Higher-Order Model Checking: Principles and Applications to Program Verification and Security

Part I: Types and Recursion Schemes
for Higher-Order Program Verification
Part II: Higher-Order Program Verification
and Language-Based Security

Naoki Kobayashi
Tohoku University

Why (Automated) Program Verification?

- Increasing Use of Software in Critical Systems
  - ATM, online banking, online shopping
  - Airplanes, automobiles
  - Nuclear power plant

  $\Rightarrow$ Reliability is becoming the primary concern

- Increase of Size/Complexity of Software

  $\Rightarrow$ Manual debugging is infeasible

Program Verification Techniques

- Model checking (c.f. 2007 Turing award)
  - Applicable to first-order procedures (pushdown model checking), but not to higher-order programs
- Type-based program analysis
  - Applicable to higher-order programs
  - Sound but imprecise
- Dependent types/theorem proving
  - Requires human intervention

Sound and precise verification techniques for higher-order programs (e.g. ML/Java programs)?

This Talk

- New program verification technique for higher-order languages (e.g. ML)
  - Sound, complete, and automatic for
    - A large class of higher-order programs
    - A large class of verification problems
  - Built on recent/new advances in
    - Type theories
    - Automata/formal language theories (esp. higher-order recursion schemes)
    - Model checking
- Applications to language-based security (part II)

Relevance to Security?
(for ASIAN audience)

- Program verification is relevant to software security
  - Prevent security holes
  - Verification techniques have been used for:
    - information flow analysis
    - access control
    - protocol verification
- Higher-order program verification brings new advantages
  - precise for higher-order programs
  - applicable to infinite-state systems

Outline

- Part I: Types and Recursion Schemes for Higher-Order Program Verification
  - Higher-order recursion schemes
  - From program verification to model checking recursion schemes [K. POPL09][K.,Tabuchi&Unno POPL10]
  - From model checking to type checking [K. POPL09][K.&Ong LICS09]
  - Type checking (=model checking) algorithm [K.PPDP09]
  - TRecS: Type-based RECursion Scheme model checker
  - Future perspectives
- Part II: Higher-order program verification for language-based security

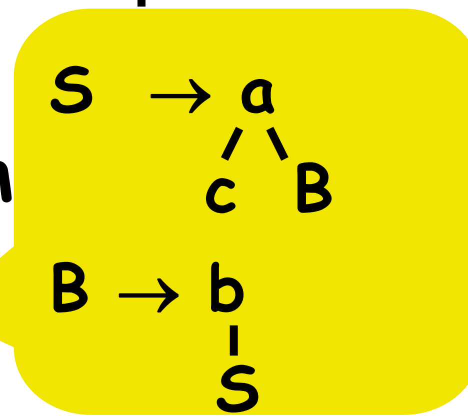
Higher-Order Recursion Scheme

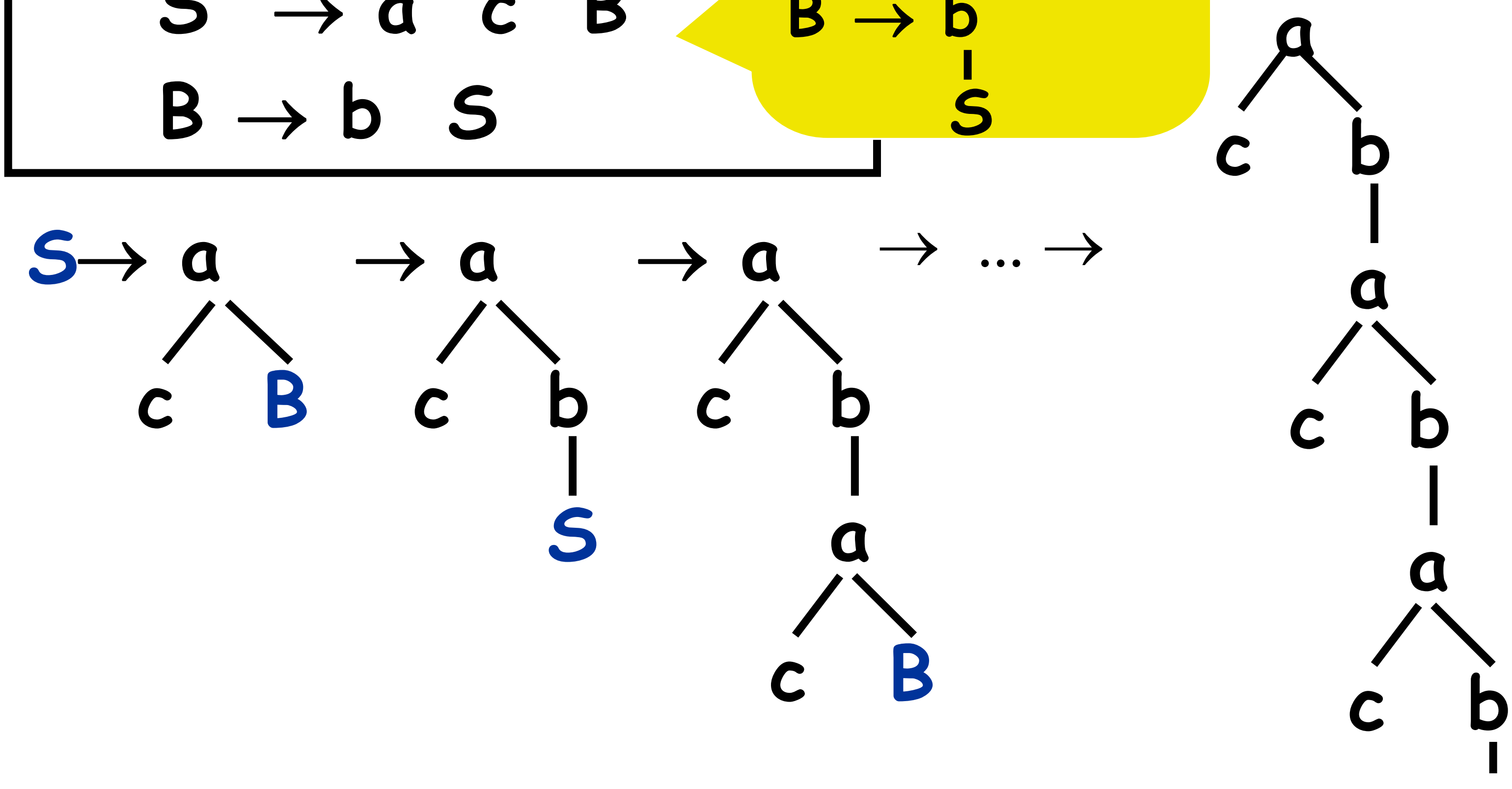
- Grammar for generating an infinite tree

Order-0 scheme
(regular tree gramm

$S \rightarrow a\ c\ B$

$B \rightarrow b\ S$





Higher-Order Recursion Scheme

- Grammar for ... ite tree

Order-1 schem...

