# 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

  ⇒ Reliability is becoming the primary concern

- Increase of Size/Complexity of Software

  ⇒ 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
  - From model checking to type checking
  - Type checking (=model checking) algorithm

  technical summary

  - TRecS: Type-based RECursion Scheme model checker
  - Future perspectives
- Part II: Higher-order program verification for language-based security

# Higher-Order Recursion Scheme

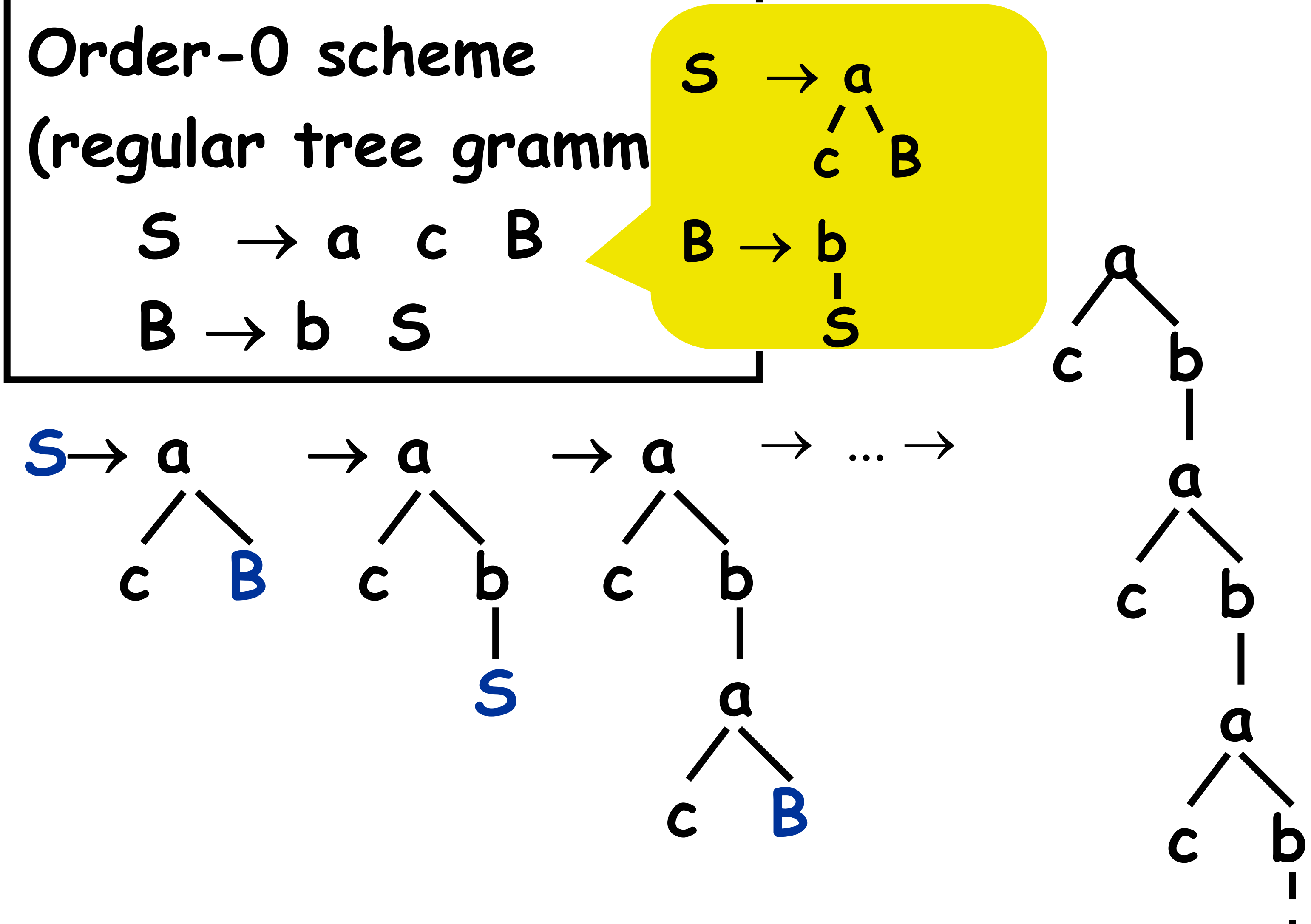
- Grammar for generating an infinite tree

