Proofs for Program Analysis

(Rewriting for verification)

Thomas Genet

IRISA

International School on Rewriting

6th July 2006

Preliminaries

Precondition of the talk : have attended to

- previous talks on
  rewriting, termination, compilation, theorem proving, security

Postcondition of the talk : be convinced that (or likely to be)

- rewriting is a convenient formal model for many kind of programs

  ...and this is well accepted by the verification community !

- many proof tools on rewriting exist

  ... but very few of them are used for a verification purpose

- Hence, there is still a lot to do !

Disclaimers

1. Of course, this talk cannot be exhaustive w.r.t.
   - proof techniques on rewriting
   - program analysis/verification

   but it tries to show what has been done in the intersection of the two

2. This talk reflects a personal selection and personal point of view

Outline

1. Program and property modeling using rewriting
2. Verification of properties on rewriting models : Inductive theorems
3. Verification of properties on rewriting models : Reachability analysis
4. A case study on a Java byte code static analysis

Rewriting models for program analysis

Program analysis=
- prove properties, and
- find errors otherwise (i.e. counter examples to the property)

on a formal model of the program

Advantages of rewriting for program modeling :
- easy to read and to understand (symbols, variables, rules)
- Turing complete
- can switch from deterministic to non-det. evaluation (strategies)
- can be executed efficiently (see P.-E. Moreau's talk)
- a lot of formal tools to reason about
- general purpose

Rewriting models for program analysis

Rewriting is used as a formal model for programs for decades

Large variety of programs/systems modeled using TRS :
- Hardware circuits
- Algebraic specifications/Functions
- Virtual machines and language semantics
- Parallel processes
- Telecommunication protocols, Cryptographic protocols
- ...

Algebraic specification/Function

For specifying Abstract Data Types in OBJ, PVS, etc.

```
sort List

nil -> List
cons(Elem, List) -> List
app(List, List) -> List
rev(List) -> List

app(cons(x, y), z) = cons(x, app(y, z))
app(nil, x) = x
rev(nil) = nil
rev(cons(x, y)) = app(rev(y), cons(x, nil))
```

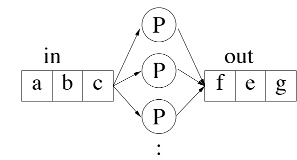

Parallel processes

- "Functional" programs

$$\left.\begin{array}{l} even(0) \to true \\ odd(0) \to false \\ even(s(x)) \to odd(x) \\ odd(s(x)) \to even(x) \end{array}\right\} \qquad even(s(s(s(0)))) \to^* false$$

- Concurrent programs

$$\begin{array}{l} P(x) \to P(x) \,||\, P(.) \\ P(.) \,||\, in([x|y]) \to P(data(x)) \,||\, in(y) \\ P(.) \,||\, in([\,]) \to in([\,]) \\ P(res(x)) \,||\, out(y) \to P(.) \,||\, out([x|y]) \end{array}$$



$$P(.) \,||\, in([a, b, c]) \to^* P(data(a)) \,||\, P(data(b)) \,||\, P(data(c)) \,||\, in([\,])$$

Parallel processes

$\mathcal{R}_1$	$\mathcal{R}_2$
$even(0) \to true$	$P(x) \to P(x) \parallel P(.)$
$odd(0) \to false$	$P(.) \parallel in([x\|y]) \to P(data(x)) \parallel in(y)$
$even(s(x)) \to odd(x)$	$P(.) \parallel in([\,]) \to in([\,])$
$odd(s(x)) \to even(x)$	$P(res(x)) \parallel out(y) \to P(.) \parallel out([x\|y])$

Concurrent "functional" programs $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$

$$\mathcal{R}_3 = \begin{cases} P(data(x)) \to P((even(x), x)) \\ P((true, x)) \to P(res(x)) \\ P((false, x)) \to P(.) \end{cases}$$

$$P(.) \parallel in([s(0), s(s(0))]) \parallel out([\,]) \to_{\mathcal{R}}^{*} P(.) \parallel in([\,]) \parallel out([s(s(0))])$$

Virtual machine/language semantics

Virtual Machine with 3 registers (program counter pc, stack s, registers r)

1 :	load 1
2 :	ifnz 7
3 :	load 1
4 :	load 2
5 :	store 1
6 :	store 2
7 :	...

$$(store_i) \quad \frac{(pc, x :: s, r)}{(pc+1, s, x \to_i r)}$$

$$(load_i) \quad \frac{(pc, s, r), x = r(i)}{(pc+1, x :: s, r)}$$

$$(ifnz_{pc'}) \quad \frac{(pc, x :: s, r)}{(pc', s, r)} \quad \text{if } x \neq 0$$

$$(ifnz_{pc'}) \quad \frac{(pc, x :: s, r)}{(pc+1, s, r)} \quad \text{if } x = 0$$

Virtual machine/language semantics

Program P $\to$ (specific TRS for P + generic TRS for the VM)

Specific TRS for P :

1 :	load 1
2 :	ifnz 7
3 :	load 1
4 :	load 2
5 :	store 1
6 :	store 2
7 :	...

$P(pp1, s, r) \to VM(load(1), pp1, s, r)$
$P(pp2, s, r) \to VM(ifnz(7), pp2, s, r)$
$P(pp3, s, r) \to VM(load(1), pp3, s, r)$
$P(pp4, s, r) \to VM(load(2), pp4, s, r)$
$P(pp5, s, r) \to VM(store(1), pp5, s, r)$
$P(pp6, s, r) \to VM(store(2), pp6, s, r)$
...

