Quis Custodiet Ipsos Custodes?

Gregor Snelting, Daniel Wasserrab, Andreas Lochbihler, Denis Lohner

funded by DFG Sn11/10, Sn11/12

LEHRSTUHL PROGRAMMIERPARADIGMEN, KIT

```
theorem nonInterferenceSecurity:
  assumes "[cf1] ≈L [cf2]" and "(-High-) ∉ ⌊HRB-slice (CFG-node (-Low-))⌋CFG" and "valid-edge a"
  and "sourcenode a = (-High-)" and "targetnode a = n" and "kind a = (λs. True)√" and "n ≜ c"
  and "final c'" and "⟨c,[cf1]⟩ ⇒ ⟨c',s1⟩" and "⟨c,[cf2]⟩ ⇒ ⟨c',s2⟩"
  shows "s1 ≈L s2"
proof -
  from High-target-Entry-edge obtain ax where "valid-edge ax" and "sourcenode ax = (-Entry-)"
    and "targetnode ax = (-High-)" and "kind ax = (λs. True)√" by blast
  from `n ≜ c` `⟨c,[cf1]⟩ ⇒ ⟨c',s1⟩` obtain n1 as1 cfs1 where "n -as1→√* n1" and "n1 ≜ c'" and "preds (kinds as1) [(cf1,undefined)]"
    and "transfers (kinds as1) [(cf1,undefined)] = cfs1" and "map fst cfs1 = s1" by(fastsimp dest:fundamental-property)
  from `n -as1→√* n1` `valid-edge a` `sourcenode a = (-High-)` `targetnode a = n` `kind a = (λs. True)√`
  have "(-High-) -a#as1→√* n1" by(fastsimp intro:Cons-path simp:vp-def valid-path-def)
  from `final c'` `n1 ≜ c'` obtain a1 where "valid-edge a1" and "sourcenode a1 = n1" and "targetnode a1 = (-Low-)" and "kind a1 = ⇑id"
    by(fastsimp dest:final-edge-Low)
```

Quis Custodiet Ipsos Custodes? [Juvenal]

Karlsruhe Institute of Technology

Who will guard the Guards?

Many software security analysis algorithms are published without soundness proof, some with a manual proof only

Quis Custodiet Ipsos Custodes? [Juvenal]

Who will guard the Guards?
Many software security analysis algorithms are published without soundness proof, some with a manual proof only

Vision of our Project:

- provide machine-checked proofs for IFC algorithms
- reaching a new level of reliability in Language Based Security
- developing new techniques to validate the underlying language description
- integrating semantics, theorem provers and program analysis with Language Based Security

Ultimate Goal: automatically generate an executable, completely machine-verified, PDG-based IFC tool

Starting Point and Goals

Developed in earlier, long-standing projects:

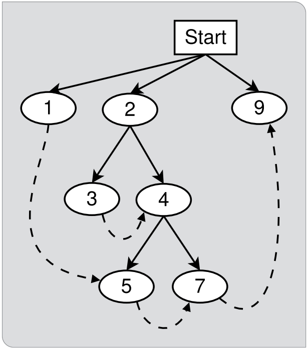
- TUM: Jinja, *Isabelle* formalization of realistic Java subset
  includes type system, operational semantics, type safety proof,
  verified JVM, verified compiler
  all proofs machine checked
- KIT: Joana, program dependence graph for full Java
  flow-sensitive, context-sensitive, object-sensitive
  scales to 100kLOC; Eclipse plug in GUI
  + IFC algorithm based on PDGs
  + manual correctness proof

Project Idea

1. verify the PDG-based IFC algorithm using Isabelle
2. support verification by innovative counter example generators

A tiny PDG

```
a = input();
while (n>0) {
  x = input();
  if (x>0)
    b = a;
  else
    c = b;
}
z = c;
```



Slicing theorem:

No path $x \to^* y \implies$ no information flow $x \to y$ guaranteed
$\exists$ Path $x \to^* y \implies$ information flow $x \to y$ possible

Backward slice: $BS(y) = \{x \mid x \to^* y\}$

Precise PDG construction for full Java is very complex
requires precise points-to analysis
Scalability: ca 100kLOC

Flow equations (intraprocedural)

$S(x)$: security level for statement/variable x

- Confidentiality: $S(x) \geq \bigsqcup_{y \in pred(x)} S(y)$
- Integrity: $S(x) \leq \bigsqcap_{y \in pred(x)} S(y)$
- required and provided levels $R(x)$, $P(x)$ (for I/O only): $R(x) \geq S(x)$ and

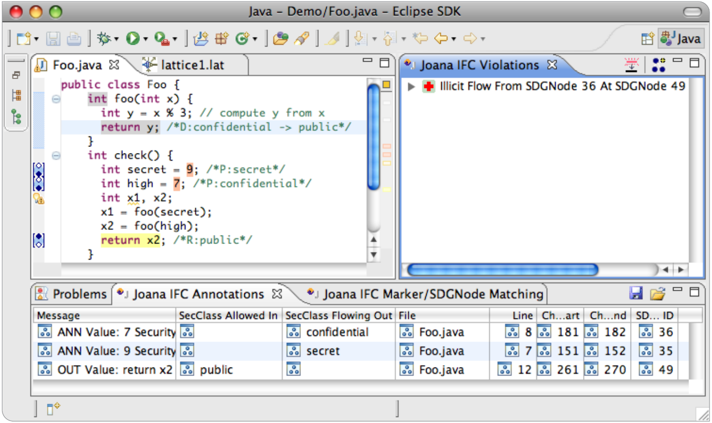
$$S(x) = \begin{cases} P(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) & \text{if } P(x) \text{ defined} \\ \bigsqcup_{y \in pred(x)} S(y) & \text{otherwise} \end{cases}$$

- for given PDG, $P(x)$, $R(x)$, S is computed by standard fixpoint iteration
- precise, interprocedural algorithm for full Java:
  C. Hammer, G. Snelting: Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs.
  International Journal of Information Security, 8, 6, December 2009.

