Type Safe Nondeterminism
A Formal Semantics of Java Threads

Andreas Lochbihler

University of Passau
Germany

01/13/2008

Funded by DFG grant Sn11/10-1

Overview

1. Motivation
2. Java threads
3. Formalisation
   - The Jinja and framework semantics
   - Deadlock vs. progress
   - Type safety for Jinja
4. Summary

The goal

Our goal:

- Formalise Java thread semantics
- Show type safety
- In a theorem prover

Benefits: Solid basis for formal verification problems

- Language based security (LBS)
- Proof carrying code (PCC)

Starting point: Jinja semantics (Nipkow, Klein, TOPLAS'06)

Type safety

Type safety

- Well-typed programs evaluate fully and
- No untrapped errors can occur

Proof technique (Wright, Felleisen '94):

Progress Semantics cannot get stuck
(as long as some threads are not deadlocked yet)

Preservation Evaluating a well-typed statement results in another well-typed statement with equal or smaller type

Challenge:
Deadlock can break progress property

Java thread features

- Dual nature of threads:
  - Objects of class `Thread`
  - Execution contexts spawned by `start()`
- Communication via shared memory
- Synchronization via locking
- Deadlocks to break progress
- Synthesized methods in `Object`:
  - `wait()`
  - `notify()`
  - `notifyAll()`

Java deadlock example

Thread (I)

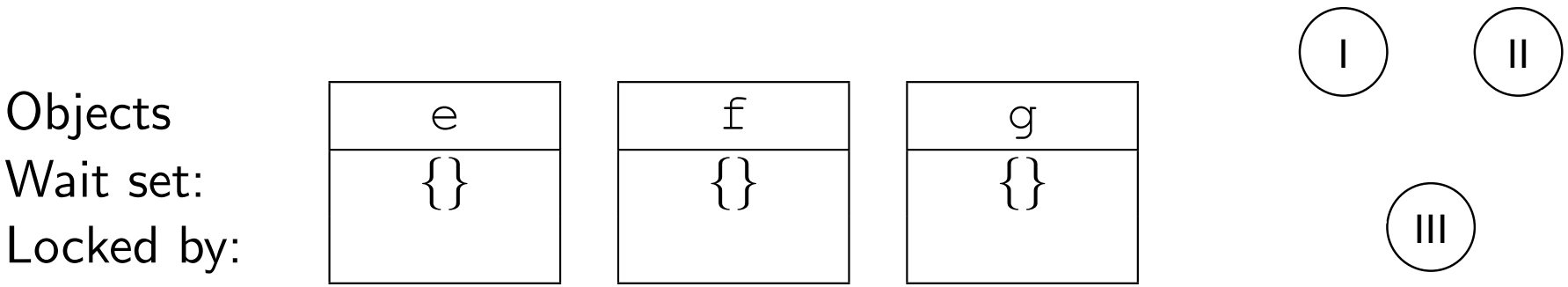
```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

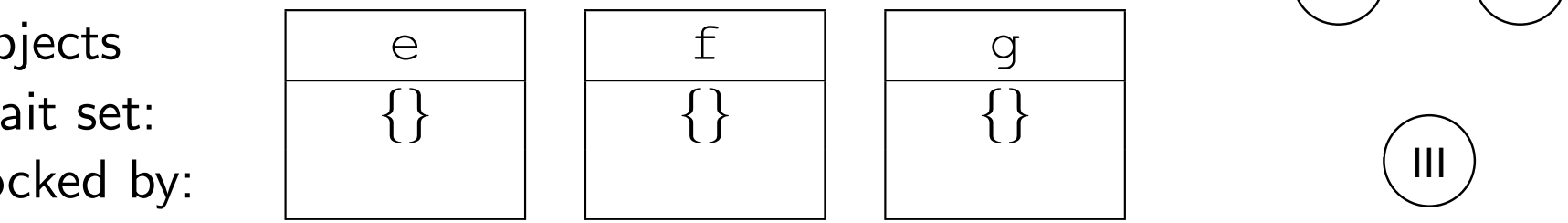
Request lock on `f`

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example

Thread (I)

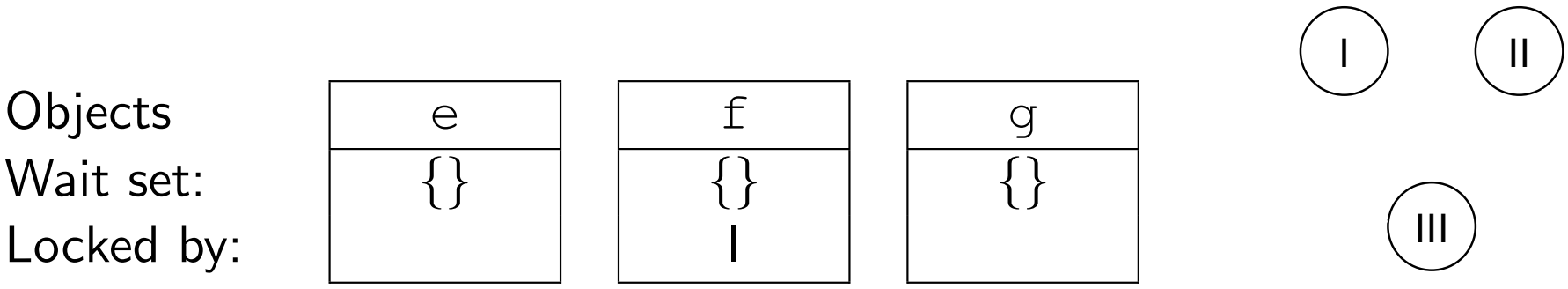
```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