$next(pp1) \to pp2$
$next(pp2) \to pp3$
$next(pp3) \to pp4$
$next(pp4) \to pp5$
$next(pp5) \to pp6$
$next(pp6) \to pp7$
...

Virtual machine/language semantics

Generic TRS for the Virtual Machine

$$(store_i) \quad \frac{(pc, x :: s, r)}{(pc+1, s, x \to_i r)}$$

becomes

$$VM(store(1), pc, stack(x, l), reg(r1, r2, r3)) \to P(next(pc), l, reg(x, r2, r3))$$

... And similar rules for each registers

Virtual machine/language semantics

$$(ifnz_{pc'})\ \frac{(pc, x :: s, r)}{(pc', s, r)}\quad \text{if } x \neq 0$$

$$(ifnz_{pc'})\ \frac{(pc, x :: s, r)}{(pc+1, s, r)}\quad \text{if } x = 0$$

becomes

$$VM(ifnz(pc'), pc, stack(s(x), s), r) \to P(pc', s, r)$$

$$VM(ifnz(pc'), pc, stack(p(x), s), r) \to P(pc', s, r)$$

$$VM(ifnz(pc'), pc, stack(0, s), r) \to P(next(pc), s, r)$$

Virtual machine/language semantics

Can model all Java Semantics (Objects, references, methods, ...)

- JavaCard (source and bytecode) [Barthe & al. 2001]
- Java (source/bytecode) [Farzan Meseguer Rosu 2004]
- Java (bytecode) [Boichut Genet Jensen Le Roux 2007]

What for ?

- "bi-simulation" proofs of defensive/offensive Java Virtual Machine
- fault detection and bounded model-checking
- static analysis

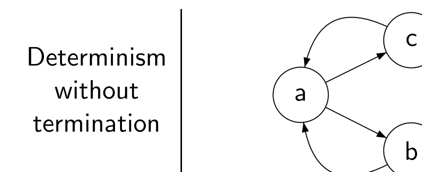
Properties of rewriting systems as program properties (Relating rewriting to verification)

- Termination
- Strong non termination
- Confluence
- Reachability
- Word problems
- Inductive theorems
- Sufficient completeness
- Complexity

Relating rewriting properties and verification

- Termination of rewriting = Termination of program
  using orderings, ... See Jurgen Giesl's talk
- Termination of rewriting = Liveness properties [Giesl Zantema 2003]
- Confluence of rewriting = Determinism of program
  using critical pairs See Franz Baader's talk
  - join critical pairs + termination
  - no critical pairs + left-linearity (Orthogonal systems)

Determinism without termination



"If execution follows different pathes, then pathes always finitely join"

Relating rewriting properties and verification

On an equational theory E (ex. an algebraic specification)

```
app(cons(x, y), z) = cons(x, app(y, z))
app(nil, x) = x
rev(nil) = nil
rev(cons(x, y)) = app(rev(y), cons(x, nil))
```

- Word problems ($s =_E t$, $s \neq_E t$) = ground properties of E

  $app([a, b], [c, d]) =_E [a, b, c, d]$

  $app([a, b], [a, b]) \neq_E [a, b]$ (See F. Baader's and C. Lynch's talks)

- Inductive theorems ($\forall \sigma : s\sigma =_E t\sigma$) = invariants of E

  $app(app(x, y), z) = app(x, app(y, z))$

  $rev(rev(x)) = x$

Relating rewriting properties and verification

On a term rewriting R

$app(cons(x, y), z) \to cons(x, app(y, z))$ $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$
$app(nil, x) \to x$
$rev(nil) \to nil$ $\mathcal{D} = \{app, rev\}$
$rev(cons(x, y)) \to app(rev(y), cons(x, nil))$ $\mathcal{C} = \{cons, nil, ...\}$

Definition

$\mathcal{R}$ is sufficiently complete if $\forall s \in \mathcal{T}(\mathcal{F}) : \exists t \in \mathcal{T}(\mathcal{C}) : s \to_{\mathcal{R}}^* t$

- Suff. completeness + Termination = (app, rev) Total relations
- Suff. completeness + Confluence + Termination = Total functions

[Nipkow Weikum 83] [Comon 86] [Kapur Narendran Zhang 87]
[Bouhoula Jacquemard 2006]

Relating rewriting properties and verification

$\mathcal{R}_1$	$\mathcal{R}_2$
$even(0) \to true$	$P(x) \to P(x) \mid\mid P(.)$
$odd(0) \to false$	$P(.) \mid\mid in([x\mid y]) \to P(data(x)) \mid\mid in(y)$
$even(s(x)) \to odd(x)$	$P(.) \mid\mid in([\,]) \to in([\,])$
$odd(s(x)) \to even(x)$	$P(res(x)) \mid\mid out(y) \to P(.) \mid\mid out([x\mid y])$

$$\mathcal{R}_3 = \begin{cases} P(data(x)) \to P((even(x), x)) \\ P((true, x)) \to P(res(x)) \\ P((false, x)) \to P(.) \end{cases}$$

- (Un)reachability ($s \not\to_{\mathcal{R}}^* t$) = safety properties
- Reachability ($s \to_{\mathcal{R}}^* t$) = fault detection

$P(.) \mid\mid in([s(0), s(s(0))]) \mid\mid out([\,]) \not\to_{\mathcal{R}}^* P(.) \mid\mid in([\,]) \mid\mid out([s(0)])$

Proofs based on the computation of $\mathcal{R}^*(E) = \{t \mid s \in E \wedge s \to_{\mathcal{R}}^* t\}$

Relating rewriting properties and verification

- Complexity of TRS = Complexity of program

  Termination by LPO $\Leftrightarrow$ polynomial complexity
  [Bonfante Marion Moyen 2001]

- Strong non termination of TRS = deadlock-free programs

Definition

Let E be a set of terms and $\mathcal{R}$ be a TRS. The TRS $\mathcal{R}$ is said to be strongly non-terminating on E if there exists no finite $\mathcal{R}$-rewrite chains from terms of E.