Implementation

JOANA Eclipse Plugin: slicing, definition of $P(x)$, $R(x)$, declassifications

displays security violations, flow through the program



Results in Karlsruhe

- precise PDGs for full Java bytecode [PASTE ’04, Hamm ’09]
  precise slicing of multithreaded programs
  [FSE ’03, SCAM ’07, Hamm ’09, JASE ’09a]
- path conditions in PDGs: precise, necessary conditions for information flow, “witnesses”
  [SAS ’96, ICSE ’02, TOSEM ’06, SCAM ’07, PLAS ’08, JASE ’09b]
- IFC for full Java, based on PDGs and path conditions
  [ISSSE ’06, ISOLA ’06, PLAS ’08, IJIS ’09]

Results in Karlsruhe

- precise PDGs for full Java bytecode [PASTE ’04, Hamm ’09]
  precise slicing of multithreaded programs
  [FSE ’03, SCAM ’07, Hamm ’09, JASE ’09a]
- path conditions in PDGs: precise, necessary conditions for information flow, “witnesses”
  [SAS ’96, ICSE ’02, TOSEM ’06, SCAM ’07, PLAS ’08, JASE ’09b]
- IFC for full Java, based on PDGs and path conditions
  [ISSSE ’06, ISOLA ’06, PLAS ’08, IJIS ’09]

Quis Custodiet: Isabelle proofs

1. Multiple Inheritance in C++ is Type Safe [OOPSLA ’06, AFP ’06]
2. PDG-based IFC is correct [TPHOLS ’08, PLAS ’09, VERIFY ’10]
3. Verified Compiler for Java Threads [FOOL ’08, ESOP ’10]

C++ Multiple Inheritance is Type Safe

Multiple Inheritance in C++

A valid C++ program:

```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```

Multiple Inheritance in C++

Karlsruhe Institute of Technology

A valid C++ program:

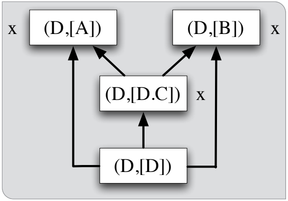
```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```

- but: `gcc` rejects it as ambiguous!
- yet, other compilers (z.B. Intel) do accept it
- problem: subobject-domination far from trivial

Subobjects and Domination

- necessary due to multiple inherits of the same super class
- Subobject: entity with the fields of the resp. class
- accessed via class path

```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```



one-step-“smaller”-relation on subobjects (reflexive transitive closure $\sqsubseteq$):

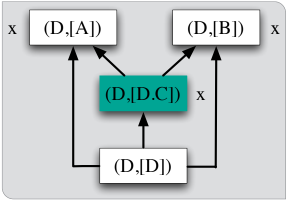
repeated: smaller subobj. contains bigger one physically in the store

shared: smaller subobj. has pointer to bigger one

Subobjects and Domination

- necessary due to multiple inherits of the same super class
- Subobject: entity with the fields of the resp. class
- accessed via class path

```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```



one-step-“smaller”-relation on subobjects (reflexive transitive closure $\sqsubseteq$):

repeated: smaller subobj. contains bigger one physically in the store

shared: smaller subobj. has pointer to bigger one

Domination: subobject “smaller” (w.r.t. $\sqsubseteq$) than all others

Subobject Formalization

Label within a class: subobject identified via class and path:

types *path* = *cname list*

types *subobj* = *cname* × *path*

Object on the heap: path selects fields of the resp. subobject:

types *subo* = *path* × (*var* ⇀ *val*)

types *obj* = *cname* × *subo set*

`this`-pointer: path denotes the subobject on which it points:

types *reference* = *addr* × *path*

may be changed via explicit and implicit casts

$\sqsubseteq$-Relation: compares path w.r.t. a class: $P,C \vdash Cs \sqsubseteq Cs'$

Dynamic Lookup

- collecting all subobjects (paths) of a class with method declaration:
  $(Cs,mthd) \in MethodDefs\ P\ C\ M$, where $mthd$ body of M in subobj. (C,Cs)
- resolve domination:
  $P \vdash C\ has\ least\ M = mthd\ via\ Cs \equiv (Cs,mthd) \in MethodDefs\ P\ C\ M \wedge$

  $$(\forall\ (Cs',mthd') \in MethodDefs\ P\ C\ M.\ P,C \vdash Cs \sqsubseteq Cs')$$

Dynamic Lookup

- collecting all subobjects (paths) of a class with method declaration:
  $\texttt{(Cs,mthd)} \in \texttt{MethodDefs P C M}$, where $\texttt{mthd}$ body of $\texttt{M}$ in subobj. $\texttt{(C,Cs)}$
- resolve domination:
  $$\texttt{P} \vdash \texttt{C has least M = mthd via Cs} \equiv \texttt{(Cs,mthd)} \in \texttt{MethodDefs P C M} \wedge$$
  $$(\forall\,\texttt{(Cs',mthd')} \in \texttt{MethodDefs P C M. P,C} \vdash \texttt{Cs} \sqsubseteq \texttt{Cs')}$$

Multiple Inheritance problem: ambiguities possible at runtime!

Dynamic Lookup

- collecting all subobjects (paths) of a class with method declaration:
  $(Cs,mthd) \in MethodDefs\ P\ C\ M$, where $mthd$ body of M in subobj. (C,Cs)
- resolve domination:
  $P \vdash C\ has\ least\ M = mthd\ via\ Cs \equiv (Cs,mthd) \in MethodDefs\ P\ C\ M \wedge$
  $(\forall\ (Cs',mthd') \in MethodDefs\ P\ C\ M.\ P,C \vdash Cs \sqsubseteq Cs')$

Multiple Inheritance problem: ambiguities possible at runtime!

A code example

```
class Top { int f(); };
class Left : Top { };
class Right : Top { };
class Bottom: Left, Right { };
...
Left* l = New Bottom();
l->f();
```

statically everything ok
At runtime:

- 2 `Top`-subobjects
  (via `Left` and `Right`)
- implicit cast of the `this`-pointer at call impossible!