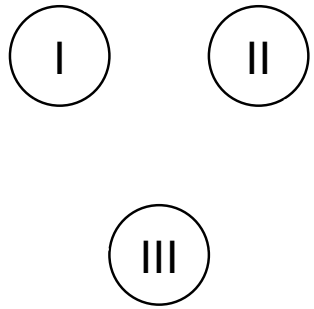
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Request lock on `g`

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

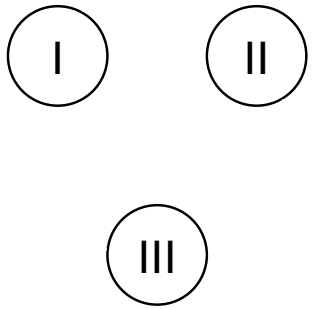
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

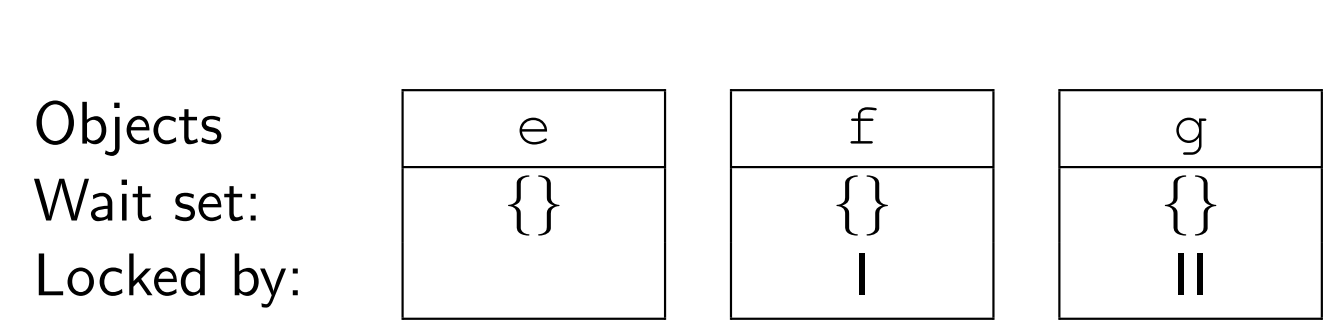
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `e`



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

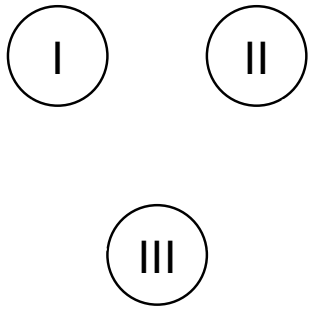
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on `g`

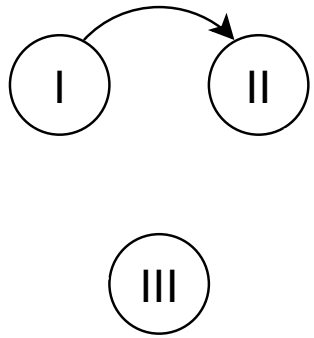
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on `g`

Thread (II)

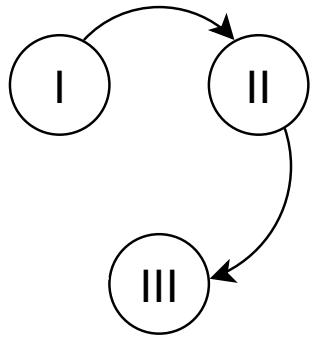
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Request lock on `e`

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on `g`

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

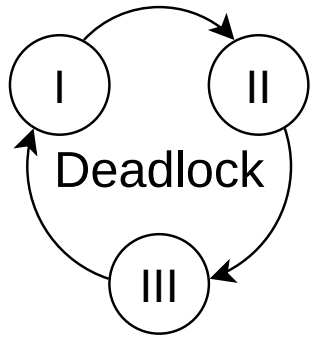
Request lock on `e`

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example with wait sets

Thread (I)