Order-0 scheme

(regular tree gramm

S → a c B

B → b S



# Higher-Order Recursion Scheme

- Grammar for ite tree

Order-1 schem

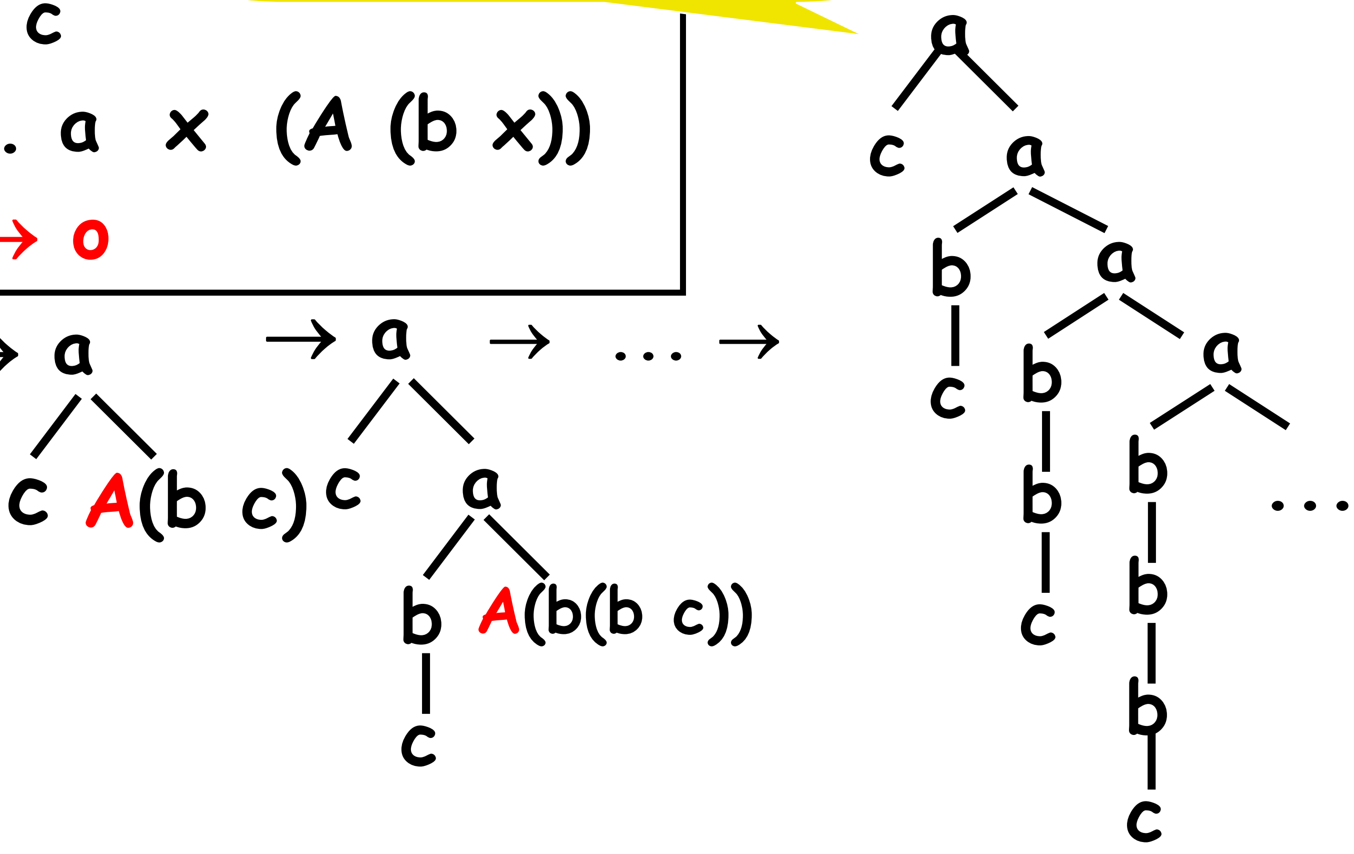
$S \to A\ c$

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

$S: o,\ A: o \to o$

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

$S \to A\ c \to$



# Model Checking Recursion Schemes

Given
G: higher-order recursion scheme
A: alternating parity tree automaton (APT)
(a formula of modal $\mu$-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 $\lambda$-calculus
    + recursion
    + tree constructors (but not destructors)
    (+ finite data domains such as booleans)

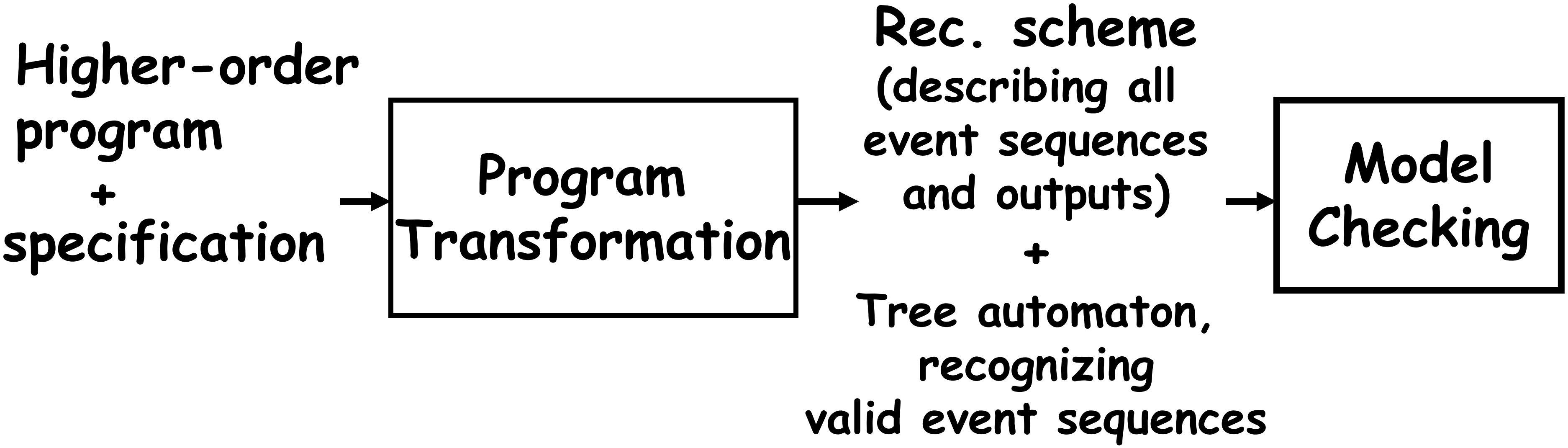
Suitable models for higher-order programs

