Towards a Software Model Checker for ML

Naoki Kobayashi
Tohoku University

Joint work with:
Ryosuke Sato and Hiroshi Unno (Tohoku University)
in collaboration with
Luke Ong (Oxford), Naoshi Tabuchi and Takeshi Tsukada (Tohoku)

This Talk

- Overview of our project to construct:

  Software Model Checker for ML,

  based on *higher-order model checking* (or, model checking of higher-order recursion schemes)

Outline

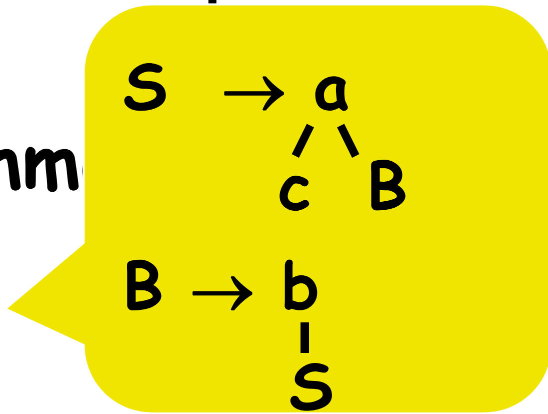
- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
- Applications to program verification
  - Verification of higher-order boolean programs
  - Dealing with infinite data domains (integers, lists,...)
- Towards a full-scale model checker for ML
- Conclusion

