Higher-Order Model Checking: From Theory to Practice

Naoki Kobayashi
Tohoku University

In collaborations with:

Luke Ong (University of Oxford)

Ryosuke Sato, Naoshi Tabuchi, Takeshi Tsukada, Hiroshi Unno (Tohoku University)

What's This Talk About?

- NOT a general survey
  (see the paper in the proceedings for this)

- BUT an overview of our recent work,
  to get
  practical applications
  (e.g. software model checker for ML)
  from
  theoretical results [Knapik et al.02;Ong06;...]
  on higher-order model checking

Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problems
- Applications
  - program verification:
    "software model checker for ML"
  - data compression
- Algorithms for higher-order model checking
- Future directions

Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problems
- Applications
  - program verification:
    "software model checker for ML"
  - data compression
- Algorithms for higher-order model checking
- Future directions

Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-0 scheme
(regular tree grammar)

$S \rightarrow a \ c \ B$

$B \rightarrow b \ S$

Higher-Order Recursion Scheme

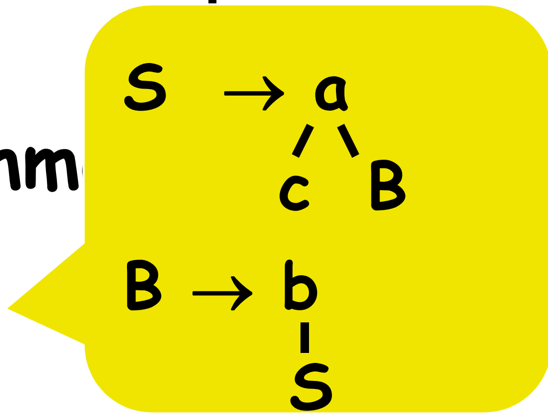
- Grammar for generating an infinite tree

Order-0 scheme

(regular tree gramm

$S \to a\ c\ B$

$B \to b\ S$



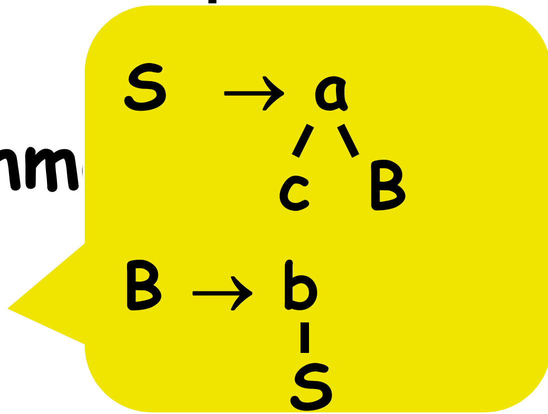
Higher-Order Recursion Scheme

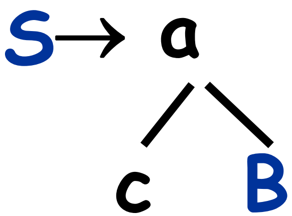
- Grammar for generating an infinite tree

Order-0 scheme
(regular tree gramm

$S \rightarrow a\ c\ B$

$B \rightarrow b\ S$



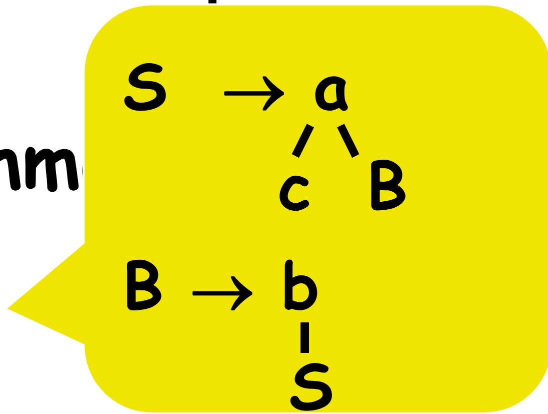


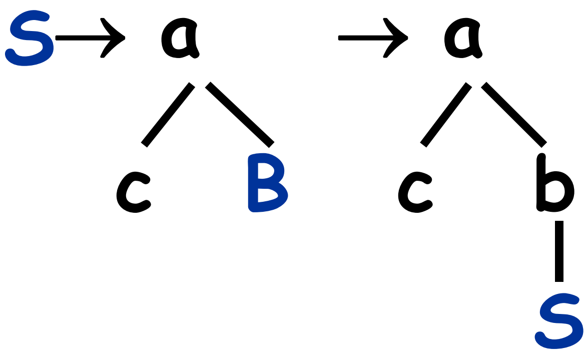
Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-0 scheme
(regular tree gramm

S → a c B
B → b S



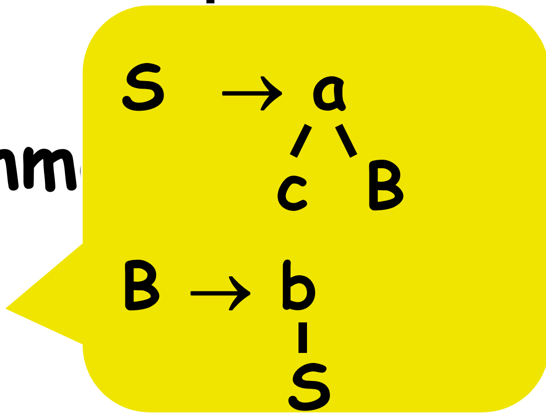


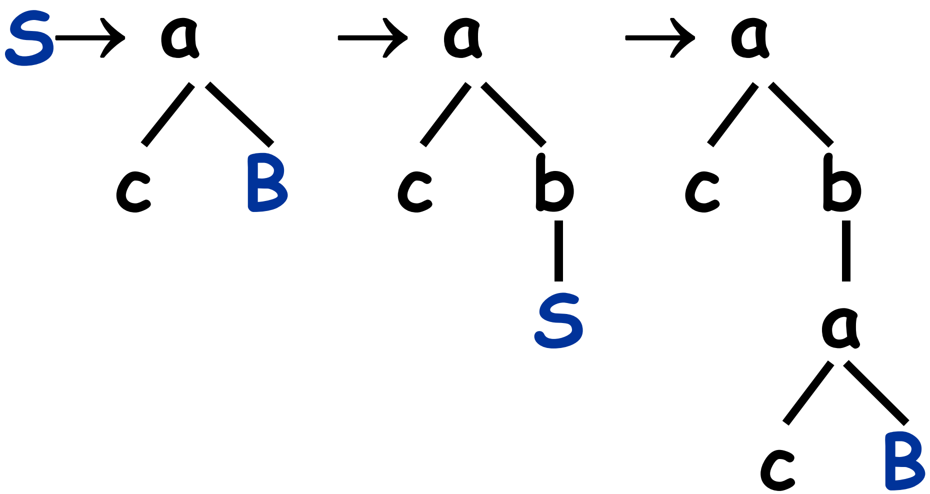
Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-0 scheme
(regular tree gramm

