy et al.)
- Verified Software ToolChain: extension of CompCert for concurrent C programs (Appel et al.)
- seL4: verified OS kernel in Isabelle/HOL (Klein, Norrish et al.)
Verified Static Analysis: Objectives

We identify two major objectives

1. building new proof methods for the working (mechanized) semanticist
   • using Abstract Interpretation theory, we can provide generic interfaces between analyses
   • we have to discover new \textit{a posteriori} validation algorithms (sound and efficient)

2. building big proofs
   • proving in the small will not give us all the lessons we want to learn
   • large case studies are important to build a new \textit{proof engineering} knowledge
This *HDR*

Connecting the dots
This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots

This HDR
Connecting the dots
A Certified Denotational Abstract Interpreter

Abstract Interpretation proposes advanced techniques for static analysis of programs that raise specific challenges for machine-checked soundness proofs. Most classical dataflow analysis techniques involve some form of iteration strategies, be it recursive programs or fixed-point computations.

A key component of the formalization is the introduction of an intermediate semantics based on a generic least-fixpoint operator. This allows us to decompose the soundness proof in an elegant manner.

We show how we manage to program and prove correct in a proof assistant (Proof Pearl) an analysis whose implementation was partially supported by ANR-U3CAT grant and FNRAE ASCERT grant.

A significant example is the state-of-the-art certified static analysis for C [7] which has proven some critical safety properties for the primary flight control software of the Airbus A340 fly-by-wire system. Taking note of such a success, we discuss the next question: should we completely trust the analyser? In spite of the nice screen code for potential bugs, security vulnerabilities or unwanted behaviours.

Conclusion

Acknowledgments

References

Appendix A

Appendix B

Appendix C

Appendix D

Appendix E

Appendix F

Appendix G

Appendix H

Appendix I

Appendix J

Appendix K

Appendix L

Appendix M

Appendix N

Appendix O

Appendix P

Appendix Q

Appendix R

Appendix S

Appendix T

Appendix U

Appendix V

Appendix W

Appendix X

Appendix Y

Appendix Z

Appendix AA

Appendix BB

Appendix CC

Appendix DD

Appendix EE

Appendix FF

Appendix GG

Appendix HH

Appendix II

Appendix JJ

Appendix KK

Appendix LL

Appendix MM

Appendix NN

Appendix OO

Appendix PP

Appendix QQ

Appendix RR

Appendix SS

Appendix TT

Appendix UU

Appendix VV

Appendix WW

Appendix XX

Appendix YY

Appendix ZZ

Appendix AAA

Appendix BBB

Appendix CCC

Appendix DDD

Appendix EEE

Appendix FFF

Appendix GGG

Appendix HHH

Appendix III

Appendix JJJ

Appendix KKK

Appendix LLL

Appendix MMM

Appendix NNN

Appendix OOO

Appendix PPP

Appendix QQQ

Appendix RRR

Appendix SSS

Appendix TTT

Appendix UUU

Appendix VVV

Appendix WWW

Appendix XHTML

Appendix XML
Abstract

Interpretation

This HDR

Connecting the dots

2012

2011

2010

2009

2008

2007

2006

ESOP’07

TPHOLs’09

ITP’10

TGC’10

ESORICS’10

APLAS’10

ESOP’11

ESOP’12

Large Scale Proofs

Verified

Abstract

Interpretation
There are three main kinds of potential failures in a static analyser. The first is (un)soundness, i.e. whether one is (un)soundness, has been formally proved correct using a proof assistant. This gives raise to the notion of certified static analysis. The next question is: should we completely trust the analyser? In spite of the nice checked soundness proofs. Most classical dataflow analysis techniques it-iteration strategies are crucial when using such accelerating operators. Abstract. Interpretation proposes advanced techniques for mathematical theory of program analysis and the solid algorithmic techniques. This paper, we show how we manage to program and prove correct in a Certified Denotational Abstract Interpreter. In the next section, we present the design of the analyser and the soundness proof into the same logic of a proof assistant. Large Scale Proofs connecting the dots.
To eliminate this gap, it is possible to merge both the analyser implementation is proved correct on paper and the analyser that actually runs on the machine. On complete lattices and allows us to decompose the soundness proof in iteration strategies are crucial when using such accelerating operators widening and narrowing are used for accelerating the convergence. Smart checked soundness proofs. Most classical dataflow analysis techniques it-erate operators on lattices without infinite ascending chains. In contrast, 

There are three main kinds of potential failures in a static analyser. The first one is (un)soundness, when the analyser guarantees that a program is safe but it is not. The second one deals with termination, when the analyser loops for one problematic issue persists, has been formally proved correct using a proof assistant. 