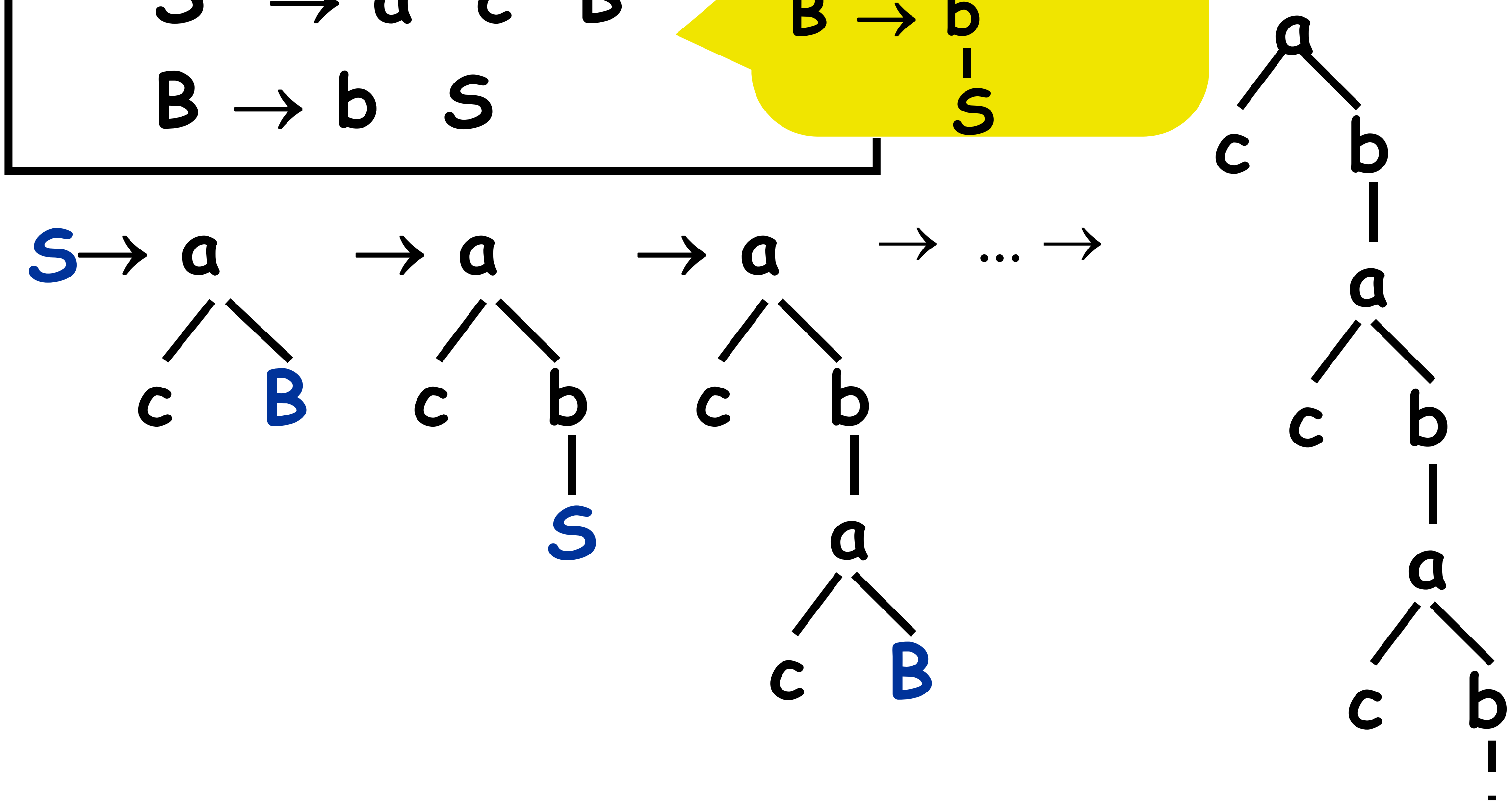
Higher-Order Recursion Scheme

- 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 … nite tree

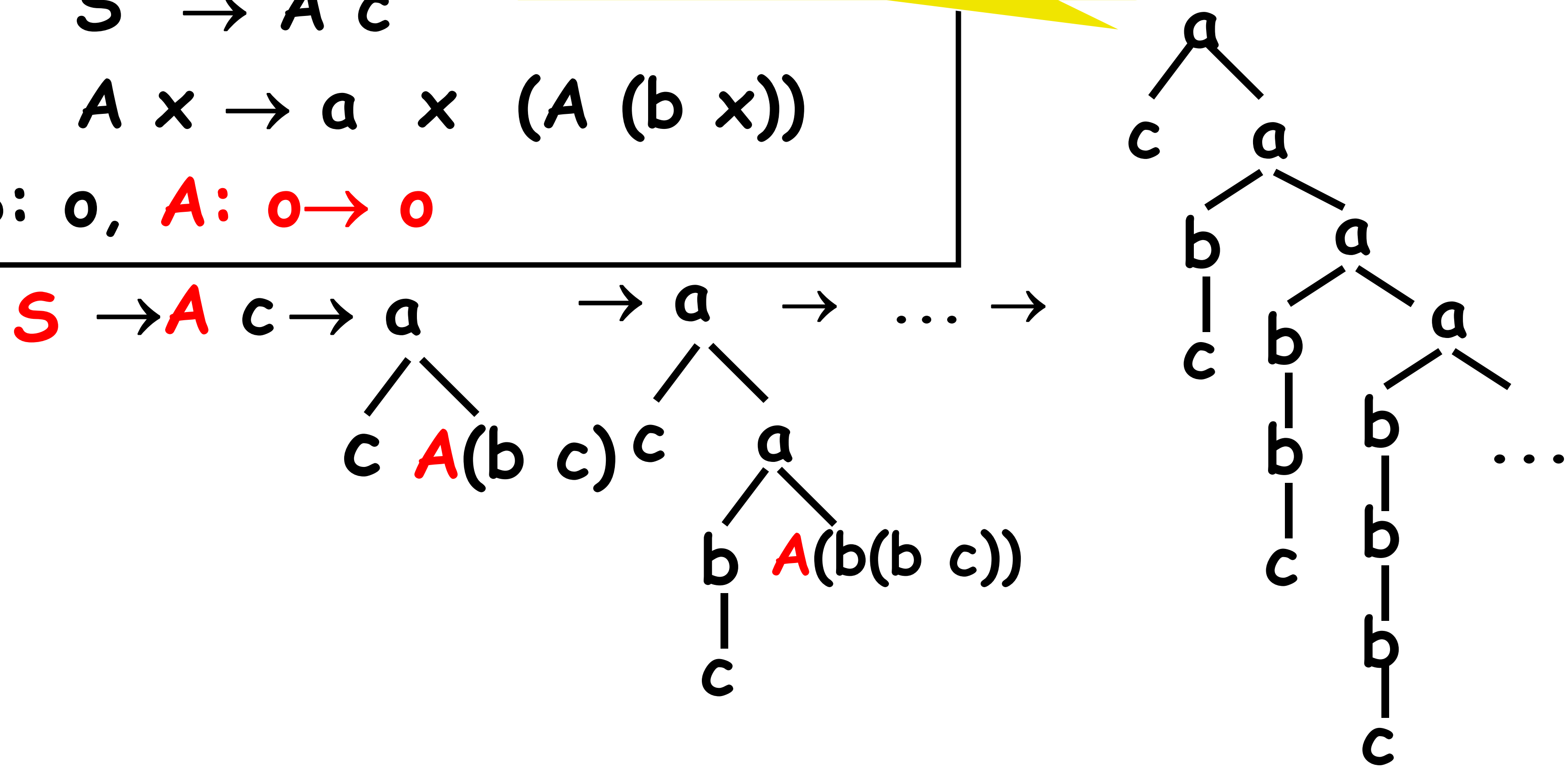
Order-1 scheme

$S \rightarrow A\ c$

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

S: o, A: o→ o

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



Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-1 scheme

$S \rightarrow A\ c$

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

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

Higher-order recursion schemes

≈

Call-by-name simply-typed λ-calculus

+

recursion, tree constructors

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 below "b"?

Higher-Order Recursion Scheme

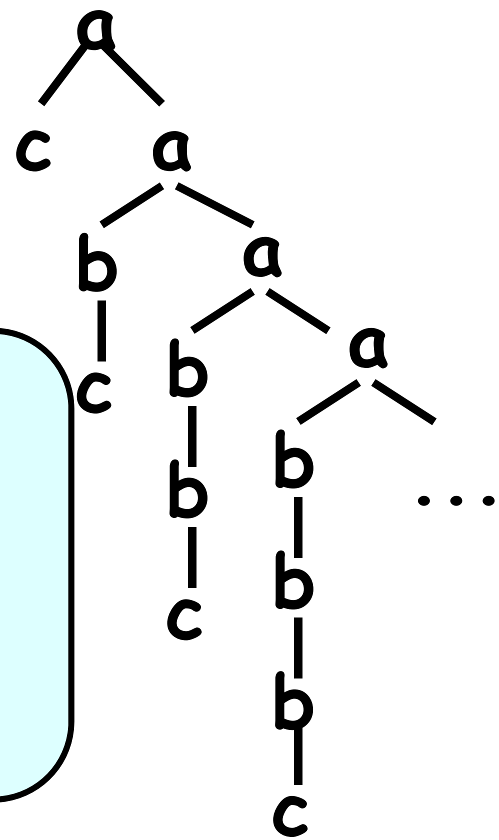
- Grammar for generating an infinite tree

Order-1 scheme

$S \rightarrow A\ c$

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

S: o, A: o→ o



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

YES!

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

NO!

Model Checking Recursion Schemes

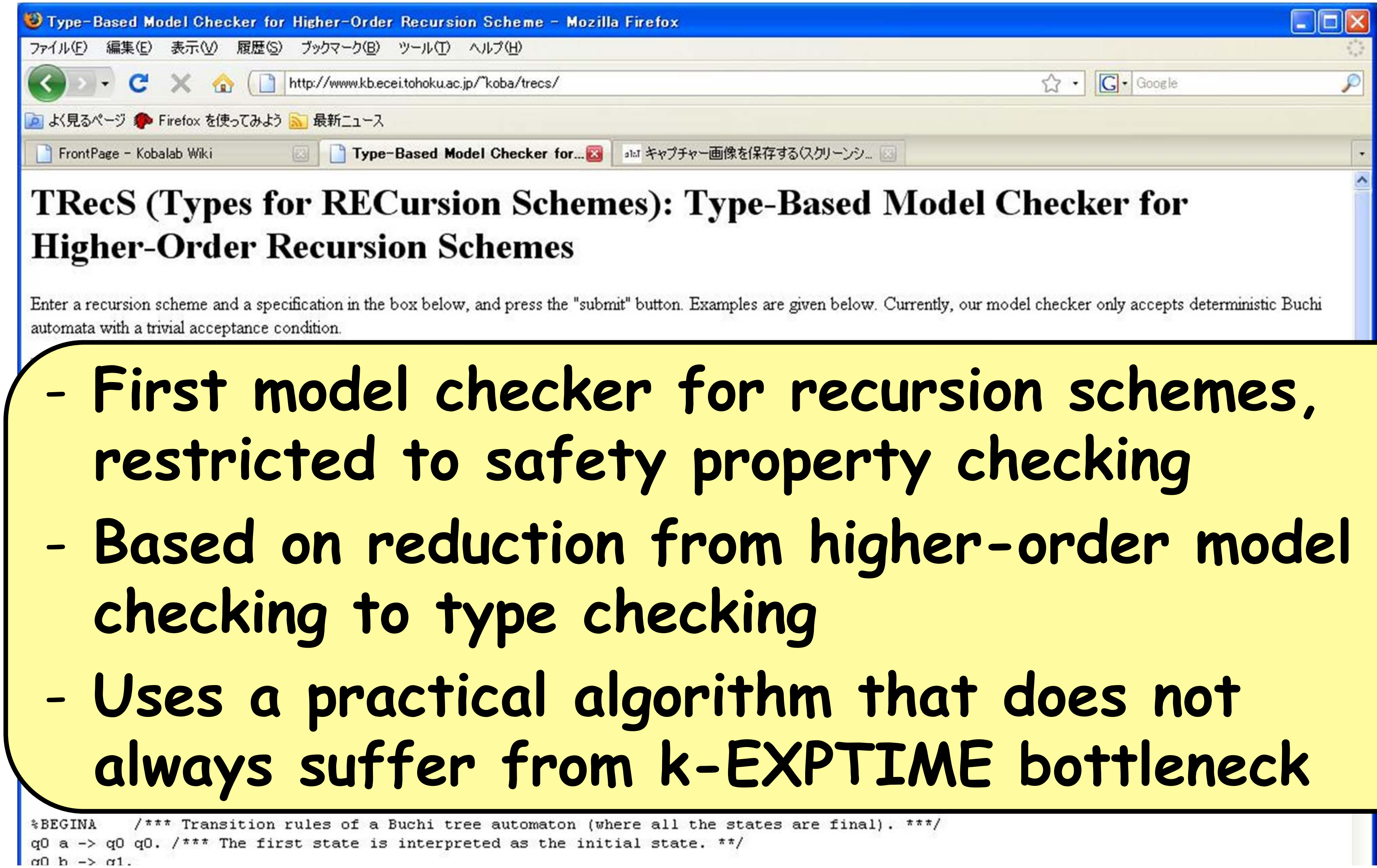
Given
G: higher-order recursion scheme
A: alternating parity tree automaton (APT)
(a formula of modal μ-calculus or MSO),
does A accept Tree(G)?