Dynamic Lookup

If lookup ambiguous at runtime, static information is used (as C++ does)

- collect minimal elements:
  $MinimalMethodDefs\ P\ C\ M \equiv (Cs, mthd) \in MethodDefs\ P\ C\ M \wedge$
  $(\forall\ (Cs', mthd') \in MethodDefs\ P\ C\ M.\ P, C \vdash Cs \sqsubseteq Cs' \longrightarrow Cs = Cs')$
- determine minimal subobjects smaller than static lookup subobject:
  $(Cs, mthd) \in MethodDefs\ P\ S\ M$, where S is the subobject of the caller
- guarantee uniqueness of the minimal subobject:
  $P \vdash S\ has\ overrider\ M = mthd\ via\ Cs \equiv$
  $(Cs, mthd) \in MethodDefs\ P\ S\ M \wedge |MethodDefs\ P\ S\ M| = 1$

Dynamic Lookup

If lookup ambiguous at runtime, static information is used (as C++ does)

- collect minimal elements:
  $\mathit{MinimalMethodDefs}\ P\ C\ M \equiv (\mathit{Cs},\mathit{mthd}) \in \mathit{MethodDefs}\ P\ C\ M \wedge$
  $(\forall\,(\mathit{Cs}',\mathit{mthd}') \in \mathit{MethodDefs}\ P\ C\ M.\ P,C \vdash \mathit{Cs} \sqsubseteq \mathit{Cs}' \longrightarrow \mathit{Cs} = \mathit{Cs}')$
- determine minimal subobjects smaller than static lookup subobject:
  $(\mathit{Cs},\mathit{mthd}) \in \mathit{MethodDefs}\ P\ S\ M$, where S is the subobject of the caller
- guarantee uniqueness of the minimal subobject:
  $P \vdash S\ \mathit{has\ overrider}\ M = \mathit{mthd}\ \mathit{via}\ \mathit{Cs} \equiv$
  $(\mathit{Cs},\mathit{mthd}) \in \mathit{MethodDefs}\ P\ S\ M \wedge |\mathit{MethodDefs}\ P\ S\ M| = 1$

Real dynamic lookup: $P \vdash (C,\mathit{Cs})\ \mathit{selects}\ M = \mathit{mthd}\ \mathit{via}\ \mathit{Cs}'$

- dyn. lookup unique: $P \vdash C\ \mathit{has\ least}\ M = \mathit{mthd}\ \mathit{via}\ \mathit{Cs}$
- dyn. lookup ambiguous: $P \vdash (C,\mathit{Cs})\ \mathit{has\ overrider}\ M = \mathit{mthd}\ \mathit{via}\ \mathit{Cs}'$

Type Safety Proof

Type Safety: Execution of a program statement e of type T in state s

- either fully evaluated value v of type smaller than T
- or controlled exception

Type Safety Theorem

$$\frac{\mathit{wf\_C\_prog}\ P \quad P,E \vdash s\ \surd \quad P,E \vdash e :: T \quad \mathcal{D}\ e\ \lfloor \mathrm{dom}\ (\mathrm{lcl}\ s)\rfloor \quad P,E \vdash \langle e,s\rangle \rightarrow^{*} \langle e',s'\rangle \quad \nexists e''\ s''.\ P,E \vdash \langle e',s'\rangle \rightarrow \langle e'',s''\rangle}{(\exists v.\ e' = \mathrm{Val}\ v \wedge P,hp\ s' \vdash v :\leq T) \vee (\exists r.\ e' = \mathit{Throw}\ r \wedge \mathit{the\_addr}\ (\mathit{Ref}\ r) \in \mathit{dom}\ (hp\ s'))}$$

Type Safety Proof

Type Safety: Execution of a program statement e of type T in state s

- either fully evaluated value v of type smaller than T
- or controlled exception

Type Safety Theorem

$$\frac{\begin{array}{c}\mathit{wf\_C\_prog}\ P \quad P,E \vdash s\ \surd \quad P,E \vdash e :: T \quad \mathcal{D}\ e\ \lfloor \mathtt{dom}\ (\mathtt{lcl}\ s)\rfloor \\ P,E \vdash \langle e,s\rangle \rightarrow^{*} \langle e',s'\rangle \quad \nexists e''\ s''.\ P,E \vdash \langle e',s'\rangle \rightarrow \langle e'',s''\rangle\end{array}}{\begin{array}{l}(\exists v.\ e' = \mathtt{Val}\ v \wedge P,hp\ s' \vdash v :\leq T) \vee \\ (\exists r.\ e' = \mathit{Throw}\ r \wedge \mathit{the\_addr}\ (\mathit{Ref}\ r) \in \mathit{dom}\ (\mathit{hp}\ s'))\end{array}}$$

Standard proof technique:

Progress: "the semantics cannot get stuck"

Preservation: "evaluating a well-typed statement results in another well-typed statement with smaller type"

Proof invariant formulated as run-time type system

CoreC++ Outline

- object-oriented core language with C++ multiple inheritance and exceptions, bases on Jinja
- big-step and small-step operational semantics with equivalence proof
- type system with compiler checks
- type safety proof of semantics w.r.t. type system
- semantics and type system executable, i.e., we have an interpreter for CoreC++ programs basing on the formal semantics
  a small tool translates simple C++ programs in CoreC++ programs

Formalization Size

LoC	Lemmas	Definitions
14,727	505	82

Proving Slicing Correct

Slicing

Karlsruhe Institute of Technology

- Slicing bases on graphs
- graphs independent of underlying concrete program syntax
- Slicing itself reachability analysis
- hence, basic slicing algorithm is language independent

Correctness of Slicing

At slicing node, all used variables have same value, regardless if original or sliced program executed

Slicing

- Slicing bases on graphs
- graphs independent of underlying concrete program syntax
- Slicing itself reachability analysis
- hence, basic slicing algorithm is language independent

Correctness of Slicing

At slicing node, all used variables have same value, regardless if original or sliced program executed

Goal: correctness proof also language independent!

- language independent framework for slicing
- instantiantable with different (formal) language semantics
- ideal starting point: abstract control flow graph

Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms

Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:

Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:

  valid edges of the graph

  valid nodes are source and target nodes of valid edges

Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:

  valid edges of the graph
  : valid nodes are source and target nodes of valid edges

  semantic information of edges
  : two kinds, different effect when traversing this edge in a state
    - update edge: updates state
    - predicate edge: checks that predicate holds in state

Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:

valid edges of the graph
: valid nodes are source and target nodes of valid edges

semantic information of edges
: two kinds, different effect when traversing this edge in a state
    - update edge: updates state
    - predicate edge: checks that predicate holds in state

def and use sets of nodes
: which variables are defined and used in a node (statement)

Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:

  valid edges of the graph
  : valid nodes are source and target nodes of valid edges

  semantic information of edges
  : two kinds, different effect when traversing this edge in a state
    - update edge: updates state
    - predicate edge: checks that predicate holds in state

  def and use sets of nodes
  : which variables are defined and used in a node (statement)