A significant example is the state-of-the-art static analyser for C [7], the Astree has been formally proved correct using a proof assistant. The next question is: should we completely trust the analyser? In spite of the nice mathematical theory of program analysis and the solid algorithmic techniques available one problematic issue persists, 

David Cachera
IRISA / ENS Cachan (Bretagne), France

Work partially supported by ANR-U3CAT grant and FNRAE ASCERT grant.

A Certified Denotational Abstract Interpreter
This HDR
PhD supervision

Verified SSA
Verified Abstract Interpretation

Sawja
Large Scale Proofs
Collaborations

- 2012: Verified SSA
- 2011: Sawja
- 2010: Large Scale Proofs
- 2009: Verified Abstract Interpretation
- 2008: Large Scale Proofs
- 2007: Sawja
- 2006: Verified SSA
Collaborations

- Verified SSA
- Verified Abstract Interpretation
- Sawja
- European Mobius Project
- Large Scale Proofs
- COST European Action
Collaborations

- 2012: Verified SSA
- 2011: Sawja
- 2010: ANSSI & INRIA
- 2009: COST European Action
- 2008: European Mobius Project
- 2007: Large Scale Proofs
- 2006: Other collaboration
This Talk

Verified SSA

Sawja

Verified Abstract Interpretation

Large Scale Proofs

2012

2011

2010

2009

2008

2007

2006
This Talk

- Verified SSA
- Sawja
- Large Scale Proofs

Verified Abstract Interpretation
joint work with
F. Besson, T. Jensen,
D. Cachera, T. Turpin
Verified Abstract Interpretation

Extend my PhD work about mechanisation of Abstract Interpretation theory

Objectives: to embed more Abstraction Interpretation techniques inside mechanized proofs

Achievements

- *A posteriori* validation of relational abstract domains
- Advanced iteration strategies for widening/narrowing
- First mechanized proof that explicitly uses a collecting semantics
  - turn a standard operational semantics into a collecting interpreter
  - the interpreter is *aligned* with the static analyzer: easier soundness proof
Program Fixpoint Collect (i:stmt) (l:pp):
    monotone (powerset(env)) (pp → powerset(env)) :=
match i with
    | Assign p x e ⇒
    | While p t i ⇒
    | [...] end.
Program Fixpoint Collect (i:stmt) (l:pp):
    monotone (P(env)) (pp → P(env)) :=

match i with
    | Assign p x e ⇒
      | While p t i ⇒
      | [...]
end.
Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (P(env)) (pp \to P(env)) :=

match i with
  | Assign p x e ⇒

  | While p t i ⇒

  | [...] end.
Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (P(env)) (pp → P(env)) :=

match i with
  | Assign p x e ⇒

  | While p t i ⇒

  | [...]
end.
Into the Depths of a Coq Development

```
Program Fixpoint Collect (i: stmt) (l: pp):  
    monotone \( P(env) \) (pp \( \to \) \( P(env) \)) :=

match i with
    | Assign p x e ⇒
    | While p t i ⇒
    | [...] end.
```
Into the Depths of a Coq Development

**Program Fixpoint** Collect (i:stmt) (l:pp):

\[
\text{match } i \text{ with } \\
\mid \text{Assign } p x e \Rightarrow \\
\mid \text{While } p t i \Rightarrow \\
\mid […] \\
\text{end.}
\]

- recursive function with dependent types
- statement of a simple imperative language
- next program point after statement \(i\)
- monotone function from \(P(\text{env})\) to \(pp \to P(\text{env})\)

set of environments reachable before the statement
Program Fixpoint Collect (i:stmt) (l:pp): monotone (P(env)) (pp \rightarrow P(env)) :=

match i with
| Assign p x e \Rightarrow
| While p t i \Rightarrow
| [...]
end.
Into the Depths of a Coq Development

recursive function with dependent types

statement of a simple imperative language

next program point after statement $i$

monotone function from $\mathcal{P}(\text{env})$ to $pp \rightarrow \mathcal{P}(\text{env})$