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

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

TRecS [K., PPDP09]

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



- First model checker for recursion schemes, restricted to safety property checking
- Based on reduction from higher-order model checking to type checking
- Uses a practical algorithm that does not always suffer from k-EXPTIME bottleneck

(Non-exhaustive) History

- 70s: (1^{st}-order) Recursive program schemes
  [Nivat;Coucelle-Nivat;...]
- 70-80s: Studies of high-level grammars
  [Damm; Engelfriet;..]
- 2002: Model checking of higher-order recursion schemes [Knapik-Niwinski-Urzyczyn02FoSSaCS]
  Decidability for “safe” recursion schemes
- 2006: Decidability for arbitrary recursion schemes
  [Ong06LICS]
- 2009: Model checker for higher-order recursion schemes [K09PPDP]
  Applications to program verification [K09POPL]

Outline

- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
- Applications to program verification
  - Verification of higher-order boolean programs
    - Rechability
    - Temporal properties
  - Dealing with infinite data domains (integers, lists,...)
- Towards a full-scale model checker for ML

Reachability verification for higher-order boolean programs

Theorem:

Given a closed term M of (call-by-name or call-by-value) simply-typed λ-calculus with:

- recursion
- finite base types (including booleans and special constant "fail")
- non-determinism,

it is decidable whether $M \to^* \text{fail}$

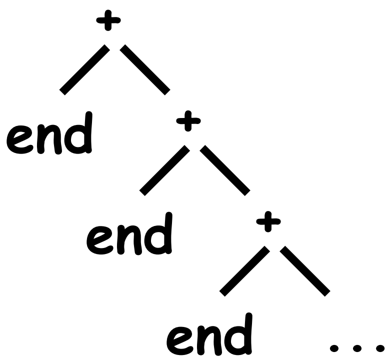
Proof: Translate M into a recursion scheme G

s.t. $M \to^* \text{fail}$ if and only if Tree(G) contains "fail".

Example

```
fun repeatEven f x = if * then x else f (repeatOdd f x)
fun repeatOdd f x = f (repeatEven f x)
fun main( ) = if (repeatEven not true) then ( ) else fail
```

Higher-order recursion scheme that generates the tree containing all the possible outputs:



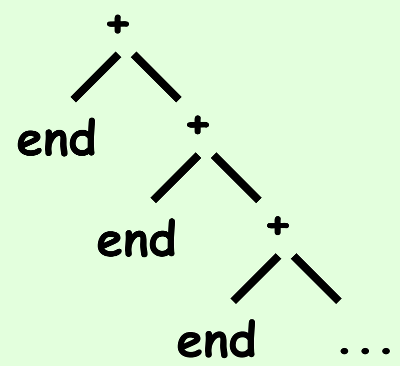
Example

```
fun repeatEven f x = if * then x else f (repeatOdd f x)
fun repeatOdd f x = f (repeatEven f x)
fun main( ) = if (repeatEven not true) then ( ) else fail
```

↓ call-by-value CPS + encoding of booleans

```
RepeatEven k f x → If TF (k x) (RepeatOdd (f k) f x)
RepeatOdd k f x → RepeatEven (f k) f x
Main → RepeatEven C Not True
C b → If b end fail
Not k b → If b (k False) (k True)
If b x y → b x y
True x y → x        False x y → y
TF x y → + x y
```

Generated tree

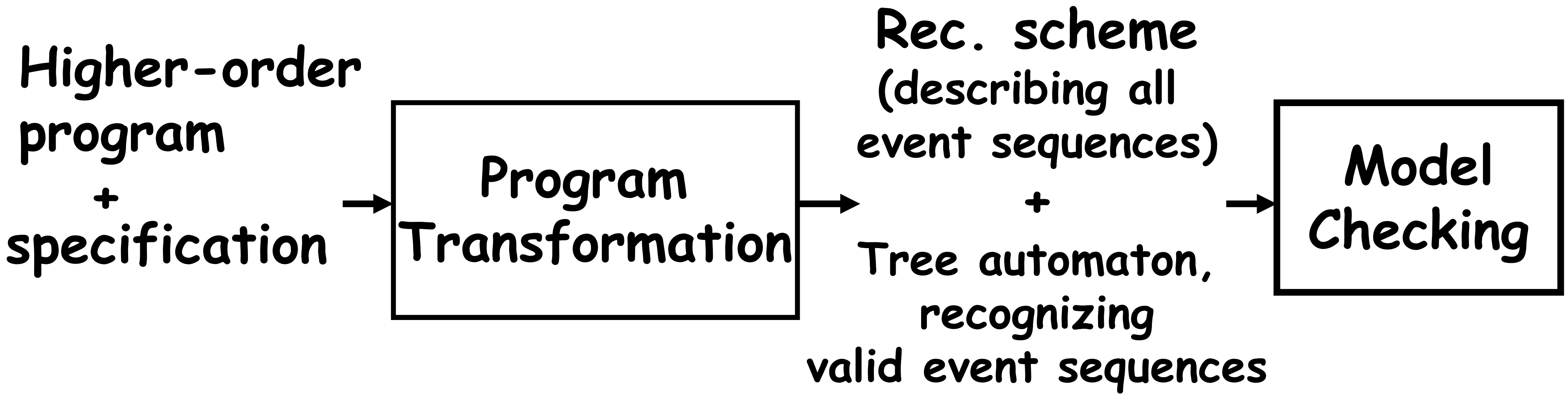


Outline

- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
- Applications to program verification
  - Verification of higher-order boolean programs
    - Rechability
    - Temporal properties
  - Dealing with infinite data domains (integers, lists,...)
- Current status and remaining challenges

Verification of temporal properties by higher-order model checking

[K. POPL 2009]