S → a c B
B → b S





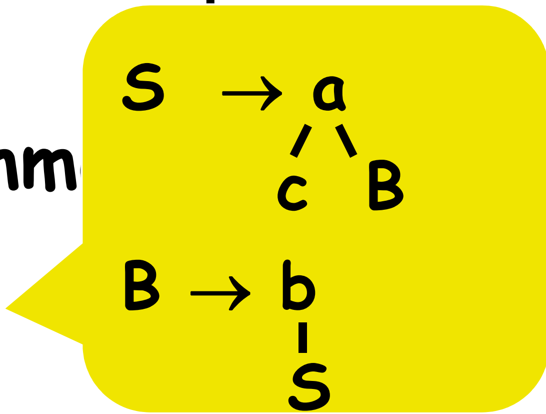
Higher-Order Recursion Scheme

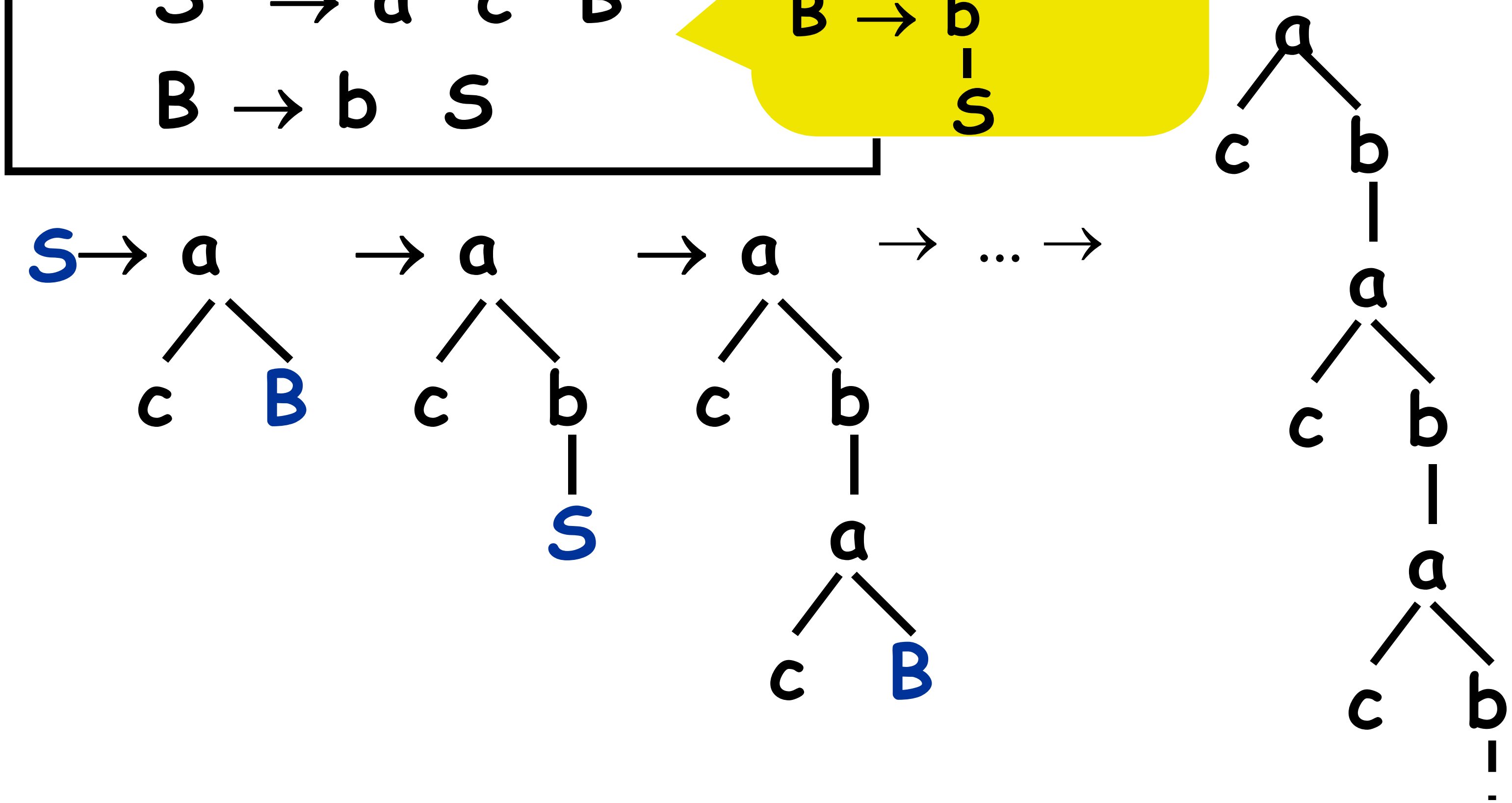
- Grammar for generating an infinite tree

Order-0 scheme
(regular tree gramm

$S \rightarrow a\ c\ B$

$B \rightarrow b\ S$





Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-1 scheme

$S \rightarrow A\ c$

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

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

S

Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-1 scheme
$S \rightarrow A\ c$
$A \rightarrow \lambda x.\ a\ x\ (A\ (b\ x))$
$S: o,\ A: o \rightarrow o$

$S \rightarrow A\ c$

Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-1 scheme
$S \rightarrow A\ c$
$A \rightarrow \lambda x.\ a\ \ x\ \ (A\ (b\ x))$
S: o, A: o$\rightarrow$ o

$S \rightarrow A\ c \rightarrow a$

$$\begin{array}{c} a \\ \diagup \quad \diagdown \\ c \quad A(b\ c) \end{array}$$

Higher-Order Recursion Scheme

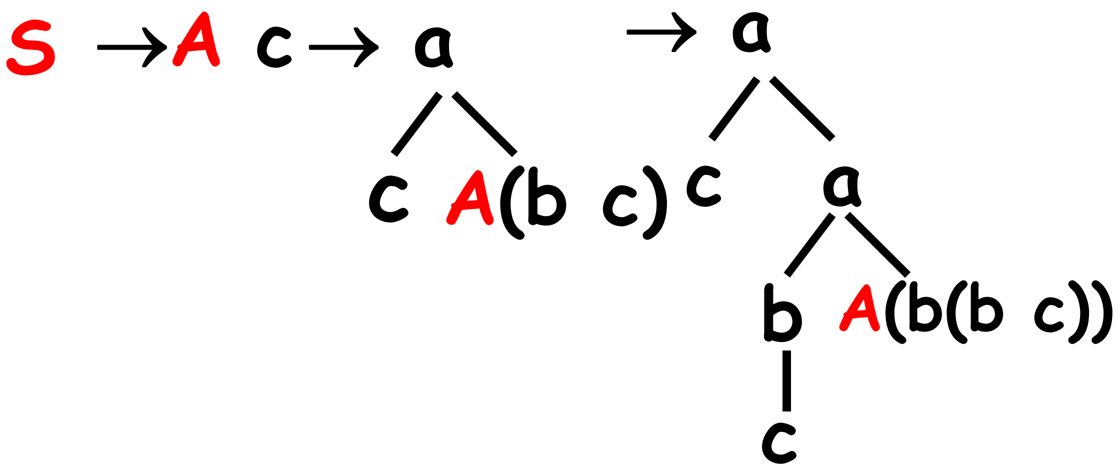
- Grammar for generating an infinite tree

Order-1 scheme

$S \rightarrow A\ c$

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

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



Higher-Order Recursion Scheme

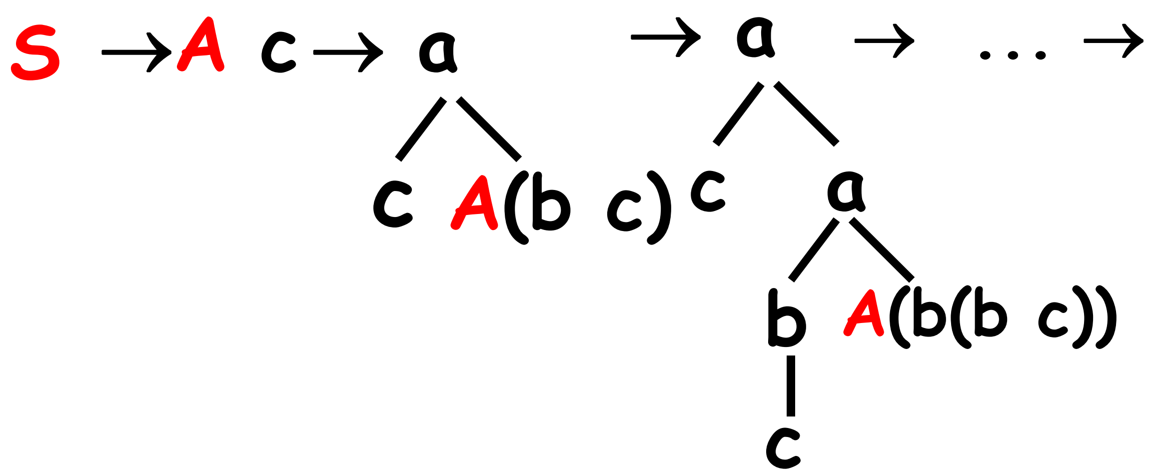
- Grammar for ... ite tree

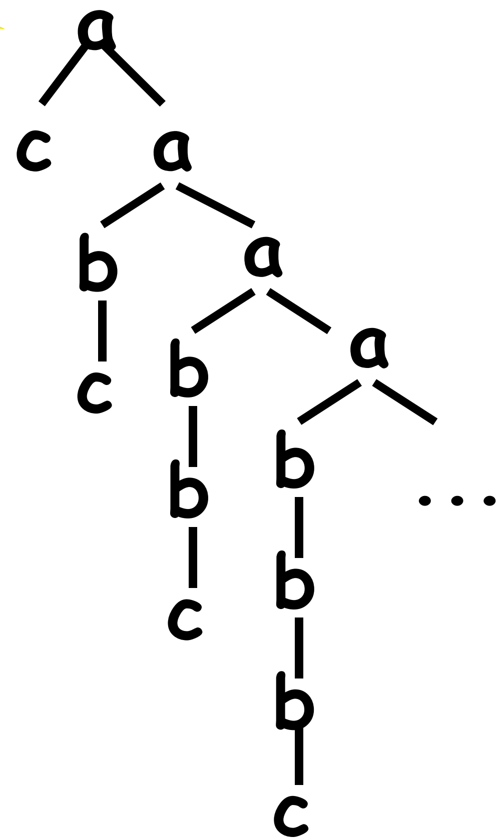
Tree whose paths are labeled by $a^{m+1}\, b^m\, c$

Order-1 schem...

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

S: o, A: o→ o





Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-1 scheme

$S \rightarrow A\ c$

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

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

Higher-order recursion schemes

$\approx$

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"?