Pattern Matching

```coq
Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (\mathcal{P}(\text{env})) (pp \rightarrow \mathcal{P}(\text{env})) :=

match i with
  | Assign p x e ⇒
    | [...]
  | While p t i ⇒
end.
```
Program Fixpoint Collect (i:stmt) (l:pp):
    monotone (P(env)) (pp → P(env)) :=
match i with
    | Assign p x e ⇒
        Mono (fun Env ⇒ ⊥ +[p ← Env] +[l ← assign x e Env]) _

    | [...] end.
Into the Depths of a Coq Development

Program Fixpoint Collect (i:stmt) (l:pp):

monotone (P(env)) (pp → P(env)) :=

match i with
| Assign p x e ⇒
  Mono (fun Env ⇒ ⊥ +[p → Env] +[l → assign x e Env] )
| [...] end.
Program Fixpoint Collect (i:stmt) (l:pp): 

match i with 
  | Assign p x e ⇒ 
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _ 

constructor of monotone functions

end.
Into the Depths of a Coq Development

Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (P(env)) (pp \rightarrow P(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p \mapsto Env] +[l \mapsto assign x e Env])
  | [...]
end.

constructor of monotone functions

(Mono f π): monotone A B
iff
  • f: A \rightarrow B
  • π is a proof term of « f monotone »
Into the Depths of a Coq Development

Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (℘(env)) (pp → ℘(env)) :=
machine i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ← Env] +[l ← assign x e Env])
  | [...] end.

constructor of monotone functions

(Mono f π) : monotone A B
iff
• f : A → B
• π is a proof term of « f monotone »
Program Fixpoint Collect (i:stmt) (l:pp):
monotone (P(env)) (pp → P(env)) :=
match i with
| Assign p x e ⇒
  Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
| [...] end.

constructor of monotone functions

the precondition is attached to p

hole for the monotonicity proof (automatically filled)

(Mono f π): monotone A B
iff
• f: A → B
• π is a proof term of « f monotone »
Into the Depths of a Coq Development

Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (℘(env)) (pp → ℘(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | [...] end.

constructor of monotone functions
the precondition is attached to p
the strongest postcondition of Env is attached to l
hole for the monotonicity proof (automatically filled)

(Mono f π): monotone A B
iff
• f: A → B
• π is a proof term of « f monotone »
Program Fixpoint Collect (i: stmt) (l: pp):
  monotone (℘(env)) (pp → ℘(env)) :=
match i with
  | Assign p x e ⇒
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | While p t i ⇒
    Mono (fun Env ⇒
      let I: ℘(env) := lfp (iter Env (Collect i p) t p) in
      (Collect i p (assert t I))
      +[p ↦ I] +[l ↦ assert (Not t) I]) _
  | [...] end.
Program Fixpoint Collect (i:stmt) (l:pp):
    monotone (P(env)) (pp \rightarrow P(env)) :=

match i with
    | Assign p x e ⇒
        Mono (fun Env ⇒ ⊥ +[p \mapsto Env] +[l \mapsto assign x e Env]) _
    | While p t i ⇒
        Mono (fun Env ⇒
            let I:P(env) := lfp (iter Env (Collect i p) t p) in
                (Collect i p (assert t I))
                +[p \mapsto I] +[l \mapsto assert (Not t) I]) _
    | [...] end.
Program Fixpoint Collect (i:stmt) (l:pp):  
    monotone (P(env)) (pp → P(env)) :=  
match i with  
| Assign p x e ⇒  
    Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _  
| While p t i ⇒  
    Mono (fun Env ⇒  
        let I:P(env) := lfp (iter Env (Collect i p) t p) in  
            (Collect i p (assert t I))  
            +[p ↦ I] +[l ↦ assert (Not t) I]) _  
| [...]  
end.
Program Fixpoint Collect (i:stmt) (l:pp):
  monotone (P(env)) (pp → P(env)) :=
match i with
| Assign p x e ⇒
  Mono (fun Env ⇒ ⊥ +[p ↦ Env] +[l ↦ assign x e Env])
| While p t i ⇒
  Mono (fun Env ⇒
    let I: P(env) := lfp (iter Env (Collect i p) t p) in
    (Collect i p (assert t I))
    +[p ↦ I] +[l ↦ assert (Not t) I])
| [...]
end.
Into the Depths of a Coq Development

Program Fixpoint Collect (i:stmt) (l:pp):
    monotone (P(env)) (pp \rightarrow P(env)) :=
match i with
| Assign p x e \Rightarrow
    Mono (fun Env \Rightarrow \bot +[p \mapsto Env] +[l \mapsto assign x e Env]) \_ 
| While p t i \Rightarrow
    Mono (fun Env \Rightarrow
        let I:P(env) := lfp (iter Env (Collect i p) t p) in
        (Collect i p (assert t I))
        +[p \mapsto I] +[l \mapsto assert (Not t) I]) \_ 
| [...]
end.