# 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

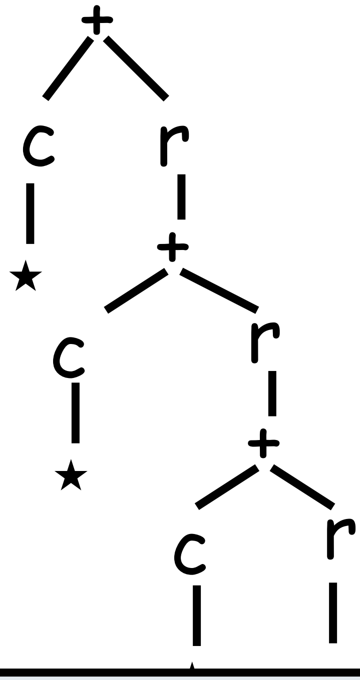
Model Checking

# 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?

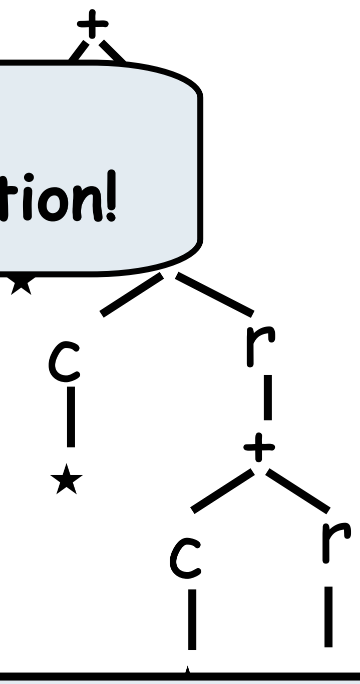
# From Program Verification to Model Checking: Example

```
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!

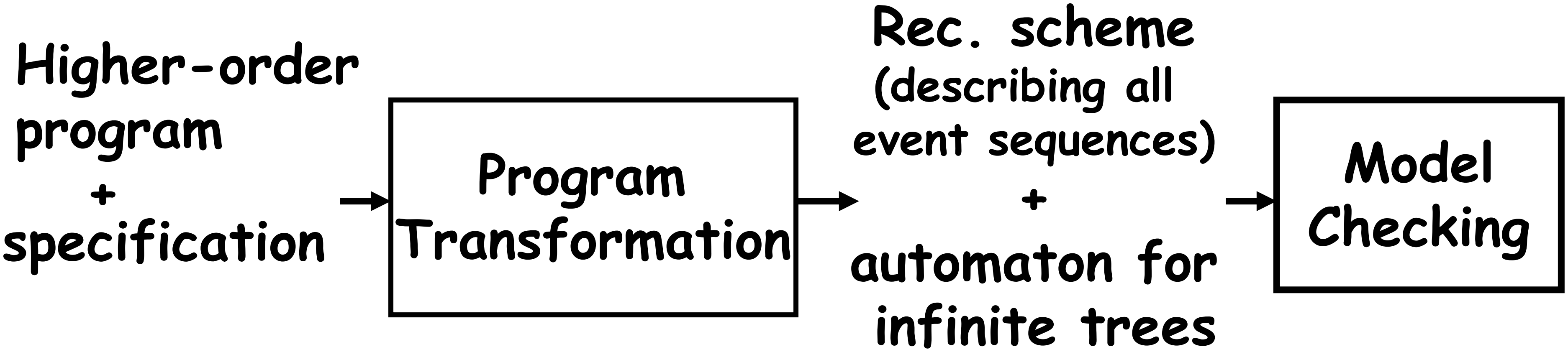


Is the file “foo” accessed according to read* close?

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

# From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]



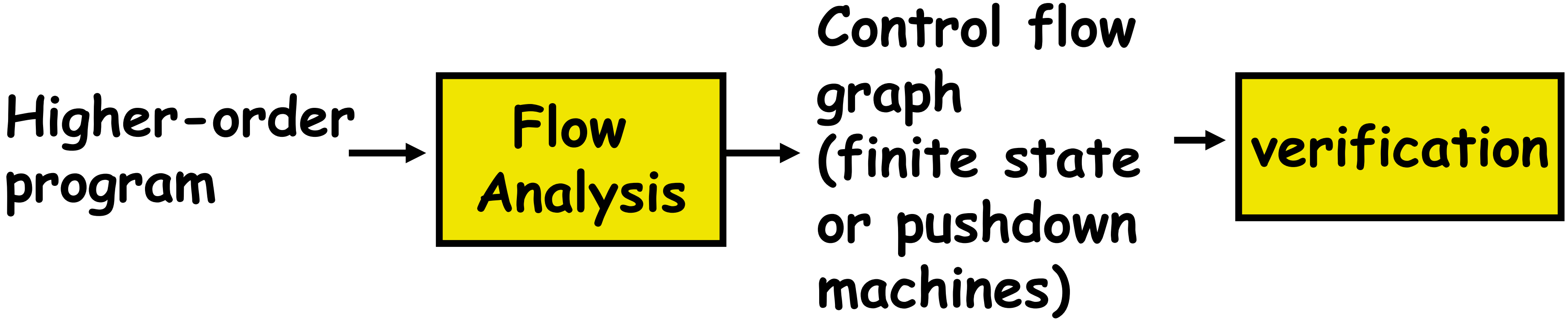
Higher-order program + specification → **Program Transformation** → Rec. scheme (describing all event sequences) + automaton for infinite trees → **Model Checking**

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, ...