- axiomatization of control flow graph properties

  structural properties: e.g., no multi-edges

  well-formedness properties: e.g., semantic effect and def/use agree

Program Dependence Graph

defined in proof context of abstract CFG

data dependence: "variable defined at one statement and used in a subsequent one, without being redefined in between"

```
n influences V in n' ≡ ∃a' as'. V ∈ Def n ∧ V ∈ Use n' ∧
        n -a'·as'→* n' ∧ (∀n''∈set (srcs as'). V ∉ Def n'')
```

control dependence: "a statement controls whether another statement is executed" (e.g., if-branches or while-body)
needs postdominator: "every terminating execution at the parameter statement has to execute the postdominating statement"

```
n' postdominates n ≡ valid_node n ∧ valid_node n' ∧
        (∀as. n -as→* Exit ⟶ n' ∈ set (srcs as))

n controls n' ≡ ∃a a' as. n -a·as→* n' ∧ n' ∉ set(srcs (a·as)) ∧
        valid_edge a' ∧ src a = n ∧ n' postdominates (trg a) ∧
        src a' = n ∧ ¬ n' postdominates (trg a')
```

Slicing

- Backward Slice: $\longrightarrow_d*$ reflexive transitive closure of control $\longrightarrow_{cd}$ and data dependence $\longrightarrow_{dd}$

  $BS\ n_c \equiv \mathit{if}\ \mathtt{valid\_node}\ n_c\ \mathit{then}\ \{n' \mid n' \longrightarrow_d* n_c\}\ \mathit{else}\ \emptyset$

- Sliced CFG: not eliminating nodes, but invalidating semantic effects! if source node of an edge not in slice, no-op as semantic effect:
  - update with identity
  - predicates *True* or *False*

  hence, traversing edge no effect, as if it were not there

- Program execution: traversing control flow paths from *Entry* to *Exit*
  - in original CFG for executions in original program
  - in sliced CFG for executions in sliced program

Correctness Proof

Following Ranganath et al. [TOPLAS '07] and Amtoft [IPL '08]:
Weak Simulation Property between original and sliced CFG

- graphs as labelled transition systems (LTS)

  LTS state: (node,state) tuple
  LTS label: edges with source node in slice
  LTS transition: silent and observable moves

$$\frac{\texttt{src a} \notin \texttt{BS}\ n_C \qquad \texttt{valid\_edge a} \qquad \texttt{pred (f a) s} \qquad \texttt{transfer (f a) s = s'}}{n_C\texttt{,f} \vdash \texttt{(src a,s) -a}\!\rightarrow_\tau \texttt{(trg a,s')}}$$

$$\frac{\texttt{src a} \in \texttt{BS}\ n_C \qquad \texttt{valid\_edge a} \qquad \texttt{pred (f a) s} \qquad \texttt{transfer (f a) s = s'}}{n_C\texttt{,f} \vdash \texttt{(src a,s) -a}\!\rightarrow \texttt{(trg a,s')}}$$

- Weak Simulation $\sim$ relation between (node,state) tuples

Correctness Proof

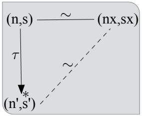
Following Ranganath et al. [TOPLAS '07] and Amtoft [IPL '08]:
Weak Simulation Property between original and sliced CFG

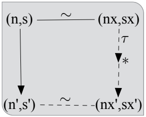
- graphs as labelled transition systems (LTS)

  LTS state: (node,state) tuple
  LTS label: edges with source node in slice
  LTS transition: silent and observable moves

$$\frac{\begin{array}{c}\mathtt{src\ a} \notin \mathtt{BS}\ n_C \\ \mathtt{valid\_edge\ a} \qquad \mathtt{pred\ (f\ a)\ s} \\ \mathtt{transfer\ (f\ a)\ s\ =\ s'}\end{array}}{n_C\mathtt{,f} \vdash \mathtt{(src\ a,s)\ \text{-}a}{\longrightarrow}_\tau\ \mathtt{(trg\ a,s')}} \qquad \frac{\begin{array}{c}\mathtt{src\ a} \in \mathtt{BS}\ n_C \\ \mathtt{valid\_edge\ a} \qquad \mathtt{pred\ (f\ a)\ s} \\ \mathtt{transfer\ (f\ a)\ s\ =\ s'}\end{array}}{n_C\mathtt{,f} \vdash \mathtt{(src\ a,s)\ \text{-}a}{\longrightarrow}\ \mathtt{(trg\ a,s')}}$$

- Weak Simulation $\sim$ relation between (node,state) tuples
- Proof: show that moves fulfil following simulation diagrams





Fundamental Property of Slicing

Correctness of Slicing

At slicing node, all used variables have same value, regardless if original or sliced program executed

weak simulation property says nothing about executions!

Fundamental Property of Slicing

Correctness of Slicing

At slicing node, all used variables have same value, regardless if original or sliced program executed

weak simulation property says nothing about executions!

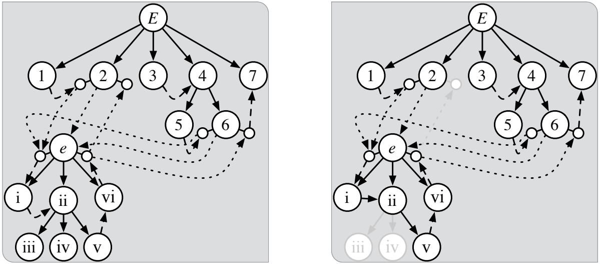
When we have a semantics which agrees to executing the CFG:

Fundamental Property of Slicing

$$\frac{n \triangleq c \qquad \langle c, s \rangle \Rightarrow \langle c', s' \rangle}{\begin{array}{l}\exists n'\ as.\ n\ \text{-}as{\longrightarrow}*\ n' \wedge preds\ (slice\_kinds\ n'\ as)\ s \wedge n' \triangleq c' \wedge \\ \quad (\forall V \in Use\ n'.\ state\_val\ (transfers\ (slice\_kinds\ n'\ as)\ s)\ V = \\ \qquad state\_val\ s'\ V)\end{array}}$$

transfers (slice_kinds n' as) s:
execution of the sliced program of *n'* in state *s*

Interprocedural Slicing

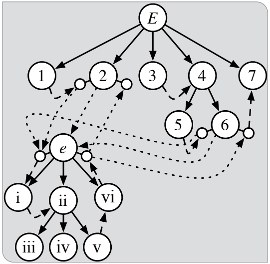


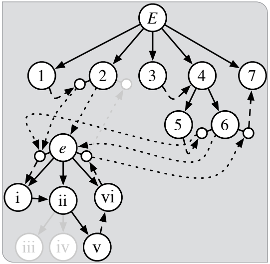
- new nodes for formal (in callee) and actual parameters (in caller)
- new edges (dotted) in dependence graph:

  call edges for calling procedures and
  parameter-in and -out edges for argument passing
- yet, simple reachability includes spurious nodes!

