# Program Verification via Higher-Order Model Checking

Naoki Kobayashi
University of Tokyo

# What's This Talk About?

- A survey of applications of
  higher-order model checking
  (model checking of higher-order recursion schemes)

  to:

  automated verification of
  higher-order functional programs
  (e.g. "software model checker" for ML)

# Outline

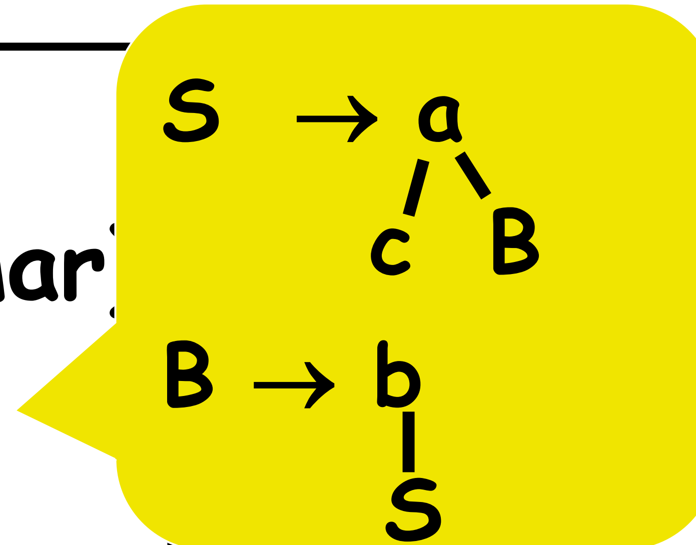
- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- Applications to program verification
  - verification of finite-data programs
  - verification of infinite-data programs
    - safety properties
    - termination
    - non-termination
    - general liveness properties
- Conclusion

# Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

**Order-0 HORS**
**(regular tree grammar)**

S → a c B
B → b S

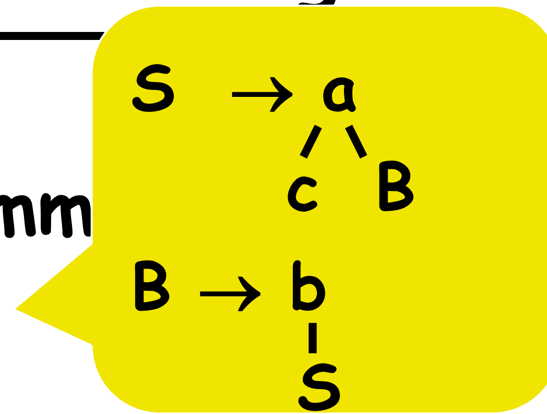


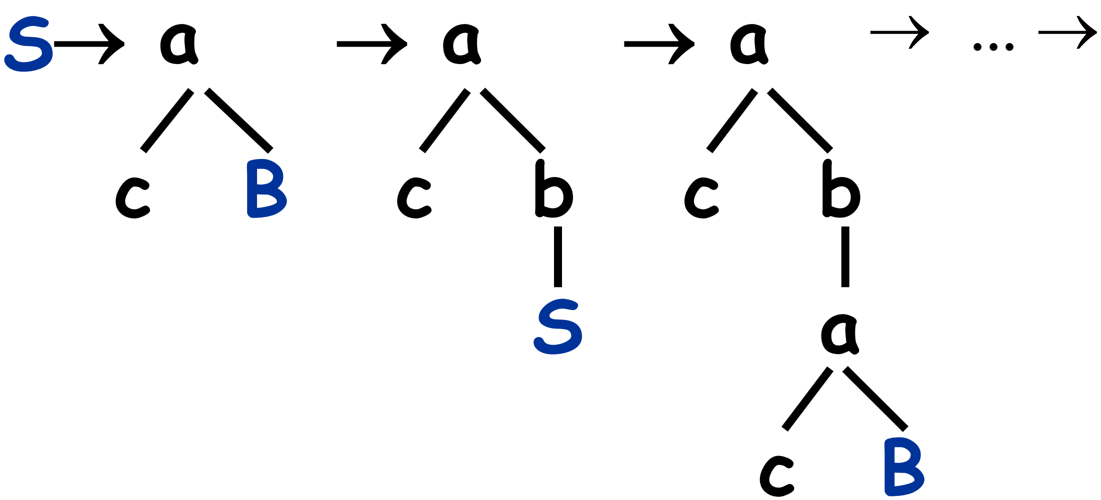
# Higher-Order Recursion Scheme (HORS)

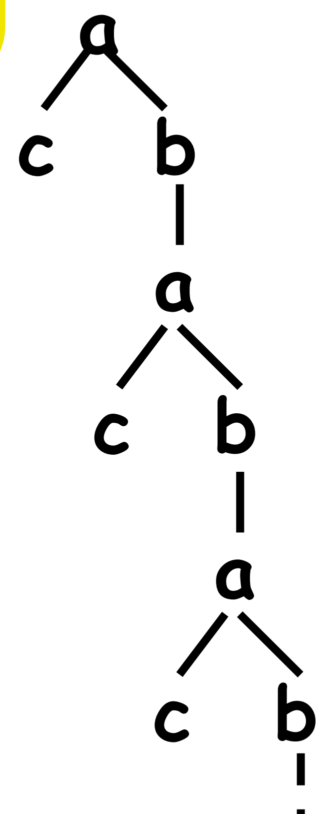
- Grammar for generating an infinite tree