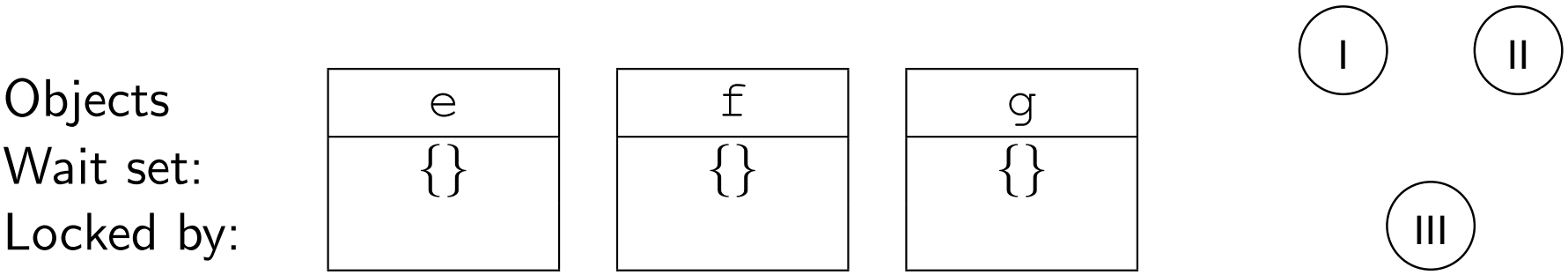
```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

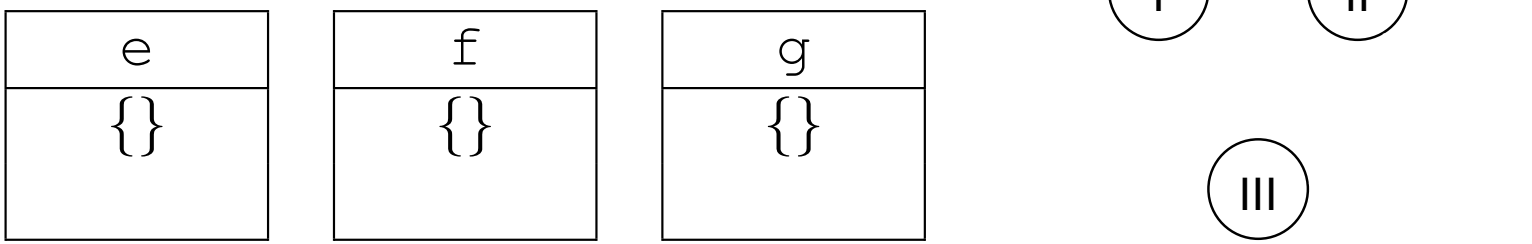
Request lock on `f`

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example with wait sets

Thread (I)

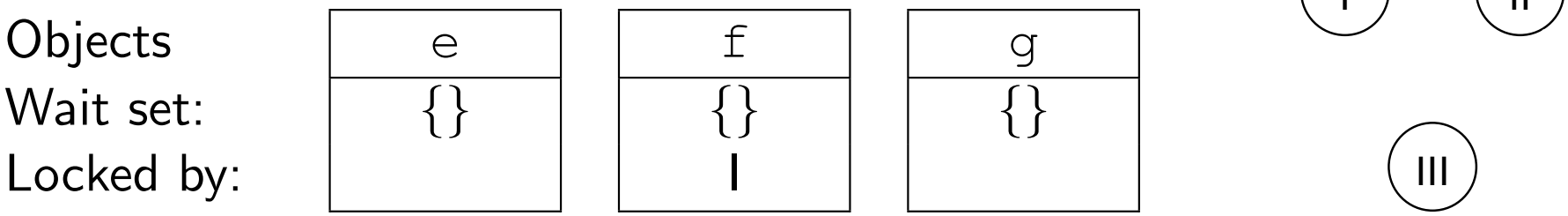
```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

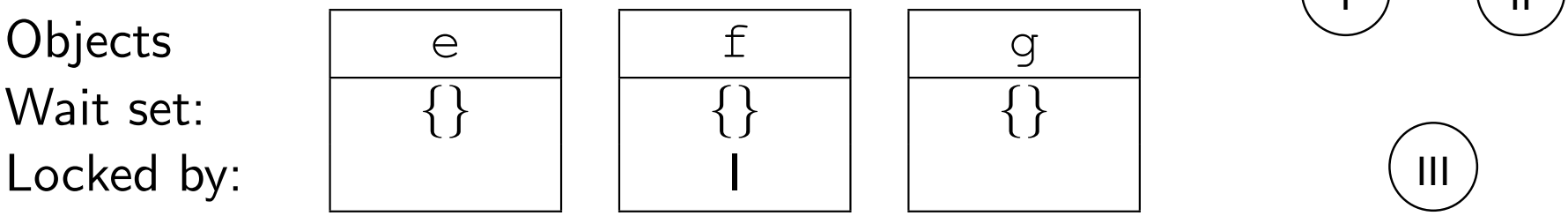
Request lock on g

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example with wait sets

Thread (I)