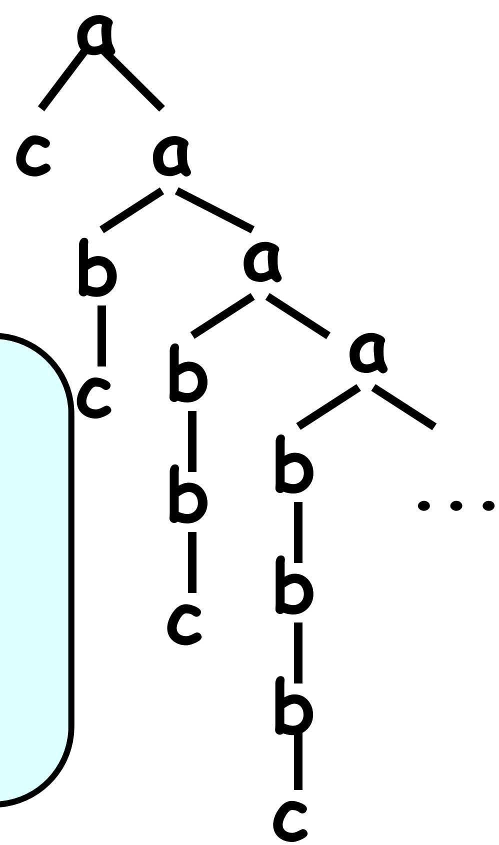
Higher-Order Recursion Scheme

- Grammar for generating an infinite tree

Order-1 scheme

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

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 below "b"?

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

(Non-exhaustive) History

- 70s: (1st-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

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problems
- Applications
  - program verification:
    "software model checker for ML"
  - data compression
- Algorithms for higher-order model checking
- Future directions

From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

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

→ **Program Transformation** →

Rec. scheme
(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)
```

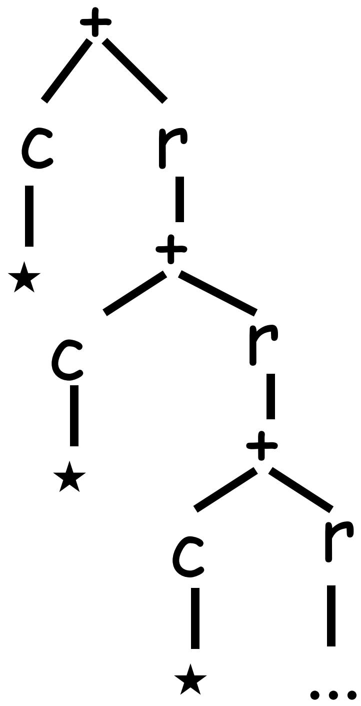
Is the file "foo" accessed according to read* close?

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)
```

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

$S \rightarrow F\ d\ \star$



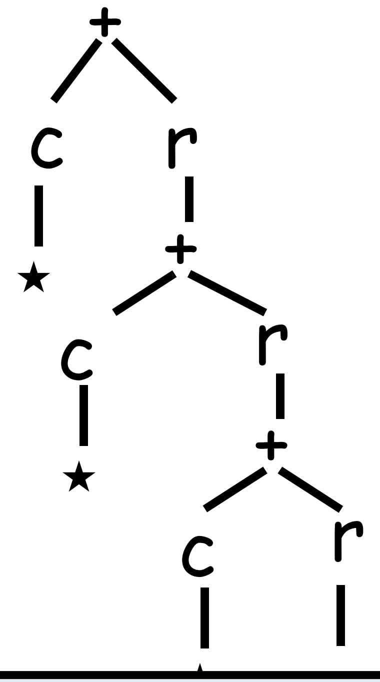
Is the file "foo" accessed according to read* close?

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 ... ing:

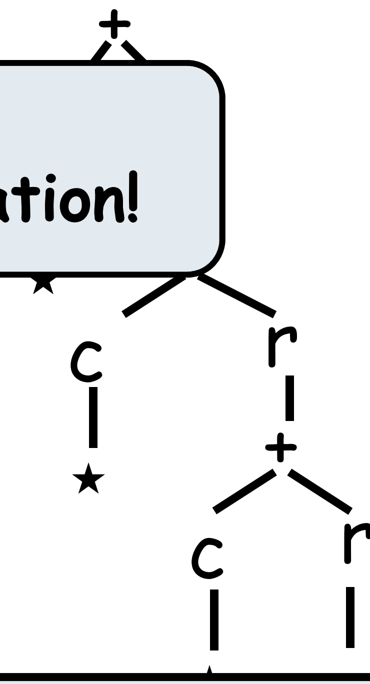
continuation parameter, expressing how "foo" is accessed after the call returns

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

$S \rightarrow F\ d\ \star$

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

CPS Transformation!



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

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

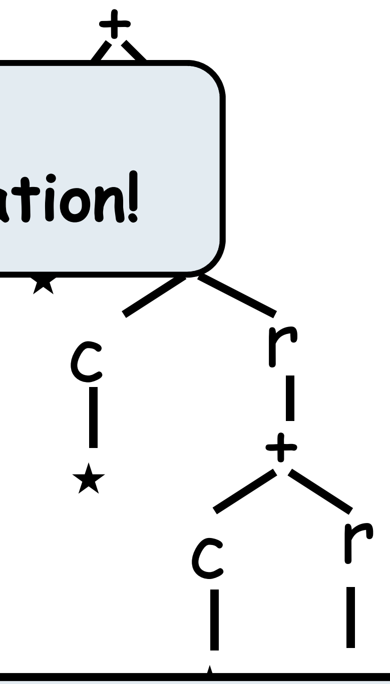
From Program Verification to Model Checking: Example

```
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?

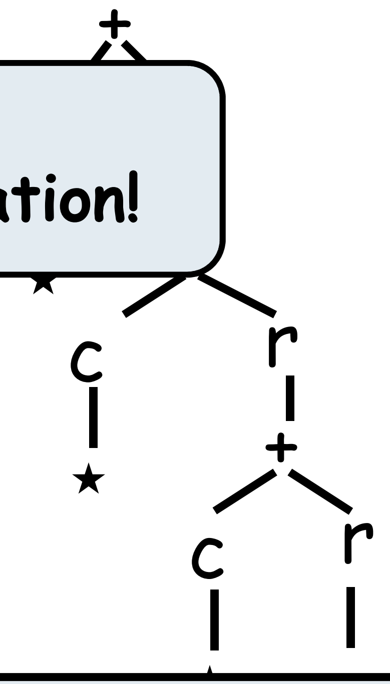
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \star$

CPS Transformation!



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

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

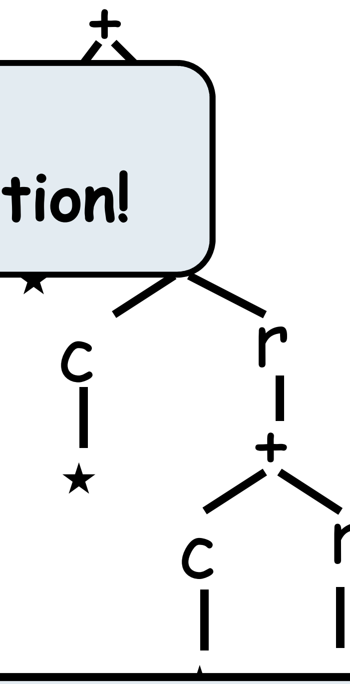
From Program Verification to Model Checking: Example

```
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?