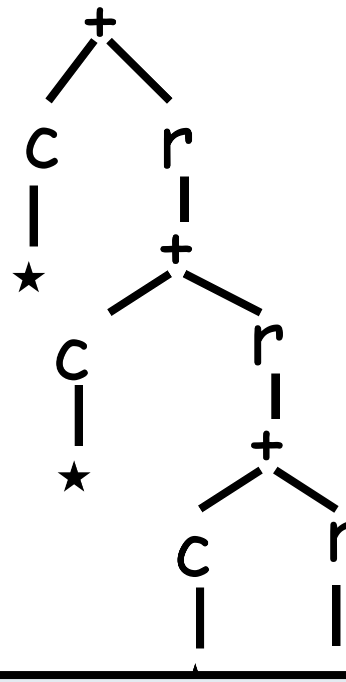


From Program Verification to Model Checking: Example

```
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 ★



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

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

From Program ... ing:

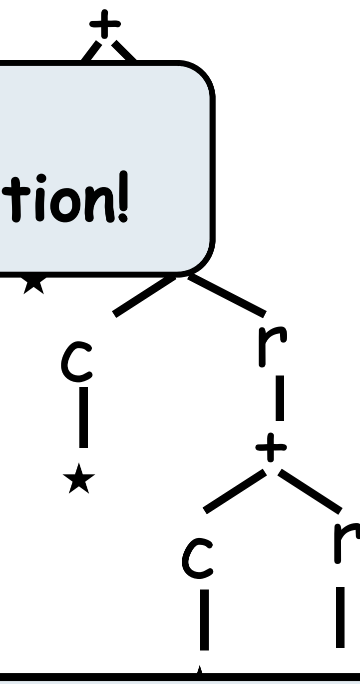
continuation parameter, expressing how "foo" is accessed after the call returns

```
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?

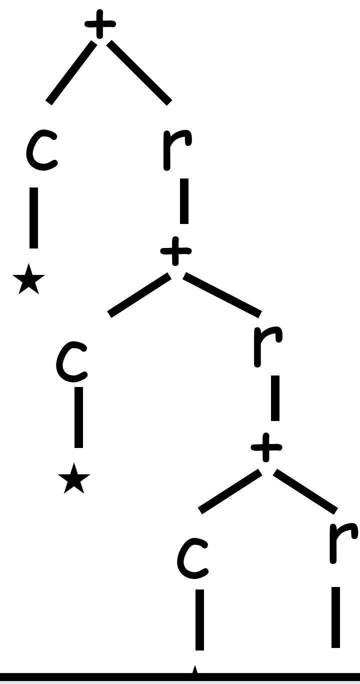
Is each path of the tree labeled by r*c?

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 ★



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

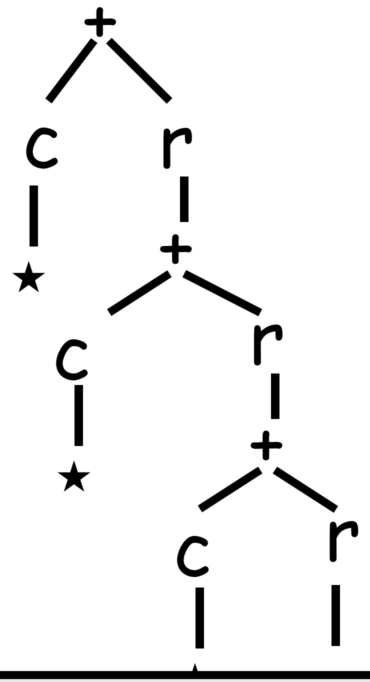
Is each path of the tree labeled by r*c?

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 ★



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

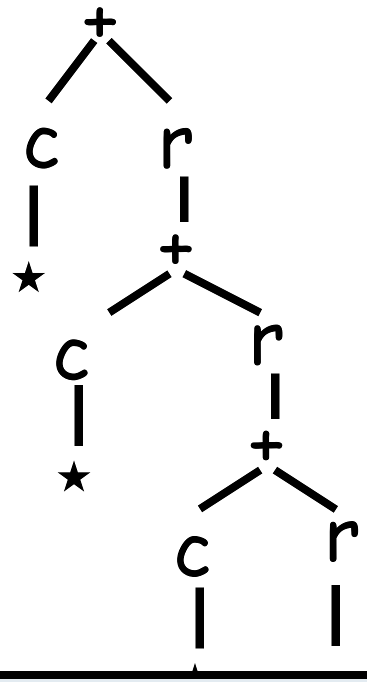
Is each path of the tree labeled by r*c?