Order-0 HORS
(regular tree gramm

S → a c B
B → b S



# Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

**Order-1 HORS**

$S \to A\ c$

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

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

# Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

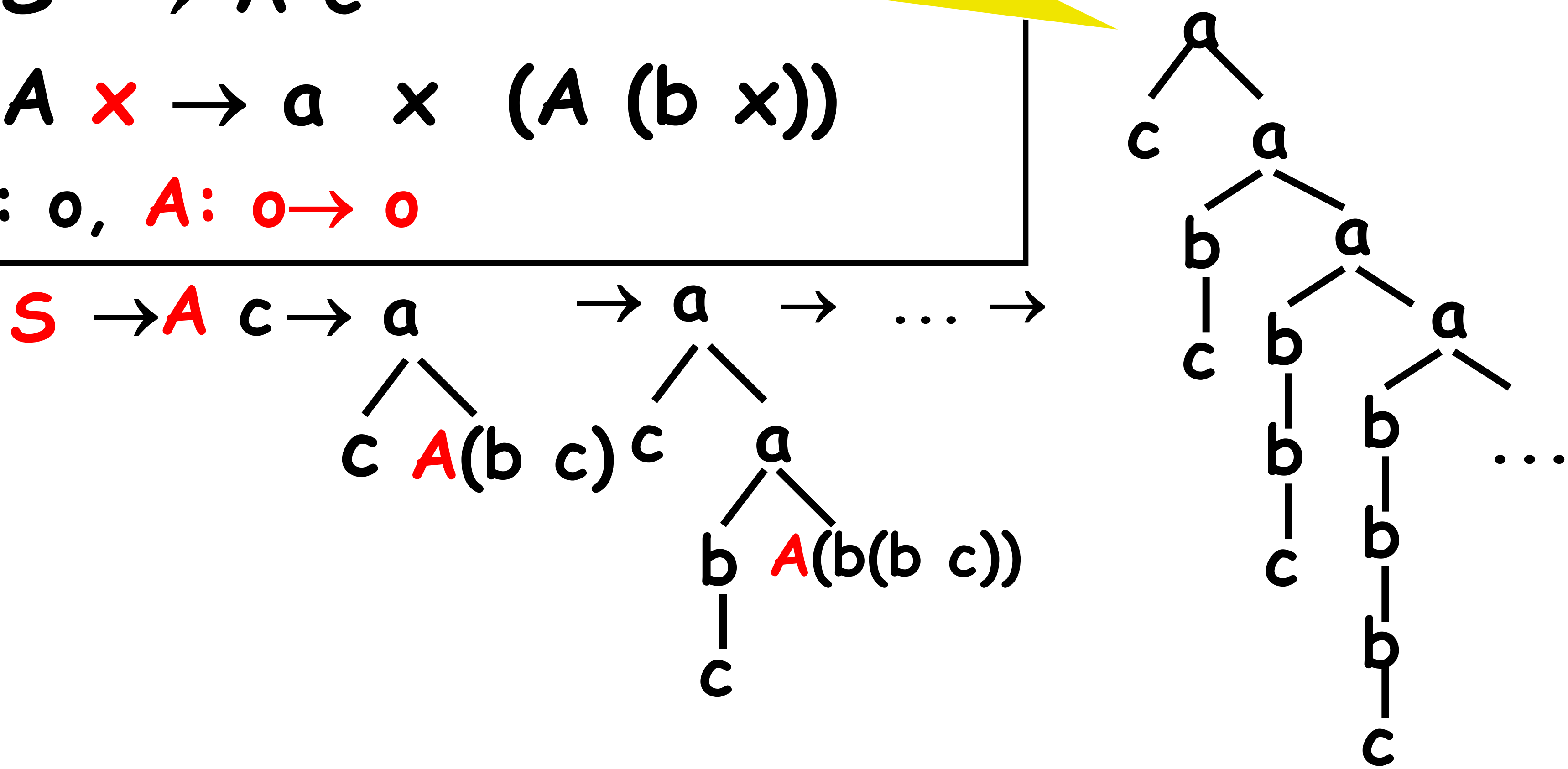
Order-1 HORS

$S \rightarrow A\ c$

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

$S: o,\ A: o \rightarrow o$

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



# Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

Order-1 HORS

$S \;\rightarrow A\ c$

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

$S$: o, $A$: o→ o

HORS

≈

Call-by-name simply-typed λ-calculus

\+

recursion, tree constructors

# Higher-Order Model Checking

Given
 G: HORS
 A: alternating parity tree automaton
 (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 below "b"?

# Higher-Order Model Checking

Order-1 HORS

$S \rightarrow A\ c$

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

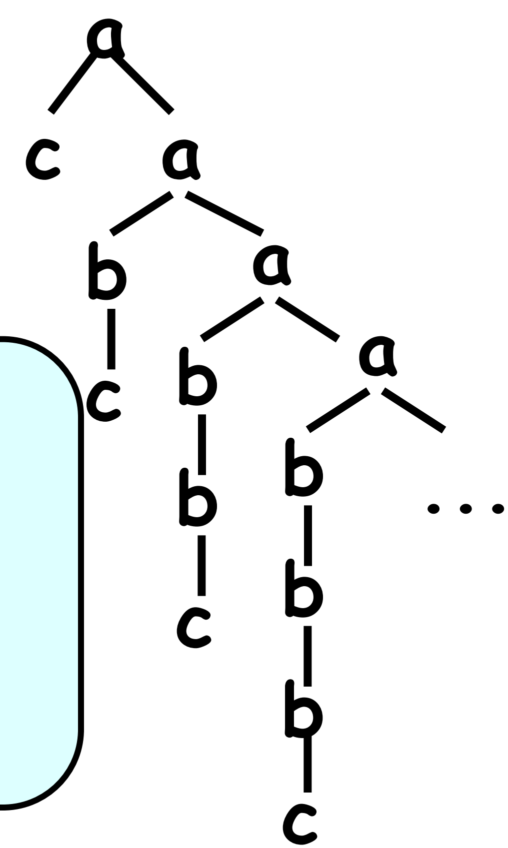
$S: o$, $A: o \rightarrow o$

Q1. Does every finite path end with "c"?

YES!

Q2. Does "a" occur below "b"?

NO!