$\mathcal{R}^*(E) \cap IRR(\mathcal{R}) = \emptyset \quad \Rightarrow \quad$ Strong non termination of $\mathcal{R}$ on E

($IRR(\mathcal{R})$ is the set of terms irreducible by $\mathcal{R}$)

[Feuillade Genet Viet-Triem-Tong 2004]

Outline

1. Program and property modeling using rewriting
2. Verification of properties on rewriting models : Inductive theorems
3. Verification of properties on rewriting models : Reachability analysis
4. A case study on a Java byte code static analysis

Proving inductive theorems

$$E = \begin{cases} 0 + x = x \\ s(x) + y = s(x + y) \end{cases} \qquad \phi \equiv \; x + (y + z) = (x + y) + z$$

prove that ϕ is an inductive theorem of E

i.e. prove that $\forall \sigma \in \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}) : [x + (y + z)]\sigma =_E [(x + y) + z]\sigma$

- Explicit induction [RRL : Kapur Zhang 88] [Larch : Garland Guttag 89] [Spike : Bouhoula Kounalis Rusinowitch 92]
- Implicit induction [Spike : Bouhoula Kounalis Rusinowitch 92]
- Inductionless induction (Proof by consistency) Survey : [Comon 94]

Explicit induction

$$E = \begin{cases} 0 + x = x \\ s(x) + y = s(x + y) \end{cases}$$

prove that $\forall \sigma \in \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}) : [x + (y + z)]\sigma =_E [(x + y) + z]\sigma$

Basic mechanism of explicit induction :

- E oriented into $\mathcal{R}$ terminating, confluent, sufficiently complete
- constructor substitutions are enough : $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{C})$
  Since $\forall t \in \mathcal{T}(\mathcal{F}) : \exists c \in \mathcal{T}(\mathcal{C}) : t =_E c$
- automatic selection of inductive positions
  $x + (y + z) =_E (x + y) + z$
- induction on $\mathcal{T}(\mathcal{C})$, here $\mathcal{C} = \{0, s\}$:
  1. prove $0 + (y + z) =_E (0 + y) + z$
  2. prove $s(a) + (y + z) =_{E'} (s(a) + y) + z)$
     where $E' = E \cup \{a + (y + z) = (a + y) + z\}$

Inductionless induction

$$E = \begin{cases} 0 + x = x \\ s(x) + y = s(x + y) \end{cases} \qquad \phi \equiv \; x + (y + z) = (x + y) + z$$

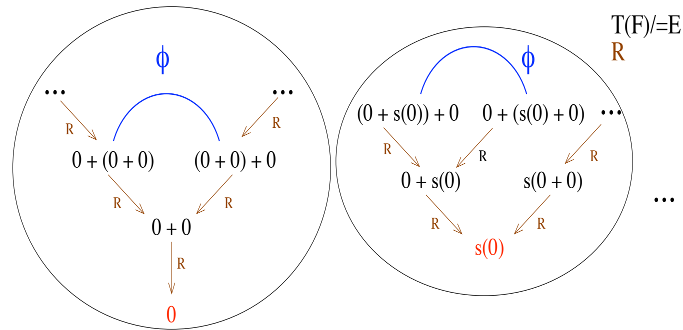
prove that $\forall \sigma \in \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}) : [x + (y + z)]\sigma =_E [(x + y) + z]\sigma$

$\Leftrightarrow$ prove $\forall \sigma \in \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}) : [(x + (y + z))\sigma]$ and $[(x + y) + z)\sigma]$ are in the same equivalence class of $\mathcal{T}(\mathcal{F})/_{=_E}$

$\Leftrightarrow$ check if ϕ consistent with $\mathcal{T}(\mathcal{F})/_{=_E}$ (using Knuth Bendix completion)

Inductionless induction

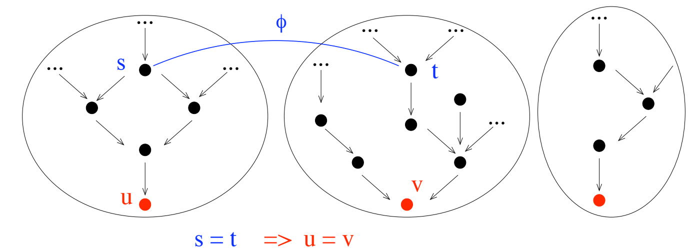
$E = \begin{cases} 0 + x = x \\ s(x) + y = s(x + y) \end{cases}$ $\phi \equiv x + (y + z) = (x + y) + z$



Same classes for E and $E \cup \{\phi\} \Rightarrow \phi$ is an inductive theorem of E

Inductionless induction

When ϕ is not an inductive theorem of E



The equation $u = v$ is produced during the completion

$\Rightarrow$ Normal forms u and v are merged in the same class !