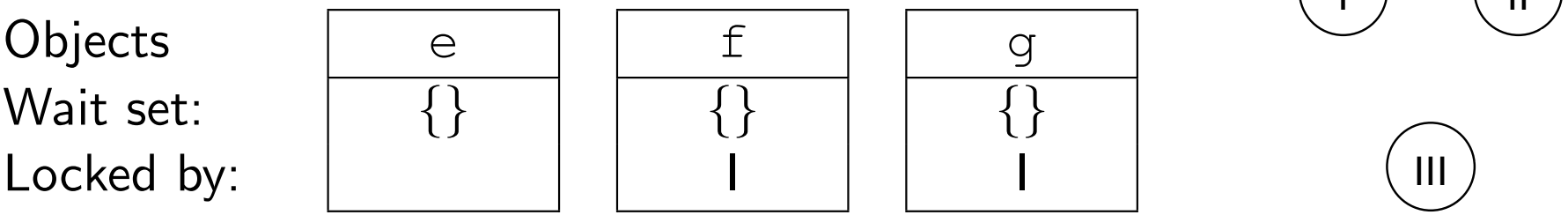
```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example with wait sets

Thread (I)

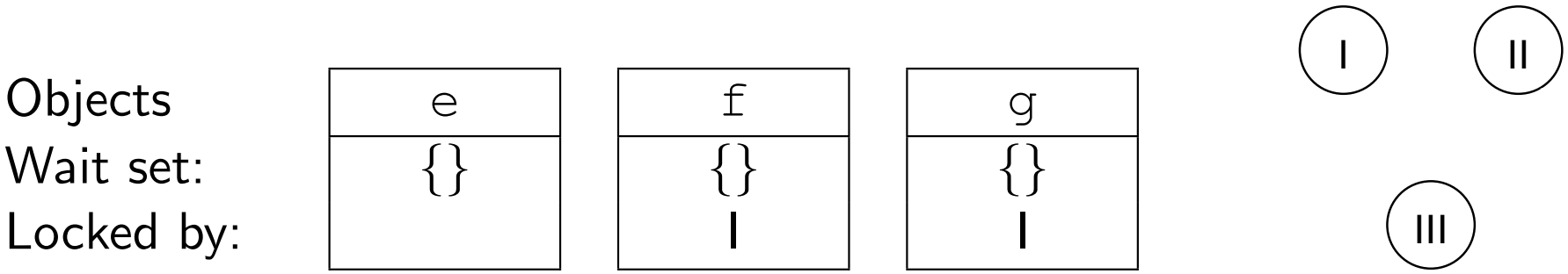
```
synchronized (f) {
  synchronized (g) {
    •••
    g.wait();
    ...
  }
}
```

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

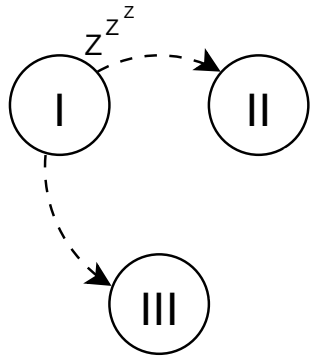
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:		I	



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

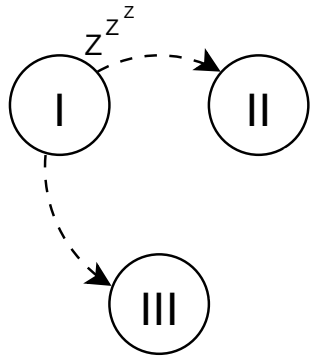
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `e`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:		I	



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

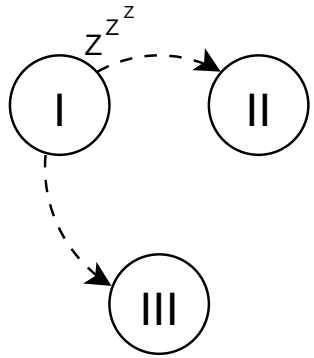
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

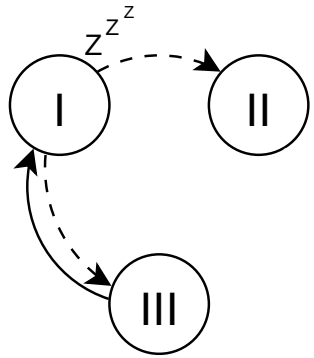
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

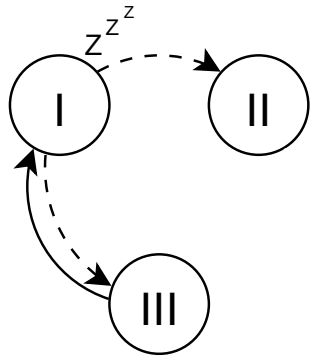
Request lock on `g`

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

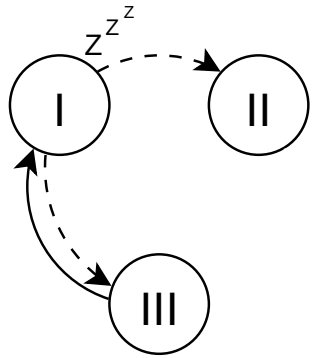
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	II



Java deadlock example with wait sets

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

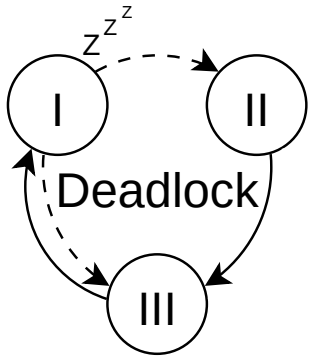
Request lock on `e`

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	II



The Jinja project (Nipkow, Klein, TOPLAS '06)

Formal semantics for a Java subset in Isabelle/HOL:

Program operations:

- Object creation
- Casts
- Literal values
- Binary operators
- Variable access and assignment
- Field access and assignment
- Method call
- Sequential composition
- If-then-else, while
- Blocks with local variables
- Exception throwing and handling

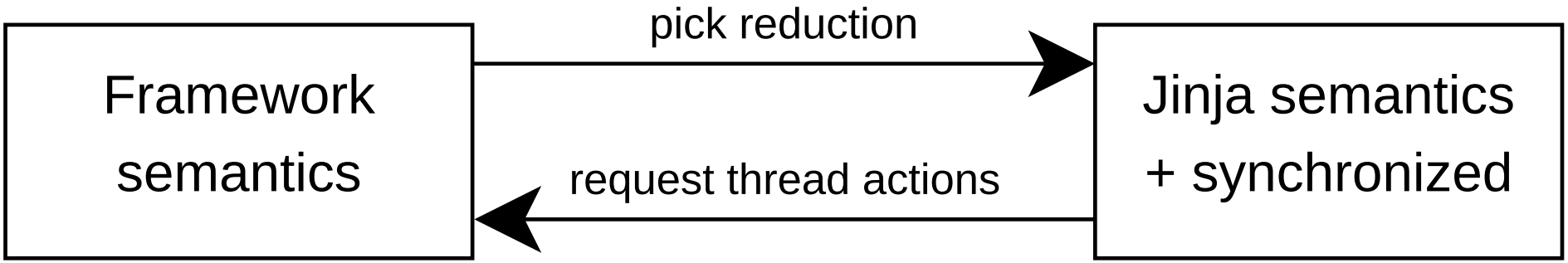
Jinja source code:

- Operational semantics
- Equivalence for small-step and big-step semantics
- Type safety proof (progress and preservation)

Bytecode:

- Jinja Virtual Machine
- Bytecode verifier
- Compiler from source to bytecode

Framework semantics



Management of

- Locks
- Threads
- Wait sets

Select thread and one of its reductions such that the thread actions are feasible

Thread actions for

- Locking and unlocking
- Thread spawning
- Wait and notify

Single-thread semantics