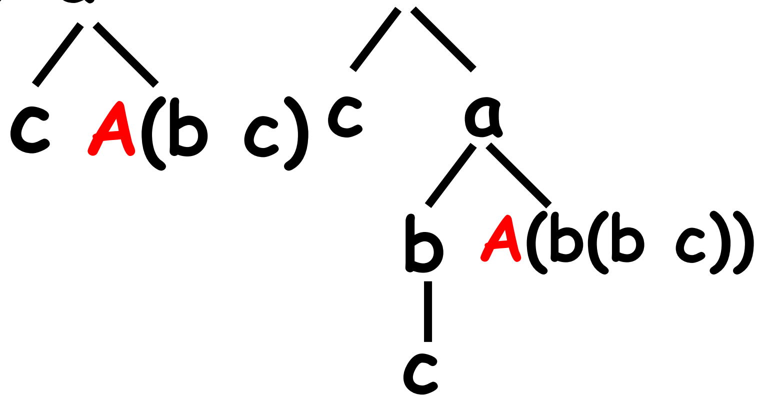
$S \rightarrow A\ c$

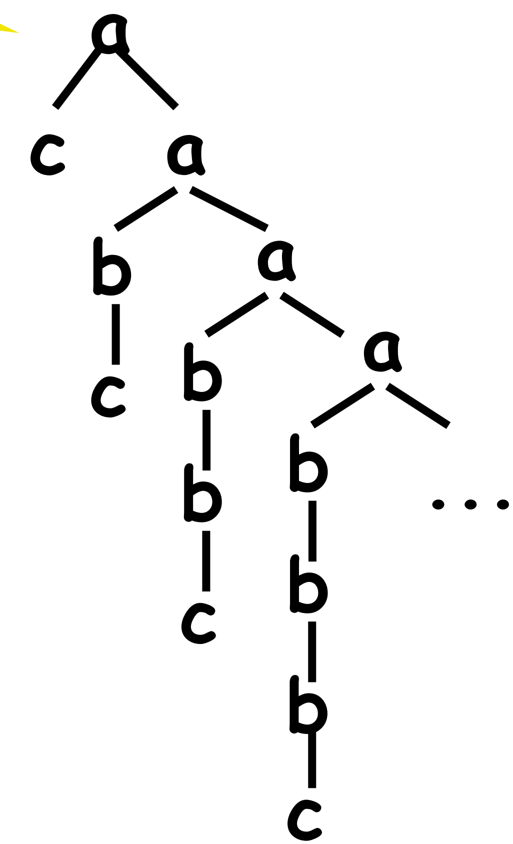
$A \rightarrow \lambda x.\ a\ \ x\ \ (A\ (b\ x))$

S: o, A: o→ o

Tree whose paths are labeled by $a^{m+1}\ b^m\ c$

$S \rightarrow A\ c \rightarrow a \quad \rightarrow a \quad \rightarrow \ \dots \ \rightarrow$





Model Checking Recursion Schemes

Given
G: higher-order recursion scheme
A: alternating parity tree automaton (APT)
(a formula of modal μ-calculus or MSO),
does A accept Tree(G)?

e.g.
- Does every finite path end with "c"?
- Does "a" occur eventually whenever "b" occurs?

n-EXPTIME-complete [Ong, LICS06]
(for order-n recursion scheme)

Why Recursion Schemes?

- **Expressive:**
  - Subsumes many other MSO-decidable tree classes (regular, algebraic, Caucal hierarchy, HPDS, ...)

- **High-level** (c.f. higher-order PDS):
  - Recursion schemes
    $\approx$
    Simply-typed λ-calculus
    \+ recursion
    \+ tree constructors (but not destructors)
    (+ finite data domains such as booleans)

Suitable models for higher-order programs

Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
- Type checking (=model checking) algorithm for recursion schemes
- TRecS: Type-based RECursion Scheme model checker
- Ongoing and future work

From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

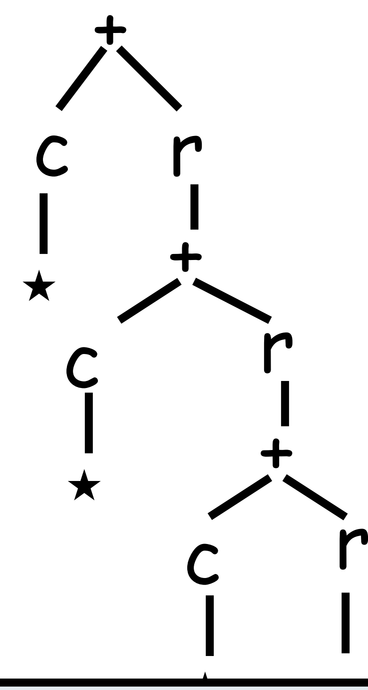
Higher-order program + specification → **Program Transformation** → Rec. scheme (describing all event sequences and outputs) + Tree automaton, recognizing valid event sequences → **Model Checking**

From Program Verification to Model Checking: Example

```
let f(x) =
   if * then close(x)
   else read(x); f(x)
in
let y = open "foo"
in
      f (y)
```

$F\ x\ k \rightarrow +\ (c\ k)\ (r(F\ x\ k))$

$S \rightarrow F\ d\ \star$



Is the file "foo" accessed according to read* close?

Is each path of the tree labeled by r*c?

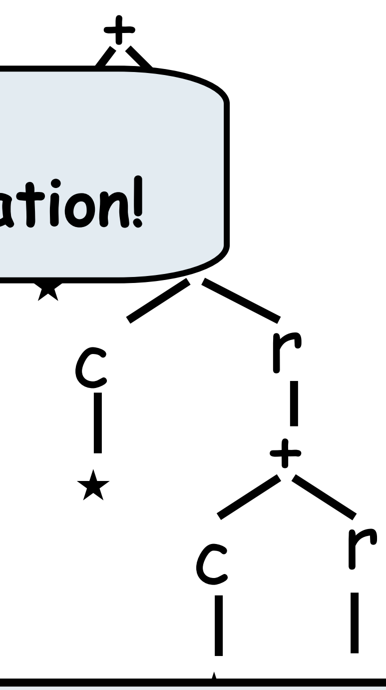
From Program Verification to Model Checking: Example

```
let f(x) =
  if * then close(x)
  else read(x); f(x)
in
let y = open "foo"
in
     f (y)
```

$$F\ x\ k \rightarrow +\ (c\ k)\ (r(F\ x\ k))$$

$$S \rightarrow F\ d\ \star$$

CPS Transformation!



Is the file "foo" accessed according to read* close?

Is each path of the tree labeled by r*c?

From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

Higher-order program + specification → **Program Transformation** → Rec. scheme (describing all event sequences) + automaton for infinite trees → **Model Checking**

Sound, complete, and automatic for:

- A large class of higher-order programs: simply-typed λ-calculus + recursion + finite base types
- A large class of verification problems: resource usage verification [Igarashi&K. POPL2002], reachability, flow analysis, ...

Comparison with Traditional Approach (Control Flow Analysis)

- Control flow analysis

Higher-order program → **Flow Analysis** → Control flow graph (finite state or pushdown machines) → **verification**

- Our approach

Higher-order program → **Program Transformation** → Recursion scheme → **verification**

Only information about infinite data domains is approximated!

Comparison with Traditional Approach (Software Model Checking)

Program Classes	Verification Methods
Programs with while-loops	Finite state model checking
Programs with 1^{st}-order recursion	Pushdown model checking
Higher-order functional programs	Recursion scheme model checking

infinite state model checking

Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
  - Goal and motivation
  - Type system equivalent to model checking
- Type checking (=model checking) algorithm
- TRecS: Type-based RECursion Scheme model checker
- Future perspectives

Goal

Construct a type system TS(A) s.t.

Tree(G) is accepted by APT A

if and only if

G is typable in TS(A)

Model Checking as
Type Checking
(c.f. [Naik & Palsberg, ESOP2005])

Why Type-Theoretic Characterization?