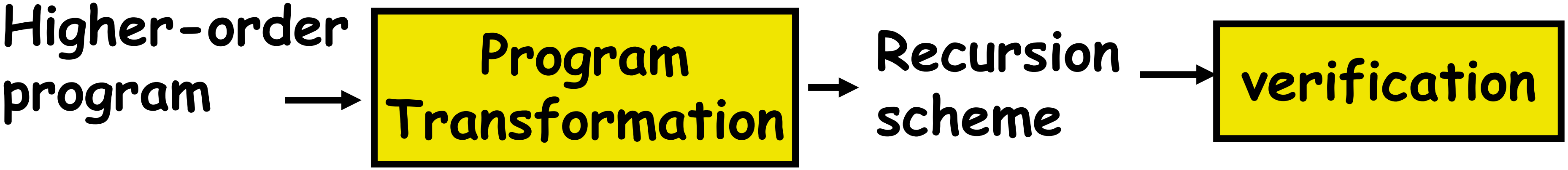
# Comparison with Traditional Approach (Control Flow Analysis)

- 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 1st-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 tree automaton *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

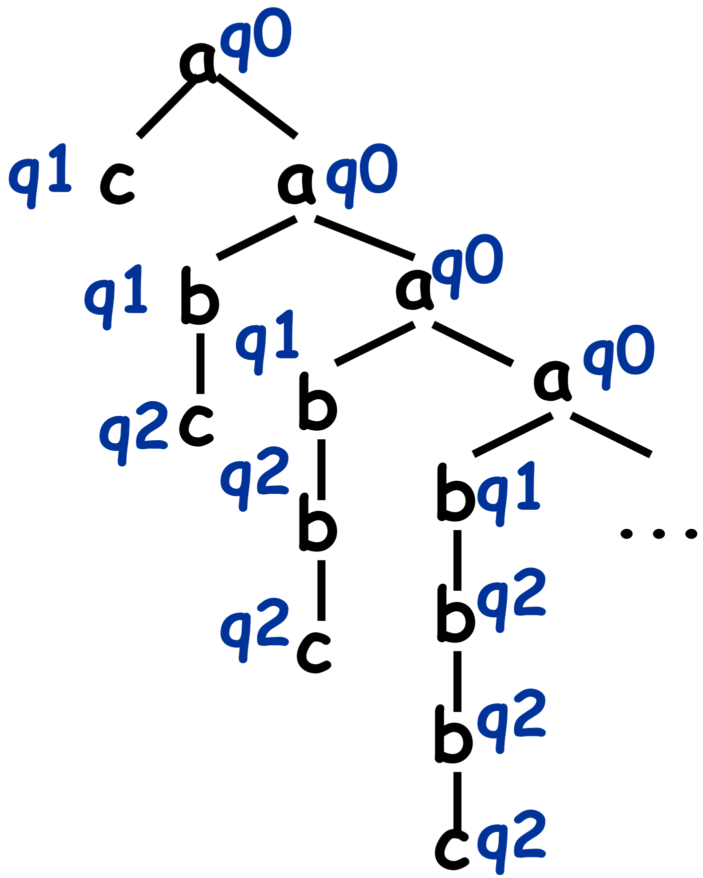
A: trivial automaton

(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



$\delta(q0,\ a) = q1\ q0$

$\delta(q1,\ b) = q2$

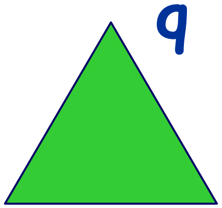
$\delta(q2,\ b) = q2$

$\delta(q1,\ c) = \varepsilon$

$\delta(q2,\ c) = \varepsilon$

# Types for Recursion Schemes

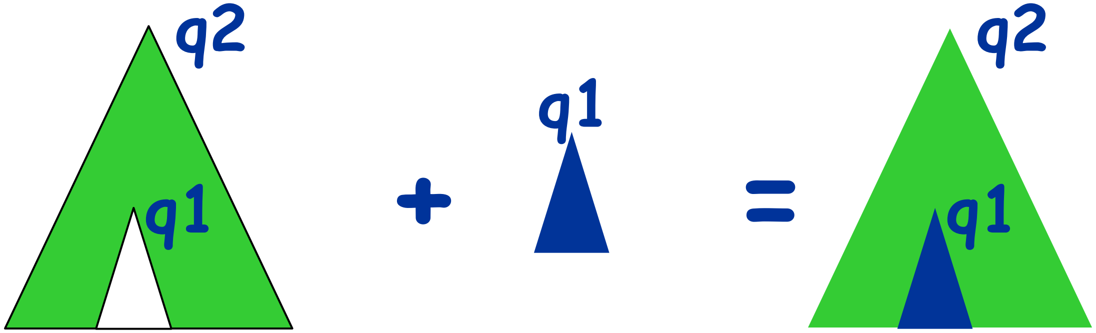
- Automaton state as the type of trees
  - q: trees accepted from state q



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

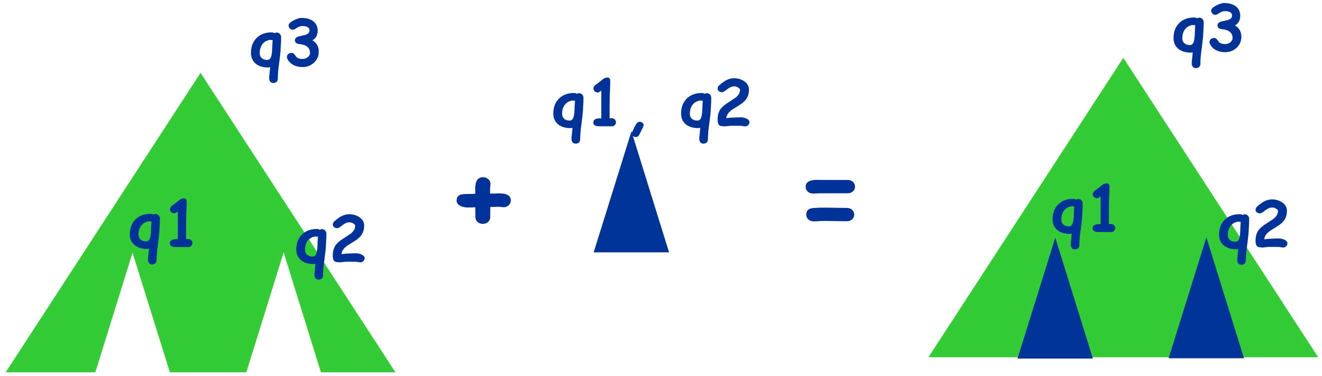