- Reduction with list of thread actions
- Type system

Modularity: Separation of thread issues from low-level Java details

Syntax

Thread actions

Locking	*Lock a*, *Unlock a*, *UnlockFail a*
Spawning	*NewThread t e h x*, *NewThreadFail*
Wait sets	*Suspend a*, *Notify a*, *NotifyAll a*

Reduction notation:

$P \vdash \langle e, (h, x)\rangle \xrightarrow{tas}\triangleright \langle e', (h', x')\rangle$	Jinja semantics
$P \vdash \langle ls \mid es, h \mid ws\rangle \xrightarrow{t:tas} \langle ls' \mid es', h' \mid ws'\rangle$	Framework semantics
$P \vdash \langle ls \mid es, h \mid ws\rangle \xrightarrow{ttas}{}^{\star} \langle ls' \mid es', h' \mid ws'\rangle$	Transitive, reflexive closure

Example reduction rules in Jinja

Thread.`start()`:

$$\frac{P \vdash C \preceq^* \mathit{Thread} \qquad h\ a = \mathit{Obj}\ C\ \mathit{fs} \qquad \mathit{ta} = \mathit{NewThread}\ t\ (\mathit{Var}\ \mathit{this}{\cdot}\mathit{run}())\ h\ [\mathit{this} \mapsto \mathit{Addr}\ a]}{P \vdash \langle \mathit{addr}\ a{\cdot}\mathit{start}(), (h, x)\rangle \xrightarrow{[\mathit{ta}]} \langle \mathit{unit}, (h, x)\rangle}$$

Object.`wait()`:

$$\frac{h\ a = q}{P \vdash \langle \mathit{addr}\ a{\cdot}\mathit{wait}(), (h, x)\rangle \xrightarrow{[\mathit{Suspend}\ a, \mathit{Unlock}\ a, \mathit{Lock}\ a]} \langle \mathit{unit}, (h, x)\rangle}$$

Monitor locking:

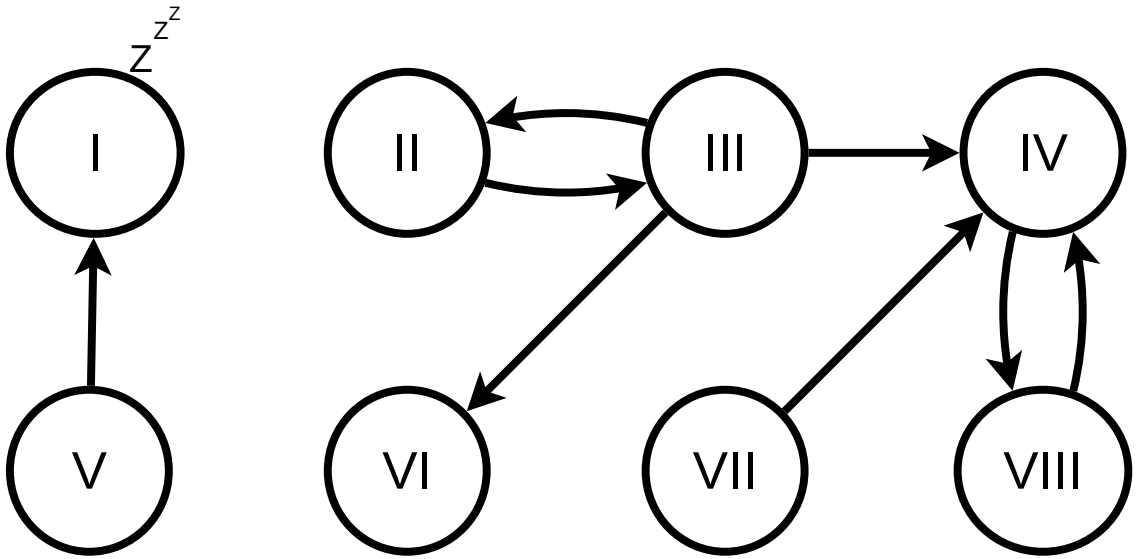
$$P \vdash \langle \mathit{sync}(\mathit{addr}\ a)\ e, s\rangle \xrightarrow{[\mathit{Lock}\ a]} \langle \mathit{sync}(\mathit{locked}(a))\ e, s\rangle$$

Unlocking at calls to `wait()`:

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\mathit{tas}} \langle e', s'\rangle \qquad \mathit{tas} = \mathit{Suspend}\ a{\cdot}\mathit{tas}'}{P \vdash \langle \mathit{sync}(\mathit{locked}(a))\ e, s\rangle \xrightarrow{\mathit{tas}\ @\ [\mathit{Unlock}\ a]} \langle \mathit{sync}(\mathit{addr}\ a)\ e', s'\rangle}$$

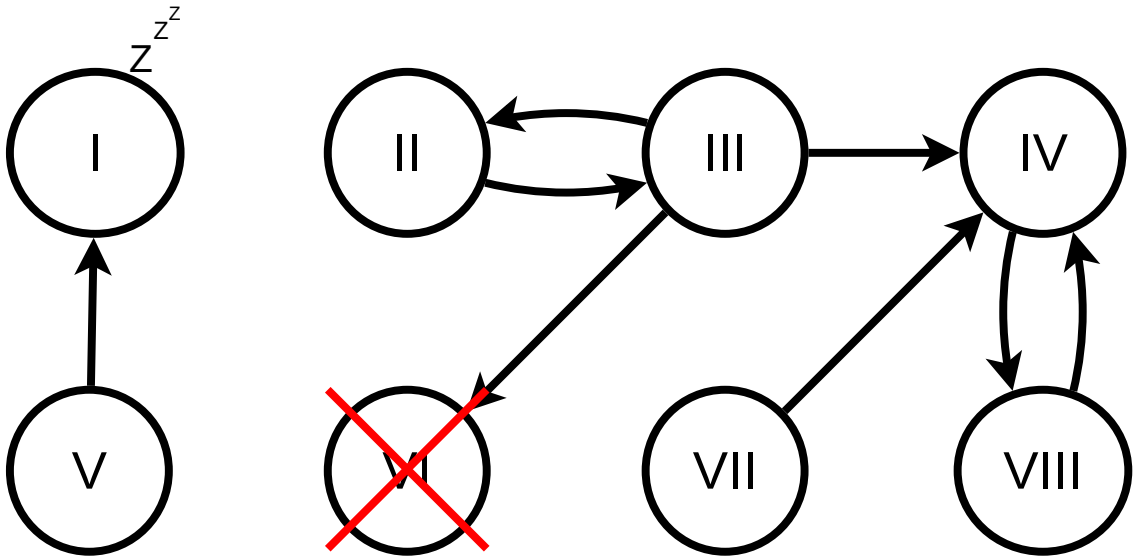
Deadlock computation

Deadlock as a greatest fixpoint:



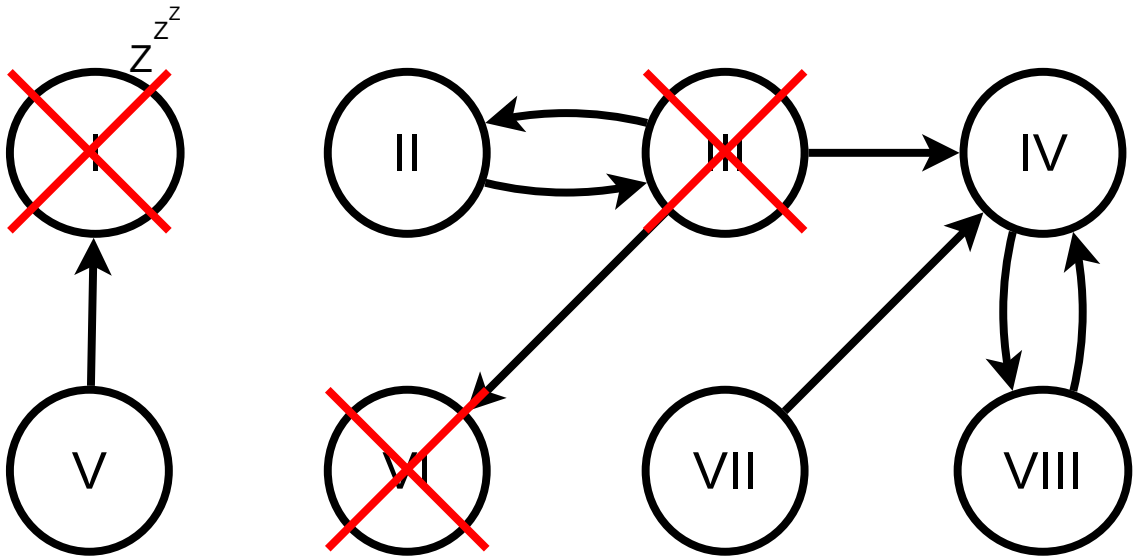
Deadlock computation

Deadlock as a greatest fixpoint:



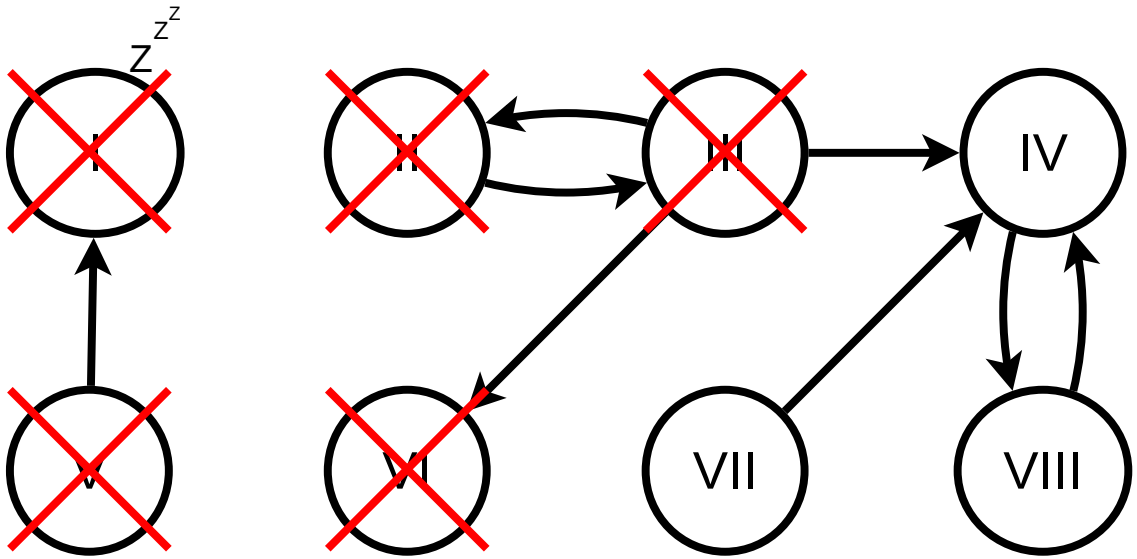
Deadlock computation

Deadlock as a greatest fixpoint:



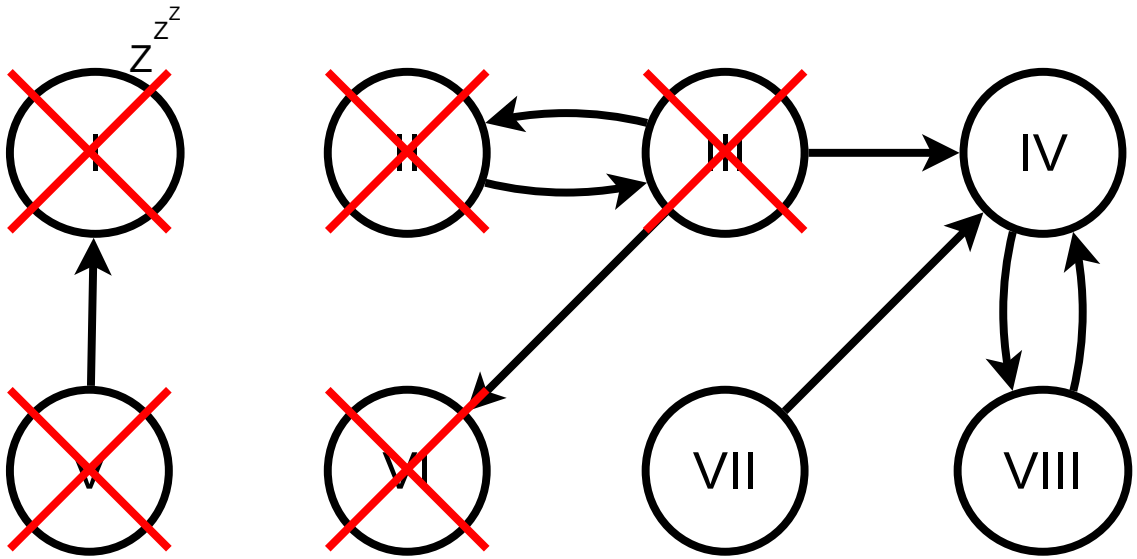