Inductionless induction

$E = \begin{cases} 0 + x = x \\ s(x) + y = s(x + y) \end{cases}$ $\phi \equiv x + (y + z) = (x + y) + z$

prove that ϕ is an inductive theorem of E

Basic mechanism of Inductionless induction

- Transform E into a terminating and confluent TRS $\mathcal{R}$
- Transform $E \cup \{\phi\}$ into a terminating and confluent TRS $\mathcal{R}'$
- If $\mathcal{R}$ and $\mathcal{R}'$ have the same set of normal forms then ϕ is an inductive theorem of E

Verification experiments on rewriting models

Rewriting based verification experiments :

- Some "Industrial" success stories
  - Hardware verification, with RRL and Larch
  - Telecommunication protocol verification, with Spike
  - Java Card virtual machine specification (but proofs done with Coq)
  - Cryptographic protocol verification, with AVISPA
  - Attack detection on cryptographic API of chips, with Otter
  - ...
- But far less than other verification techniques
  - model-checking
    commonly used for hardware verification (Intel)
  - static analysis (ex. Astree analyzer)
    proof of the absence of Run Time Error on 1.000.000 lines of C code of Flight control software (Airbus)
  - proof assistants on "functional" models (ex. Coq)
    certification of a complete JavaCard system (Trusted logic)

What are the weak points of rewriting for verification ?

- Initially rewriting proof techniques not designed for verification
  - Automated proof of complex properties on small theories
  - The research effort mainly focuses on automation
- Automation of proof is important but
  - How to guide the proof when the automatic tool fails ?
  - How to succeed when the automatic proof explodes ?
- Many proof techniques need Termination
- Proving termination is hard on real-size systems
  - Proving termination of one system is difficult
  - Termination of TRS is not modular
    $\neq$ Termination of functional models

$\Rightarrow$ Less scalable than user-assisted proofs on "functional" models

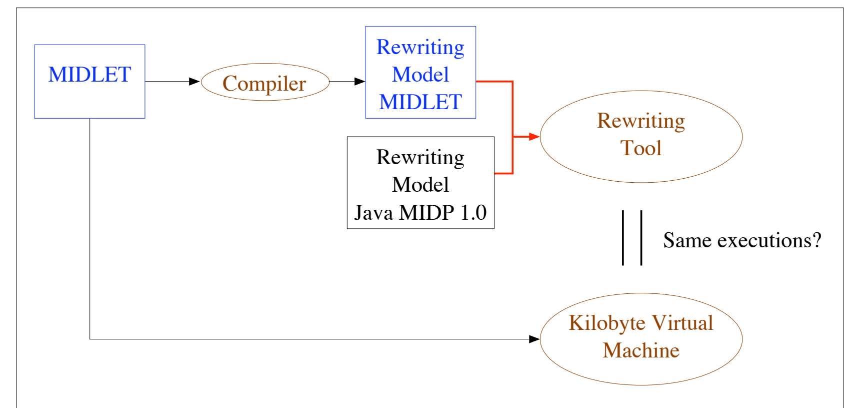
Where does rewriting may succeed in verification ?

- As a modeling language
  - simple
  - close to execution
  - general purpose
- In decision procedures for specific equational theories
  using (unfailing) completion or specific procedures in proof assistants
  [Bonacina Hsiang 90] [Armando Ranise Rusinowitch 2003]
- But also for verification of safety properties on models !
  The same rewriting model can be used for
  - simulation (execution/animation using $\rightarrow_{\mathcal{R}}$)
  - efficient fault detection (using $s \rightarrow_{\mathcal{R}}^{*} t$)
  - proof (using $s \not\rightarrow_{\mathcal{R}}^{*} t$) (Ex. AVISPA)
- Ex. for fault detection rewriting tools compete with model-checkers
  (Ex. Maude vs SPIN) [Eker & al. 2002]