# Higher-Order Model Checking

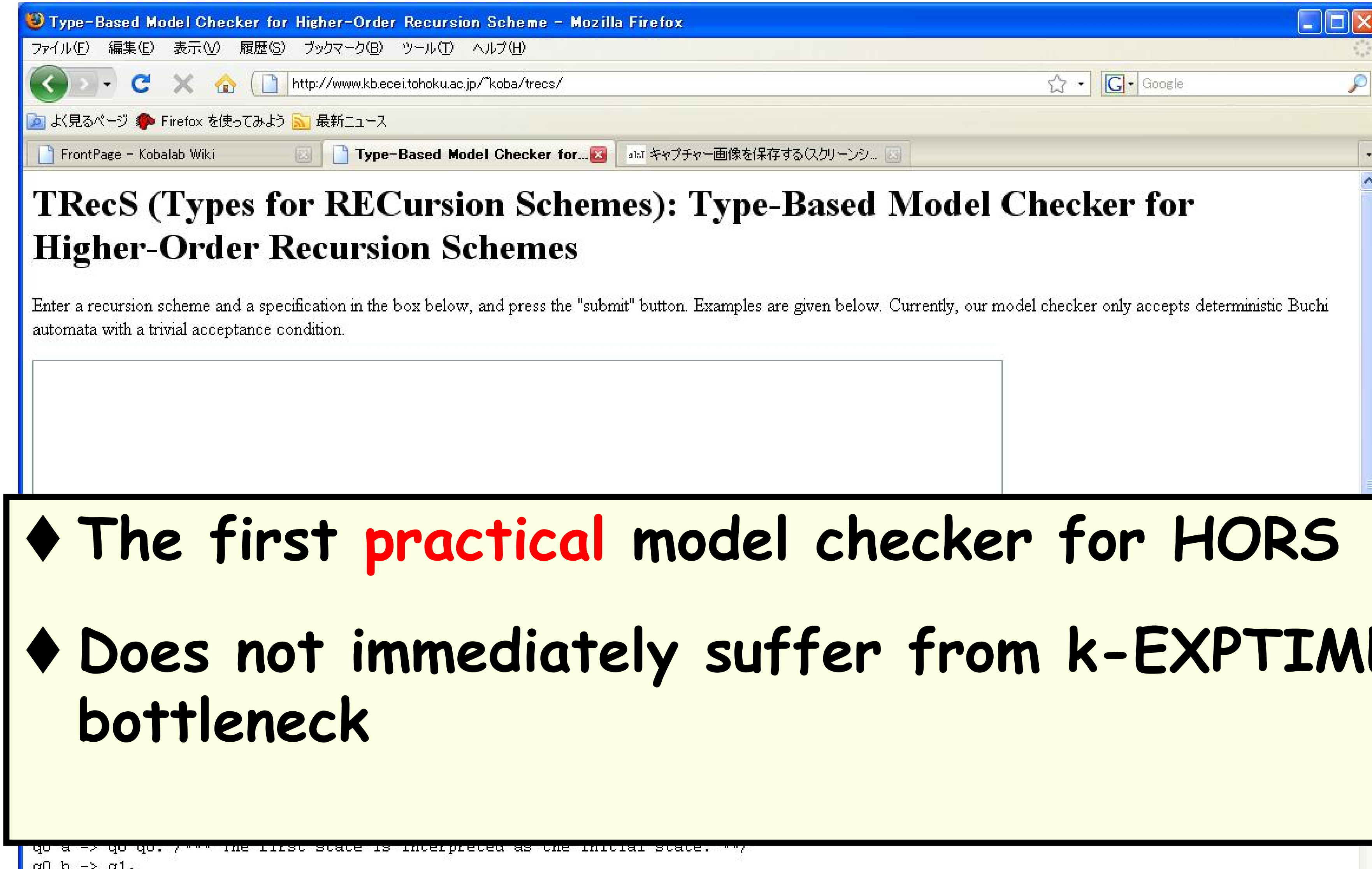
Given
G: HORS
A: alternating parity tree automaton (APT)
(a formula of modal $\mu$-calculus or MSO),
does A accept Tree(G)?

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

k-EXPTIME-complete [Ong, LICS06] $k \left\{ 2^{2^{\cdot^{\cdot^{2^{p(x)}}}}} \right.$
(for order-k HORS)

# TRecS [K. PPDP09]

http://www-kb.is.s.u-tokyo.ac.jp/~koba/trecs/

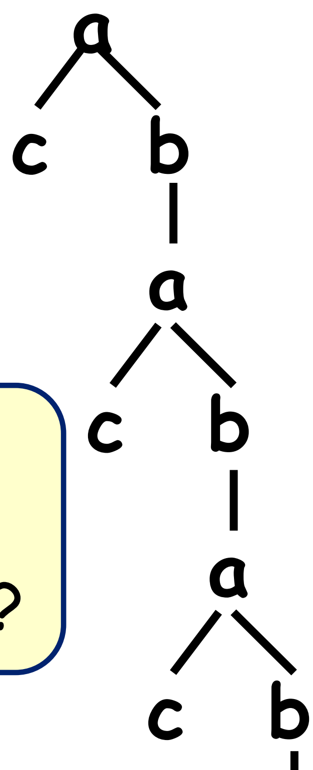


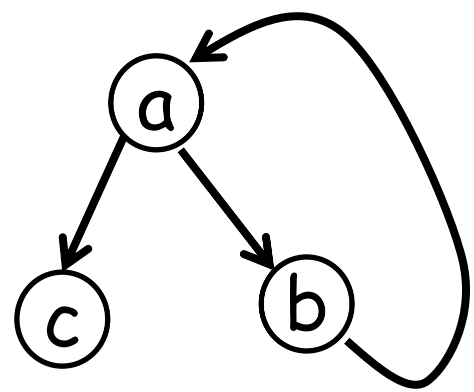

- The first **practical** model checker for HORS
- Does not immediately suffer from k-EXPTIME bottleneck

# HO Model Checking as Generalization of Finite State/Pushdown Model Checking

- order-0 ≈ finite state model checking
- order-1 ≈ pushdown model checking

infinite tree ≈ transition system




Does “a” occur below “b”?

Is there a transition sequence in which “a” occurs after “b”?

# Why HO Model Checking Works? (despite k-EXPTIME completeness)

- Fixed-parameter polynomial time in the size of grammars (under certain assumptions)
- A “certificate” can be checked in polynomial time (cf. NP problems)
- For finite-state models, HO model checking can actually be faster than finite state model checking
  - HORS can compactly represent finite-state systems
    - An order-k HORS of size x can represent a system with states

$$\left.2^{2^{\cdot^{\cdot^{2^{p(x)}}}}}\right\} k$$

  - k-EXPTIME algorithm for HO model checking ≈ PTIME algorithm for finite-state model checking

# Why HO Model Checking Works? (despite k-EXPTIME completeness)

- Fixed-parameter polynomial time in the size of grammars (under certain assumptions)
- A "certificate" can be checked in polynomial time (cf. NP problems)
- For finite-state models, HO model checking can actually be faster than finite state model checking
  - HORS can compactly represent finite-state systems
    - An order-k HORS of size x can represent a system with $\left.2^{2^{\cdot^{\cdot^{2^{p(x)}}}}}\right\} k$ states
  - PTIME algorithm for HO model checking >> PTIME algorithm for finite-state model checking

# Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- Applications to program verification
  - verification of finite-data HO programs
  - verification of infinite-data HO programs
    - safety properties [K+ PLDI 2011]…
    - termination [Kuwahara+ ESOP 2014]
    - non-termination [Kuwahara+ CAV 2015]
    - general liveness properties (ongoing)
- Conclusion