Deadlock computation

Deadlock as a greatest fixpoint:



Deadlock computation

Deadlock as a greatest fixpoint:



Threads in deadlock: IV, VII, VIII

Deadlock formalisation

Coinductive definition in the framework:

Set of threads in deadlock = greatest set D of threads satisfying:
For all threads t, t is in D iff t is

1. not in a wait set and
   - t can make progress on its own and
   - in every possible reduction, t requests a lock which is held by another thread in D,

or

2. in a wait set and all other threads
   - are in D or
   - have terminated,

Independent of type systems and language-specific constructs

Progress in the multithreaded setting

Theorem (Progress): If

- there is a thread not in deadlock and
- all threads can make progress on their own and
- locks are held only by non-final threads and
- the semantics behaves well w.r.t. thread actions,

then the framework semantics can make progress.
Formally:

$$\frac{\begin{array}{c} es\ t = (e, x) \qquad \neg\ final\ e \qquad t \notin deadlocked\ P\ ls\ es\ ws\ c \\ wf\text{-}progress\ P\ es\ c \qquad es \vdash_f ls\ \surd \qquad ex\text{-}red\ P\ ls\ es\ c \end{array}}{\exists\, t'\ tas'\ es'\ ls'\ ws'\ c'.\ P \vdash \langle ls | es, c | ws \rangle \xrightarrow{t':tas'} \langle ls' | es', c' | ws' \rangle}$$

Type safety for multithreaded Jinja

Type safety:
For well-formed classes, during execution of a set of well-formed threads, every thread with expression type T either

- gets fully evaluated with type $T' \leq T$, or
- raises a controlled exception, or
- deadlocks with type $T' \leq T$

$$\textit{wf-J-prog}\ P \qquad es \vdash_i Es\ \surd \qquad P,Es \vdash es,h \Uparrow\surd\Uparrow \qquad \Uparrow\mathcal{D}\Uparrow es\ h$$

$$es \vdash_e ls\ \surd \qquad \vdash es \Uparrow\surd_{\&}\Uparrow \qquad P \vdash \langle ls|es,h|ws\rangle \xrightarrow{ttas}{}^{\star} \langle ls'|es',h'|ws'\rangle$$

$$\nexists\, t\ tas\ es''\ ls''\ ws''\ h''.\ P \vdash \langle ls'|es',h'|ws'\rangle \xrightarrow{t:tas} \langle ls''|es'',h''|ws''\rangle$$

$$Es' = Es\ [\mathcal{I}\rightsquigarrow]_{P,\cdot} \vdash \cdot,\cdot,\cdot\ \surd \qquad \textit{flatten}\ (\textit{map snd}\ tas)$$

$$\begin{aligned}
&Es \trianglelefteq Es' \wedge \\