# Types for Recursion Schemes

- Automaton state as the type of trees
  - q1→ 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

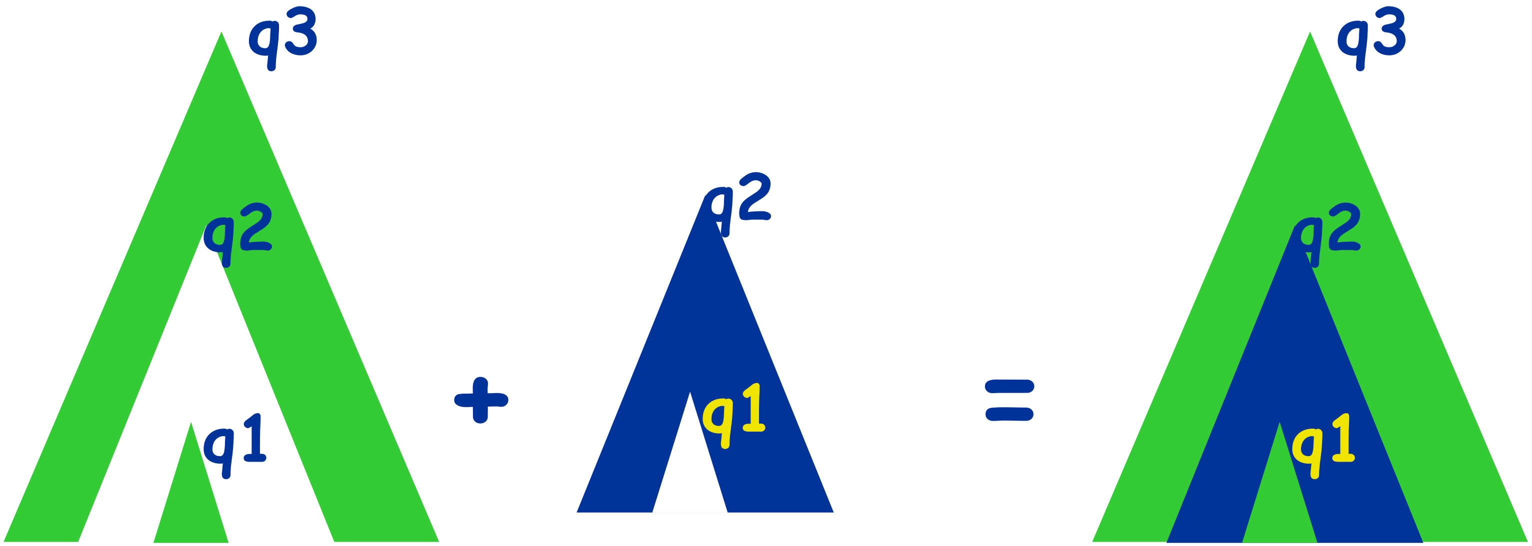


# Types for Recursion Schemes

- Automaton state as the type of trees

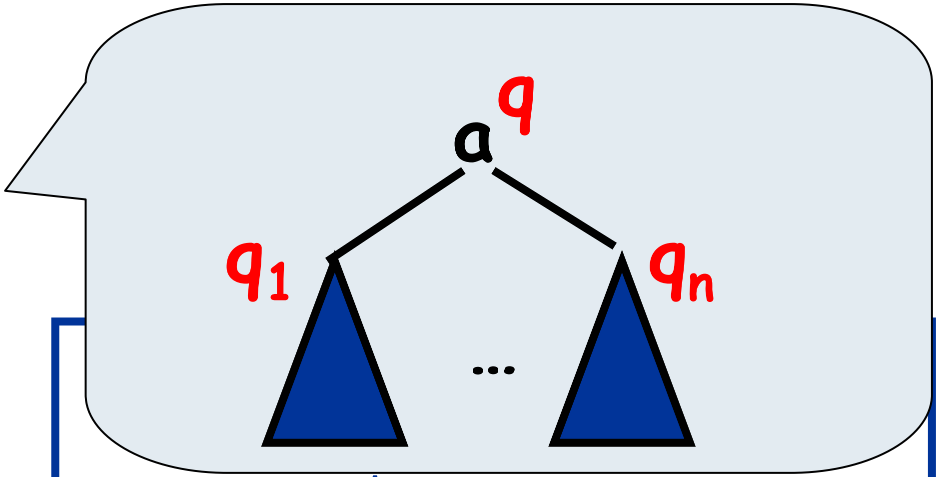
  (q1 → q2) → q3:

  functions that take a function of type q1 → 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

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

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

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

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

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

# Naïve Type Checking Algorithm

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

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 $\Gamma$

⇕