# From Program Verification to HO Model Checking

[K. POPL 2009]

Higher-order program + specification (on events or output)

→ **Program Transformation** →

HORS (describing all event sequences or outputs) + Tree automaton, recognizing valid event sequences or outputs

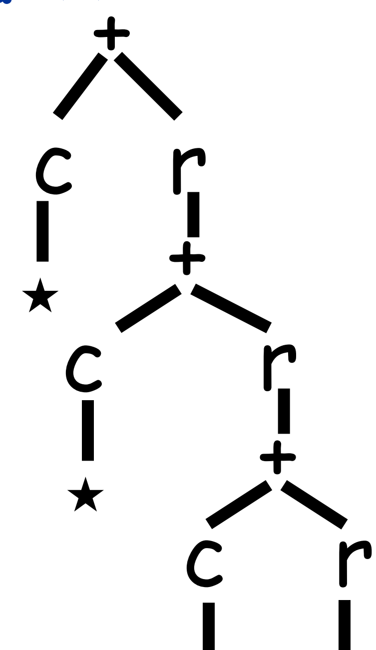
→ **Model Checking**

# 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 ... g:

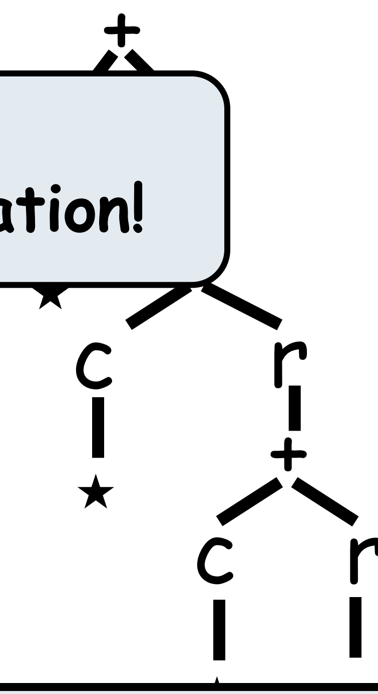
continuation parameter, expressing how "foo" is accessed after the call returns

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

F x k → + (c k) (r(F x k))

S → F d ★

CPS Transformation!




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

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

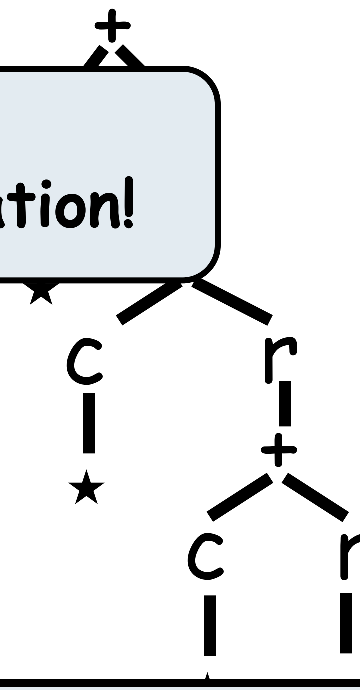
# From Program Verification to Model Checking: Example

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

F x k → + (c k) (r(F x k))

S → F d ★

CPS Transformation!




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

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

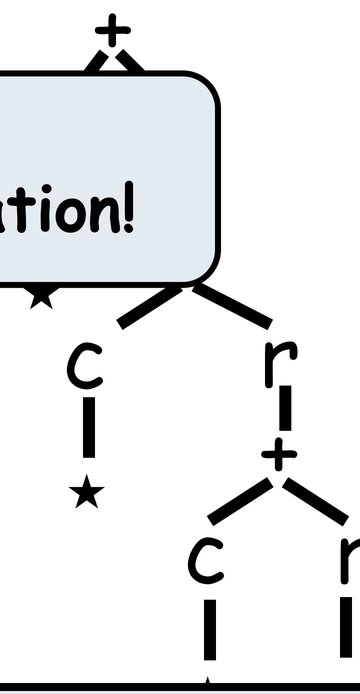
# From Program Verification to Model Checking: Example

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

F x k → + (c k) (r(F x k))

S → F d ★

CPS Transformation!




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

⇒

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

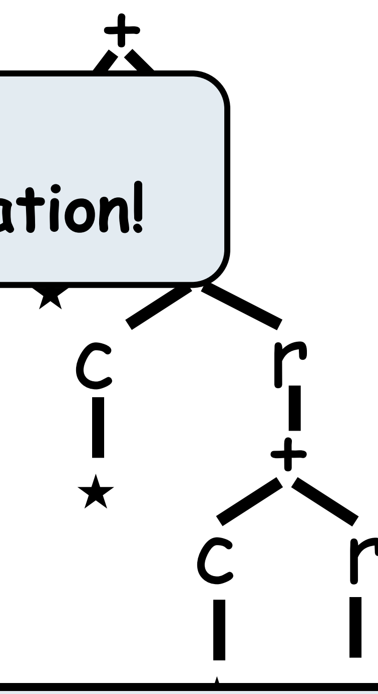
# From Program Verification to Model Checking: Example

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

F x k → + (c k) (r(F x k))

S → F d ★

CPS Transformation!




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

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

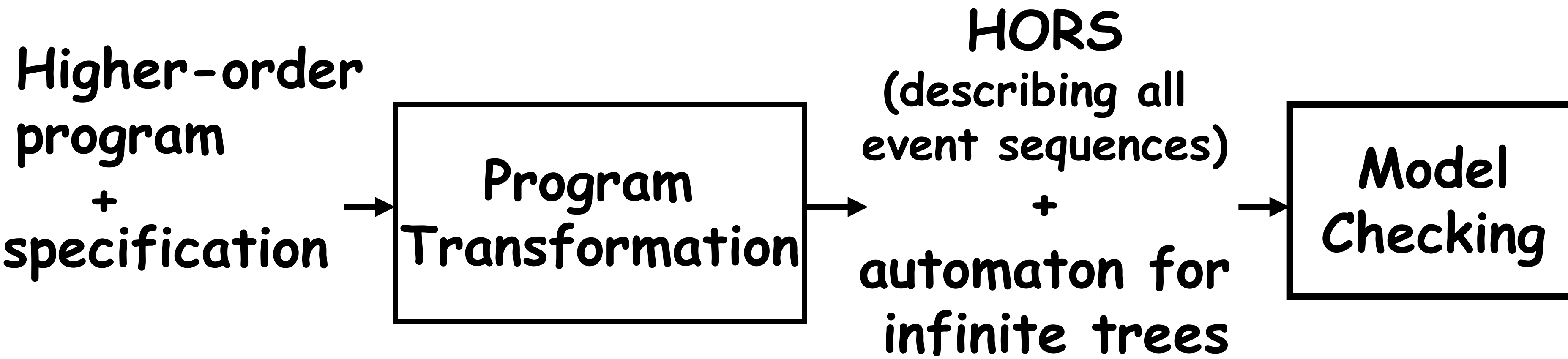
# From Program Verification to HO Model Checking

Higher-order program + specification → **Program Transformation** → HORS (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 (e.g. booleans) + exceptions + ...
- A large class of verification problems:
  resource usage verification (or typestate checking),
  reachability, flow analysis, strictness analysis, ...