&(\forall\, t\ e'.\ \exists\, x'.\ es'\ t = (e', x') \longrightarrow \\
&\quad (\exists\, v.\ e' = \textit{Val}\ v \wedge (\exists\, E\ T.\ Es'\ t = (E, T) \wedge P,h' \vdash v :\leq T)) \vee \\
&\quad (\exists\, a.\ e' = \textit{Throw}\ a \wedge a \in \textit{dom}\ h') \vee \\
&\quad (t \in \textit{deadlocked}\ P\ ls'\ es'\ ws'\ h' \wedge (\exists\, E\ T.\ Es'\ t = (E, T) \wedge P,E,h' \vdash e' \leq; T))
\end{aligned}$$

Type safety for multithreaded Jinja

Type safety:
For well-formed classes, during execution of a set of well-formed threads, every thread with expression type T either

- gets fully evaluated with type $T' \leq T$, or
- raises a controlled exception, or
- deadlocks with type $T' \leq T$

$$\frac{\begin{array}{c} \mathit{wf\text{-}J\text{-}prog}\ P \qquad \mathit{es} \vdash_i \mathit{Es}\ \surd \qquad P,\mathit{Es} \vdash \mathit{es},h \Uparrow\surd\Uparrow \qquad \Uparrow\mathcal{D}\Uparrow \mathit{es}\ h \\ \mathit{es} \vdash_{\mathrm{e}} \mathit{ls}\ \surd \qquad \vdash \mathit{es} \Uparrow\surd_{\&}\Uparrow \qquad P \vdash \langle \mathit{ls}|\mathit{es},h|\mathit{ws}\rangle \xrightarrow{\mathit{ttas}}{}^{\star} \langle \mathit{ls}'|\mathit{es}',h'|\mathit{ws}'\rangle \\ \nexists\, t\ \mathit{tas}\ \mathit{es}''\ \mathit{ls}''\ \mathit{ws}''\ h''.\ P \vdash \langle \mathit{ls}'|\mathit{es}',h'|\mathit{ws}'\rangle \xrightarrow{\mathit{t:tas}} \langle \mathit{ls}''|\mathit{es}'',h''|\mathit{ws}''\rangle \\ \mathit{Es}' = \mathit{Es}\ [\mathcal{I}\rightsquigarrow]_P \qquad \vdash \cdot,\cdot,\cdot\ \surd\ \mathit{flatten}\ (\mathit{map\ snd\ tas}) \end{array}}{\begin{array}{l} \mathit{Es} \trianglelefteq \mathit{Es}' \wedge \\ (\forall\, t\ e'.\ \exists\, x'.\ \mathit{es}'\ t = (e', x') \longrightarrow \\ \quad (\exists\, v.\ e' = \mathit{Val}\ v \wedge (\exists\, E\ T.\ \mathit{Es}'\ t = (E, T) \wedge P,h' \vdash v :\leq T)) \vee \\ \quad (\exists\, a.\ e' = \mathit{Throw}\ a \wedge a \in \mathit{dom}\ h') \vee \\ \quad (t \in \mathit{deadlocked}\ P\ \mathit{ls}'\ \mathit{es}'\ \mathit{ws}'\ h' \wedge (\exists\, E\ T.\ \mathit{Es}'\ t = (E, T) \wedge P,E,h' \vdash e' \leq: T)) \end{array}}$$

The "Quis Custodiet" project

Quis Custodiet Ipsos Custodes?

Who's watching the guards?