loop invariant
least fixpoint on complete lattices
builds the fixpoint equation

I == Env \cup (Collect i p (assert t I) p)
Verified Abstract Interpretation

Lessons learned

Abstract Interpretation is still marginally employed in PL mechanized proofs
  • Not always the fastest path to prove an analysis
  • Lack of good tutorials

Still a rewarding proof technique
  • It gives a better understanding of an analysis (collecting semantics)
  • Putting everything in a same framework facilitates combination of analyses

Current limitations
  • Still no symbolic derivation of a best abstract transformer
  • Hard to draw definite conclusions about the scalability of the a posteriori approach on large programs: need to test this ideas on a more realistic langage
This Talk

Verified SSA

Sawja

Large Scale Proofs
joint work with G. Barthe, F. Dabrowski, T. Rezk

Verified Abstract Interpretation
Large Scale Proofs

Objectives:
• Go beyond the state of the art in terms of the complexity of the mechanized static analysis
• Attempt to capture large fragments of the Java semantics

Achievements:
• Soundness of an information flow type checker for Java bytecode
• Soundness of a data race analysis for Java bytecode
Information Flow Type Checker
Information Flow Type Checker

Objectives

• Proving non-interference of Java bytecode programs
• Public outputs should not depend on secret inputs
Objectives

• Proving non-interference of Java bytecode programs
• Public outputs should not depend on secret inputs
• Non-interference can be enforced by a type system

[D. Volpano and G. Smith, A Type-Based Approach to Program Security, Theory and Practice of Software Development, 1997.]

\[
\begin{array}{ll}
\text{CONST} & \vdash n : L \\
\text{VAR} & \vdash x \in \mathbb{V}_\tau \vdash x : \tau \\
\text{BINOP} & \vdash e_1 : \tau \vdash e_2 : \tau \\
\text{EXP-SUBTYP} & \vdash e : \tau_1 \vdash \tau_1 \sqsubseteq \tau_2 \\
\text{ASSIGN} & \vdash x \in \mathbb{V}_\tau \vdash e : \tau \\
\text{SEQ} & \vdash S_1 : \tau \vdash S_2 : \tau \\
\text{IF} & \vdash e : \tau \vdash S_1 : \tau \vdash S_2 : \tau \\
\text{WHILE} & \vdash e : \tau \vdash S : \tau \\
\text{STM-SUBTYP} & \vdash S : \tau_2 \vdash \tau_1 \sqsubseteq \tau_2 \\
\end{array}
\]
Information Flow Type Checker

Objectives

- Proving non-interference of Java bytecode programs
- Public outputs should not depend on secret inputs
- Non-interference can be enforced by a type system

[Volpano97]

Achievements: mechanized proof of a type checker for a bytecode language handling

- unstructured control flow
- operand stack
- exceptions
- objects and array dynamically allocated
- classes and virtual method calls

66 typing rules...
Data Race Analysis
A Challenging Example

class List{ T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            temp.val = new T();
            temp.val.f = new A();
            temp.next = l;
            l = temp
        }
        while (*) {
            T t = new T();
            t.data = l;
            t.start();
            t.f = ...;
            return;
        }
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*){
            List m = this.data;
            while (*) { m = m.next; }
            synchronized(m){ m.val.f = ...;}
            return;
        }
    }
}
## Data Race Analysis

### A Challenging Example

```java
class List { T val; List next; }

class Main() {
    void main() {
        List l = null;
        while (*) {
            List temp = new List();
            1:    temp.val = new T();
            2:    temp.val.f = new A();
            3:    temp.next = l;
                  l = temp }
        while (*) {
            T t = new T();
            4:    t.data = l;
                  t.start();
            5:    t.f = ...;
        }
        return;
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run() {
        while (*) {
            6:    List m = this.data;
            7:    while (*) { m = m.next; }
            8:    synchronized(m) { m.val.f = ...; }
        }
        return;
    }
}
```

1. We create a link list `l`

Threads: M

1. `temp` creates a new list and assigns it to `l`.
2. `T t` creates a new thread and assigns the list to `t.data`.
3. The thread starts and assigns a value to `t.f`.
4. The loop continues, and the thread repeats the process.