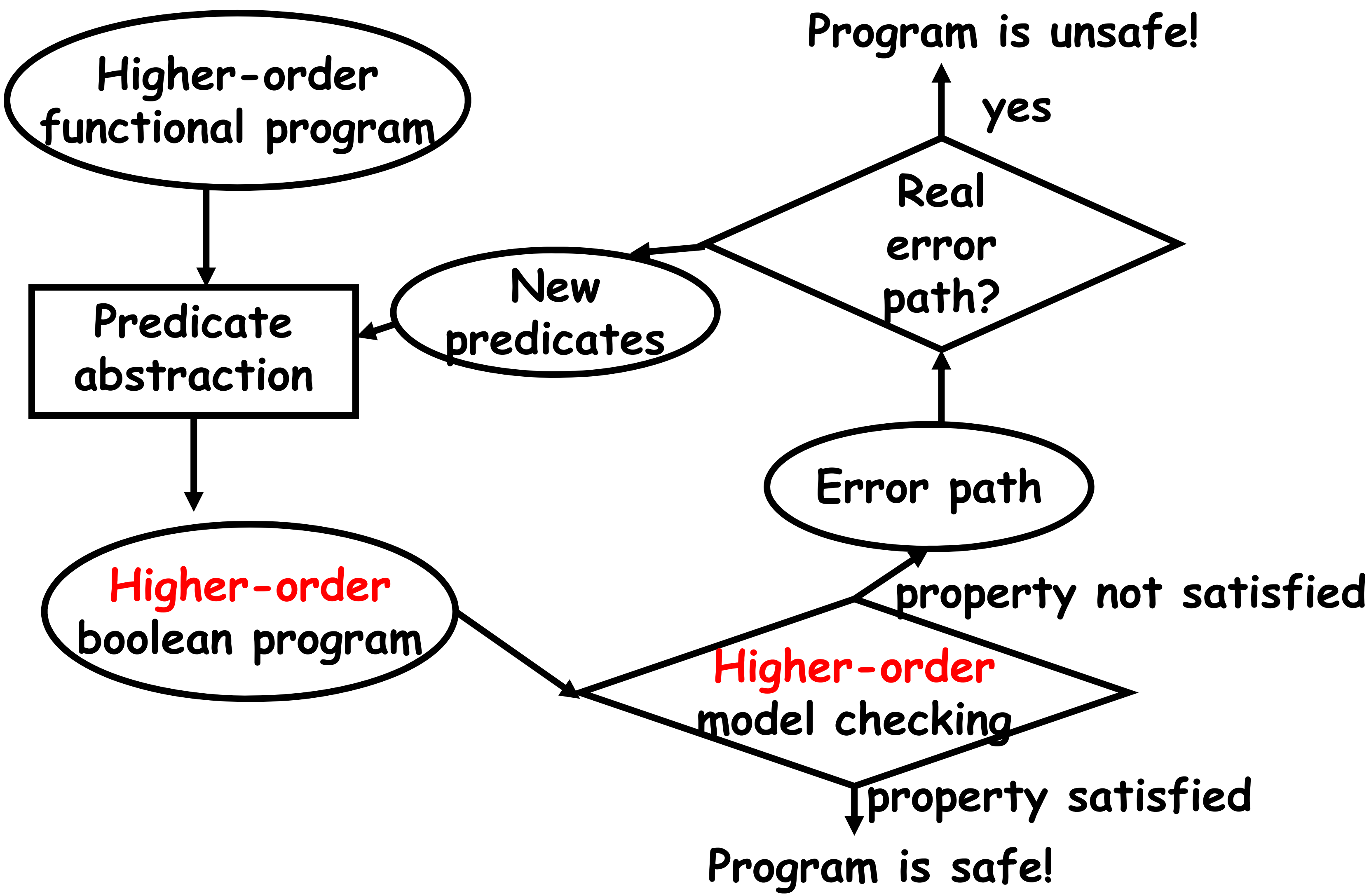
From Program Verification to Model Checking Recursion Schemes

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 (e.g. booleans)
- A large class of verification problems:
  resource usage verification (or typestate checking),
  reachability, flow analysis,...

Combination with Predicate Abstraction and CEGAR [K&Sato&Unno,PLDI11]



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	Higher-order model checking

infinite state model checking

Applications to Program Verification: Summary

- **Sound, complete, and automatic** for simply-typed programs with recursion and **finite base types** (e.g. booleans)

- **Sound (but incomplete) and automatic** for simply-typed programs with recursion and **infinite base types** (e.g. integers, lists, ...) by combination with predicate abstraction and CEGAR

Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problems
- Applications
  - program verification:
    "software model checker for ML"
  - data compression
- Algorithms for higher-order model checking
- Future directions

Applications to Data Compression

- Compressed data as higher-order grammars (c.f. Kolmogorov complexity)
  - Hyper-exponential compression ratio
- Data processing without decompression using higher-order model checking

Compressed Data as Recursion Schemes

$$\underbrace{a(a(a(.....(a}_{2^n}(e))...)))$$

$\Downarrow$ Compression ratio : $O(n/2^n)$

$$S = \underbrace{Twice(Twice(...(Twice}_{n}\ a)...))\ e$$

$$Twice\ f\ x = f(f(x))$$

Compressed Data as Recursion Schemes

$$\underbrace{a(a(a(\ldots\ldots(a}_{\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}n}(e))\ldots)))$$

$$\Big\Downarrow \text{ compression}$$

$$S = \underbrace{((\mathrm{Twice}\ \mathrm{Twice})\ \ldots\ \mathrm{Twice})}_{n}\ a\ e$$

$$\mathrm{Twice}\ f\ x = f(f(x))$$

Applications to Data Compression

- Compressed data as higher-order grammars
  - Hyper-exponential compression ratio

- Data processing without decompression using higher-order model checking
  - pattern match queries
  - associated data processing to compute:
    - matching positions
    - the number of matches
    - ... (whatever expressed by transducers)

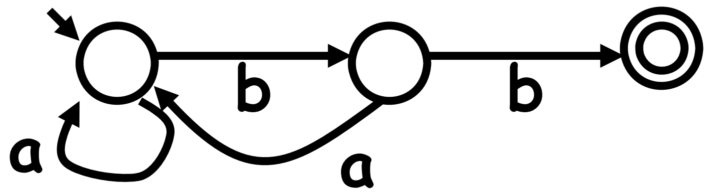
Pattern Matching without Decompression by Higher-Order Model Checking

Does Tree(G) match a pattern P?

e.g. contains "bb"?

Is Tree(G) accepted by M_P?

e.g. accepted by the following automaton?



Example: a Fibonacci word

Fibonacci word:
w_0=b, w_1=a, w_2=w_1w_0=ab, w_3=w_2w_1=aba,....,
w_n=$w_{n-1}w_{n-2}$

$\Downarrow$ Compression (case $n=2^m$)

$S = \overbrace{\text{Twice(Twice(...(Twice}}^{m}\text{ Next)..)) Fst b a e}$
Next k u v = k v (Concat v u)
Concat f g x = f(g(x))
Twice f x = f(f(x))

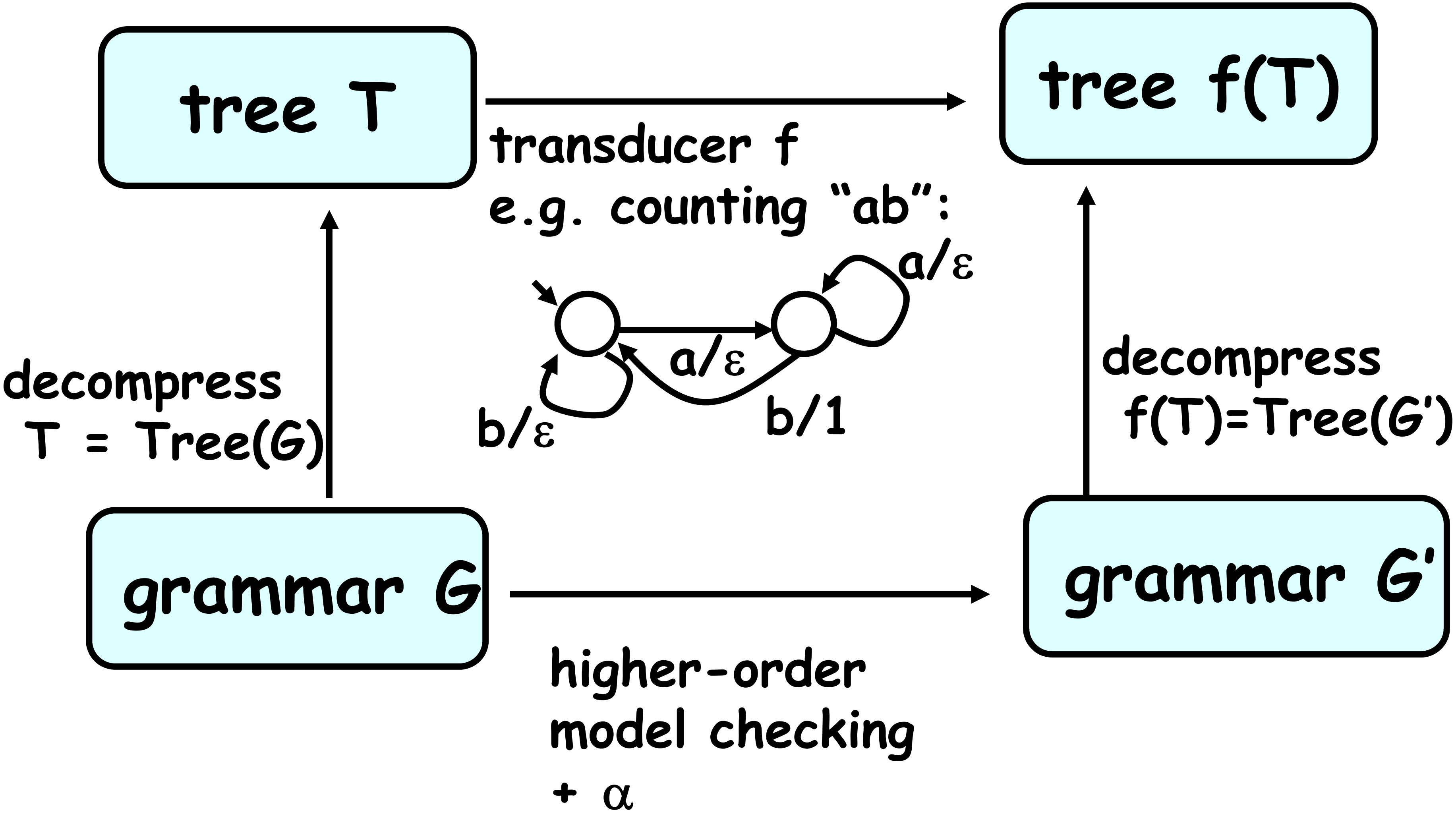
Query: Does w_{1024} contain "bb"?
(Note: $|w_{1024}| > 10^{200}$)