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)
```

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

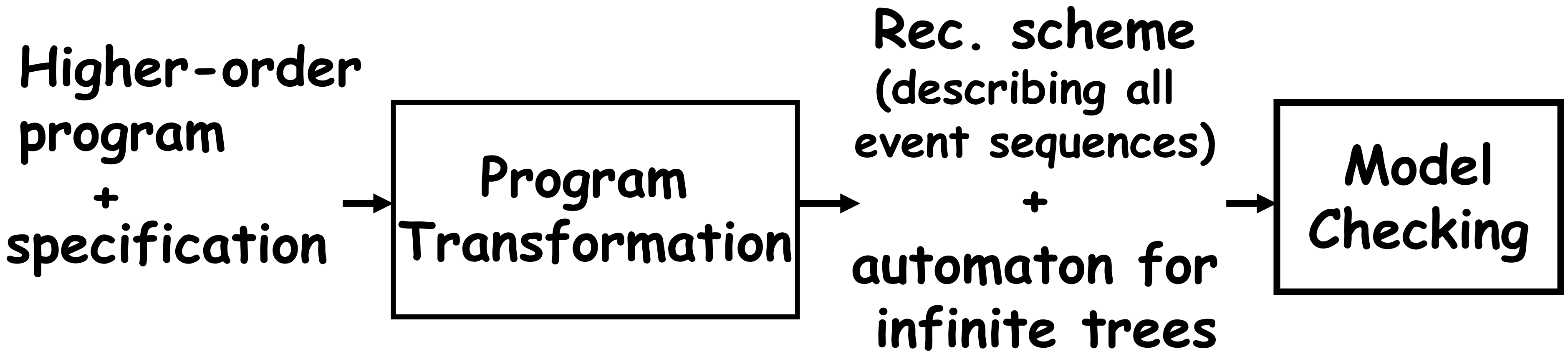
$S \rightarrow F\ d\ \star$



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

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

Program Verification by Higher-order Model Checking



Sound, complete, and automatic for:

- A large class of higher-order programs: finitary PCF (simply-typed λ-calculus + recursion + finite base types)
- A large class of verification problems: resource usage verification (or typestate checking), reachability, flow analysis,...

Comparison with Other 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 with arbitrary recursion	Higher-order model checking

} infinite state model checking (Pushdown model checking, Higher-order model checking)

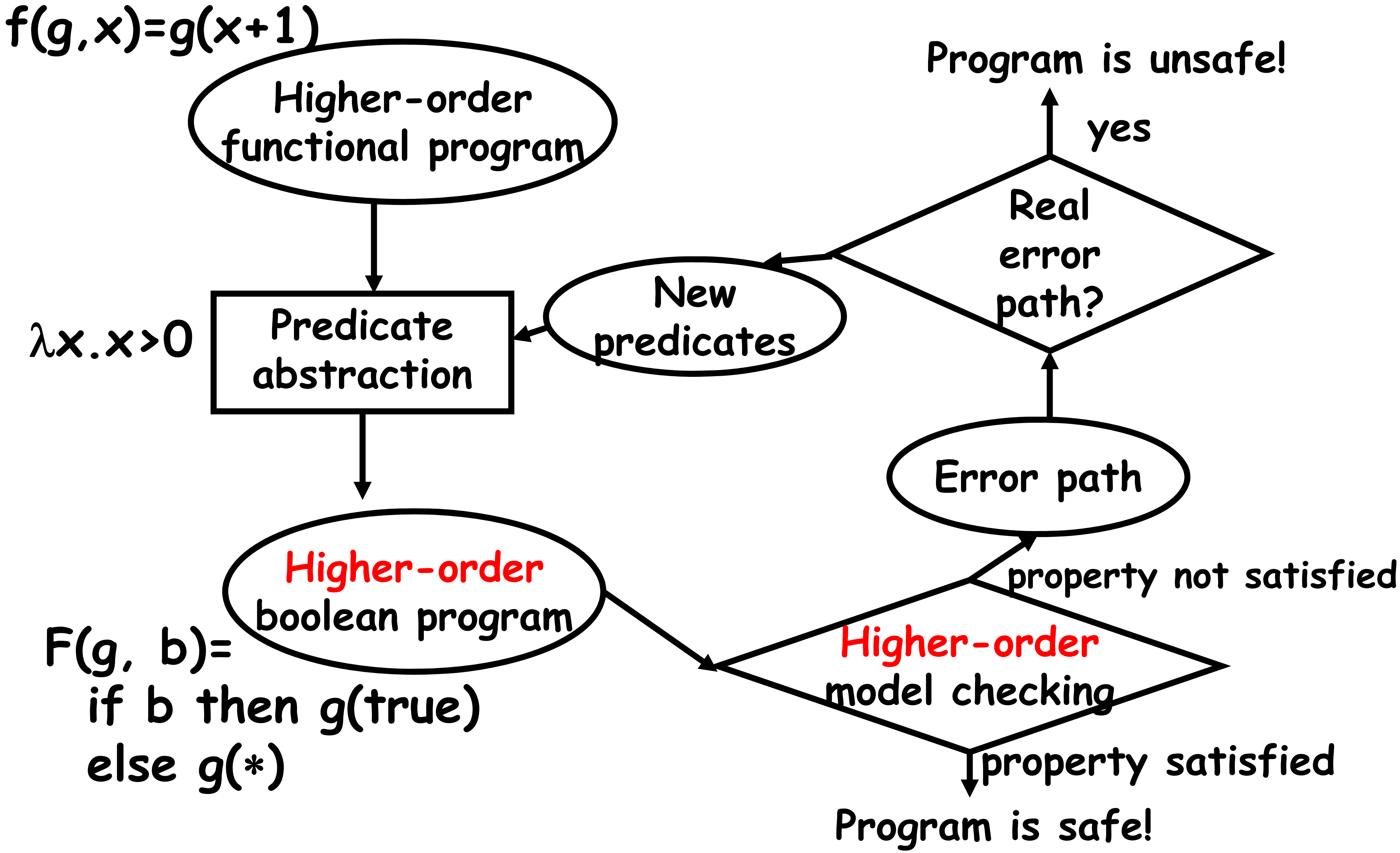
Outline

- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
- Applications to program verification
  - Verification of higher-order boolean programs
  - Dealing with infinite data domains (integers, lists,...)
- Current status and remaining challenges

Dealing with Infinite Data Domains

- Abstractions of data structures by tree automata [K.,Tabuchi&Unno, POPL 2010]
- Predicate abstraction and CEGAR [K-Sato-Unno, PLDI 2011] (c.f. BLAST, SLAM, …)

Predicate Abstraction and CEGAR for Higher-Order Model Checking



What are challenges?

- ♦ **Predicate abstraction**
  - How to consistently abstract a program, so that the resulting HOBP is a safe abstraction?

```
let sum n k = if n ≤ 0 then k 0
                  else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x ≥ m))
```

- ♦ **CEGAR** (counterexample-guided abstraction refinement)
  - How to find new predicates to abstract each term to guarantee progress (i.e. any spurious counterexample is eliminated)?

What are challenges?

- **Predicate abstraction**
  - How to consistently abstract a program, so that the resulting HOBP is a safe abstraction?

```
let sum n k = if n ≤ 0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x ≥ m))
```

Abstracted with
$\lambda x.x \geq m$

- **CEGAR**
  - How to find new predicates to abstract each term to guarantee progress (i.e. any spurious counterexample is eliminated)?

What are challenges?

- Predicate abstraction
  - How to consistently abstr
    so that the resulting HOB

> Should be abstracted with λx.x≥n

```
let sum n k = if n ≤ 0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x ≥ m))
```

> Abstracted with λx.x≥m

- CEGAR
  - How to find new predicates to abstract each term to guarantee progress (i.e. any spurious counterexample is eliminated)?

What are challenges?

- Predicate abstraction
  - How to consistently abstr
    so that the resulting HOB

Should be abstracted with $\lambda x.x \geq n$

```
let sum n k = if n ≤ 0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x ≥ m))
```

Abstracted with $\lambda x.x \geq m$

Should be abstracted with $\lambda x.x \geq n-1$

- CEGAR
  - How to find new predicates to abstract each term to guarantee progress (i.e. any spurious counterexample is eliminated)?

Abstraction Types as Abstraction Interface

$int[P_1, \ldots, P_n]$

Integers that should be abstracted by $P_1, \ldots, P_n$

e.g.

3: int[$\lambda x.x>0$, even?] $\Rightarrow$ (true, false)

$x{:}int[P_1, \ldots, P_n] \rightarrow int[Q_1, \ldots, Q_m]$

Assuming that argument x is abstracte ... ,...,P_n,
abstract the return value by $Q_1, \ldots, Q_m$

x>0?

e.g. $\lambda x.x+x$: ($x{:}int[\lambda x.x>0] \rightarrow int[\lambda y.y>x]$) $\Rightarrow$ $\lambda b.b$

$\lambda x.x+x$: ($x{:}int[\lambda x.x>1$, even?$] \rightarrow$ int[...

x+x>x?

$\Rightarrow \lambda(b_1, b_2).$if b_1 then ... *

Type-based Predicate Abstraction

$$\frac{\Gamma \vdash M_1 : (x{:}\tau_2 \to \tau) \Rightarrow N_1 \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow N_2}{\Gamma \vdash M_1 M_2 : [M_2/x]\tau \Rightarrow N_1 N_2}$$

source program

abstraction type

abstract program

$$\frac{\Gamma, x{:}\tau_x \vdash M : \tau \Rightarrow N}{\Gamma \vdash \lambda x.M : (x{:}\tau_x \to \tau) \Rightarrow \lambda x.N}$$

Type-based Predicate Abstraction

$$\frac{\Gamma \vdash M_1 : (x{:}\tau_2 \to \tau) \Rightarrow N_1 \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow N_2}{\Gamma \vdash M_1 M_2 : [M_2/x]\tau \Rightarrow N_1 N_2}$$

$$\frac{\Gamma,\ x{:}\tau_x \vdash M : \tau \Rightarrow N}{\Gamma \vdash \lambda x.M : (x{:}\tau_x \to \tau) \Rightarrow \lambda x.N}$$

Example (predicate abstraction)