- **Simpler** decidability proof of model checking recursion schemes
  - Previous proofs [Ong, 2006][Hague et. al, 2008] made heavy use of game semantics
- **More efficient** model checking algorithm
  - Known algorithms [Ong, 2006][Hague et. al, 2008] **always** require n-EXPTIME

Outline

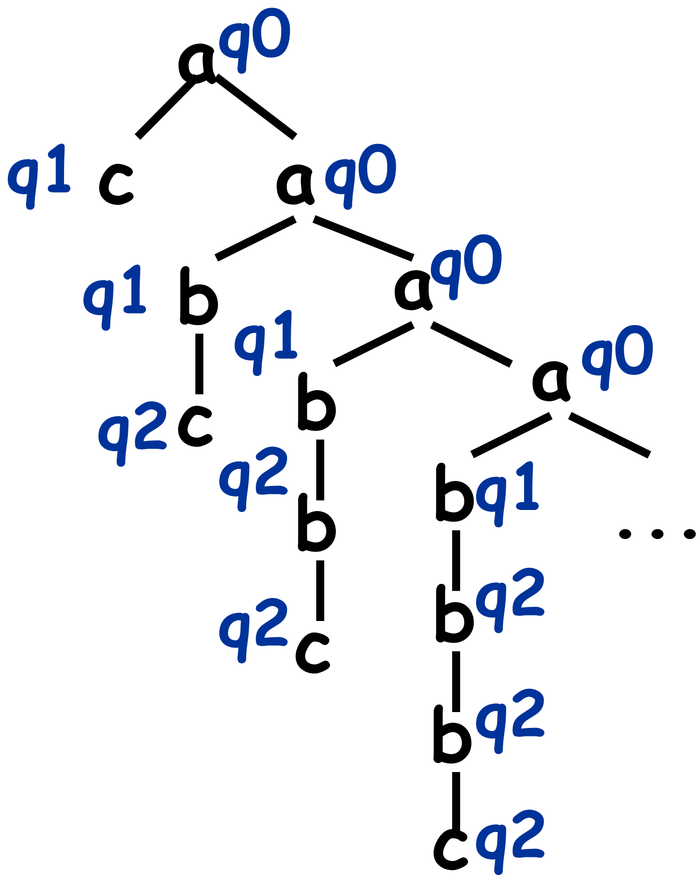
- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
  - Goal and motivation
  - Type system
- Type checking (=model checking) algorithm
- TRecS: Type-based RECursion Scheme model checker
- Future perspectives

Model Checking Problem
(Simple Case, for safety properties)

Given
G: higher-order recursion scheme
A: trivial automaton
(Büchi tree automaton where all the states are accepting states)
does A accept Tree(G)?

See [K.&Ong, LICS09] for the general case

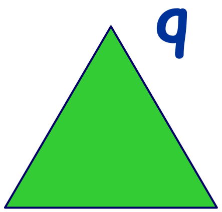
(Trivial) tree automaton for infinite trees



$\delta(q0,\ a) = q1\ q0$
$\delta(q1,\ b) = q2$
$\delta(q2,\ b) = q2$
$\delta(q1,\ c) = \varepsilon$
$\delta(q2,\ c) = \varepsilon$

Types for Recursion Schemes

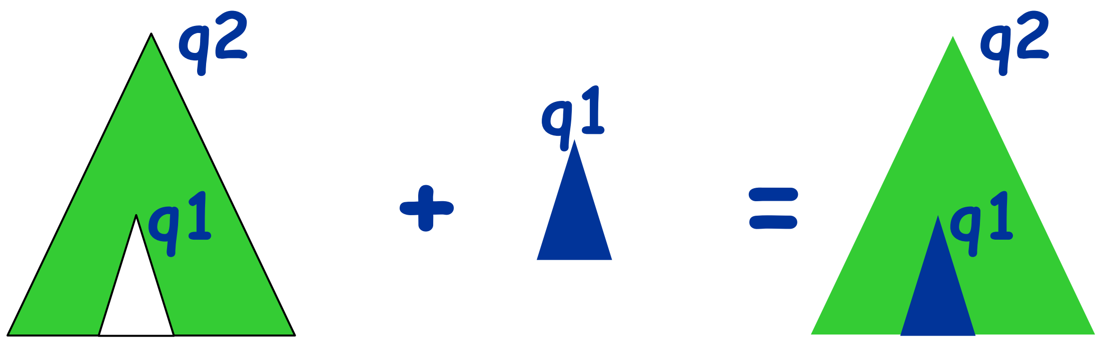
- Automaton state as the type of trees
  - q: trees accepted from state q



  - q1∧q2: trees accepted from both q1 and q2

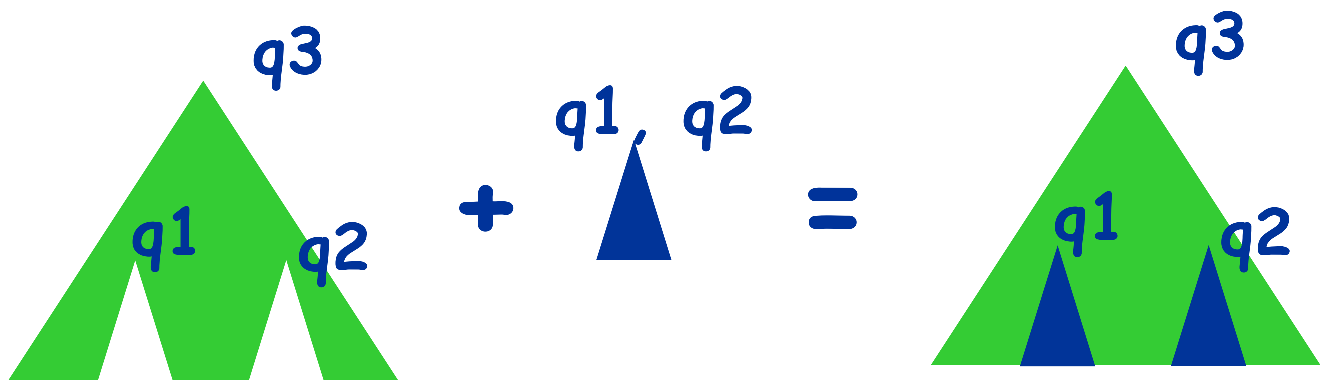
Types for Recursion Schemes

- Automaton state as the type of trees
  - $q1 \rightarrow q2$: functions that take a tree of type q1 and return a tree of q2



Types for Recursion Schemes

- Automaton state as the type of trees
  - $q1 \wedge q2 \rightarrow q3$:
    functions that take a tree of type $q1 \wedge q2$ and return a tree of type $q3$