The diagram shows the link list and thread interactions, highlighting the potential data race condition where `t.f` might overwrite the value assigned by `temp.val.f`.
Data Race Analysis
A Challenging Example

```java
class List { T val; List next; }

class Main() {
    void main() {
        List l = null;
        while (*) {
            List temp = new List();
            1:    temp.val = new T();
            2:    temp.val.f = new A();
            3:    temp.next = l;
            l = temp
        }
        while (*) {
            T t = new T();
            4:    t.data = l;
            5:    t.start();
        }
        return;
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run() {
        while (*) {
            6:    List m = this.data;
            7:    while (*) { m = m.next; }
            8:    synchronized(m) { m.val.f = ...; }
        }
        return;
    }
}
```

1. We create a link list \( l \)
2. We create several threads that all share the list \( l \)
Data Race Analysis
A Challenging Example

class List { T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            temp.val = new T();
            temp.val.f = new A();
            temp.next = l;
            l = temp
        }
        while (*) {
            T t = new T();
            t.data = l;
            t.start();
            t.f = ...;
        }
        return;
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*){
            List m = this.data;
            while (*) { m = m.next; }
            synchronized(m){ m.val.f = ...;}
        }
        return;}
}

1. We create a link list \( l \)

2. We create several threads that all share the list \( l \)

3. Each thread chooses a cell, takes a lock on it and updates it.
Data Race Analysis
A Challenging Example

class List{ T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            1: temp.val = new T();
            2: temp.val.f = new A();
            3: temp.next = l;
            l = temp }
        while (*) {
            T t = new T();
            4: t.data = l;
            t.start();
            5: t.f = ...;}
        return;
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*){
            6: List m = this.data;
            7: while (*) { m = m.next; }
            8: synchronized(m){ m.val.f = ...;}
        return;}}

1. We create a link list l
2. We create several threads that all share the list l
3. Each thread chooses a cell, takes a lock on it and updates it.

Threads: M T1 T2 ... Tn
Data Race Analysis
A Challenging Example

class List{ T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            1:    temp.val = new T();
            2:    temp.val.f = new A();
            3:    temp.next = l;
                 l = temp }
        while (*) {
            T t = new T();
            4:    t.data = l;
                 t.start();
            5:    t.f = ...;}
        return;
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*){
            6:    List m = this.data;
            7:    while (*) { m = m.next; }
            8:    synchronized(m){ m.val.f = ...;}}
        return;}}

Definition [Data Race]
• The situation where two different processes attempt to access to the same memory location and at least one access is a write.

Question
• Is this program data race free?
Effective Static Data Race Detection for Java
Naik’s PhD 2008

```java
class List { T val; List next; }

class Main() {
    void main() {
        List l = null;
        while (*) {
            List temp = new List();
            1: temp.val = new T();
            2: temp.val.f = new A();
            3: temp.next = l;
            l = temp }
        while (*) {
            T t = new T();
            4: t.data = l;
            t.start();
            5: t.f = ...;}
        return;
    }
}

class T extends java.lang.Thread {
    A f;
    List data;
    void run() {
        while (*) {
            List m = this.data;
            6: m = m.next; }
            synchronized(m) {
                m.val.f = ...;}
        return;
    }
}
```
Effective Static Data Race Detection for Java
Naik’s PhD 2008

Original pairs

Reachable pairs

Aliasing pairs

Escaping pairs

Unlocked pairs

The first set of potential races is based on field safety

(1,val,1)  (1,val,2)  (2,f,2)  (3,next,3)
(4,data,4)  (5,f,5)  (2,f,5)  (5,f,8)  (4,data,6)
(3,next,7)  (1,val,8)  (2,f,8)  (8,f,8)

class List{ T val; List next; } 

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            1:    temp.val = new T();
            2:    temp.val.f = new A();
            3:    temp.next = l;
                    l = temp }
        while (*) { 
            T t = new T();
        4:    t.data = l;
                t.start();
        5:    t.f = ...;}
        return;
    }
}

class T extends java.lang.Thread { 
    A f;
    List data;
    void run(){
        while(*){
            6:    List m = this.data;
            7:    while (*) { m = m.next; }
            8:    synchronized(m){ m.val.f = ...;}
                    return;}}

(1,val,1)  (1,val,2)  (2,f,2)  (3,next,3)
(4,data,4)  (5,f,5)  (2,f,5)  (5,f,8)  (4,data,6)
(3,next,7)  (1,val,8)  (2,f,8)  (8,f,8)
Effective Static Data Race Detection for Java
Naik's PhD 2008

class List{ T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            1:    temp.val = new T();
            2:    temp.val.f = new A();
            3:    temp.next = l;
            l = temp }
        while (*) {
            T t = new T();
            4:    t.data = l;
            t.start();
            5:    t.f = ...;}
        return;
    }
}

class T extends java.lang.Thread {
    A.f;
    List data;
    void run(){
        while(*){
            6:    List m = this.data;
            7:    while (*) { m = m.next; }
            8:    synchronized(m){ m.val.f = ...;}
            return;}}
Effective Static Data Race Detection for Java
Naik’s PhD 2008