# From Program Verification to HO Model Checking




For finite-data HO programs, automated verification comes almost free from HO model checking!

# Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- Applications to program verification
  - verification of finite-data HO programs
  - verification of infinite-data HO programs
    - safety properties
    - termination
    - non-termination
    - general liveness properties
- Conclusion

# Verification of Higher-order Programs with Infinite Data (integers, lists, trees, ...)

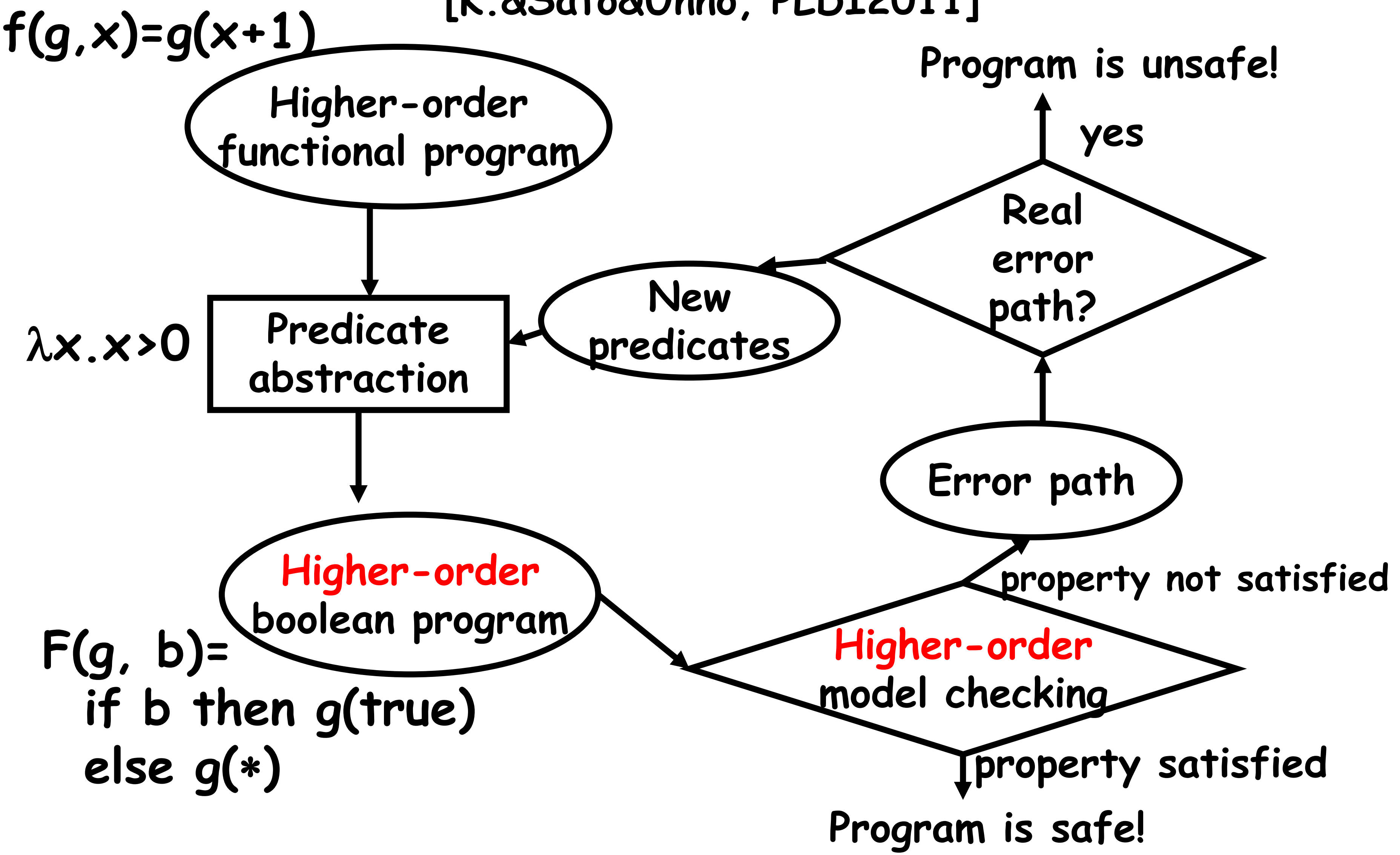
- For safety properties (e.g. reachability), overapproximation by abstraction of infinite data suffice.
- For other properties (e.g. termination), combinations of problem reduction and abstraction are required.

# Verification of Higher-order Programs with Infinite Data (integers, lists, trees, ...)

- For safety properties (e.g. reachability), overapproximation by abstraction of infinite data suffice.
- For other properties (e.g. termination), combinations of problem reduction and abstraction are required.

# Predicate Abstraction and CEGAR for Higher-Order Model Checking

[K.&Sato&Unno, PLDI2011]



# Dealing with algebraic data types (e.g. lists)

- ♦ **Abstraction approach:**
  - automata-based [K+ POPL10][Unno+ APLAS 10]...
  - pattern-based [Ong&Ramsay POPL11]
- ♦ **Encoding approach** [Sato+ PEPM13] :
  - algebraic data as functions

length  function from indices to elements

$[\ \tau\ \text{list}\ ] = \text{int} \times (\text{int} \rightarrow [\tau]\ )$

$\text{nil} = (0,\ \lambda x.\ \text{fail}\ )$

$\text{cons} = \lambda x.\lambda(\text{len},f).$
$(\text{len}+1,\ \lambda i.\text{if } i=0 \text{ then } x \text{ else } f(i-1))$

$\text{hd}\ (\text{len},f) = f(0)$

...

# Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- Applications to program verification
  - verification of finite-data programs
  - verification of infinite-data programs
    - safety properties
    - termination
    - non-termination
    - general liveness properties
- Conclusion

# Termination Verification

- Goal: prove that a program terminates for every input (and non-determinism)
- Naive approach: abstract a program to a finite data program, and apply HO model checking
  - Problem: many terminating programs are turned into non-terminating ones by abstraction

    e.g. f(x) = if x<0 then 1 else 1+f(x-1)     terminating

    → $f(b_{x<0})$ = if $b_{x<0}$ then 1 else 1+f(*)     non-terminating
- Our approach [Kuwahara+, ESOP14]

  (cf. [Rybalchenko&Podelski] for termination of imperative programs):
  - Reduce termination to *binary reachability*
  - Reduce binary reachability to *plain* reachablity

# From Termination to Binary Reachability for HO Programs

- Every non-terminating computation must contain an infinite chain of recursive calls:

  $\text{main}() \to^{*} C_0[f\ v_0]$

  $f\ v_i \ \to^{+} C_{i+1}[f\ v_{i+1}]$ for i=0,1,2,...

  for some function f

$\Rightarrow$ A sufficient (and necessary) condition for termination:

$\text{Call}_f = \{ (v, w) \mid \text{main}() \to^{*} C[f\ v],\ f\ v \to^{+} D[f\ w]\}$

is well-founded for every function f

$\Rightarrow$ To prove termination, it suffices to

- pick a well-founded relation $W_f$ ; and
- prove $\text{Call}_f \subseteq W_f$

for each f

# From Binary Reachability to Plain Reachability

- Goal: check $Call_f \subseteq W_f$
  (where $Call_f=\{(v, w) | main() \rightarrow^* C[f\ v],\ f\ v \rightarrow^+ D[f\ w]\}$)
- Approach: reduction to a plain reachability problem by program transformation
  - To each function, add an extra argument to record the argument of an ancestor call of f.
  - Assert that $W_f$ holds when f is called

```
fib n =
   if n<2 then n
   else fib(n-1)+fib(n-2)
main() = fib(rand())

Wfib = {(m,n) | m>n≥0 }
```

⇒

```
fib m n =
   assert(m>n≥0);
   let m'= if * then m else n in
    if n<2 then n
    else fib m' (n-1)+fib m' (n-2)
main() = fib ⊥ (rand())
```

# Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- Applications to program verification
  - verification of finite-data programs
  - verification of infinite-data programs
    - safety properties
    - termination
    - non-termination
    - general liveness properties
- Conclusion

# Verifying Non-Termination (or Disproving Termination) of HO programs

- Goal: prove that a program is non-terminating for some input (or for some non-deterministic choice)
  - complementary to termination verification
- Unsound approach: overapproximate a program by a finite data program, and apply HO model checking

  f(x) = if x<0 then 1 else 1+f(x-1)　　　terminating

  → f($b_{x<0}$) = if $b_{x<0}$ then 1 else 1+f(*)　　　non-terminating
- Our approach [Kuwahara+, CAV15]:
  - combine over- and under-approximation
  - construct a program that outputs an approximation of the computation tree of the original program
  - use HO model checking to check that the computation tree contains infinite computation

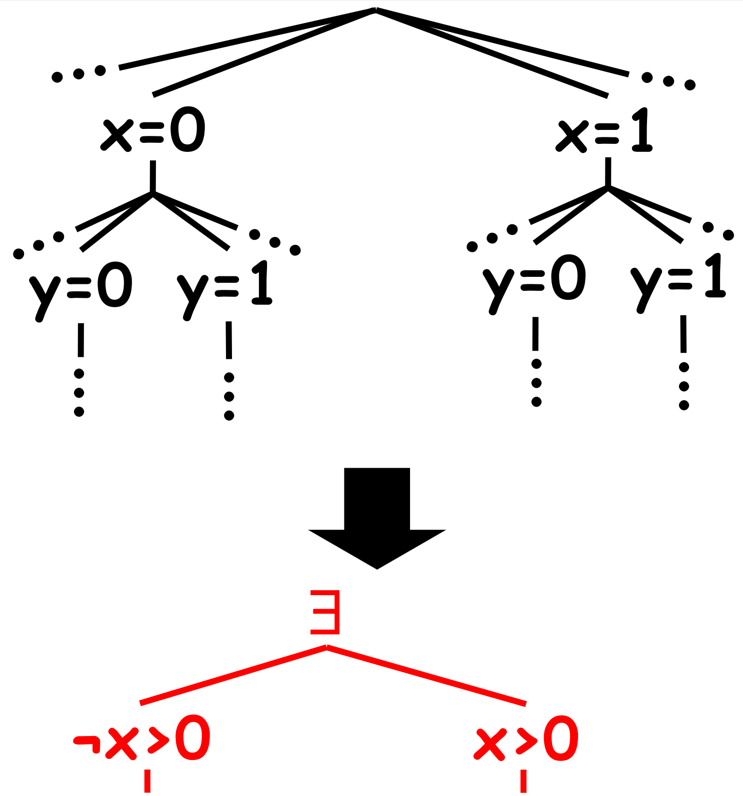
# Our Approach: Combination of Under-/Over-approximation

pred: x>0

```
let x=* in
let y=* in
   f(x+y)
```

Only one of the branches needs to be non-terminating

```
∃( ...
   /* case ¬x>0 */
, ...
   /* case x>0 */
)
```