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

where

$H(\Gamma) = \{ F_k:\tau \in \Gamma \mid \Gamma \vdash t_k:\tau \}$

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

Filter out invalid type bindings

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

# Naïve Algorithm Does NOT Work

S has type $q_0$

$\Updownarrow$

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

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

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

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 $q_0$

$\Leftarrow$

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

where

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

Challenges:

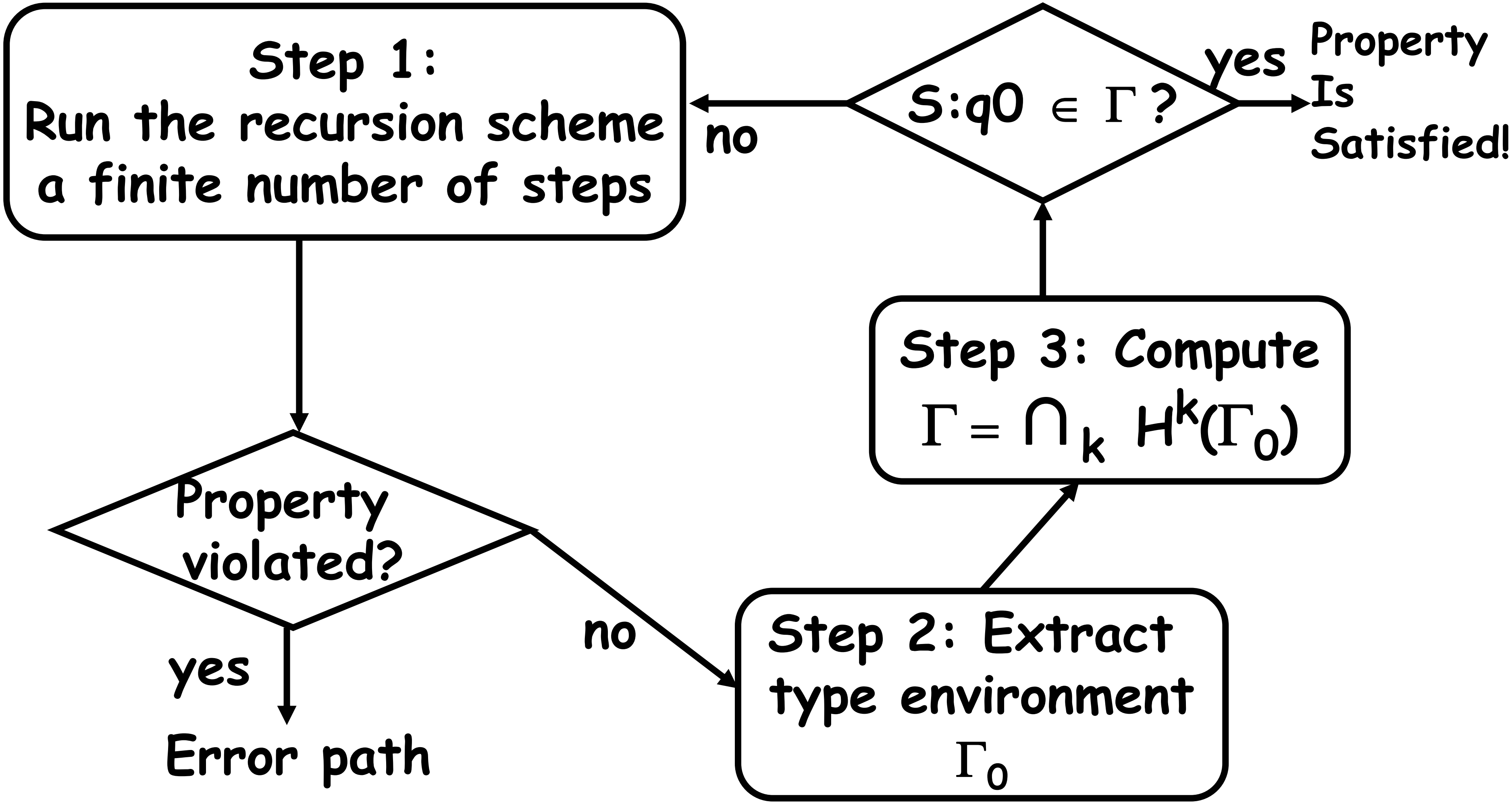
(i) How can we find an appropriate $\Gamma_0$ ?