Using a points-to abstraction we compute the pairs that may touch the same heap location

Original pairs:
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Reachable pairs:
(1, X, 1) (1, X, 2) (2, X, 2) (3, X, X, 3)
(4, data, 4) (5, X, 5) (2, f, X, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Aliasing pairs:
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, X, X, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Escaping pairs:

Unlocked pairs:

class List{ T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            1:    temp.val = new T();
            2:    temp.val.f = new A();
            3:    temp.next = l;
            l = temp }
        while (*) {
            T t = new T();
            4:    t.data = l;
            t.start();
            5:    t.f = ...;}
        return;
    }
}

class T extends java.lang.Thread { 
    A f;
    List data;
    void run(){
        while(*){
            6:    List m = this.data;
            7:    while (*) { m = m.next; }
            8:    synchronized(m){ m.val.f = ...;}}
        return;}}
Effective Static Data Race Detection for Java
Naik’s PhD 2008

<table>
<thead>
<tr>
<th>Original pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)</td>
</tr>
<tr>
<td>(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)</td>
</tr>
<tr>
<td>(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reachable pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)</td>
</tr>
<tr>
<td>(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)</td>
</tr>
<tr>
<td>(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Aliasing pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)</td>
</tr>
<tr>
<td>(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)</td>
</tr>
<tr>
<td>(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Escaping pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)</td>
</tr>
<tr>
<td>(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)</td>
</tr>
<tr>
<td>(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unlocked pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>The pairs of program points where the target location may be shared at that points</td>
</tr>
</tbody>
</table>

```java
class List{ T val; List next; }
class Main() {
    void main(){
        List l = null;
        while (*){
            List temp = new List();
            temp.val = new T();
            temp.val.f = new A();
            temp.next = l;
            l = temp 
        }
        return;
    }
}
class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*){
            List m = this.data;
            while (*) { m = m.next; }
            synchronized(m){ m.val.f = ...;}
            return;}}
```
Effective Static Data Race Detection for Java
Naik’s PhD 2008

Original pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Reachable pairs
(1, x, 1) (1, x, 2) (2, x) (3, x, t, 3)
(4, o, a, 4) (5, f, 5) (2, x, 5) (5, f, 6) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Aliasing pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, x, 5) (5, x, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Escaping pairs
(1, x, 1) (1, x, 2) (2, x) (3, x, t, 3)
(4, o, a, 4) (5, f, 5) (2, x, 5) (5, f, 8) (4, o, a, 6)
(3, next, 7) (1, x, 8) (2, x, 8) (8, f, 8)

Unlocked pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, x, 8)

class List { T val; List next; }

class Main() {
    void main(){
        List l = null;
        while (*) {
            List temp = new List();
            1: temp.val = new T();
            2: temp.val.f = new A();
            3: temp.next = l;
            l = temp }
        while (*) {
            T t = new T();
            4: t.data = l;
            t.start();
            5: t.f = ...;}
        return;
    }
}
class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*){
            6: List m = this.data;
            7: while (*) { m = m.next; }
            8: synchronized(m){ m.val.f = ...;}}
}
Effective Static Data Race Detection for Java
Naik’s PhD 2008

Original pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Reachable pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Aliasing pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Escaping pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

Unlocked pairs
(1, val, 1) (1, val, 2) (2, f, 2) (3, next, 3)
(4, data, 4) (5, f, 5) (2, f, 5) (5, f, 8) (4, data, 6)
(3, next, 7) (1, val, 8) (2, f, 8) (8, f, 8)

If each original pair has been removed at least one time, the program is data-race free.
Effective Static Data Race Detection for Java
Naik’s PhD 2008

Original pairs

Reachable pairs

Aliasing pairs

Escaping pairs

Unlocked pairs

If each original pair has been removed at least one time, the program is data-race free.

This program is data-race free!
Proof Architecture

Original pairs
Reachable pairs
Aliasing pairs
Escaping pairs
Unlocked pairs

Potential Races

Static Analyses

Points-to Analysis
Must-Lock Analysis
Must-Not Thread Escape Analysis
Conditional Must-Not Alias Analysis

Standard Semantics
Points-to Semantics
Counting Semantics

 Semantics

: sound w.r.t.
: safe instrumed w.r.t.
: use the result of
Lessons learned

- Complex analyses are generally a composition of several sub-analyses. Mechanized proof make explicit the interactions between them.
- Directly analyzing bytecode programs is a methodological mistake. About 1.5 man year effort and 15K loc for each proof...