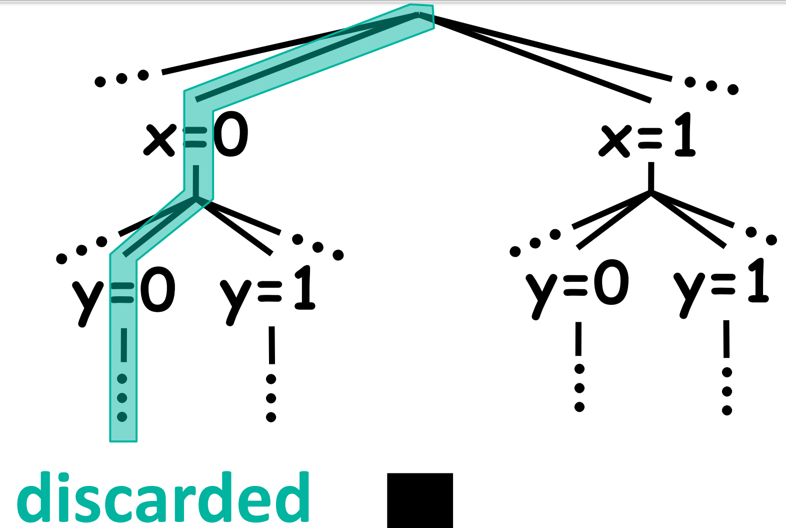


# Our Approach: Combination of Under-/Over-approximation

```
let x=* in
let y=* in
   f(x+y)
```

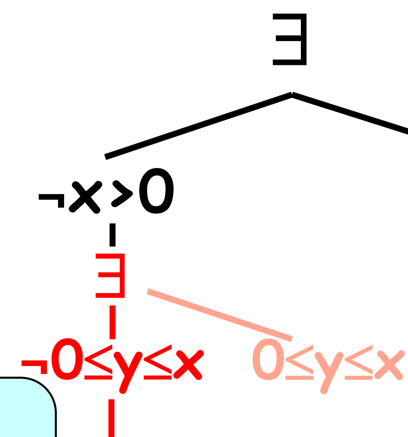
pred: x>0

pred: 0≤y≤x




```
∃( /* case ¬x>0 */
  ∃(...
     /* case ¬0≤y≤x */ )
, ...
)
```

Under-approximation:
case for $\neg x>0 \wedge 0 \leq y \leq x$
is discarded

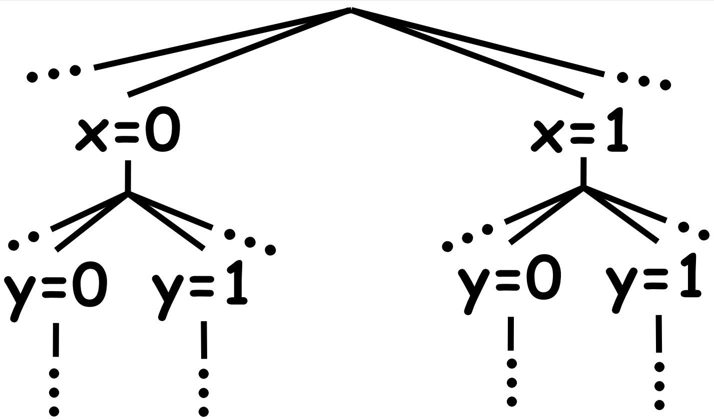


# Our Approach: Combination of Under-/Over-approximation

pred: x>0

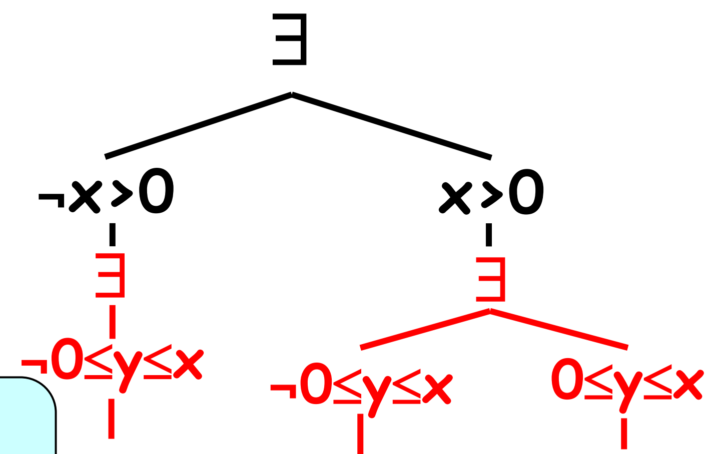
pred: 0≤y≤x

```
let x=* in
let y=* in
   f(x+y)
```




```
∃( /* case ¬x>0 */
  ∃(...
     /* case ¬0≤y≤x */ )
, ...
)
```

Under-approximation: case for $\neg x>0 \wedge 0 \le y \le x$ is discarded



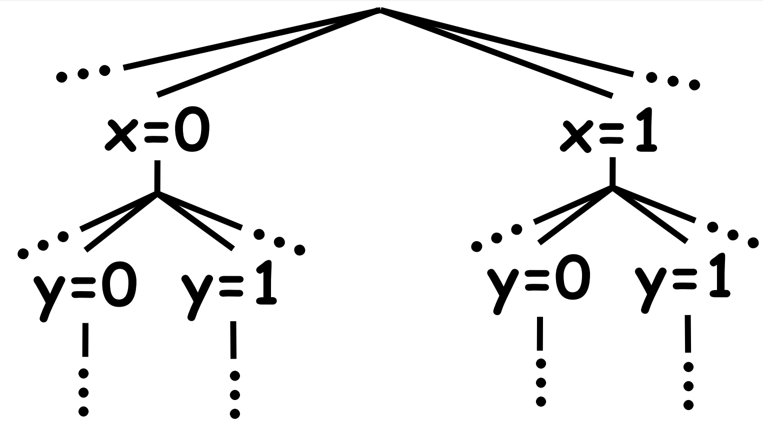
# Our Approach: Combination of Under-/Over-approximation

```
let x=* in
let y=* in
  f(x+y)
```

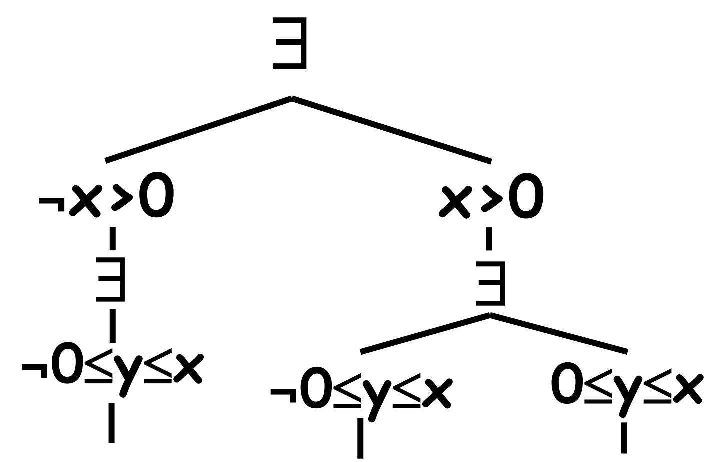
pred: x>0

pred: 0≤y≤x

pred: x+y>0




```
∃( /* case ¬x>0 */
  ∃(/* case ¬0≤y≤x */
     ... )
, ...
)
```