"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



Step 1:
Run the recursion scheme
a finite number of steps

Property violated?

yes

Error path

no

Step 2: Extract type environment $\Gamma_0$

Step 3: Compute $\Gamma = \cap_k H^k(\Gamma_0)$

$S{:}q0 \in \Gamma$?

no

yes

Property Is Satisfied!

# 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

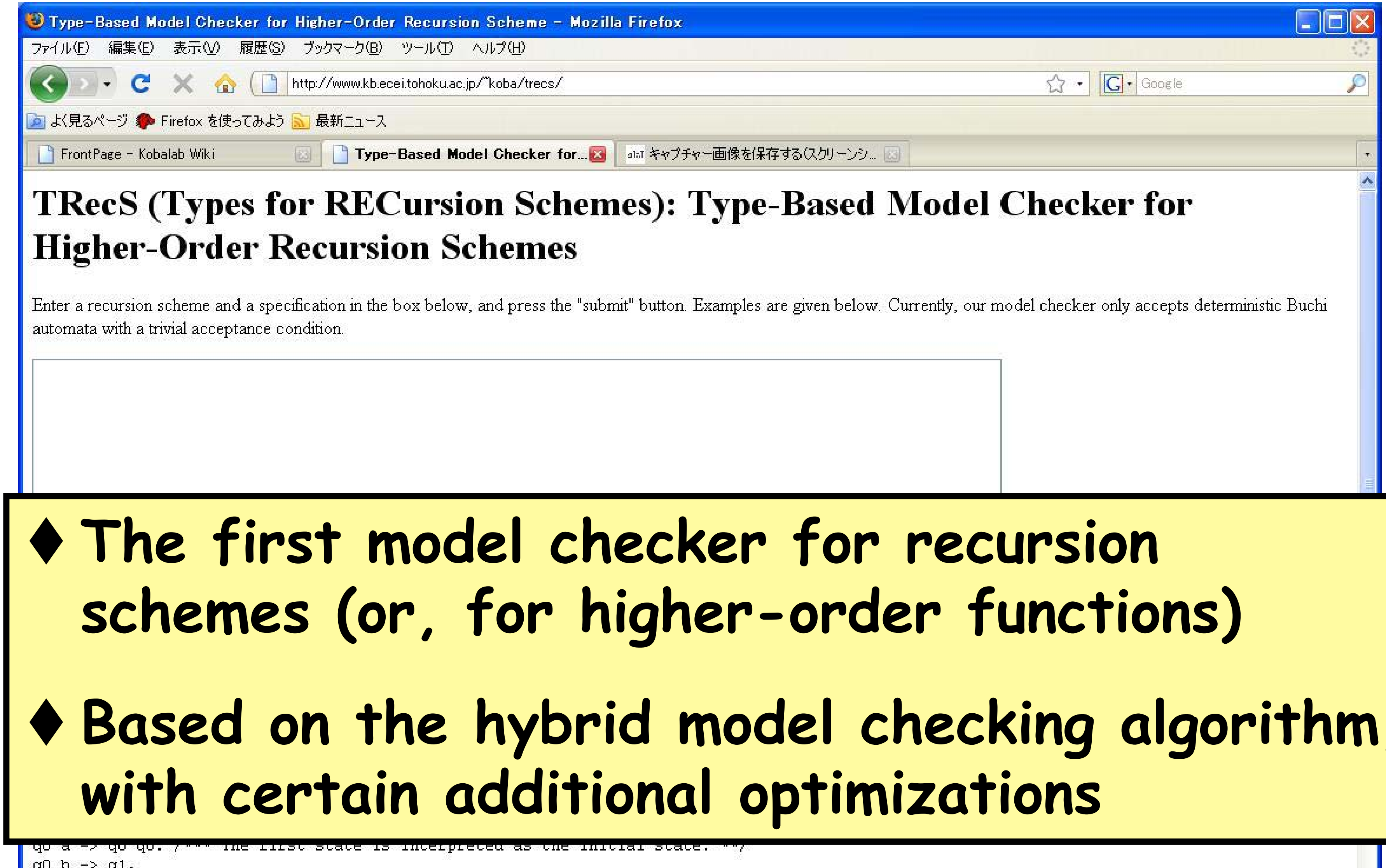
(ii) outputs a type environment
if Tree(G) is accepted by A

# 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/



## TRecS (Types for RECursion Schemes): Type-Based Model Checker for Higher-Order Recursion Schemes

Enter a recursion scheme and a specification in the box below, and press the "submit" button. Examples are given below. Currently, our model checker only accepts deterministic Buchi automata with a trivial acceptance condition.