```
let sum n k = if n≤0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

Abstraction type environment:

sum: (n:int[]→ (int[λx.x≥n] →★) →★)

```
let sum n k = if * then k true
              else sum ( ) (λb.k(if b then true else *))
in sum ( ) (λb.assert(b))
```

Example (predicate abstraction)

```
let sum n k = if n≤0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

Abstraction type environment:

sum: (n:int[]→ (int[λx.x≥n] →★) →★)

```
let sum n k = if * then k true
              else sum ( ) (λb.k(if b then true else *))
in sum ( ) (λb.assert(b))
```

Example (predicate abstraction)

```
let sum n k = if n≤0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

Abstraction type environment:

sum: (n:int[]→ (int[λx.x≥n] →★) →★)

```
let sum n k = if * then k true
              else sum ( ) (λb.k(if b then true else *))
in sum ( ) (λb.assert(b))
```

Example (predicate abstraction)

```
let sum n k = if n≤0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

Abstraction type environment:

sum: (n:int[]→ (int[λx.x≥n] →★) →★)

```
let sum n k = if * then k true
              else sum ( ) (λb.k(if b then true else *))
in sum ( ) (λb.assert(b))
```

Example (predicate abstraction)

```
let sum n k = if n≤0 then k 0
              else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

Abstraction type environment:

sum: (n:int[]→ (int[λx.x≥n] →★) →★)

```
let sum n k = if * then k true
              else sum ( ) (λb.k(if b then true else *))
in sum ( ) (λb.assert(b))
```

x≥n-1

Example (predicate abstraction)

```
let sum n k = if n≤0 then k 0
                  else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

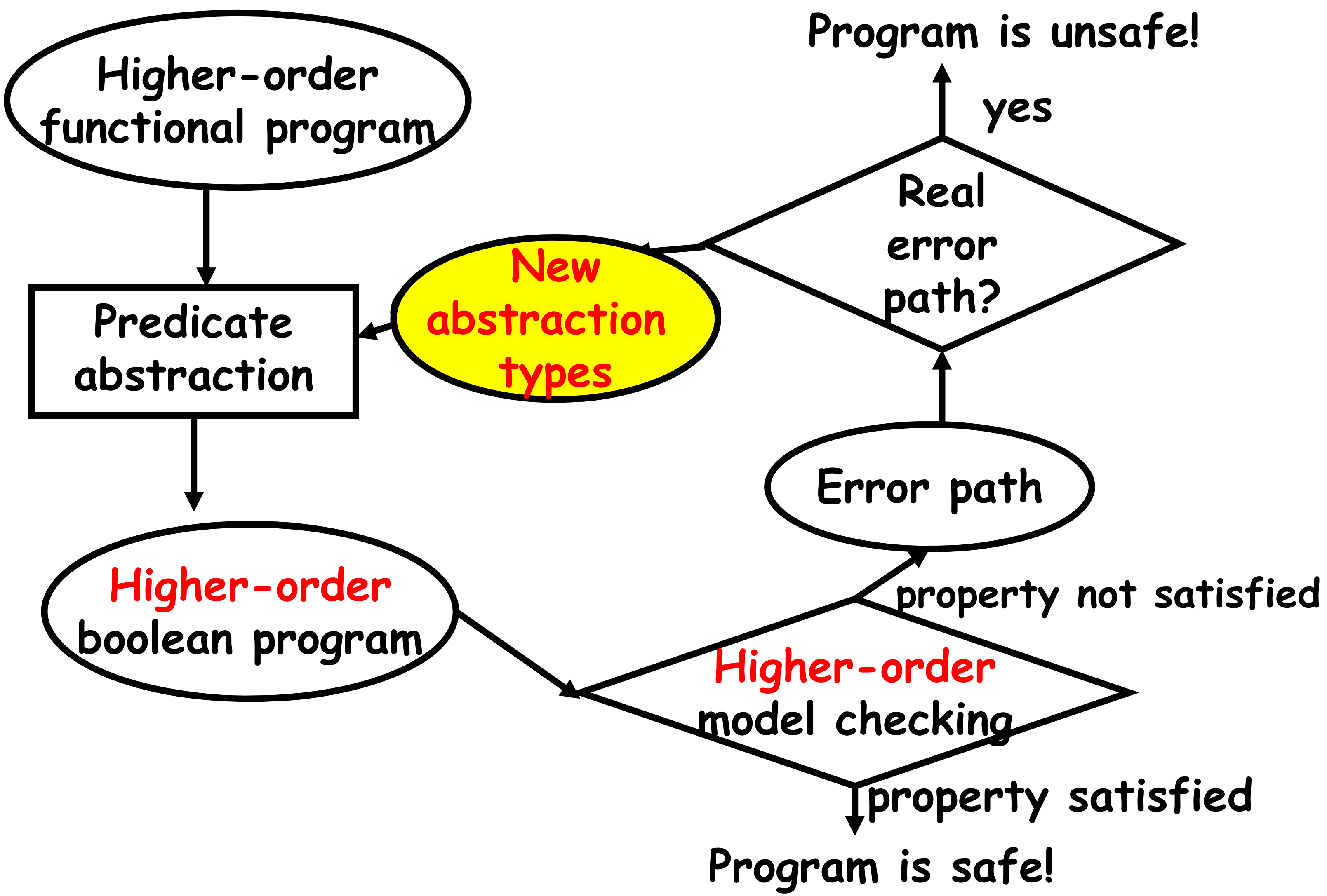
Abstraction type environment:

sum: (n:int[]→ (int[λx.x≥n] →★) →★)