Interprocedural Slicing

Karlsruhe Institute of Technology



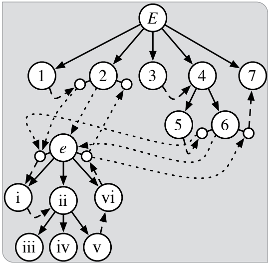


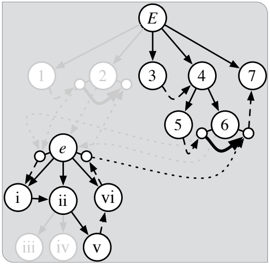
- new nodes for formal (in callee) and actual parameters (in caller)
- new edges (dotted) in dependence graph:

  call edges for calling procedures and
  parameter-in and -out edges for argument passing
- yet, simple reachability includes spurious nodes!
- context-sensitivity can eliminate such spurious nodes

Algorithm of Horwitz, Reps, Binkley (HRB)

standard for interprocedural context-sensitive slicing [TOPLAS '90]





- 2 phases: first only ascends to callee, second only descends to callers
- context-sensitivity via summary edges (bold)
  efficient computable [Reps et al.: SIGSOFT '94]
- but no correctness proof!

Summary Edges and HRB Slice

- in actual algorithm: complex algorithm $\mathcal{O}(n^3)$
- in formalization: simple declarative description

Summary Edge

If m formal in-parameter and m' formal out-parameter node, $m \longrightarrow_d * m'$ and n and n' corresponding actual parameter nodes at call site, then $n \longrightarrow_{sum} n'$

Summary Edges and HRB Slice

- in actual algorithm: complex algorithm $\mathcal{O}(n^3)$
- in formalization: simple declarative description

Summary Edge

If m formal in-parameter and m' formal out-parameter node, $m \longrightarrow_d{}^* m'$ and n and n' corresponding actual parameter nodes at call site, then $n \longrightarrow_{sum} n'$

Formalizing the two phases of the HRB algorithm as sets:

$$sum\_SDG\_slice1\ n = \{n'.\ n' \longrightarrow_{\{cd,dd,call,in,sum\}*}\ n\}$$
$$sum\_SDG\_slice2\ n = \{n'.\ n' \longrightarrow_{\{cd,dd,out,sum\}*}\ n\}$$

Summary Edges and HRB Slice

- in actual algorithm: complex algorithm $\mathcal{O}(n^3)$
- in formalization: simple declarative description

Summary Edge

If m formal in-parameter and m' formal out-parameter node, $m \longrightarrow_d{*}\ m'$ and n and n' corresponding actual parameter nodes at call site, then $n \longrightarrow_{sum}\ n'$

Formalizing the two phases of the HRB algorithm as sets:

$$\mathit{sum\_SDG\_slice1}\ n = \{n'.\ n' \longrightarrow_{\{cd,dd,call,in,sum\}}{*}\ n\}$$
$$\mathit{sum\_SDG\_slice2}\ n = \{n'.\ n' \longrightarrow_{\{cd,dd,out,sum\}}{*}\ n\}$$

HRB slice as combination of this two sets:

$$\frac{n' \in \mathit{sum\_SDG\_slice1}\ n}{n' \in \mathit{HRB\_slice}\ n} \qquad \frac{n'' \in \mathit{sum\_SDG\_slice1}\ n \quad \langle n''\ \mathit{is\ actual\ out\text{-}parameter\ node}\rangle \quad n' \in \mathit{sum\_SDG\_slice2}\ n''}{n' \in \mathit{HRB\_slice}\ n}$$

Correctness Proof

- using the same Weak Simulation Property
- but: due to context-sensitivity we need call history
  - remembers call sites previously visited, but not returned to
  - we use a node stack
- LTS state now (node stack,state) tuple
- much more complicated definition of moves and simulation relation

Correctness Proof

- using the same Weak Simulation Property
- but: due to context-sensitivity we need call history
  - remembers call sites previously visited, but not returned to
  - we use a node stack
- LTS state now (node stack,state) tuple
- much more complicated definition of moves and simulation relation

But finally, same result as for intraprocedural slicing:

Fundamental Property of Slicing

$$\frac{n \triangleq c \qquad \langle c, s\rangle \Rightarrow \langle c', s'\rangle}{\begin{array}{c}\exists n'\ as.\ n\ \text{-}as{\rightarrow}*\ n' \wedge \mathit{preds}\ (\mathit{slice\_kinds}\ n'\ as)\ s \wedge n' \triangleq c' \wedge \\ (\forall V \in \mathit{Use}\ n'.\ \mathit{state\_val}\ (\mathit{transfers}\ (\mathit{slice\_kinds}\ n'\ as)\ s)\ V = \\ \mathit{state\_val}\ s'\ V)\end{array}}$$

But much more effort...

Instantiations

While: standard while language with procedures

- source code language
- complex CFG construction (label semantics)
- proving conditions mainly by inductive reasoning

Jinja byte code: quite sophisticated object-oriented language

  - features exception throwing and catching
  - fully object oriented
  - but: no points-to analysis yet
    $\Longrightarrow$ far from precise (“heap as a whole”)
- byte code language
- “simple” CFG construction
- proving conditions mainly by reasoning by case distinction

Application to Information Flow Control

IFC: check if secret information may leak to public output

- variables partitioned in H (secret) and L (public)
- Low Equality $=_L$: two states agree in values of all L variables
- Classical Noninterference: $\forall s\ s'. s =_L s' \longrightarrow [\![c]\!]s =_L [\![c]\!]s'$
  differing values in initial H variables no effect on final L values

Application to Information Flow Control

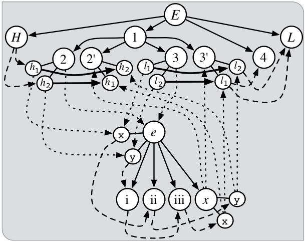
IFC: check if secret information may leak to public output

- variables partitioned in H (secret) and L (public)
- Low Equality $=_L$: two states agree in values of all L variables
- Classical Noninterference: $\forall s\ s'. s =_L s' \longrightarrow [\![c]\!]s =_L [\![c]\!]s'$
  differing values in initial H variables no effect on final L values

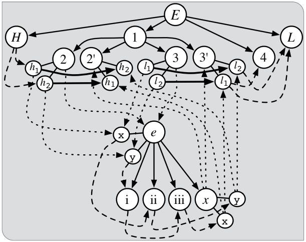
Proof that Slicing guarantees Classical Noninterference:

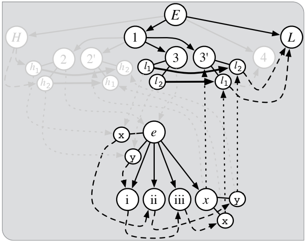
- enhance CFG by adding two nodes:
  - High immediately after Entry, defines all H variables
  - Low immediately before Exit, uses all L variables
- additional nodes also appear in Dependence Graph
- if $\mathit{High} \notin \mathit{BS}\ \mathit{Low}$, no influence from High to Low

Application to Information Flow Control



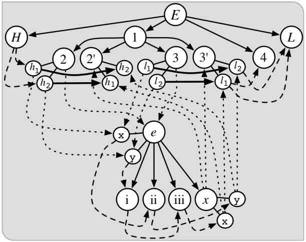
Application to Information Flow Control

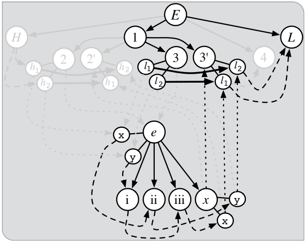




No influence from *High* to *Low*. Noninterferent?

Application to Information Flow Control





No influence from *High* to *Low*. Noninterferent!

Slicing Guarantees Noninterference

$$\frac{s_1 =_L s_2 \qquad High \notin HRB\_slice\ Low \qquad initial\ n \qquad n \triangleq c \qquad final\ n' \qquad n' \triangleq c' \qquad \langle c, s_1 \rangle \Rightarrow \langle c', s_1' \rangle \qquad \langle c, s_2 \rangle \Rightarrow \langle c', s_2' \rangle}{s_1' =_L s_2'}$$

Proof mainly by Correctness of Slicing

Slicing Outline

- language-independent framework for slicing via dependence graphs
- dynamic, static intra- and interprocedural slicing proved correct
- two instantiations:
  - a simple While source code language and
  - a sophisticated object-oriented byte code language
- first proof that slicing can guarantee classical noninterference

Formalization Size Intraprocedural Slicing

	LoC	Lemmas	Definitions
Framework	6,872	209	43
Instantiations			
While	3,177	51	17
Jinja	5,517	100	27
IFC Noninterference			
Proof	558	15	2
CFG lifting	1,470	12	3
Total	17,594	387	92

Slicing Outline

Formalization Size Interprocedural Slicing	LoC	Lemmas	Definitions
Framework	18,988	579	104
Instantiations (w/o semantics)			
While	6,758	127	29
Jinja	3,429	64	30
IFC Noninterference			
Proof	1,502	20	2
CFG lifting	2,025	8	10
Total	32,702	798	175

Ongoing and future work

Karlsruhe Institute of Technology

Points-to analysis:

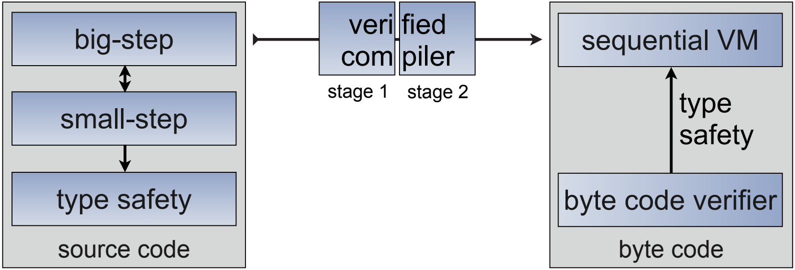
- language: Jinja (byte code)
- bases on abstract dataflow framework [Kildall '73] formalization
- Goals:
  1. verify *Correctness* (machine checked)
  2. improve precision of PDG formalization

IFC:

- formalization of
  - *suitable* noninterference definition (supporting I/O)
  - the PDG-based IFC algorithm [IJIS '09] (without declassification)
- language independent
- bases on slicing framework
- Goal: verify Correctness of the algorithm

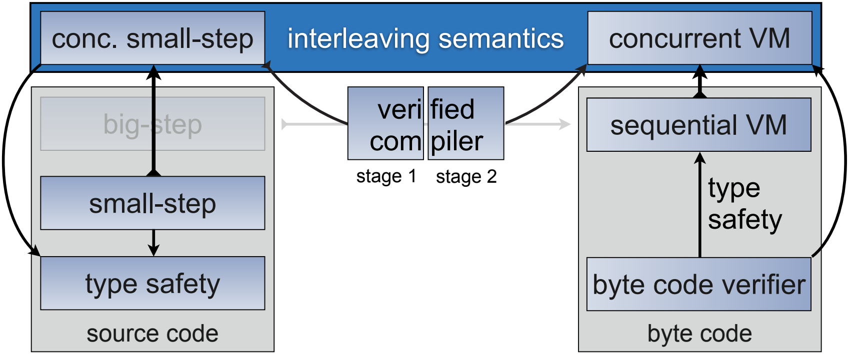
JinjaThreads

Jinja [Klein, Nipkow '06]



Java features	not modelled
classes, objects & fields	reflection & class loading
inheritance & late binding	interfaces
exceptions	threads
imperative features	

JinjaThreads



Java concurrency features	not modelled
dynamic thread creation	java.util.concurrent
synchronisation	Java Memory Model
wait / notify	
join & thread interruption	

Interleaving small-step semantics

single-thread semantics

$t \vdash \langle x, h\rangle \xrightarrow{ta} \langle x', h'\rangle$

NewThread x
Lock l / Unlock l
Wait w / notify w / ...

interleaving

multithreaded semantics

$\langle\!\langle \sigma, h\rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h'\rangle\!\rangle$

locks
thread-local states
wait sets

$$\frac{\text{typeof}_h\ a = \text{Class } C \qquad P \vdash C \preceq \text{Thread} \qquad P \vdash C \text{ sees run() = body}}{t \vdash \langle (\text{addr } a).\text{start}(), h\rangle \xrightarrow{[\text{NewThread body}]} \langle \text{Unit}, h\rangle}$$

Type safety

progress

$$\frac{P \vdash (\sigma, h)\checkmark \qquad \neg\ \text{final}\ \sigma}{\exists t\ ta\ \sigma'\ h'.\quad \langle\!\langle \sigma, h \rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h' \rangle\!\rangle}$$

preservation

$$\frac{P \vdash (\sigma, h)\checkmark \qquad \langle\!\langle \sigma, h \rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h' \rangle\!\rangle}{P \vdash (\sigma', h')\checkmark}$$

Type safety

progress

$$\frac{P \vdash (\sigma, h)\checkmark \qquad \neg\ \text{final}\ \sigma}{\exists t\ ta\ \sigma'\ h'.\ \langle\!\langle \sigma, h \rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h' \rangle\!\rangle}$$

preservation

$$\frac{P \vdash (\sigma, h)\checkmark \qquad \langle\!\langle \sigma, h \rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h' \rangle\!\rangle}{P \vdash (\sigma', h')\checkmark}$$

Generic preservation lemma

If single-thread semantics preserves prop. thread-locally,

$\Rightarrow$ multithreaded semantics preserves property globally.