Why simulation is necessary ?

```
TRS R
state(xframe(invokeVirtual(main),m,pc,stack(loc(test,x),s),l),f,h,k) -> state(frame(name(main,test),pp0,nil_sta
 state(xframe(invokeVirtual(init),m,pc,stack(loc(test,x),s),l),f,h,k) -> state(frame(name(init,test),pp0,nil_st
 state(xframe(invokeVirtual(set),m,pc,stack(l1,stack(loc(B,x),s)),l),f,h,k) -> state(frame(name(set,B),pp0,nil_
 state(xframe(invokeVirtual(init),m,pc,stack(loc(B,x),s),l),f,h,k) -> state(frame(name(init,B),pp0,nil_stack,lo
 state(xframe(invokeVirtual(set),m,pc,stack(l1,stack(loc(A,x),s)),l),f,h,k) -> state(frame(name(set,A),pp0,nil_
 state(xframe(invokeVirtual(init),m,pc,stack(loc(A,x),s),l),f,h,k) ->
 state(frame(name(init,A),pp0,nil_stack,locals(loc(A,x),nil_local,nil_local)),stack(stored_frame(m,pc,s,l),f),h
 state(frame(name(init,javalangObject),pp0,nil_stack,locals(x,nil_local,nil_local)),stack(stored_frame(m,pc,s,l
 state(xframe(invokeNonVirtual(x),m,pc,stack(y,s),l),f,h,k) -> state(frame(x,pp0,nil_stack,locals(y,nil_local,n
 state(xframe(new(A),m,pc,s,l),f,heaps(h0,heap(a,h),h2),k) -> state(frame(m,next(pc),stack(loc(A,a),s),l),f,hea
 state(xframe(new(B),m,pc,s,l),f,heaps(h0,h1,heap(a,h)),k) -> state(frame(m,next(pc),stack(loc(B,a),s),l),f,hea
 state(xframe(new(test),m,pc,s,l),f,heaps(heap(a,h),h1,h2),k) -> state(frame(m,next(pc),stack(loc(test,a),s),l)
 state(xframe(putField(x),m,pc,stack(loc(A,y),s),l),f,heaps(h0,heap(a,h),h2),k) -> state(xframe(xGetField(A,x,h
 state(xframe(putField(x),m,pc,stack(loc(B,y),s),l),f,heaps(h0,h1,heap(a,h)),k) -> state(xframe(xGetField(B,x,h
 state(xframe(putField(x),m,pc,stack(loc(test,y),s),l),f,heaps(heap(a,h),h1,h2),k) -> state(xframe(xGetField(te
 state(xframe(putField(x),m,pc,stack(z,stack(loc(A,y),s)),l),f,heaps(h0,heap(a,h),h2),k) -> state(xframe(xPutFi
 state(xframe(putField(x),m,pc,stack(z,stack(loc(B,y),s)),l),f,heaps(h0,h1,heap(a,h)),k) -> state(xframe(xPutFi
 state(xframe(putField(x),m,pc,stack(z,stack(loc(test,y),s)),l),f,heaps(heap(a,h),h1,h2),k) -> state(xframe(xPu
 state(xframe(resultPutField(x),m,pc,stack(z,stack(loc(A,y),s)),l),f,heaps(h0,heap(a,h),h2),k) -> state(frame(m
 state(xframe(resultPutField(x),m,pc,stack(z,stack(loc(B,y),s)),l),f,heaps(h0,h1,heap(a,h)),k) -> state(frame(m
 state(xframe(resultPutField(x),m,pc,stack(z,stack(loc(test,y),s)),l),f,heaps(heap(a,h),h1,h2),k) -> state(fram
 state(xframe(returnVoid,a,b,c,z),stack(stored_frame(m,pc,s,l),f),h,k)-> state(frame(m,next(pc),s,l),f,h,k)
 xPutField(B,fieldf,x,y,stack(objectB(ff),h),zero,zero) -> xPutField(B,fieldf,wait,y,stack(objectB(x),h),wait,w
 xframe(aLoad(local0),m,pc,s,locals(l0,l1,l2)) -> frame(m,next(pc),stack(l0,s),locals(l0,l1,l2))
 xframe(aLoad(local1),m,pc,s,locals(l0,l1,l2)) -> frame(m,next(pc),stack(l1,s),locals(l0,l1,l2))
 xframe(aLoad(local2),m,pc,s,locals(l0,l1,l2)) -> frame(m,next(pc),stack(l2,s),locals(l0,l1,l2))
 xframe(aStore(local0),m,pc,stack(x,s),locals(l0,l1,l2)) -> frame(m,next(pc),s,locals(x,l1,l2))
 xframe(aStore(local1),m,pc,stack(x,s),locals(l0,l1,l2)) -> frame(m,next(pc),s,locals(l0,x,l2))
 xframe(aStore(local2),m,pc,stack(x,s),locals(l0,l1,l2)) -> frame(m,next(pc),s,locals(l0,l1,x))
 xframe(dup,m,pc,stack(x,s),l) -> frame(m,next(pc),stack(x,stack(x,s)),l)
 xframe(goto(x),m,pc,s,l) -> frame(m,x,s,l)
 xframe(iConst(x),m,pc,s,l) -> frame(m,next(pc),stack(x,s),l)
```


Building a reliable rewriting semantics for Java



Outline

1. Program and property modeling using rewriting
2. Verification of properties on rewriting models : Inductive theorems
3. Verification of properties on rewriting models : Reachability analysis
4. A case study on a Java byte code static analysis

Reachability analysis

If $\mathcal{R}$ models a program and s, t model program states

- (Un)reachability ($s \not\rightarrow_{\mathcal{R}}{}^* t$) = safety properties
- Reachability ($s \rightarrow_{\mathcal{R}}{}^* t$) = fault detection

Generalized using sets of reachable terms

- E, set of all initial terms
- Bad, set of all errors/faults/attacks of interest
- $\mathcal{R}^*(E) = \{t \mid s \in E \wedge s \rightarrow_{\mathcal{R}}{}^* t\}$ set of reachable ters
- Program safety $\equiv \mathcal{R}^*(E) \cap Bad = \emptyset$

Why focusing on reachability analysis ?

- safety properties are one of the main concerns in verification
- reachability analysis can deal with non terminating TRS
- reachability analysis can be adapted to the size of the model/property

What are the proof techniques for reachability ?

Usual proof techniques of rewriting do not apply !

$$E = \begin{cases} l_1 = r_1 \\ \dots \\ l_n = r_n \end{cases} \qquad \mathcal{R} = \begin{cases} l_1 \rightarrow r_1 \\ \dots \\ l_n \rightarrow r_n \end{cases} \qquad \begin{array}{ccc} s & =_E & t \\ & \searrow_{\mathcal{R}^*} \swarrow & \\ & u & \end{array}$$

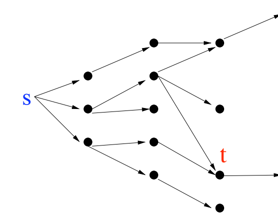
- Completion [Knuth-Bendix 70]
- Unfailing completion [Bachmair Dershowitz Plaisted 89]
- Proof of inductive theorems [Reddy 90]
  [Comon 94] [Kapur Zhang 95] [Bouhoula Rusinowitch 95] ...