Applications to Data Compression

- Compressed data as higher-order grammars
  - Hyper-exponential compression ratio

- Data processing without decompression using higher-order model checking
  - pattern match queries
  - associated data processing to compute:
    - matching positions
    - the number of matches
    - ... (whatever expressed by transducers)

Data Transformation without Decompression



Applications to Data Compression: Summary

- Compressed data as higher-order grammars
  - Hyper-exponential compression ratio

- Data processing without decompression using higher-order model checking
  - pattern match queries; and
  - associated data processing expressed by transducers

Outline

- What is higher-order model checking?
- Applications
  - program verification:
    "software model checker for ML"
  - data compression
- Algorithms for higher-order model checking
  - from model checking to typing
  - practical algorithms
- Future directions

Difficulty of higher-order model checking

- Extremely high worst-case complexity
  - n-EXPTIME complete [Ong, LICS06]

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

  - Earlier algorithms [Ong06;Aehlig06;Hague et al.08] almost **always** suffer from n-EXPTIME bottleneck.

Our approach: from model checking to typing

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])

Model Checking Problem

Given
  G: higher-order recursion scheme
    (without safety restriction)
  A: alternating parity tree automaton (APT)
    (a formula of modal μ-calculus or MSO),
does A accept Tree(G)?

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

Model Checking Problem: Restricted version

Given

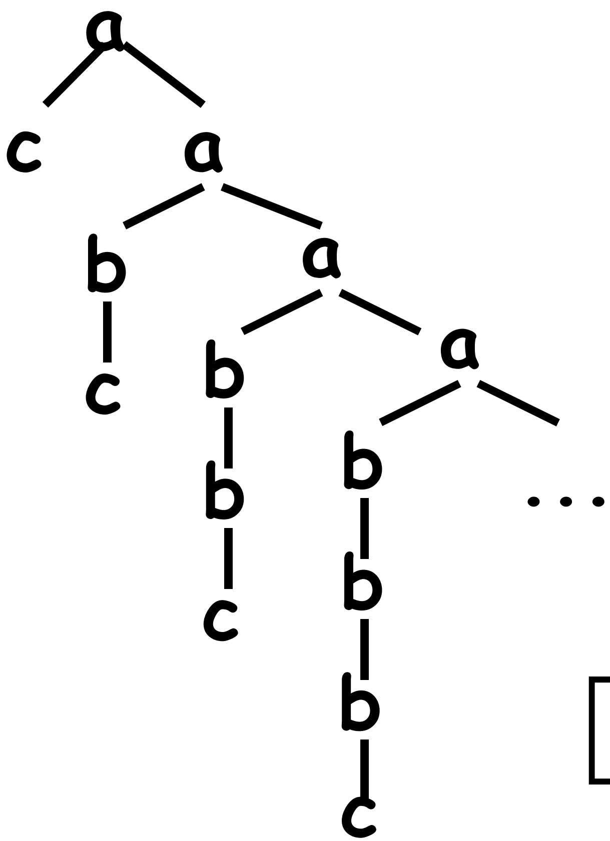
G: higher-order recursion scheme (without safety restriction)

A: trivial automaton [Aehlig CSL06] (Büchi tree automaton where all the states are accepting states)

does A accept Tree(G)?

See [K.&Ong, LICS09] for the general case (full modal μ-calculus model checking)

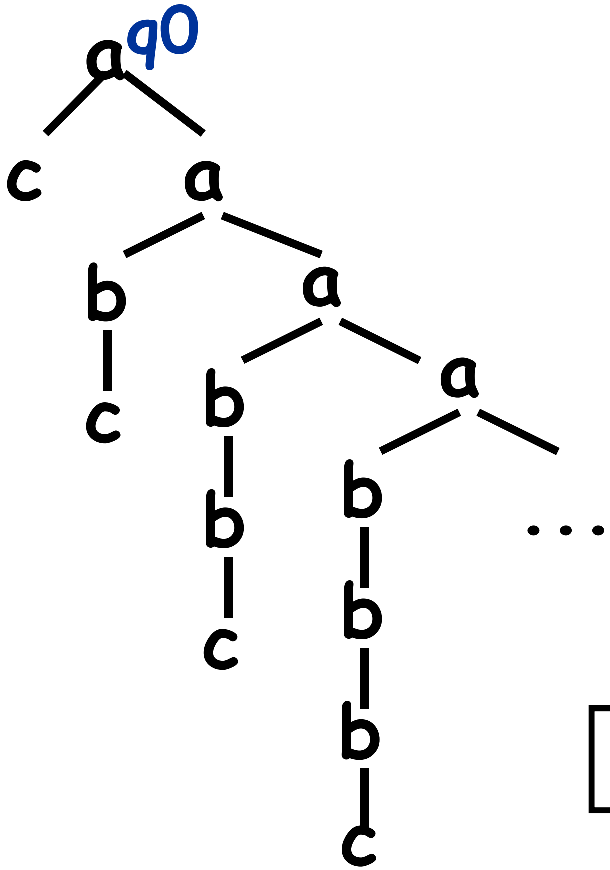
Trivial tree automaton for infinite trees



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

"a" does not occur below "b"

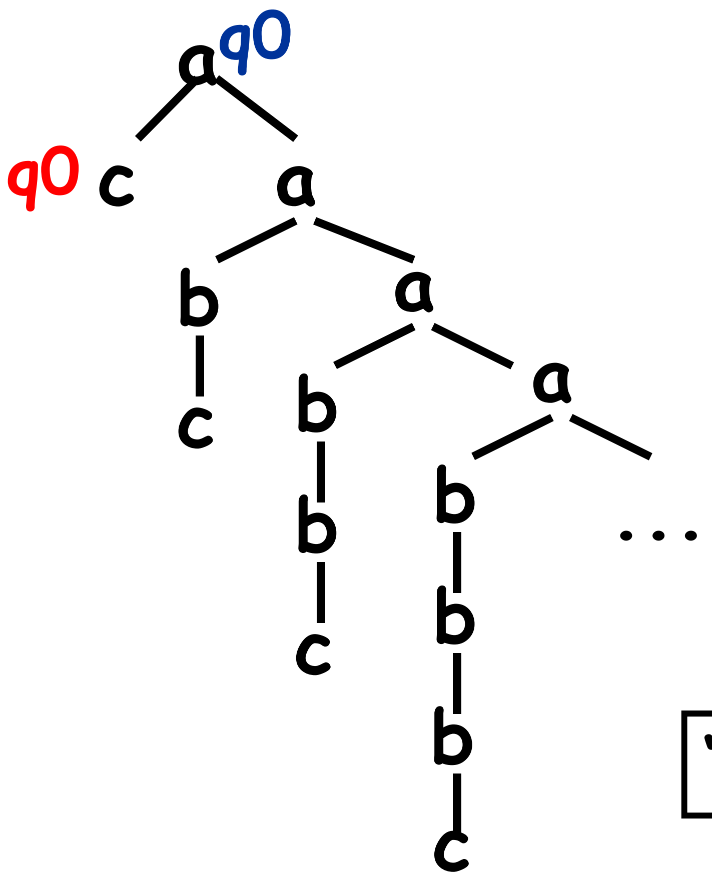
Trivial tree automaton for infinite trees



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

“a” does not occur below “b”