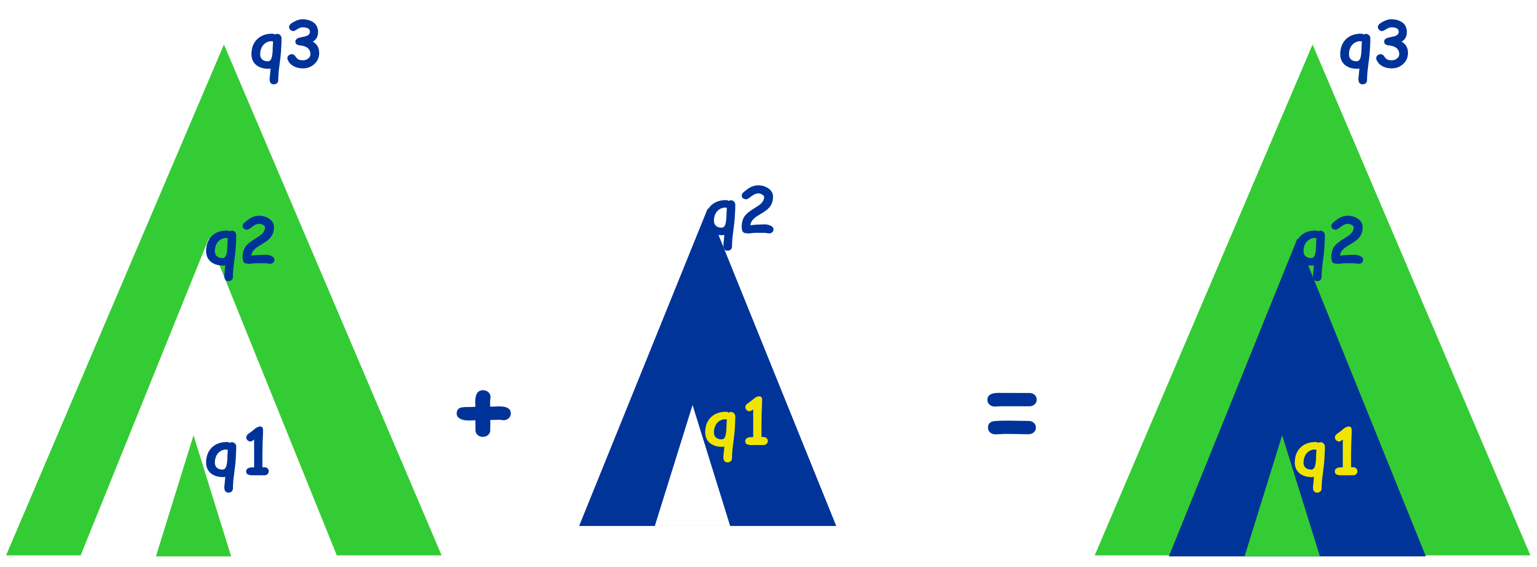


Types for Recursion Schemes

- **Automaton state as the type of trees**

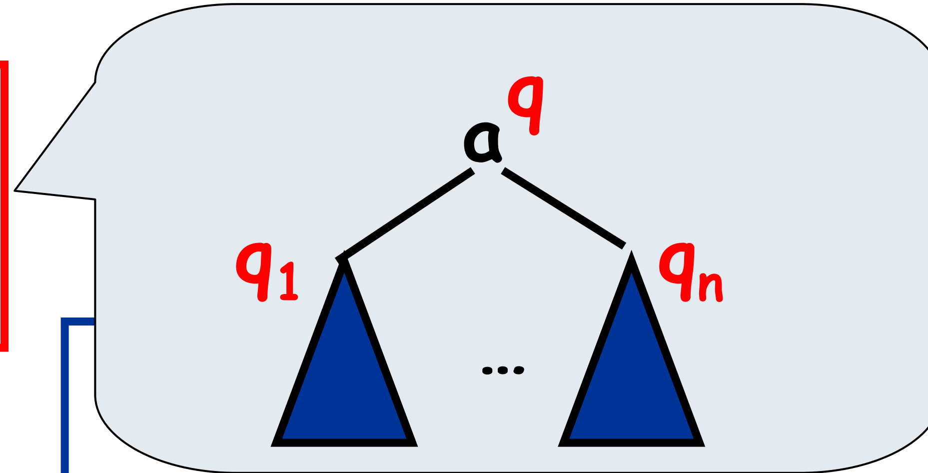
  $(q1 \to q2) \to q3$:

  functions that take a function of type $q1 \to q2$ and return a tree of type $q3$



Typing

$$\frac{\delta(q,\ a) = q_1 \ldots q_n}{\vdash a : q_1 \rightarrow \ldots \rightarrow q_n \rightarrow q}$$



$$\frac{\Gamma, x{:}\tau_1, \ldots, x{:}\tau_n \ \vdash t{:}\tau}{\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau}$$

$$\frac{\Gamma \vdash t_2{:}\tau_i\ (i=1,..n)}{\Gamma \vdash t_1\ t_2{:}\tau}$$

$$\frac{\Gamma \vdash t_k : \tau \text{ (for every } F_k{:}\tau \in \Gamma)}{\vdash \{F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n\} : \Gamma}$$

Soundness and Completeness [K., POPL2009]

Let

- G: Rec. scheme with initial non-terminal S
- A: Trivial automaton with initial state q_0
- TS(A): Intersection type system derived from A

Then,

Tree(G) is accepted by A
if and only if
S has type q_0 in TS(A)

Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
- Type checking (=model checking) algorithm
  - Naive algorithm
  - Practical algorithm
- TRecS: Type-based RECursion Scheme model checker
- Future perspectives

Typing

$$\frac{\delta(q,\ a) = q_1 \ldots q_n}{\vdash a : q_1 \rightarrow \ldots \rightarrow q_n \rightarrow q}$$

$$\Gamma, x{:}\tau \ \vdash x : \tau$$

$$\frac{\Gamma, x{:}\tau_1, \ldots, x{:}\tau_n \ \vdash t{:}\tau}{\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau}$$

$$\frac{\Gamma \vdash t_1{:}\ \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau \qquad \Gamma \vdash t_2{:}\tau_i\ (i=1,\ldots n)}{\Gamma \vdash t_1\ t_2{:}\tau}$$

$$\frac{\Gamma \vdash t_k : \tau \text{ (for every } F_k{:}\tau \in \Gamma)}{\vdash \{F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n\} : \Gamma}$$

Naïve Type Checking Algorithm

S has type q_0

(i) $\Gamma \vdash t_k : \tau$
for each $F_k : \tau \in \Gamma$
(ii) $S : q_0 \in \Gamma$
for some Γ

Recursion Scheme:
$\{F_1 \to t_1, \ldots, F_m \to t_m\}$

Filter out invalid type bindings

$S : q_0 \in \mathrm{gfp}(H) = \bigcap_k H^k(\Gamma_{max})$
where
$H(\Gamma) = \{ F_k : \tau \in \Gamma \mid \Gamma \vdash t_k : \tau \}$
$\Gamma_{max} = \{ F : \tau \mid \tau :: \mathrm{sort}(F) \}$

All the possible type bindings
E.g. for $F : o \to o$,
$\{F : \top \to q0,\ F : q0 \to q0,$
$F : q1 \to q0,$
$F : q0 \wedge q1 \to q0, \ldots\}$

Naïve Algorithm Does NOT Work

S has type q_0

$\Updownarrow$

$S{:}q_0 \in gfp(H) = \bigcap_k H^k(\Gamma_{max})$

where $H(\Gamma) = \{ F{:}\tau \in \Gamma \mid \Gamma \vdash G(F){:}\tau \}$

$\Gamma_{max} = \{F{:}\tau \mid \tau :: sort(F) \}$

This is huge!

sort	# of types ($Q=\{q_0,q_1,q_2,q_3\}$)
o	4 (q_0,q_1,q_2,q_3)
$o \to o$	$2^4 \times 4 = 64$ ($\wedge S \to q$, with $S \in 2^Q$, $q \in Q$)
$(o\to o) \to o$	$2^{64} \times 4 = 2^{66}$
$((o\to o) \to o) \to o$	$2^{2^{66}} \times 4 > 10^{10000000000000000000}$

Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
- Type checking (=model checking) algorithm for recursion schemes
  - Naive algorithm
  - Practical algorithm
- TRecS: Type-based RECursion Scheme model checker
- Future perspectives

More Efficeint Algorithm?