But

$$\begin{array}{ccc} a & =_E & b \\ \Vert_E & & \\ c & & \end{array} \qquad \begin{array}{ccc} a & \xrightarrow{R} & b \\ \downarrow_R & \swarrow_R & \\ c & & \end{array}$$

Reachability with termination

If s and t ground, $s \rightarrow_{\mathcal{R}}{}^* t$ and $s \not\rightarrow_{\mathcal{R}}{}^* t$ decidable if $\mathcal{R}$ terminates



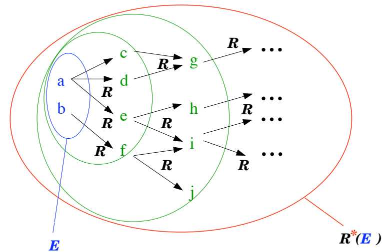
But :

- program models are frequently non terminating
- only finite set E of initial terms

Reachability without termination

Classes of TRS for which $s \to_{\mathcal{R}}^* t$ and $s \not\to_{\mathcal{R}}^* t$ are decidable

- syntactic restrictions on $\mathcal{R}$
- construction of reachable terms : $\mathcal{R}^*(E)$



$$s \to_{\mathcal{R}}^* t$$
$$\Updownarrow$$
$$t \in \mathcal{R}^*(\{s\})$$

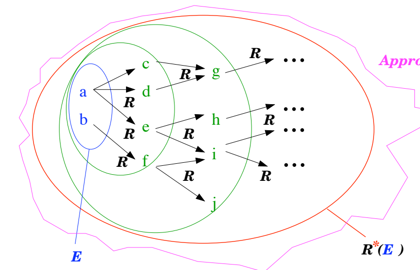
- E and $\mathcal{R}^*(E)$ represented by tree automata

[Salomaa 1988][Dauchet,Tison 1990][Jacquemard 1996][Gyenizse,Vàgvölgyi 1998]
[Takai,Kaiji,Seki 2000][Seki,Takai,Fujinaka,Kaiji 2002]

Reachability without termination (Tree Automata approx.)

Criteria for $s \not\to_{\mathcal{R}}^* t$ by over-approximation

- (nearly) no syntactic restrictions on $\mathcal{R}$
- construction of : $Approx \supseteq \mathcal{R}^*(E)$



if $s \in E$

$t \notin Approx$
$$\Downarrow$$
$t \notin \mathcal{R}^*(E)$
$$\Downarrow$$
$s \not\to_{\mathcal{R}}^* t$

- E and $Approx$ represented by tree automata

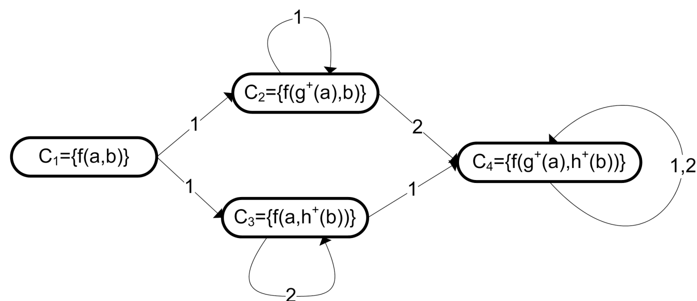
[Jacquemard 96][Genet 98][Takai 04]

Intuition behind the over-approximations

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\to_{\mathcal{R}}^* f(a,h(g(b)))$?

using $App = \{g(g(x)) = g(x), h(h(x)) = h(x)\}$



$f(a,b) \not\to_{\mathcal{R}/App}^* f(a,h(g(b)))$ $\Rightarrow$ $f(a,b) \not\to_{\mathcal{R}}^* f(a,h(g(b)))$

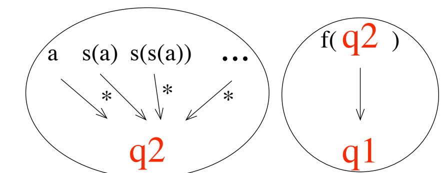
[Genet,Viet Triem Tong 2002][Meseguer,Palomino,Marti-Oliet 2003][Takai 2004]

How to represent regular equivalence classes ?

- Representation of $f(s^*(a))$ by tree grammar/automata

Tree grammar G			Tree automata A		
$\{f(s^*(a))\}$	axiome : N_1		$\{f(s^*(a))\}$	final state : q_1	
N_1	$:=$	$f(N_2)$	$f(q_2)$	$\to$	q_1
N_2	$:=$	$s(N_2)$	$s(q_2)$	$\to$	q_2
N_2	$:=$	a	a	$\to$	q_2

$N_2 \to_G^* s(s(a))$ | $s(s(a)) \to_A^* q_2$



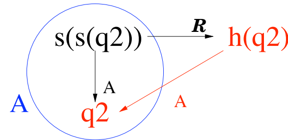
$\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ by Tree Automata Completion

$\mathcal{A}^0$	$\mathcal{A}^*_{\mathcal{R}}$
$f(q_2) \to q_1$ $s(q_2) \to q_2$ $a \to q_2$	$h(q_2) \to q_2$
$\{f(s^*(a))\}$	$\{f([s,h]^*(a)\}$

$s(s(a)) \to^*_{\mathcal{A}^0} q_2$
$s(s(s(a))) \to^*_{\mathcal{A}^0} q_2$
$h(a) \to^*_{\mathcal{A}^*_{\mathcal{R}}} q_2$
$h(s(a)) \to^*_{\mathcal{A}^*_{\mathcal{R}}} q_2$
$h(s(h(a))) \to^*_{\mathcal{A}^*_{\mathcal{R}}} q_2$
$f(h(s(h(a)))) \to^*_{\mathcal{A}^*_{\mathcal{R}}} q_1$

$\mathcal{A}^0$ completed into $\mathcal{A}^1_{\mathcal{R}}$ for $\mathcal{R} = \{s(s(x)) \to h(x)\}$

- matching of $s(s(x))$ in $\mathcal{A}^0$: $s(s(q_2)) \to_{\mathcal{A}^0} s(q_2) \to_{\mathcal{A}^0} q_2$