```
let sum n k = if * then k true
                  else sum ( ) (λb.k(if b then true else *))
in sum ( ) (λb.assert(b))
```

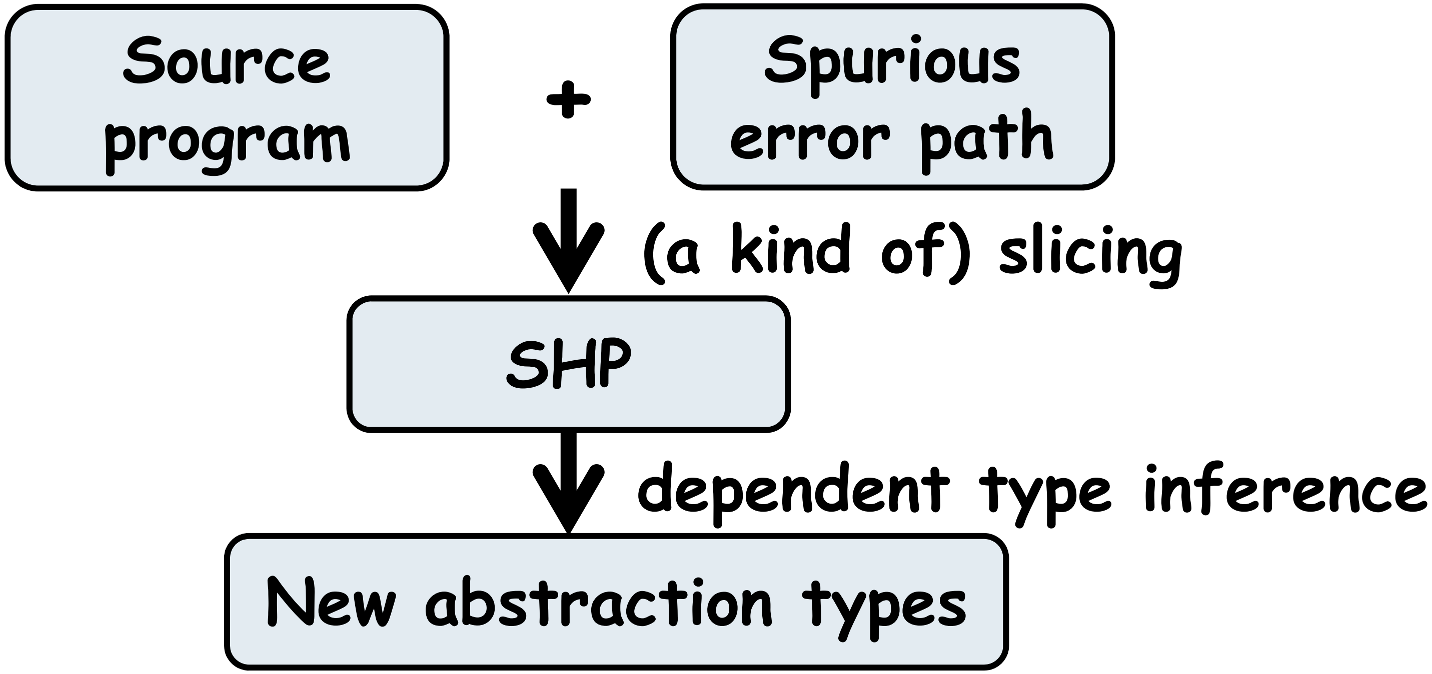
x≥n-1

Predicate Abstraction and CEGAR for Higher-Order Model Checking



Finding new abstraction types from a spurious error path

- Reduction to a dependent type inference problem for SHP (straightline higher-order program) that exactly corresponds to the spurious path



Example (predicate discovery)

```
let sum n k = if n≤0 then k 0
                 else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

sum: (n:int[]→ (int[ ] →★) →★)

```
let sum n k = if * then k ( )
                 else sum ( ) (λx.k ( ))
in sum ( ) (λx.assert(*))
```

spurious error path (with k = λx.assert(*)):

sum ( ) k → if * then k( ) else ... → k( ) → assert(*) → fail

Example (predicate discovery)

```
let sum n k = if n≤0 then k 0 else sum (n-1) (λx.k(x+n))
in sum m (λx.assert(x≥m))
```

Spurious error path:

sum ( ) k → if ∗ then k( ) else ... → k( ) → assert(∗) → fail

Example (predicate discovery)

let sum n k = if n≤0 then k 0 else sum (n-1) (λx.k(x+n))
in sum m (λx. if x≥m then () else fail)

↓

Spurious error path:
sum ( ) k → if ∗ then k( ) else ... → k( ) → assert(∗) → fail

Straightline higher-order program (SHP):
let sum n k = if (n≤0) then k 0 else _
in sum m (λx.if x≥m then _ else fail)

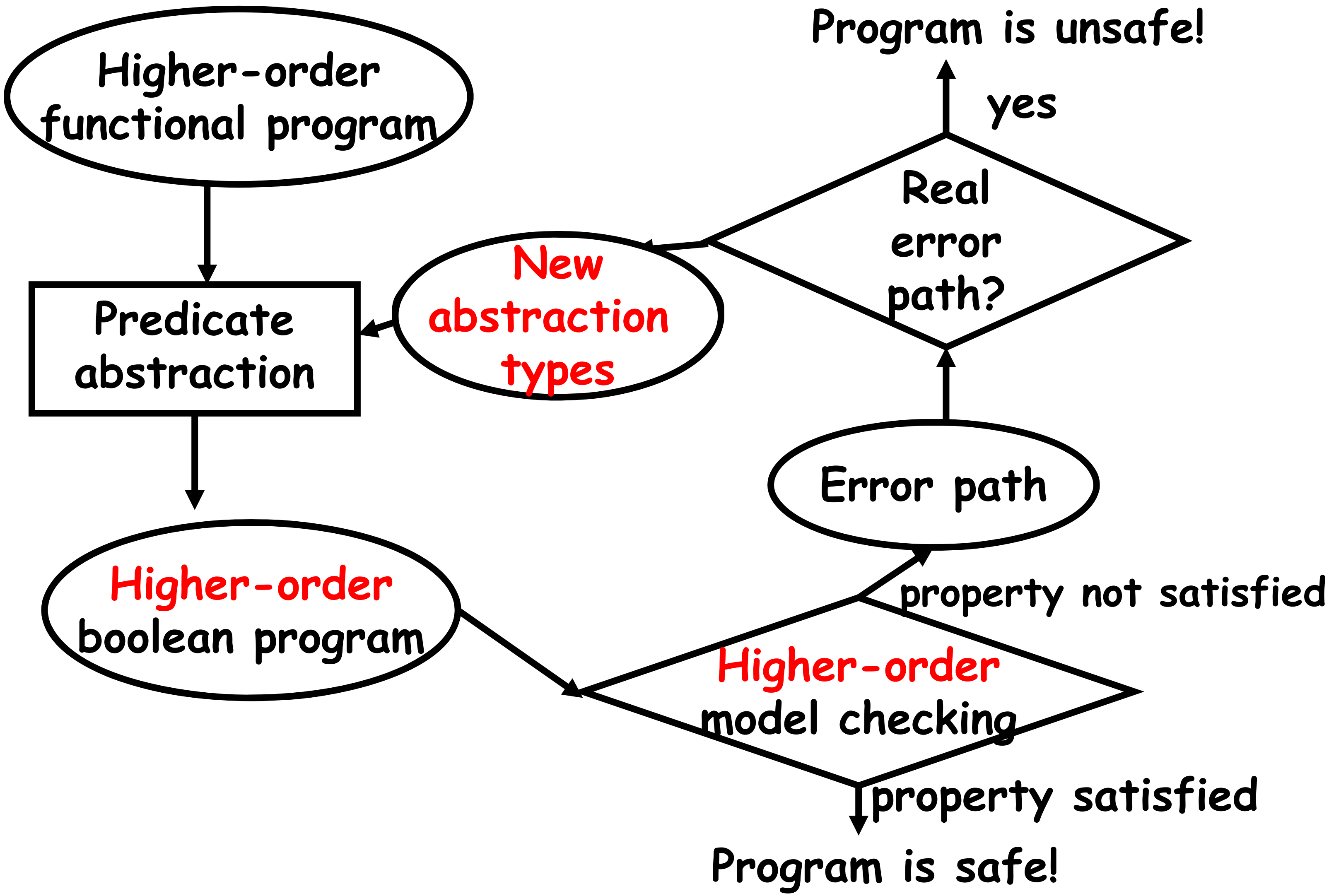
↓ Dependent type inference with interpolants [Unno&K. PPDP09]