# Our Approach: Combination of Under-/Over-approximation

```
let x=* in
let y=* in
    f(x+y)
```

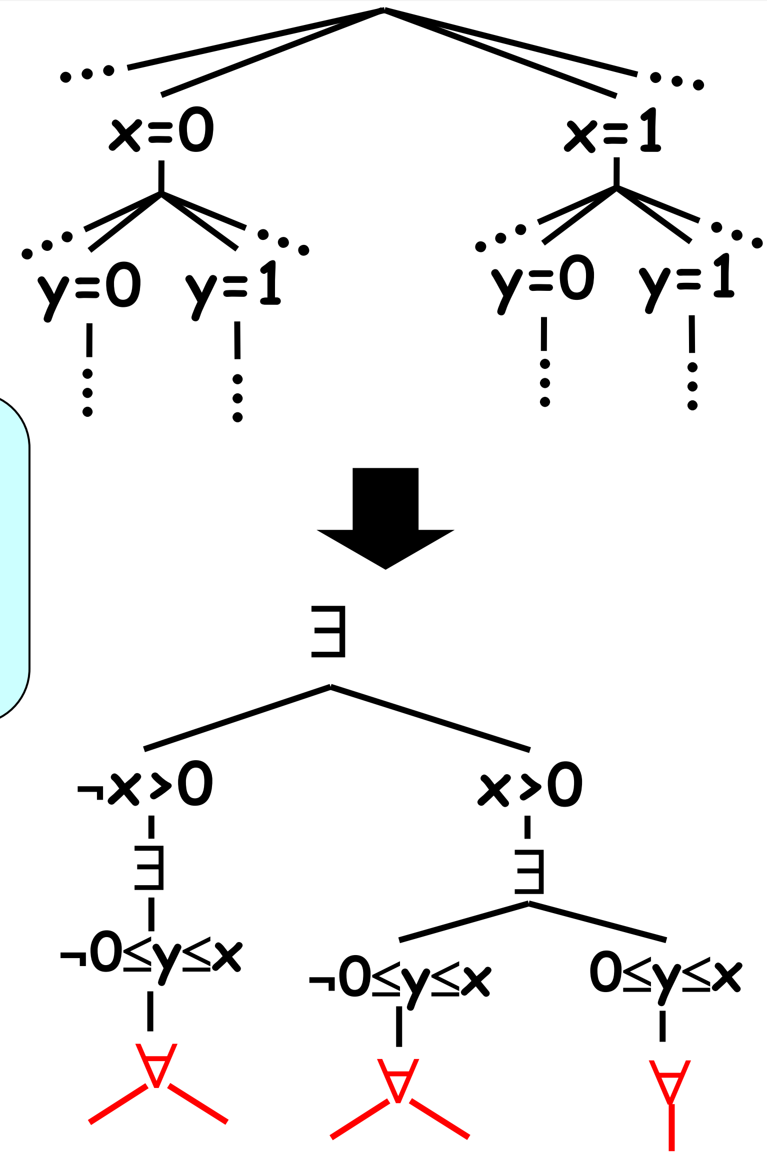
pred: x>0

pred: 0≤y≤x

pred: x+y>0

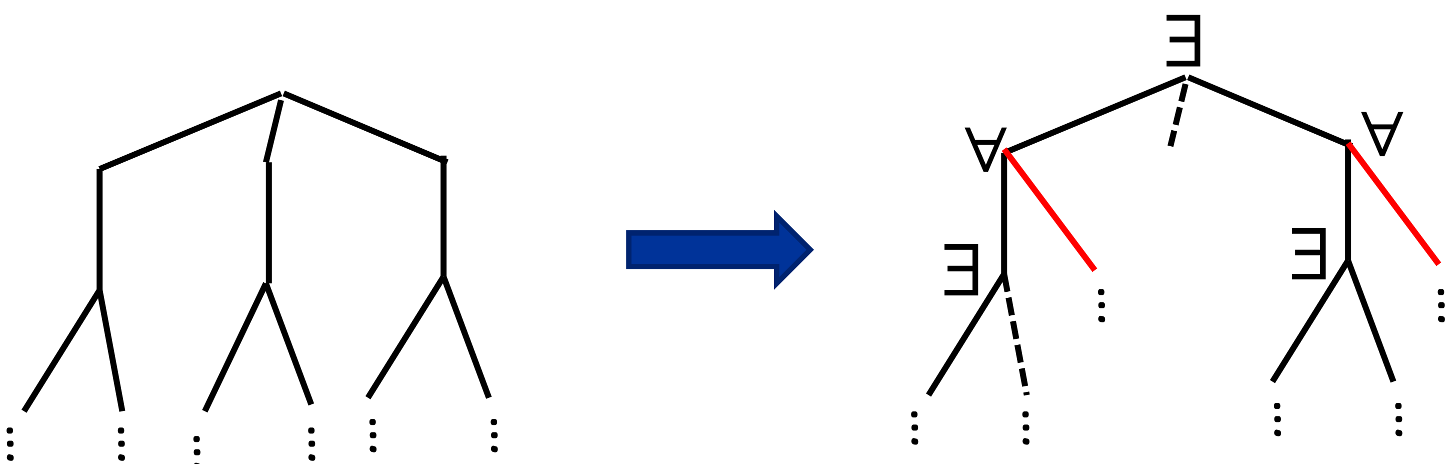
Overapproximation: both branches should have an infinite path (since we don't know which branch is valid)

```
∃( /* case
  ∃( /* case
   ∀( f true /*case x+y>0 */,
      f false /*case ¬x+y>0 */)
  )
 , ...
)
```



# Non-Termination Verification: Summary

- Underapproximate non-deterministic computation, and check that one of the branches has a non-terminating path
- Overapproximate deterministic computation, and check that all the branches have non-terminating paths
- Check them by using HO model checking



# Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- Applications to program verification
  - verification of finite-data HO programs
  - verification of infinite-data HO programs
    - safety properties [K+ PLDI2011],...
    - Termination [Kuwahara+ ESOP 2014]
    - non-termination [Kuwahara+ CAV 2015]
    - general liveness properties (ongoing)
- Conclusion

# Verification of LTL properties of HO programs

- Reduce to fair termination [Vardi 91]
- Extend the termination verification method [Kuwahara+ 14] for proving fair termination

# Conclusion

- Higher-order model checking enables automated verification of functional programs
  - Various properties (including both safety and liveness properties) can be checked by an appropriate combination with abstraction and program transformation
- Do not worry too much about k-EXPTIME completeness of HO model checking
  - depending on inputs, recent HO model checkers can process inputs of thousands of lines in a few seconds