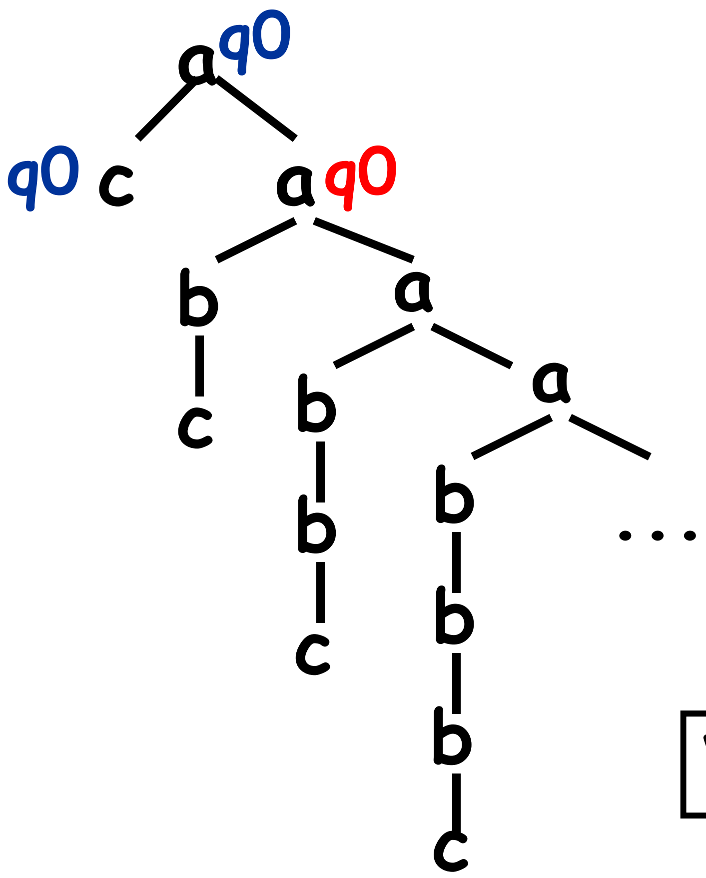
Trivial tree automaton for infinite trees



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

“a” does not occur below “b”

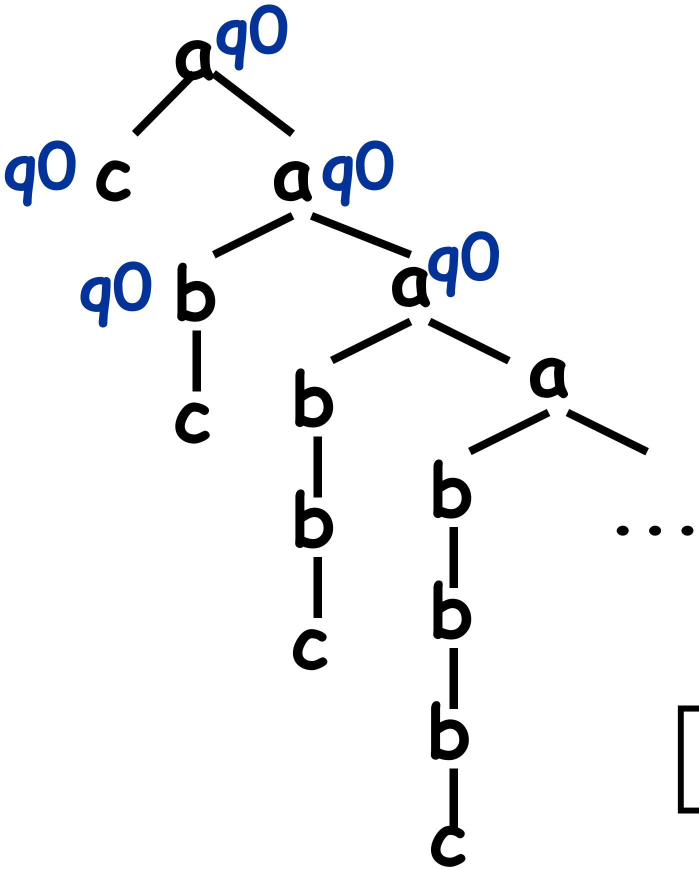
Trivial tree automaton for infinite trees



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

“a” does not occur below “b”

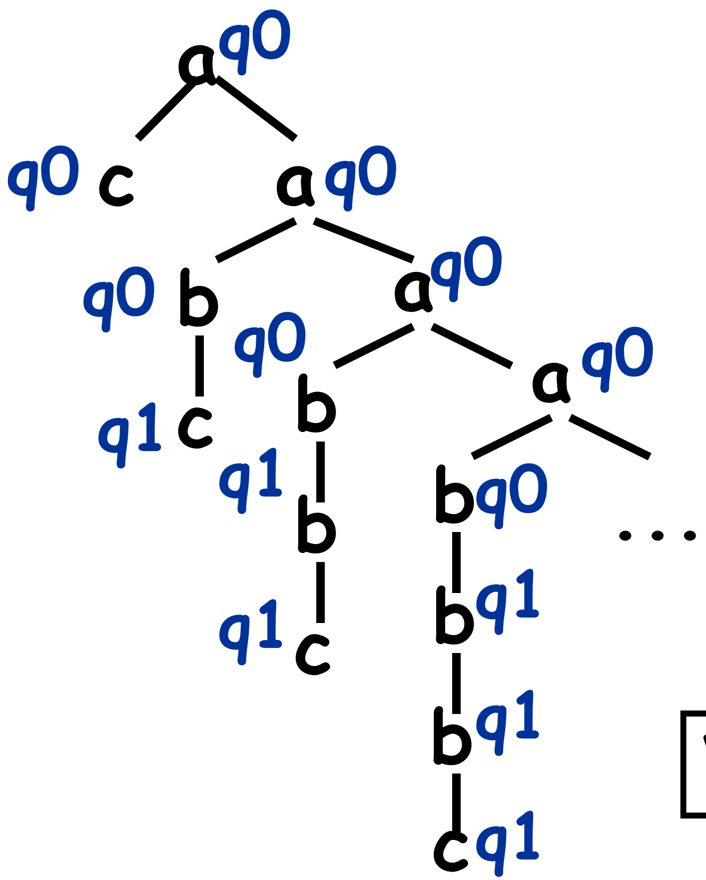
Trivial tree automaton for infinite trees



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

“a” does not occur below “b”

Trivial tree automaton for infinite trees



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

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

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

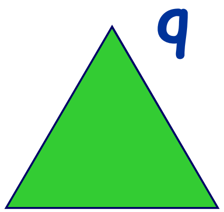
$\delta(q0,\ c) = \varepsilon$

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

"a" does not occur below "b"

Types for Recursion Schemes

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



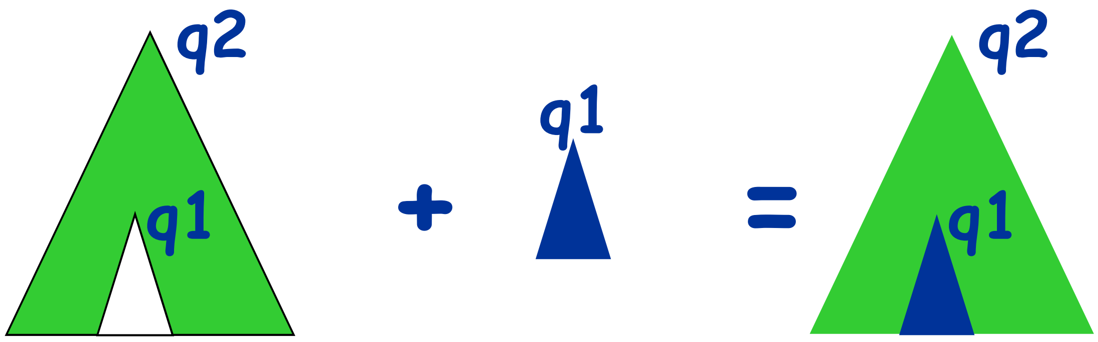
  - $q1 \wedge q2$: trees accepted from both q1 and q2

Is Tree(G) accepted by A?

Does Tree(G) have type q_0?

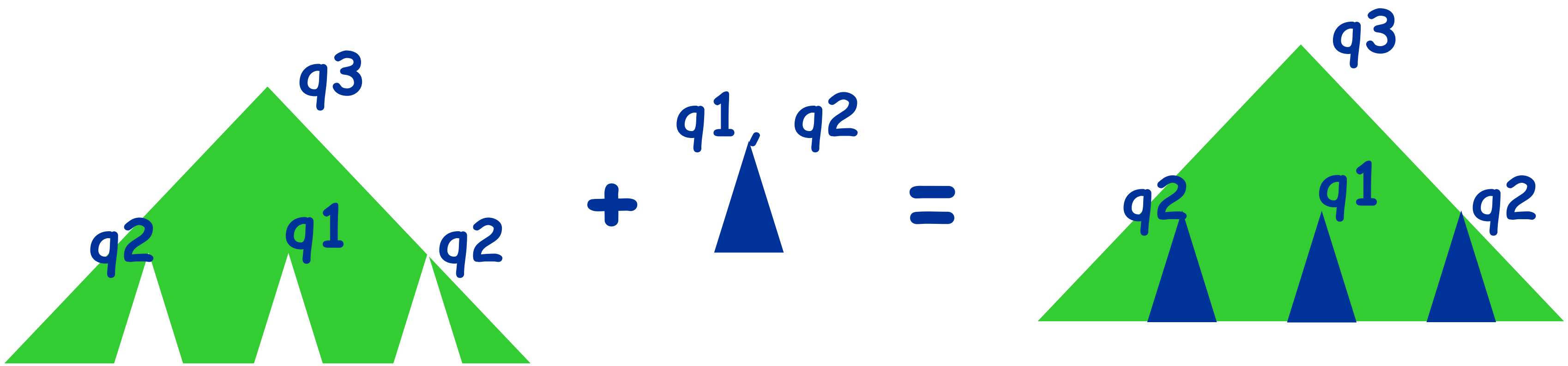
Types for Recursion Schemes

- 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
  - $q1 \wedge q2 \to q3$:
    functions that take a tree of type $q1 \wedge q2$ and return a tree of type q3