- $\mathcal{A}^1_{\mathcal{R}} = \mathcal{A}^0 \cup \{h(q_2) \to q_2\} = \mathcal{A}^*_{\mathcal{R}}$ Fixpoint ! $\boxed{\mathcal{L}(\mathcal{A}^*_{\mathcal{R}}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}^0))}$

Decidable classes covered by completion

[Dauchet Tison 90], [Salomaa 88], [Coquidé 91], [Jacquemard 96]

- An interesting class for verification : [Réty 99]
  - $\mathcal{R}$ does not dupplicate data (right linear)
  - recursive calls of $\mathcal{R}$ consume data
  - $\mathcal{R}$ no embedded function calls
  - E finite number of function calls on data

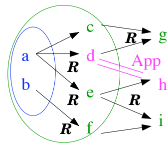
$app(nil, x) \to x$ $app(cons(x,y), z) \to cons(x, app(y,z))$ $\} = \mathcal{R}$ $\{app(app([a,a,\ldots],[b,b,\ldots]),[c,c,\ldots])\} = E$	$\mathcal{R}^*(E)$ $\|$ $\{ [a,a,\ldots,b,b,\ldots,c,c,\ldots] \}$

  - *even*, *odd*, *plus*, ...

- Other interesting decidable classes not covered (yet ?)
  - Right-linear finite-path overlapping [Takai Kaji Seki 2000]
  - PA-processes [Lugiez Schnoebelen 1998]
  - PRS (subclass of) [Bouajjani Touili 2005]
  - ... (See Denis Lugiez talk for details)

Over-approximation

- Outside of decidable classes : $A^*_{\mathcal{R}}$ infinite



[Genet 98]
[Meseguer & al. 2003]
[Takai 2004]

- Approximation principle : equivalence $=_{App}$ on $\mathcal{L}(A^*_{\mathcal{R}})$
- Rewriting modulo approx. : $s \to_{\mathcal{R}/App} t \Leftrightarrow s =_{App} s' \to_{\mathcal{R}} t' =_{App} t$
- Safe approximation : $\boxed{s \not\to^*_{\mathcal{R}/App} t \;\Rightarrow\; s \not\to^*_{\mathcal{R}} t}$
- Construction of $A^*_{\mathcal{R}/App}$ by propagation of $=_{App}$ on $A^*_{\mathcal{R}}$

$$\begin{array}{ccc|} s_1 & =_{App} & s_2 \\ \downarrow A^*_{\mathcal{R}} & & A^*_{\mathcal{R}} \downarrow \\ q_1 & & q_2 \end{array} \quad \Rightarrow \quad q_1 = q_2 \text{ applied to } A^*_{\mathcal{R}}$$

Timbuk Demo (I)

- Timbuk = LGPL implementation of Tree Automata Completion
  `http ://www.irisa.fr/lande/genet/timbuk`

$$\mathcal{R}_1 = \begin{cases} app(nil, x) \to x \\ app(cons(x,y), z) \to cons(x, app(y,z)) \\ rev(nil) \to nil \\ rev(cons(x,y)) \to app(rev(y), cons(x, nil)) \end{cases}$$

$$\mathcal{R}_2 = \begin{cases} app(nil, x) \to x \\ app(cons(x,y), z) \to cons(x, app(y,z)) \end{cases}$$

- Rewriting : $rev([a,b,c]) \to^*_{\mathcal{R}_1} [c,b,a]$
- Decidable classes :

$$app(app([a,a,\ldots],[b,b,\ldots]),[c,c,\ldots]) \to^*_{\mathcal{R}_2} [a,a,\ldots,b,b,\ldots,c,c,\ldots]$$

$$app(app([a,a,\ldots],[b,b,\ldots]),[c,c,\ldots]) \not\to^*_{\mathcal{R}_2} [a,c|x]$$

Timbuk Demo (II)

$even(0) \rightarrow true$	$P(x) \rightarrow P(x) \mid\mid P(.)$
$odd(0) \rightarrow false$	$P(.) \mid\mid in([x\|y]) \rightarrow P(data(x)) \mid\mid in(y)$
$even(s(x)) \rightarrow odd(x)$	$P(.) \mid\mid in([\,]) \rightarrow in([\,])$
$odd(s(x)) \rightarrow even(x)$	$P(res(x)) \mid\mid out(y) \rightarrow P(.) \mid\mid out([x\|y])$
	$P(data(x)) \rightarrow P((even(x), x))$
	$P((true, x)) \rightarrow P(res(x))$
	$P((false, x)) \rightarrow P(.)$

Approximated case, $=_{App}$ such that :

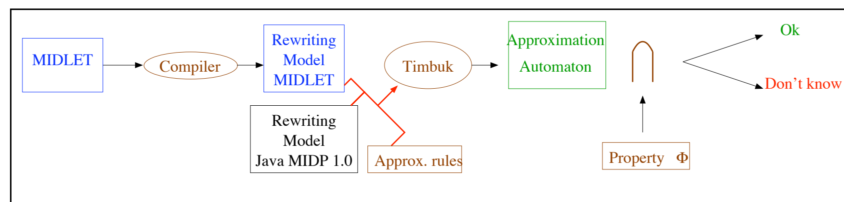
$(x \mid\mid y) =_{App} x =_{App} y \qquad cons(x, y) =_{App} y \qquad out(x) =_{App} out(y)$

- $P(.) \mid\mid in([s(0), s(s(0))]) \mid\mid out([\,]) \not\rightarrow_{\mathcal{R}}^* x \mid\mid out([s(0)])$
- $P(.) \mid\mid in(l_{nat}) \mid\mid out([\,]) \not\rightarrow_{\mathcal{R}}^* x \mid\mid out(l_{oneOdd})$