Conclusions

- Verified static analysis should learn from what others do with (traditional) static analysis platforms.
Traditional Static Analysis Platforms

Standard architecture of a (Java) static analysis platform
• A parser
• An IR (stackless, SSA)
• A generic fixpoint solver
• Some examples of analysis (control flow analysis is almost mandatory for Java)

Well known platforms: Soot, Wala

Problem: programmed in Java (mutable internal states, visitor patterns...)

Reboot in Coq?...
Traditional Static Analysis Platforms

Standard architecture of a (Java) static analysis platform
• A parser
• An IR (stackless, SSA)
• A generic fixpoint solver
• Some examples of analysis (control flow analysis is almost mandatory for Java)

Well known platforms: Soot, Wala

Problem: programmed in Java (mutable internal states, visitor patterns...)

Reboot in Coq?... or let’s start with OCaml first!
This Talk

Sawja
joint work with N. Barré, F. Besson, D. Demange, T. Jensen, L. Hubert, F. Kirchner, V. Monfort, T. Turpin, P. Vittet

Verified SSA

Large Scale Proofs

Verified Abstract Interpretation
Sawja (Static Analysis Workshop for JAva)
http://sawja.inria.fr

An efficient OCaml library to analyze Java bytecode programs
• A parser generates a hash-consed program representation
• BIR: stackless intermediate representation (with/without SSA)
• Fixpoint solvers (worklist, Bourdoncle iterations)
• Various control flow analyses

Applications: static analyses to strengthen the security of Java programs
• Secure initialisation of objects [ESORICS’10]
• Secure copying of objects [ESOP’11]

Each time a similar methodology
• We prove correct in Coq the analysis on a core language close to BIR
• We implement the analysis with Sawja and run large scale experiments on Java programs to check that our analyses are precise enough
Sawja
Lessons learned

- OCaml + BIR + generic fixpoint solvers make the prototyping of static analysis really easy (3 man month)
- Proving a static analysis on a core language is relatively easy (2 man month)
- An OCaml static analysis platform is competitive with Java platform

From OCaml to Coq?
Lessons learned

• OCaml + BIR + generic fixpoint solvers make the prototyping of static analysis really easy (3 man month)
• Proving a static analysis on a core language is relatively easy (2 man month)
• An OCaml static analysis platform is competitive with Java platform

From OCaml to Coq?

Most proof challenges rely in the generation of IR
• stackless representation of bytecode (paper proof) [APLAS’11]
• SSA form (Coq proof) [ESOP’12]
Sawja

Lessons learned
• OCaml + BIR + generic fixpoint solvers make the prototyping of static analysis really easy (3 man month)
• Proving a static analysis on a core language is relatively easy (2 man month)
• An OCaml static analysis platform is competitive with Java platform

From OCaml to Coq?

Most proof challenges rely in the generation of IR
• stackless representation of bytecode (paper proof) [APLAS’11]
• SSA form (Coq proof) [ESOP’12]

The last challenge for Sawja is concurrency... (stay tuned)
This Talk

Verified SSA
joint work with G. Barthe, D. Demange

Sawja

Large Scale Proofs

Verified Abstract Interpretation
The Static Single Assignment (SSA) representation is widely used in modern compilation/analysis platforms but it has never been used in formal proofs.

We provide

- a formal semantics of SSA that follows closely (but in a rigorous way) the informal semantics used in the literature
- a verified SSA generator, using the most efficient algorithm of the literature and an original \textit{a posteriori} validator
- a mechanized proof of an emblematic SSA-based static analysis (Global Value Numbering)
- an integration into the CompCert C compiler
Verified SSA Representation

Lessons learned

- Compiler gurus are (almost) right. Our work make precise their informal explanations

In SSA, «If two expressions are textually the same, they are sure to evaluate the same result»

- As we did for abstract interpretation we try not to reinvent the wheel in our mechanized proofs to make formal proofs benefit from past research

- In return, we expect our work will help those who need to understand these theories
Conclusions

We have explored various virgin lands in the area of verified static analyses
- numerical relational abstract interpreter
- alias analysis
- concurrency
- information flow
- shape
- SSA-based

This variety gives a better understanding of
- the high potential of this approach,
- but also its limits (human effort)

In reaction to these limits, we have built a platform that
- gives the foundations of a verified static analysis platform
- provides a semantically-grounded platform to the static analysis community
- has been applied to design new security mechanisms for Java
Perspectives