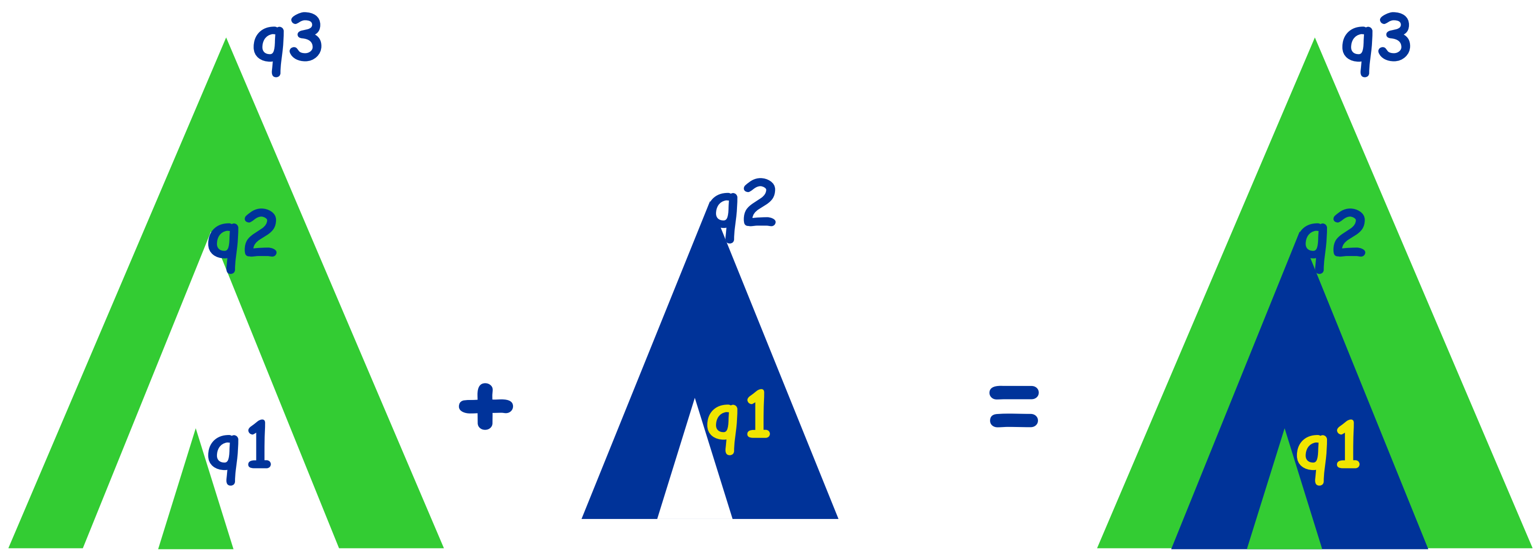


Types for Recursion Schemes

- 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

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

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

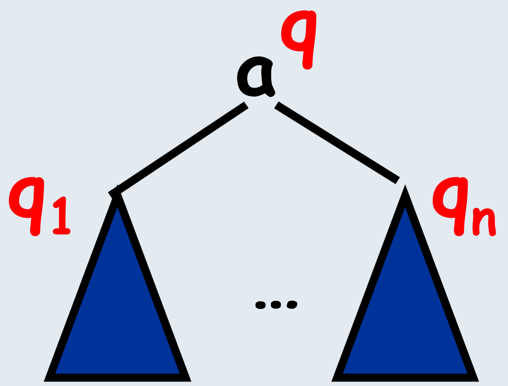
$$\frac{\Gamma, x{:}\tau_1, \ldots, x{:}\tau_n \vdash t{:}\tau}{\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge \tau_n \to \tau}$$

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

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

Typing

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



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

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

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

Typing

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

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

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

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

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

Soundness and Completeness
[K., POPL2009]

Tree(G) is accepted by A

if and only if

S has type q_0 in TS(A),

i.e. $\exists\Gamma.(S{:}q_0 \in \Gamma \wedge\ \vdash \{F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n\} : \Gamma)$

if and only if

$\exists\Gamma.(S{:}\ q_0 \in \Gamma \wedge \forall(F_k{:}\tau) \in \Gamma.\ \Gamma \vdash t_k : \tau\ )$

$G = \{F_1 \rightarrow t_1,\ \ldots,\ F_m \rightarrow t_m\}$ (with $S{=}F_1$)

A: Trivial automaton with initial state q_0

TS(A): Intersection type system for A

Soundness and Completeness

[K., POPL2009]

Tree(G) is accepted by A

if and only if

S has type q_0 in TS(A),

i.e. $\exists\Gamma.(S{:}q_0 \in \Gamma \land |\!- \{F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n\} : \Gamma)$

if and only if

$\exists\Gamma.(S{:}\ q_0 \in \Gamma \land \forall(F_k{:}\tau) \in \Gamma.\ \Gamma |\!- t_k : \tau)$

if and only if

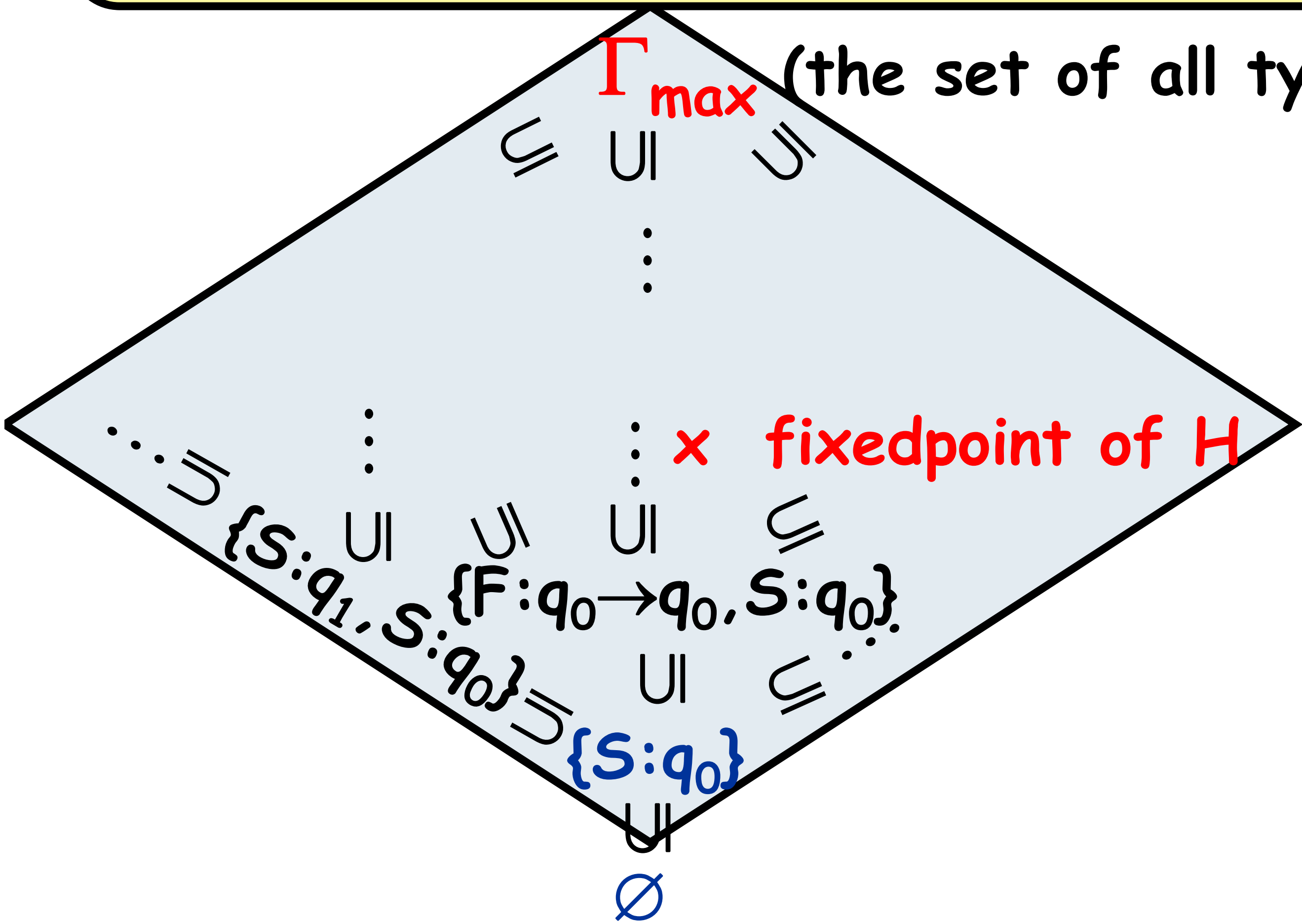
$\exists\Gamma.(S{:}\ q_0 \in \Gamma \land \Gamma = H(\Gamma))$