Outline

1. Program and property modeling using rewriting
2. Verification of properties on rewriting models : Inductive theorems
3. Verification of properties on rewriting models : Reachability analysis
4. A case study on a Java byte code static analysis

Approximation for proving properties on Java programs



Approximation for proving properties on Java programs (II)

Encoding Java semantics into rewriting

$$(pop) = \frac{(m, pc, x :: s, l)}{(m, pc + 1, s, l)}$$

$$\mathcal{R} = \left| \begin{array}{lcl} frame(name(foo, A), pp2, s, l) & \rightarrow & xframe(pop, name(foo, A), pp2, s, l) \\ xframe(pop, m, pc, stack(x, s), l) & \rightarrow & frame(m, next(pc), s, l) \\ next(pp2) & \rightarrow & pp3 \end{array} \right.$$

Initial terms/$\mathcal{A}^0$

$\mathcal{L}(\mathcal{A}^0) = \{$Initial Java program states to analyze$\}$

Reachable terms

$\mathcal{R}^*(\mathcal{L}(\mathcal{A}^0)) = \{$Any program state that is reachable by $\mathcal{R}\}$

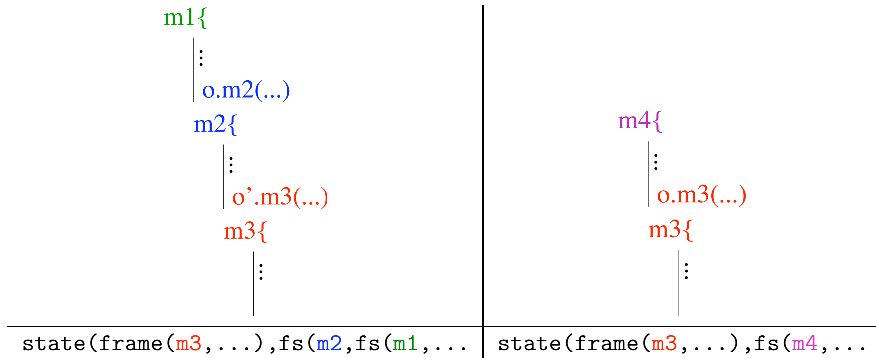
Properties : $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^0)) \cap Bad = \emptyset$

$Bad = \{$Program states where a malicious action has been performed $\}$

Approximation Methodology for Java Programs

Basic approximation — 0-CFA

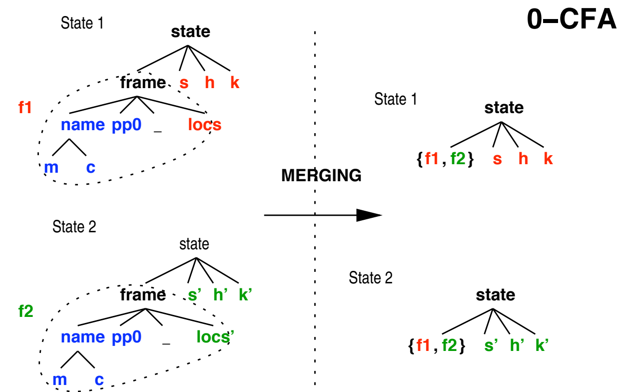
- Integers/Booleans are abstracted by their type
- Objects are abstracted by their class
- No difference between two calls to a method m



Approximation Methodology for Java Programs

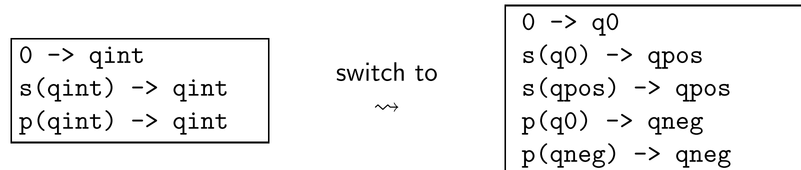
Basic approximation — 0-CFA

- Integers/Booleans are abstracted by their type
- Objects are abstracted by their class
- No difference between two calls to a method m



Approximation Methodology for Java Programs (II)

```
class TestList{
  public static void main(String[] argv){
    List lpos=null;
    InvList lneg=null;
    int x;
    boolean pos;
    pos= true;
    try {x=System.in.read();}
    catch(java.io.IOException e){x=0;}
    while (x != 0){
      if (pos) {lpos= new List(x, lpos);pos=false;}
      else {lneg= new InvList(x,lneg);pos=true;}
      [...]
```



Reachability Analysis with Timbuk in practice

The good

- better automation w.r.t. proof assistants (integrated in AVISPA)
- better control w.r.t. usual completion based methods
  - possible to adapt the approximation to the property to prove
  - possible to limit the size of the approximation on bigger models
    Ex. Needham Schroeder Public Key protocol (unbounded number of agents, sessions, intruder actions)

NSPK exact	NSPK approximated
partial (4 steps)	completed (6 steps)
4500 states	16 states

The bad

- limited to "regular approximations"
  - no inductive proofs
  - limited counting
  - limited relations between variables
- no way to discriminate between real attacks and approximations
- control on the proof is still limited w.r.t. proof assistant

Further research

- Trace reconstruction [Boichut Genet 2006]
- Cover more decidable classes with completion
- Optimize completion to do static analysis on « real » programs

  "Prove simple properties on big programs"
- Prove termination of completion for a given *App*
- Verification of safety/security properties on Java Midlets
- Certification of approximations (fixpoint checking)