S has type q_0

$\Leftarrow$

$S{:}q_0 \in \bigcap_k H^k(\cancel{\Gamma_{max}}\ \Gamma_0)$

where

$$H(\Gamma) = \{\ F{:}\tau \in \Gamma \mid \Gamma \vdash G(F){:}\tau\ \}$$

Challenges:

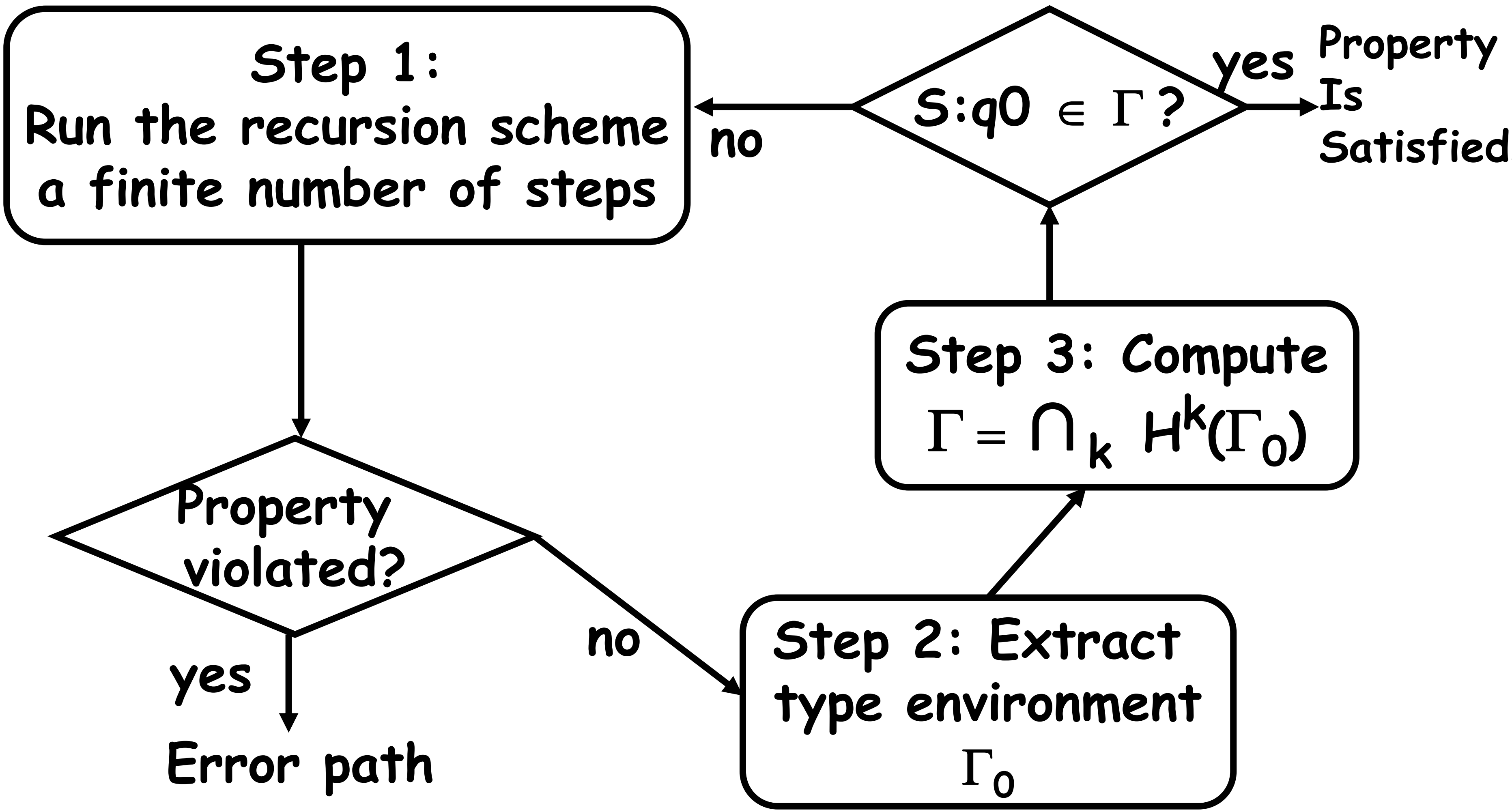
(i) How can we find an appropriate Γ_0?

"Run" the recursion scheme (finitely many steps), and extract type information

(ii) How can we guarantee completeness?

Iteratively repeat (i) and type checking

Hybrid Type Checking Algorithm



Soundness and Completeness of the Hybrid Algorithm

Given:

- Recursion scheme G
- Deterministic trivial automaton A,

the algorithm eventually terminates, and:

(i) outputs an error path
if Tree(G) is not accepted by A

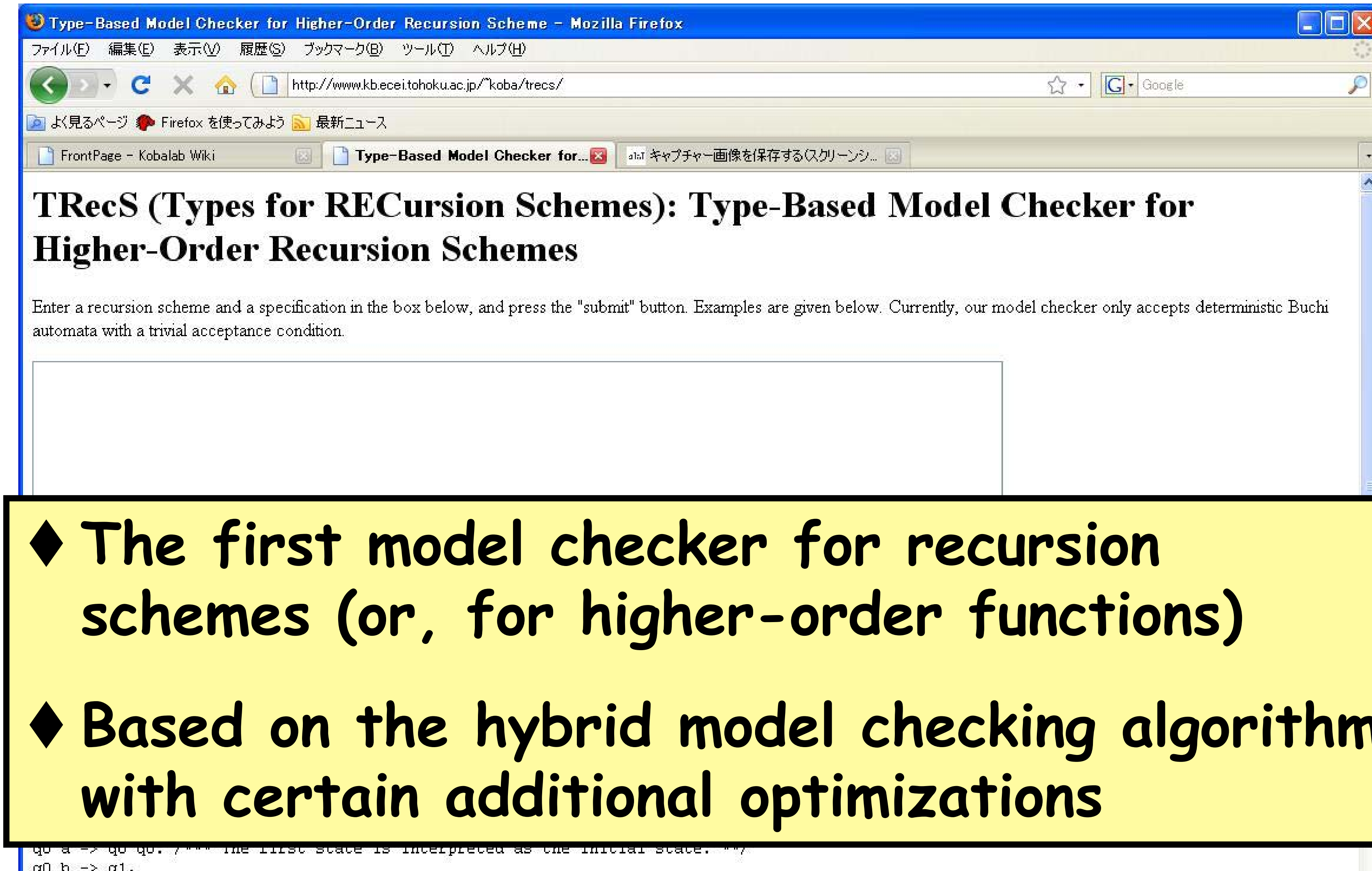
(ii) outputs a type environment
if Tree(G) is accepted by A

Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
- Type checking (=model checking) algorithm for recursion schemes
- TRecS: Type-based RECursion Scheme model checker
- Future perspectives

TRecS

http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/



- The first model checker for recursion schemes (or, for higher-order functions)
- Based on the hybrid model checking algorithm, with certain additional optimizations

Experiments

	order	rules	states	result	Time (msec)
Twofiles	4	11	4	Yes	2
FileWrong	4	[illegible]	[illegible]	[illegible]	[illegible]
TwofilesE	4	[illegible]	[illegible]	[illegible]	[illegible]
FileOcamlC	4	23	4	Yes	5
Lock	4	11	3	Yes	5
Order5	5	9	4	Yes	2

Taken from the compiler of Objective Caml, consisting of about 60 lines of O'Caml code

(Environment: Intel(R) Xeon(R) 3Ghz with 2GB memory)

(A simplified version of) FileOcamlC

```
let readloop fp =
  if * then () else readloop fp; read fp
let read_sect() =
  let fp = open "foo" in
  {readc=fun x -> readloop fp;
   closec = fun x -> close fp}
let loop s =
  if * then s.closec() else s.readc();loop s
let main() =
  let s = read_sect() in loop s
```

Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
- Type checking (=model checking) algorithm for recursion schemes
- TRecS: Type-based RECursion Scheme model checker
- Discussion
  - Advantages of our approach
  - Remaining challenges

Advantages of our approach

(1) Sound, complete and automatic for a large class of higher-order programs

- no false alarms!
- no annotations

Advantages of our approach

(1) Sound, complete and automatic for a large class of higher-order programs
  - no false alarms!
  - no annotations

(2) Subsumes finite-state/pushdown model checking
  - Order-0 rec. schemes ≈ finite state systems
  - Order-1 rec. schemes ≈ pushdown systems

Advantages of our approach

(3) Take the best of model checking and types

- Types as certificates of successful verification
  $\Rightarrow$ applications to PCC (proof-carrying code)

- Counter-example when verification fails
  $\Rightarrow$ error diagnosis,
  CEGAR (counter-example-guided abstraction refinement)

Advantages of our approach

(4) Encourages structured programming

Previous techniques:

- Imprecise for higher-order functions and recursions, hence discourage using them

```
Main:
 fp1 := open "r" "foo";
 fp2 := open "w" "bar";
Loop:
 c1 := read fp1;
 if c1=eof then goto E;
 write(c1, fp2);
 goto Loop;
E:
 close fp1;
 close fp2;
```

v.s.

```
let copyfile fp1 fp2 =
   try write(read fp2, fp1);
         copyfile fp1 fp2
    with
       Eof -> close(fp1);close(fp2)
let main =
   let fp1 = open "r" file in
   let fp2 = open "w" file in
       copyfile fp1 fp2
```

Advantages of our approach

(4) Encourages structured programming

Our technique:

- No loss of precision for higher-order functions and recursions
- Performance penalty? -- Not necessarily!
  - n-EXPTIME in the specification size, but polynomial time in the program size
  - Compact representation of large state space

    e.g. recursion schemes generating $a^m(c)$

    $S \rightarrow F_1\ c,\ F_1\ x \rightarrow F_2(F_2\ x), \ldots,\ F_n\ x \rightarrow a(a\ x)$

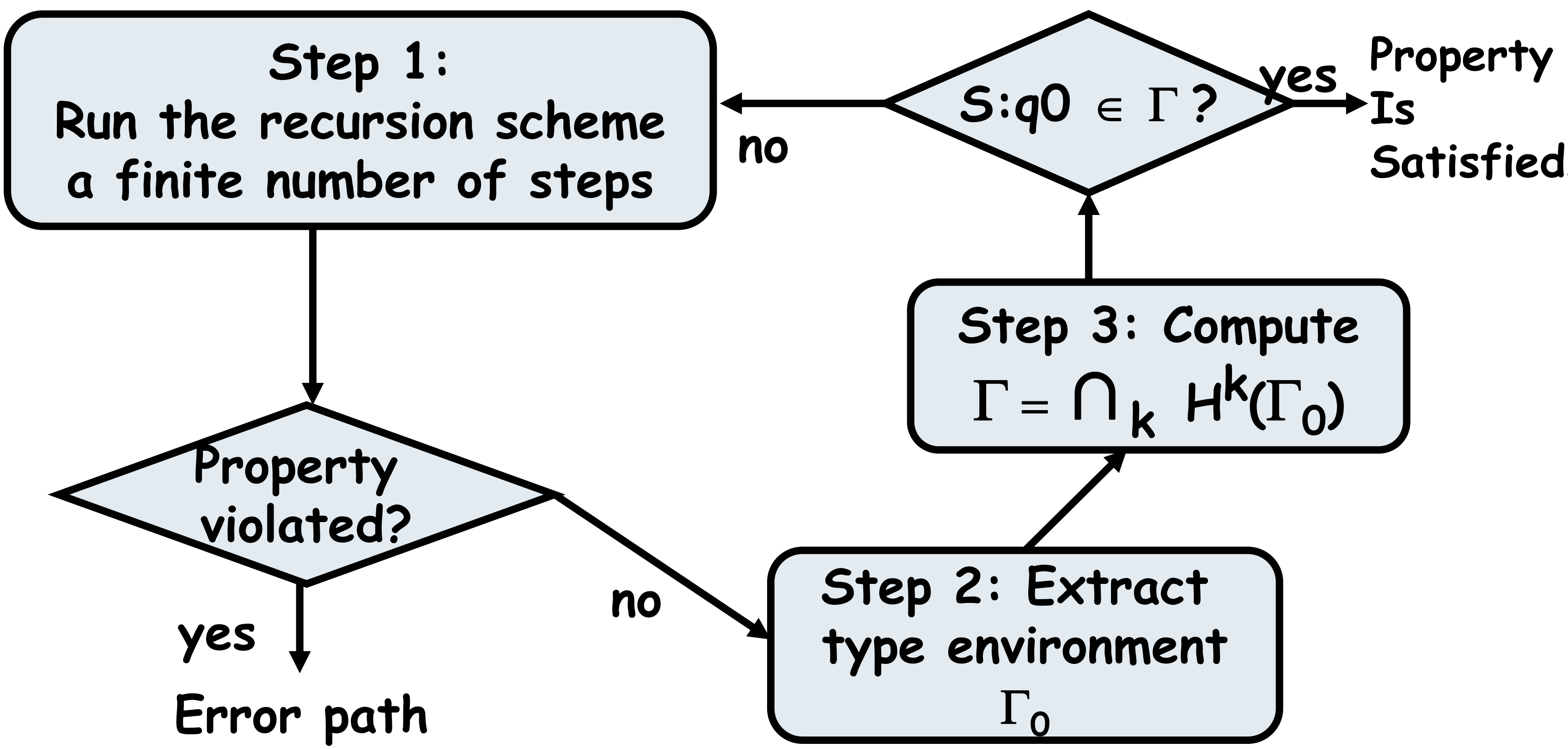
    vs

    $S \rightarrow a\ G_1,\ G_1 \rightarrow a\ G_2, \ldots,\ G_m \rightarrow c \quad (m=2^n)$