```
q0 a -> q0 q0. /*** The first state is interpreted as the initial state. ***/
q0 b -> q1.
```

- ♦ 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 | | | | |
| **TwofilesE** | 4 | | | | |
| **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
  ⇒ applications to PCC (proof-carrying code)

- **Counter-example** when verification fails
  ⇒ 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 $a^m(c)$

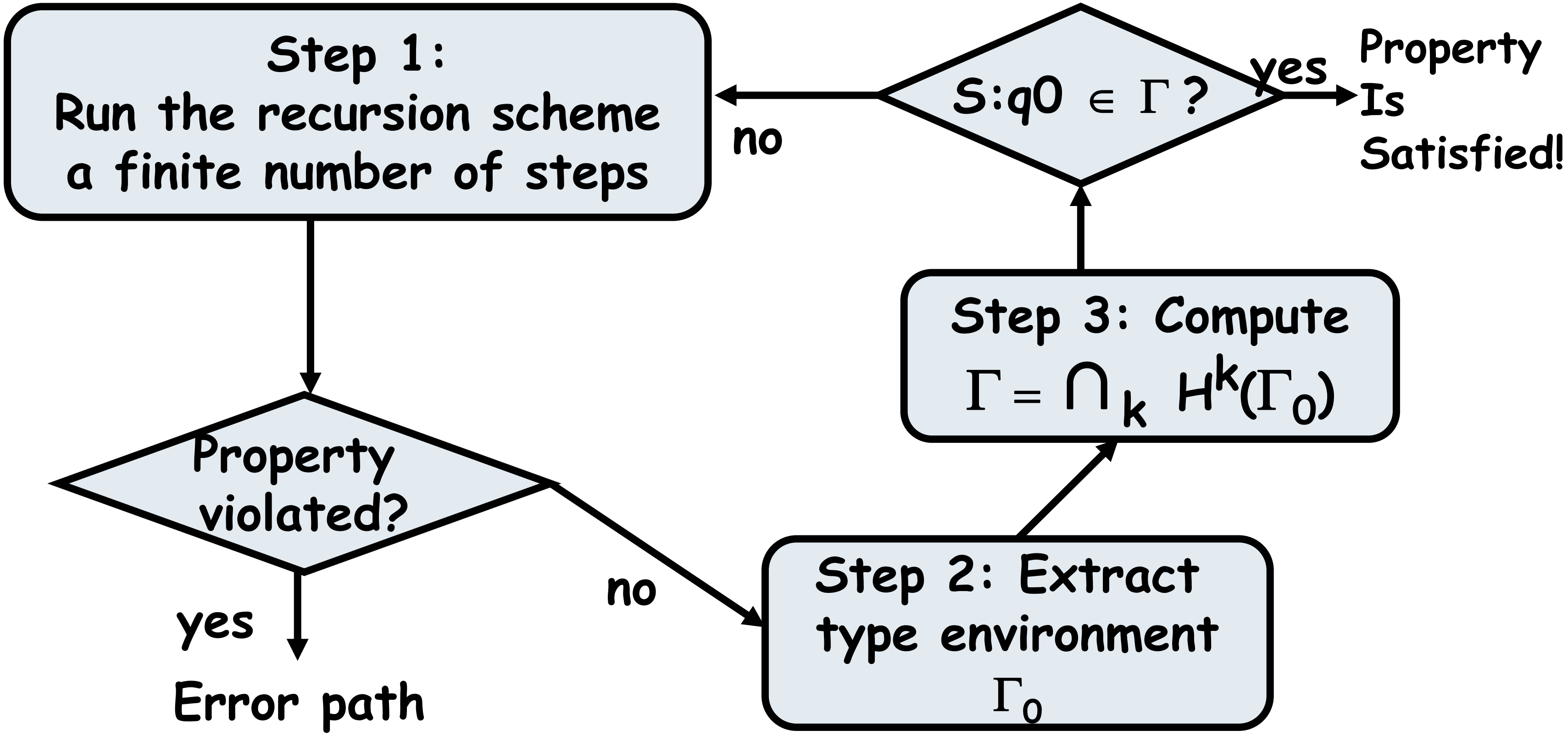
$S \to F_1\ c,\ F_1\ x \to F_2(F_2\ x), \ldots, F_n\ x \to a(a\ x)$

vs

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

# Advantages of our approach

## (5) A good combination with testing: Verification through testing



Step 1: Run the recursion scheme a finite number of steps

Property violated?

yes

Error path

no

Step 2: Extract type environment $\Gamma_0$

Step 3: Compute $\Gamma = \bigcap_k H^k(\Gamma_0)$

$S{:}q0 \in \Gamma$ ?

yes

Property Is Satisfied!

no

# 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:
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