Typing for SHP: sum: (n:int → ({x:int | x≥n} → ★) → ★

↓

Abstraction type: sum: (n:int[] → (x:int[λx.x≥n] → ★) → ★

Predicate Abstraction and CEGAR for Higher-Order Model Checking



Summary (up to this point)

- Higher-order model checking provides a sound and complete verification method for higher-order boolean programs

- Combination with predicate abstraction and CEGAR provides a sound verification method for simply-typed higher-order programs
  - Dependent types are used in the background

Outline

- ♦ Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
- ♦ Applications to program verification
  - Verification of higher-order boolean programs
  - Dealing with infinite data domains (integers, lists,...)
- ♦ Current status and remaining challenges
- ♦ Conclusion

Current Status of MoCHi

- Reachability verification for:
  - Call-by-value simply-typed λ-calculus with recursion, booleans and integers (or, call-by-value PCF)
- Ongoing work to support:
  - Exceptions
  - Algebraic data types

How far is the goal?
(“software model checker for ML”)

- Missing features:
  - algebraic data types
  - exceptions
  - let-polymorphism
  - modules
  - references

Inline let-definitions
or use intersection types

- Scalability problems
  - bottleneck: predicate discovery and higher-order model checking

How far is the goal?
("software model checker for ML")

- Missing features:
  - algebraic data types
  - exceptions   exception handlers as auxiliary continuations
  - let-polymorphism
  - modules
  - references
- Scalability problems
  - bottleneck: predicate discovery and higher-order model checking

Dealing with Exceptions

Extend CPS transformation by:

$$[\text{try } e_1 \text{ with } x \rightarrow e_2]\ k\ h = [e_1]\ k\ (\lambda x.[e_2]k\ h)$$

$$[\text{raise } e]\ k\ h = [e]\ h\ h$$

Ordinary continuation (k)

Exception handler (h)

How far is the goal?
(“software model checker for ML”)

- Missing features:
  - algebraic data types
  - exceptions
  - let-polymorphism
  - modules
  - references
- Scalability problems
  - bottleneck: predicate discovery and higher-order model checking

Dealing with algebraic data types

- Algebraic data types as functions

length — function from indices to elements

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

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

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

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

$\text{tl}\ (\text{len}, f) = \text{assert}(\text{len}>0);\ (\text{len}-1, \lambda i.\ f(i+1))$

Pros:

- Can reuse predicate abstraction and cegar for integers
- Generalization of container abstraction [Dillig-Dillig-Aiken]

Cons:

- More burden on model checker and cegar

How far is the goal?
("software model checker for ML")

- Missing features:
  - algebraic data types
  - exceptions
  - let-polymorphism
  - modules
  - references

store passing
(and stores as functions)?

- Scalability problem
  - bottleneck: predicate discovery and higher-order model checking

Problems on Predicate Abstraction and Discovery

- Too specific predicates are discovered

  ```
  let copy n = if n=0 then 0 else 1+copy(n-1)
  in assert(copy(copy m) = m)
  ```

  - discovered predicates (for return values)
    r=0, r=1, r=2, ...
  - what we want:
    r=n (for argument n)

- Supported predicates are limited
  - only linear constraints on base types
    - let rec rev l = … (* list reverse *)
      in assert(rev(rev l) = l)

How far is the goal?
("software model checker for ML")

- Missing features:
  - algebraic data types
  - exceptions
  - let-polymorphism
  - modules
  - references
- Scalability problems
  - bottleneck: predicate discovery and higher-order model checking

Higher-Order Model Checker TRecS [PPDP09]: Current Status

- Can verify recursion schemes of a few hundred lines in a few seconds
- Can become a bottleneck if:
  - The order of a program is very high (after CPS)
  - Many irrelevant predicates are used in abstractions

Direct support of call-by-value semantics?

BDD-like implementation techniques?

FAQ

Does HO model checking scale?
(It shouldn't, because of k-EXPTIME completeness)

Answer:

Don't know yet.

But there is a good hope it does, because:

(i) worst-case complexity is linear time in the program size (for safety properties)

$$O(|G| \times \underbrace{2^{2^{\cdot^{\cdot^{2^{(AQ)^{1+\varepsilon}}}}}}}_{k})$$

(ii) the worst-case behavior seems to come from the expressive power of higher-order functions

Recursion schemes generating $a^{2^m} c$

Order-1:

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

Order-0:

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

Exponential time algorithm for order-1

$\approx$

Polynomial time algorithm for order-0

Recursion schemes generating $a^{2^m} c$

Order-1:

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

Order-0:

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

k-EXPTIME algorithm for order-k

≈

Polynomial time algorithm for order-0

Recursion schemes generating $a^{2^m} c$

Order-1:

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

Order-0:

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

(fixed-parameter)
Polynomial time algorithm for order-k [K11FoSSaCS]
>>
Polynomial time algorithm for order-0

FAQ

Does HO model checking scale?
(It shouldn't, because of n-EXPTIME completeness)

Answer:

Don't know yet.

But there is a good hope it does, because:

(i) worst-case complexity is linear time in the program size (for safety properties)

(ii) the worst-case behavior seems to come from the expressive power of higher-order functions

Outline

- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
- Applications to program verification
  - Verification of higher-order boolean programs
  - Dealing with infinite data domains (integers, lists,...)
- Current status and remaining challenges
- Conclusion

Conclusion

- Higher-order model checking is useful for verification of functional programs
- MoCHi: software model checker for a tiny subset of ML
- A long way to construct a scalable, full-scale software model checker for ML
  - Support of more features: algebraic data structures,...
  - Better predicate abstraction and discovery
  - Better algorithms and implementations of higher-order model checker
  - Modular verification

Exciting research topics for the next decade!

References

- A short survey:
  [K, LICS11]
- Applications to program verification
  [K,POPL09] [K&Tabuchi&Unno, POPL10]
  [K&Sato&Unno, PLDI11]
- From model checking to type checking
  [K,POPL09] [K&Ong,LICS09] [Tsukada&K, FoSSaCS10]
- HO model checking algorithms
  [K, PPDP09] [K, FoSSaCS11]
- Complexity of HO model checking
  [K&Ong, ICALP09]