Advantages of our approach

(5) A good combination with testing:

Verification through testing



Outline

- Higher-order recursion schemes
- From program verification to model checking recursion schemes
- From model checking to type checking
- Type checking (=model checking) algorithm for recursion schemes
- TRecS: Type-based RECursion Scheme model checker
- Discussion
  - Advantages of our approach
  - Remaining challenges

Challenges

(1) More efficient recursion scheme model checker

- More results on language-theoretic properties of recursion schemes (e.g. pumping lemmas)

- BDD-like representation for higher-order functions

Challenges

(2) A software model checker
(on top of a recursion scheme model checker)

- predicate abstraction and CEGAR
  for infinite base types (e.g. integers)

- automaton abstraction for algebraic
  data types [K. et al. POPL2010]

- imperative features and concurrency

Challenges

(3) Extend the model checking problem:
$\text{Tree}(G) \models \varphi$

- Beyond "simply-typed" recursion schemes
  [Tsukada&K., FOSSACS 2010]
  - polymorphism
  - recursive types
- Beyond regular properties (MSO)
  Is there a more expressive, decidable logic?

Conclusion (for Part I)

- New program verification technique based on model checking recursion schemes
  - Many attractive features
    - Sound and complete for higher-order programs
    - Take the best of model-checking and type-based techniques
  - Many interesting and challenging topics

References

- K., Types and higher-order recursion schemes for verification of higher-order programs, POPL09

  From program verification to model-checking, and
  from model-checking to typing
- K.&Ong, Complexity of model checking recursion schemes for fragments of the modal mu-calculus, ICALP09 Complexity of model checking
- K.&Ong, A type system equivalent to modal mu-calculus model-checking of recursion schemes, LICS09

  From model-checking to type checking
- K., Model-checking higher-order functions, PPDP09

  Type checking (= model-checking) algorithm
- K., Tabuchi & Unno, Higher-order multi-parameter tree transducers and recursion schemes for program verification, POPL10 Extension to transducers and its applications

Higher-Order Model Checking: Principles and Applications to Program Verification and Security

Part I: Types and Recursion Schemes
for Higher-Order Program Verification
Part II: Higher-Order Program Verification
and Language-Based Security

Naoki Kobayashi
Tohoku University

Language-Based Security

- Enforcing software security
  - at a programming language level
  - by using programming language techniques
    (types, program analysis, compilation, run-time monitoring, ...)
- Applications
  - information flow/integrity
    - Can I run this program without leaking secret information?
    - Can I trust the output of this program?
  - access control
  - protocol verification
- Advantages (c.f. operating system-based approach)
  - static guarantees
  - high-level assurance

Program Verification Techniques for Security

- Played a key role in language-based security
  - Information flow
    - types [Volpano&Smith] [Myer 99] ...
    - model checking (self-composition)[Barth et al. 04] ...
    - combination [Terauchi 05][Unno&K. 06]
  - Access control (e.g. JVM)
    - types [Pottier et al. 01][Higuichi&Ohori, 03]
    - model checking [Nitta et al. 01]
  - Protocol verification
    - types [Gordon&Jefferey] [Kikuchi&K. 09]
    - model checking

This Talk

- Higher-order model checking for language-based security
  - Applications
    - information flow
    - access control (stack inspection)
  - Advantages
    - more precise than previous type-based approach
      (more programs can be statically checked to be safe)
    - more faithful modeling of software
      than previous model-checking
      (higher-order functions and recursion)
  - Limitations

Outline

- ♦ Model checking higher-order boolean programs
- ♦ Information flow
  - Problem definition
  - Reduction to higher-order model checking
- ♦ Stack-based access control
  - Problem definition
  - Reduction to higher-order model checking

Model-checking Higher-Order Boolean Programs (HBP)

- Language:
  simply-typed λ + recursion + booleans

  $$M \text{ (terms) } ::= x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{fix}(f,x,M) \mid M_1 M_2 \mid \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3$$

  $$\tau \text{ (types) } ::= \mathsf{bool} \mid \tau \to \tau$$

- Model checking problem:
  Given M:bool and $b \in \{\mathsf{true}, \mathsf{false}\}$,
  decide whether $M \Downarrow b$

Decidable, by a straightforward encoding into recursion scheme model checking
($\mathsf{true} = \lambda x.\lambda y.x$, $\mathsf{false} = \lambda x.\lambda y.y$)

Outline

- Model checking higher-order boolean programs
- Information flow analysis
  - Problem definition
  - Reduction to higher-order model checking
- Stack-based access control
  - Problem definition
  - Reduction to higher-order model checking

Information Flow Analysis

- Static program analysis to check flow of information

```
pub := if s < passwd then 0 else 2
```

Insecure

```
tmp := if s < passwd then 0 else 2;
pub := tmp mod 2
```

Secure

Information Flow Analysis (IFA) for Higher-Order Boolean Programs

secret input | public input | public output

- Given a program M: $\tau \to \sigma \to$ bool, check whether

  $M\ u\ w \Downarrow b$ iff $M\ v\ w \Downarrow b$

  for all $u, v \in Val(\tau)$ and $w \in Val(\sigma)$

  and $b \in \{true, false\}$

IFA via Higher-Order Model Checking

IFA Problem:
Given a program $M: \tau \rightarrow \sigma \rightarrow \text{bool}$,
check whether

$$M\ u\ w \Downarrow b \quad \text{iff} \quad M\ v\ w \Downarrow b$$

for all $u,v \in Val(\tau)$, $w \in Val(\sigma)$ and $b \in \{\text{true}, \text{false}\}$

"Procedure" based on higher-order model-checking
1. Enumerate all u, v, w (up to $\approx_\tau$ and $\approx_\sigma$)
2. Check $M\ u\ w \Downarrow b$ and $M\ v\ w \Downarrow b$ for each u,v,w,b

Theorem: IFA is decidable if $\tau = \sigma = \text{bool}$

IFA via Higher-Order Model Checking: Limitations

- IFA for HBP is undecidable in general, due to undecidability of finitary PCF [Loader01]

"Procedure" based on higher-order model-checking
1. Enumerate all u, v, w (up to $\approx_\tau$ and $\approx_\sigma$) No such algorithm!
2. Check M u w$\Downarrow$b and M v w$\Downarrow$b for each u,v,w,b

Solution: Over-approximate definable functions, to get a sound but incomplete IFA algorithm

- Only finite base types are allowed

Solution: Use self-composition [Barthe et al. 04] and predicate abstraction

Outline

- Model checking higher-order boolean programs
- Information flow analysis
- Stack-based access control
  - Problem definition
  - Reduction to higher-order model checking

Java's Stack-Based Access Control (SBAC)

- Prevent untrusted code's indirect access to resources

```
System: /* trusted, allowed to access files */
   void grepfile(file, key) {
         fp = open(file);
         ... /* print a line that contains key */
   }

Applet:  /* untrusted, not allowed to access files */
    grepfile("/etc/passwd", "kobayashi");
```

Security Violation!