Type safety

Karlsruhe Institute of Technology

progress

$$\frac{P \vdash (\sigma, h)\checkmark \quad \neg\ \text{final}\ \sigma \quad (\sigma, h) \notin \text{deadlock}}{\exists t\ ta\ \sigma'\ h'.\ \langle\!\langle \sigma, h \rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h' \rangle\!\rangle}$$

preservation

$$\frac{P \vdash (\sigma, h)\checkmark \quad \langle\!\langle \sigma, h \rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h' \rangle\!\rangle}{P \vdash (\sigma', h')\checkmark}$$

Deadlock

- all unfinished threads wait for
  - locks held by other threads
  - unfinished other threads
  - notification from wait set
- independent of concrete single-thread semantics
- coinductive characterisation

Generic preservation lemma

If single-thread semantics preserves prop. thread-locally,

$\Rightarrow$ multithreaded semantics preserves property globally.

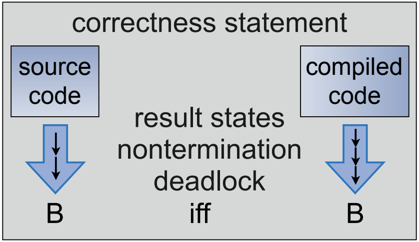
Deadlock characterisation

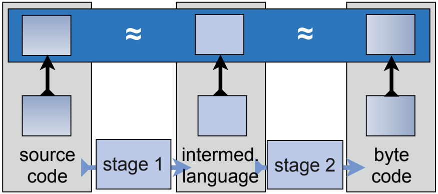
$$\frac{\text{thr } \sigma\ t = \lfloor x \rfloor \qquad t \vdash \langle x, h \rangle \rightarrow \\ \forall ta.\ t \vdash \langle x, h \rangle \xrightarrow{ta} \Longrightarrow\ \exists lt \in ta.\ \exists t' \in \text{deadlocked } (\sigma, h).\ \text{must-wait } \sigma\ t\ t'\ lt}{t \in \text{deadlocked } (\sigma, h)}$$

$$\frac{\sigma\ t = \lfloor x \rfloor \qquad t \in \text{wait-sets } \sigma \qquad \forall t \notin \text{deadlocked } (\sigma, h).\ \text{final } (\sigma\ t)}{t \in \text{deadlocked } (\sigma, h)}$$

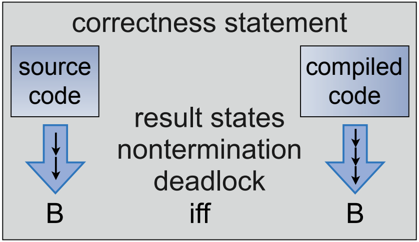
$$\text{deadlock} = \{\, (\sigma, h) \mid \forall t.\ \text{final } (\sigma\ t) \vee t \in \text{deadlocked } (\sigma, h) \,\}$$

Compiler correctness

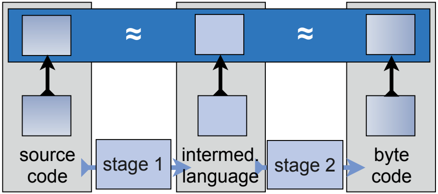




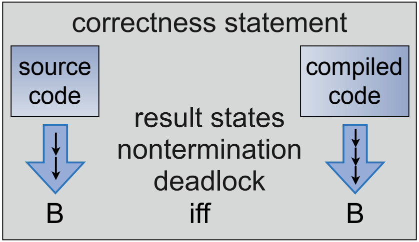
Compiler correctness



⇑



Compiler correctness



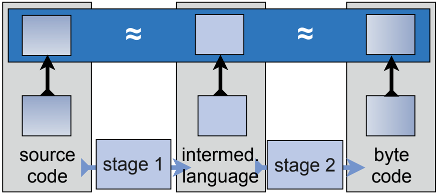
$\Uparrow$

delay bisimulation $\approx$

$(\sigma_1, h) \approx (\sigma_2, h)$

$\downarrow \tau$

(σ'_1, h')



Compiler correctness

correctness statement

source code → B

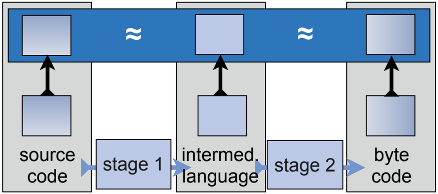
compiled code → B

result states
nontermination
deadlock
iff

⇑

delay bisimulation ≈

$$\begin{array}{ccc} (\sigma_1, h) & \approx & (\sigma_2, h) \\ \downarrow \tau & & \downarrow \tau^* \\ (\sigma'_1, h') & \approx & (\sigma'_2, h') \end{array}$$



Compiler correctness

correctness statement

source code $\Downarrow$ B

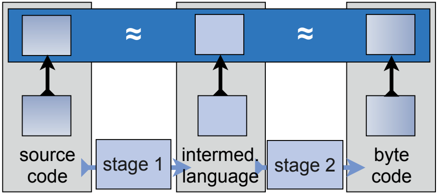
result states
nontermination
deadlock
iff

compiled code $\Downarrow$ B

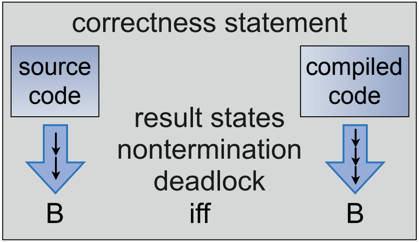
$\Uparrow$

delay bisimulation $\approx$

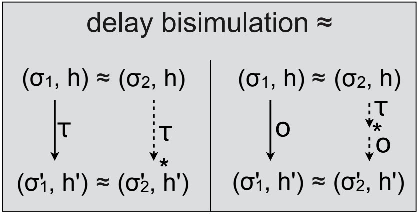
$$\begin{array}{ccc|c} (\sigma_1, h) & \approx & (\sigma_2, h) & (\sigma_1, h) \approx (\sigma_2, h) \\ \downarrow \tau & & \vdots\, \tau^* & \downarrow o \\ (\sigma'_1, h') & \approx & (\sigma'_2, h') & (\sigma'_1, h') \end{array}$$