⇒ Reach a new level of reliability in Language Based Security

⇒ Integrate semantics, theorem provers and program analysis with LBS

CoreC++ (Wasserrab et al., OOPSLA'06)
Multiple inheritance in C++ is type-safe

JinjaThreads in the Archive of Formal Proofs (`afp.sourceforge.net`):

- Framework formalisation: 5k lines (approx. 250 lemmata)
- Jinja source code add-ons: 5k lines

The "Quis Custodiet" project

Quis Custodiet Ipsos Custodes?

Who's watching the guards?

⇒ Reach a new level of reliability in Language Based Security

⇒ Integrate semantics, theorem provers and program analysis with LBS

CoreC++ (Wasserrab et al., OOPSLA'06)
Multiple inheritance in C++ is type-safe

JinjaThreads in the Archive of Formal Proofs (`afp.sourceforge.net`):

- Framework formalisation: 5k lines (approx. 250 lemmata)
- Jinja source code add-ons: 5k lines

The "Quis Custodiet" project

Quis Custodiet Ipsos Custodes?

Who's watching the guards?

⇒ Reach a new level of reliability in Language Based Security

⇒ Integrate semantics, theorem provers and program analysis with LBS

CoreC++ (Wasserrab et al., OOPSLA'06)
Multiple inheritance in C++ is type-safe

JinjaThreads in the Archive of Formal Proofs (`afp.sourceforge.net`):

- Framework formalisation: 5k lines (approx. 250 lemmata)
- Jinja source code add-ons: 5k lines

Summary

Formal semantics for multithreaded Java (subset) and type safety

- Features the most important thread primitives
- Proofs are machine-checked

- Generic framework for lifting single-thread semantics
- Deadlock formalisation and progress theorem

Starting point for:

- Language based security
- Proof carrying codes

Future work:

- Multithreaded byte code in Jinja
- Integrate duality of Java threads fully
- Include the Java Memory Model

Summary

Formal semantics for multithreaded Java (subset) and type safety

- Features the most important thread primitives
- Proofs are machine-checked

- Generic framework for lifting single-thread semantics
- Deadlock formalisation and progress theorem

Starting point for:

- Language based security
- Proof carrying codes

Future work:

- Multithreaded byte code in Jinja
- Integrate duality of Java threads fully
- Include the Java Memory Model