A callee is given the least privilege in the call sequence

λ-calculus with SBAC

[Pottier et al. 01][Gordon&Fournet 01]

M ::= x | fix(f,x,M) | M1M2
   | R[M] | check r then M | ...

Evaluate M with permissions R

Check that r is in the current permission

```
let grepfile f k =
  {open}[
    check open then ...] in
∅[ grepfile ...]
→*
∅[{open}[
   check open then ...
]]
```

```
System: /* trusted */
   void grepfile(file, key) {
          fp = open(file);
          ...
   }
Applet:  /* untrusted*/
 grepfile("/etc/passwd","koba");
```

Fails, as the current permission is $\emptyset \cap \{open\} = \emptyset$

λ-calculus with SBAC

[Pottier et al. 01][Gordon&Fournet 01]

M ::= x | fix(f,x,M) | M1M2
| R[M] | check r then M | ...

Evaluate M with permissions R

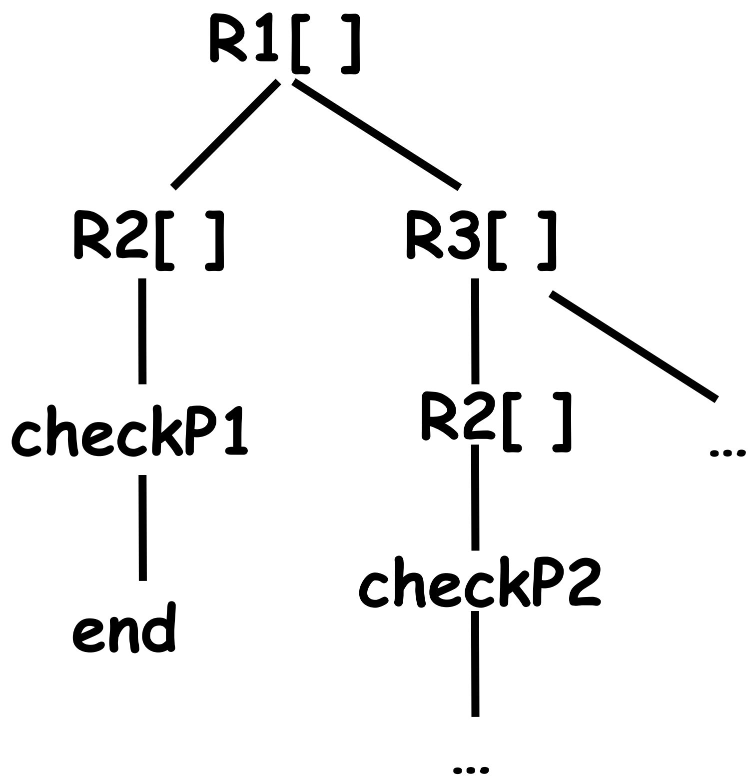
Check that r is in the current permission

Static SBAC Problem:

Given a (closed) SBAC program M, decide whether $M \to^* \text{FAIL}$

From SBAC to Higher-Order Model Checking

- Transform a SBAC program into a recursion scheme that *generates* "call tree"



From SBAC to Higher-Order Model Checking

- Transform a SBAC program into a recursion scheme that generates "call tree"

```
let f x = Trusted[check Trusted in x] in
let g x = Untrusted[f x] in
    g d
```

Continuation that takes a call tree as an additional argument

$F\ x\ k \rightarrow k\ (\text{trusted}\ (\text{checkT}\ \text{end}))\ x$
$G\ x\ k \rightarrow F\ x\ (\lambda t.\lambda x.k\ (\text{untrusted}\ t)\ x)$
$S \rightarrow G\ d\ (\lambda t.\lambda x.t)$

From SBAC to Higher-Order Model Checking

- Transform a SBAC program into a recursion scheme that generates "call trees"

```
let f x = Trusted[check Trusted in x] in
let g x = Untrusted[f x] in
    g d
```

Continuation that takes a call tree as an additional argument

$F\ x\ k \rightarrow k\ (\mathrm{trusted}\ (\mathrm{checkT}\ \mathrm{end}))\ x$
$G\ x\ k \rightarrow F\ x\ (\lambda t.\lambda x.k\ (\mathrm{untrusted}\ t)\ x)$
$S \rightarrow G\ d\ (\lambda t.\lambda x.t)$

$S \rightarrow G\ d\ (\lambda t.\lambda x.t)$
$\rightarrow F\ x\ (\lambda t.\lambda x.\ (\lambda t.\lambda x.t)\ (\mathrm{trusted}\ t)\ x)$
$\rightarrow F\ x\ (\lambda t.\lambda x.\ (\mathrm{untrusted}\ t))$
$\rightarrow (\lambda t.\lambda x.\ (\mathrm{untrusted}\ t))\ (\mathrm{trusted}\ (\mathrm{checkT}\ \mathrm{end}))\ x$
$\rightarrow \mathrm{untrustd}\ (\mathrm{trusted}\ (\mathrm{checkT}\ \mathrm{end}))$

From SBAC to Higher-Order Model Checking

- Transform a SBAC program into a recursion scheme that generates "call trees"

```
let f x = Trusted[check Trusted in x] in
let g x = Untrusted[f x] in
     g d
```

$$F\ x\ k \rightarrow k\ (\mathrm{trusted}\ (\mathrm{checkT}\ \mathrm{end}))\ x$$
$$G\ x\ k \rightarrow F\ x\ (\lambda t.\lambda x.k\ (\mathrm{untrusted}\ t)\ x)$$
$$S \rightarrow G\ d\ (\lambda t.\lambda x.t)$$

$S \rightarrow^* \mathrm{untrusted}\ (\mathrm{trusted}\ (\mathrm{checkT}\ \mathrm{end}))$ =

untrusted
|
trusted
|
checkT
|
end

From SBAC to Higher-Order Model Checking

- Transform a SBAC program into a recursion scheme that generates "call trees"

```
let f x = Trusted[check Trusted in x] in
let g x = Untrusted[f x] in
     g d
```

↓

$F\ x\ k \rightarrow k\ (\mathrm{frameT}\ (\mathrm{checkT}\ \mathrm{end}))\ x$
$G\ x\ k \rightarrow F\ x\ (\lambda t.\lambda x.k\ (\mathrm{frameU}\ t)\ x)$
$S \rightarrow G\ d\ (\lambda t.\lambda x.t)$

Static SBAC problem is decidable for simply-typed programs with finite base types if the set of permissions is finite

Limitations of higher-order model checking for SBAC

- Applicable only to simply-typed programs with recursion and finite base types

For infinite base types (e.g. integers),
use predicate abstractions
to get a sound but incomplete algorithm

Summary (for Part II)

- Higher-order model checking provides:
  - sound, complete, and certifying verification methods
  - for various security-related problems,
  - with some intrinsic restrictions on target programs
    - simply-typed
    - only finite base types
    - closed programs of low order types
      (c.f. undecidability of λ-definability)
- For practical programming languages,
  - predicate abstraction may be applicable
    to get a sound but incomplete method
  - more studies are required to evaluate effectiveness

Lessons Learned

- As a researcher on program verification
  - Keep an eye on new theoretical results (esp. on decision problems)
  - Do not worry too much about the worst-case complexity (e.g. SAT, recursion schemes), or undecidability (e.g. termination analysis)
- As a researcher on language-based security
  - Keep an eye on new verification techniques (e.g. program analysis based on linear programming [Terauchi&Aiken], SAT solvers, recursion schemes)