- 2012: Verified SSA
- 2011: Sawja
- 2010: Large Scale Proofs
- 2009: Verified Abstract Interpretation
Perspectives

1. A Verified CompCert C Analyzer

2. A Verified Software Toolchain for Java
A Verified CompCert C Analyzer
The Verasco Project (Univ. Rennes, Inria Paris & Saclay, Verimag, Airbus)

Verified abstract interpretation: it’s time to scale these techniques to a realistic tool
• static analysis inside CompCert
• targets the formal verification of some of the key components of the Astrée tool
  • interprocedural
  • abstraction of the memory
  • relational numerical abstraction both for machine integers and floats

Preliminary results:
• an intra-procedural interval analysis of integers on the RTL CompCert IR and its application for a WCET analysis (joint work with A. Maroneze and S. Blazy)
Perspectives

1. A Verified CompCert C Analyzer
2. A Verified Software Toolchain for Java
A Last Challenge: Concurrency

We need to give a semantics to the various Sawja IRs (bytecode, BIR)

- May seem a easy task if we build on top of the previous mechanized semantics (M6, Jinja, Bicolano)
- (Un)fortunately much research remains to be done when formalizing the semantics of concurrent Java programs...
A Last Challenge: Concurrency

We need to give a semantics to the various Sawja IRs (bytecode, BIR)

- May seem a easy task if we build on top of the previous mechanized semantics (M6, Jinja, Bicolano)
- (Un)fortunately much research remains to be done when formalizing the semantics of concurrent Java programs...

Java Memory Model (JMM), the official specification

- Major revision in 2004 (JSR-133)
- Guarantees for programmers
  - A (safe) formal semantics for all Java programs (incl. those with races)
  - Data-race free programs execute like in a interleaving model (SC)
- Guarantees for optimizers
  - Allows all various hardware reordering semantics
  - Allows aggressive compiler optimizations
A Last Challenge: Concurrency

We need to give a semantics to the various Sawja IRs (bytecode, BIR)

- May seem a easy task if we build on top of the previous mechanized semantics (M6, Jinja, Bicolano)
- (Un)fortunately much research remains to be done when formalizing the semantics of concurrent Java programs...

Java Memory Model (JMM), the official specification

- Major revision in 2004 (JSR-133)
- Guarantees for programmers
  - A (safe) formal semantics for all Java programs (incl. those with races)
  - Data-race free programs execute like in a interleaving model (SC)
- Guarantees for optimizers
  - Allows all various hardware reordering semantics
  - Allows aggressive compiler optimizations
Bad News
Bad News

JMM is formally broken

- The initial definitions and theorems were flawed
- Aspinall, Sevcik [ECOOP’08] and later Lochbihler [ESOP’12] provides formal patches in order to patch the guarantees for the programmers
- but nobody has fully patched/proved the guarantees for the optimizers
Bad News

JMM is formally broken

- The initial definitions and theorems were flawed
- Aspinall, Sevcik [ECOOP’08] and later Lochbihler [ESOP’12] provides formal patches in order to patch the guarantees for the programmers
- but nobody has fully patched/proved the guarantees for the optimizers
Bad News

JMM is formally broken
- The initial definitions and theorems were flawed
- Aspinall, Sevcik [ECOOP’08] and later Lochbihler [ESOP’12] provides formal patches in order to patch the guarantees for the programmers
- but nobody has fully patched/proved the guarantees for the optimizers

To patch the JMM properly, we need to link, in a same formal proof
- hardware semantics with weak memory models
- formalization of aggressive compiler optimizations

So we need to build a formally-verified Java compiler too!
Perspectives

- Verified Java Toolchain
- A Verified CompCert C Analyzer
- Verified SSA
- Verified Abstract Interpretation
- Sawja
- Large Scale Proofs
Perspectives

Verified Software Toolchain for Java
- a verified static analysis platform
- a formal definition of (a new) JMM
- a verified implementation of JMM

A verified Java compiler is full of challenges
- aggressive optimizations
- concurrent implementations of
  - monitors
  - data-structures
  - garbage collector
- Just In Time compiler
Perspectives

Verified Software Toolchain for Java
- a verified static analysis platform
- a formal definition of (a new) JMM
- a verified implementation of JMM

A verified Java compiler is full of challenges
- aggressive optimizations
- concurrent implementations of
  - monitors
  - data-structures
  - garbage collector
- Just In Time compiler

First step achieved during my stay at Purdue University:
- we provide an intermediate semantic step between JMM and x86