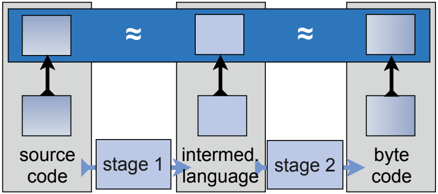


Compiler correctness



⇑





Compiler correctness

correctness statement
delay bisimulation ≈

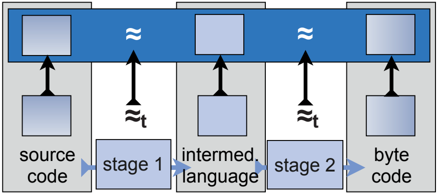
$$
\begin{tikzcd}
(\sigma_1, h) \arrow[d, "\tau"] & \approx & (\sigma_2, h) \arrow[d, dashed, "\tau^*"] \\
(\sigma'_1, h') & \approx & (\sigma'_2, h')
\end{tikzcd}
\qquad
\begin{tikzcd}
(\sigma_1, h) \arrow[dd, "o"] & \approx & (\sigma_2, h) \arrow[d, dashed, "\tau^*"] \\
 & & \cdot \arrow[d, dashed, "o"] \\
(\sigma'_1, h') & \approx & (\sigma'_2, h')
\end{tikzcd}
$$

define $(\sigma_1, h) \approx (\sigma_2, h)$:

- locks and wait sets of σ_1 and σ_2 are the same
- thread-local states x_1 and x_2 satisfy:
  $(x_1, h) \approx_t (x_2, h)$

Observable steps

- heap access
- synchronisation
- thread creation
- external method calls



Compiler correctness

correctness statement

delay bisimulation ≈

$(\sigma_1, h) \approx (\sigma_2, h)$ $\downarrow\tau$ τ (σ'_1, h')

$(\sigma_1, h) \approx (\sigma_2, h)$ $\downarrow o$ $\tau\downarrow *$

define $(\sigma_1, h) \approx (\sigma_2, h)$:

- locks and wait sets of σ_1 and σ_2 are the same
- thread-local states x_1 and x_2 satisfy:

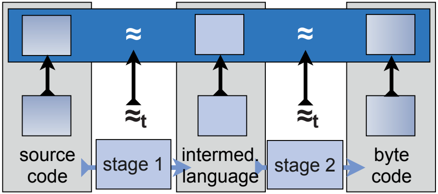
$$(x_1, h) \approx_t (x_2, h)$$

Theorem

If $\approx_t$ is a single-thread delay bisimulation,
then $\approx$ is a multithreaded delay bisimulation.

Observable steps

- heap access
- synchronisation
- thread creation
- external method calls



Statistics

Formalisation

LoC	lemmas	definitions
60,225	2812	463

$\Rightarrow$ 3 times the size of Jinja

Statistics

Formalisation

LoC	lemmas	definitions
60,225	2812	463

$\Rightarrow$ 3 times the size of Jinja

Build times:

4GB,	1×2.6GHz,	x86:	3:30h
40GB,	1×2.53GHz,	x86_64:	1:30h
40GB,	8×2.53GHz,	x86_64:	0:30h

Statistics

Formalisation

LoC	lemmas	definitions
60,225	2812	463

$\Rightarrow$ 3 times the size of Jinja

Build times:

4GB, $1\times$2.6GHz,	x86:	3:30h
40GB, $1\times$2.53GHz,	x86_64:	1:30h
40GB, $8\times$2.53GHz,	x86_64:	0:30h

Essential Isabelle features:

- Isar
- locales as a module system
- (co-)inductive definitions and proofs by (co-)induction

Statistics

Karlsruhe Institute of Technology

Formalisation

LoC	lemmas	definitions
60,225	2812	463

$\Rightarrow$ 3 times the size of Jinja

Build times:

4GB,	1×2.6GHz,	x86:	3:30h
40GB,	1×2.53GHz,	x86_64:	1:30h
40GB,	8×2.53GHz,	x86_64:	0:30h

Essential Isabelle features:

- Isar
- locales as a module system
- (co-)inductive definitions and proofs by (co-)induction

JinjaThreads hits the limits

- locales and parallelisation devour lots of memory
- very little support for refactoring

JinjaThreads summary

- formal small-step semantics for multithreaded Java
  source code and byte code
- type system and type safety proof
- verified compiler from source code to byte code
- available in the Archive of Formal Proofs
  `http://afp.sourceforge.net/entries/JinjaThreads.shtml`

JinjaThreads summary

- formal small-step semantics for multithreaded Java source code and byte code
- type system and type safety proof
- verified compiler from source code to byte code
- available in the Archive of Formal Proofs
  `http://afp.sourceforge.net/entries/JinjaThreads.shtml`

Current and future work:

- Java Memory Model
- extract executable Java interpreter

Conclusion

Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS '09] incl. points-to, threads
  requires generalized noninterference (cmp. [Askarov '08])
  proof will require $>$ 100000 LOC Isabelle text
- extend compiler formalization/proof with memory model

Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS '09] incl. points-to, threads
  requires generalized noninterference (cmp. [Askarov '08])
  proof will require $>$ 100000 LOC Isabelle text
- extend compiler formalization/proof with memory model
- automatically generate an executable, completely machine-verified, PDG-based IFC tool (?!)

Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS '09] incl. points-to, threads
  requires generalized noninterference (cmp. [Askarov '08])
  proof will require $>$ 100000 LOC Isabelle text
- extend compiler formalization/proof with memory model
- automatically generate an executable, completely machine-verified, PDG-based IFC tool (?!)

Quis Custodiet Ipsos Custodes?

Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS '09] incl. points-to, threads
  requires generalized noninterference (cmp. [Askarov '08])
  proof will require $>$ 100000 LOC Isabelle text
- extend compiler formalization/proof with memory model
- automatically generate an executable, completely machine-verified, PDG-based IFC tool (?!)

Quis Custodiet Ipsos Custodes?
Isabelle!