for $H(\Gamma) = \{ F_k{:}\tau \in \Gamma \mid \Gamma |\!- t_k{:}\tau \}$

Function to filter out invalid type bindings

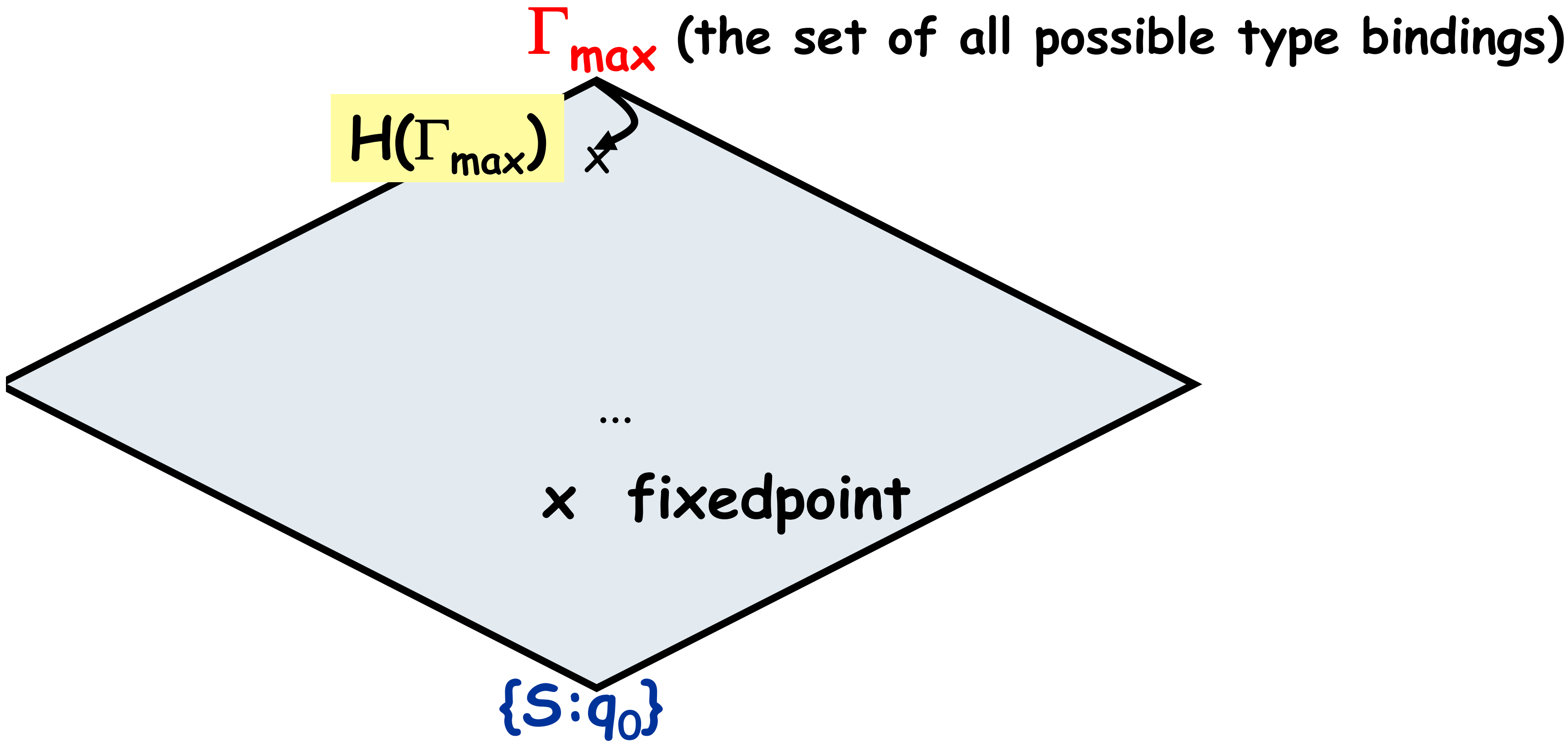
Type checking (=model checking) problem

Is there a fixedpoint of H greater than $\{S:q_0\}$?
(where $H(\Gamma) = \{\, F_j:\tau \in \Gamma \mid \Gamma \vdash t_j:\tau \,\}$)



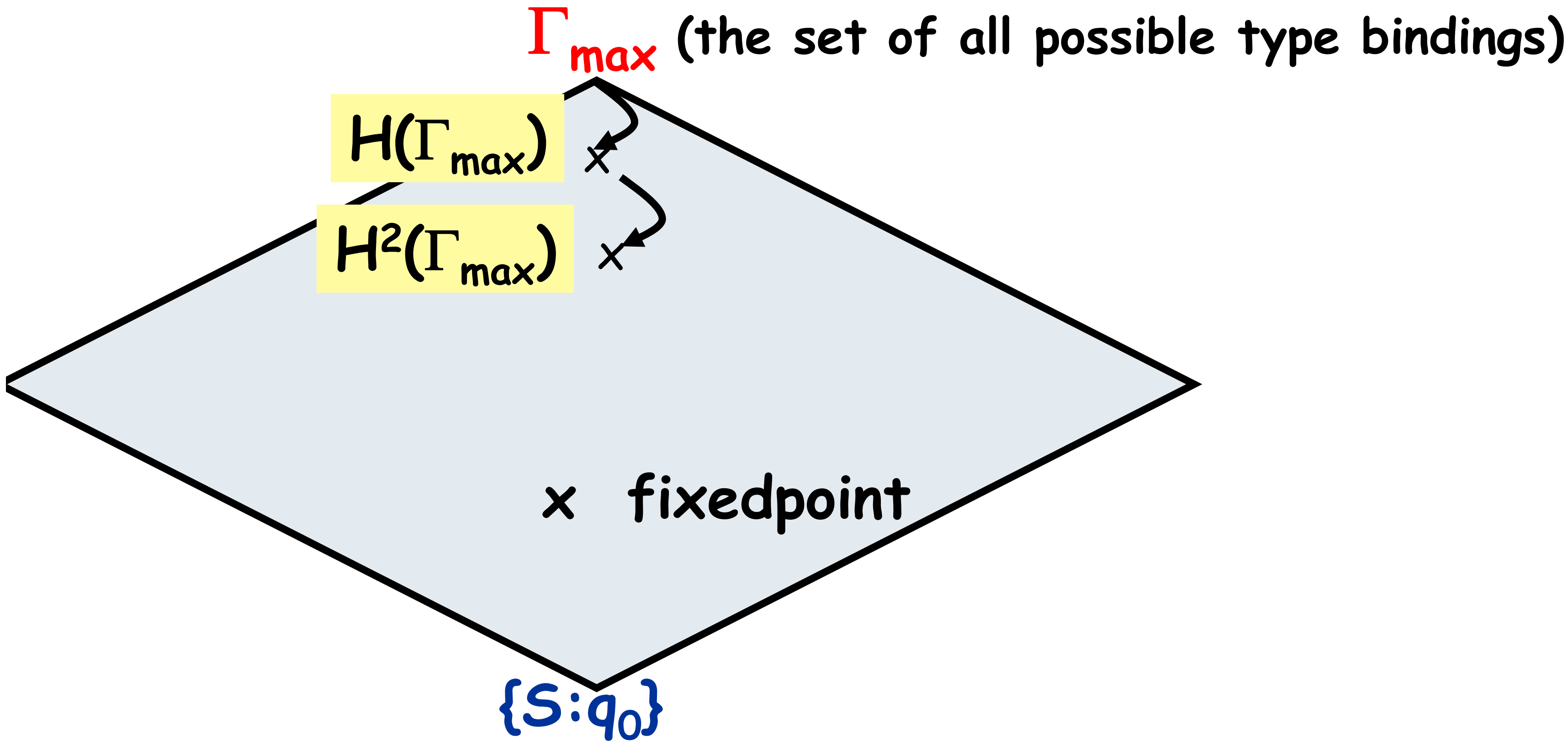
Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint Γ_{gfp} of H
   $(H(\Gamma) = \{ F_j:\tau \in \Gamma \mid \Gamma \vdash t_j:\tau \})$
2. Check whether $S:q_0 \in \Gamma_{gfp}$



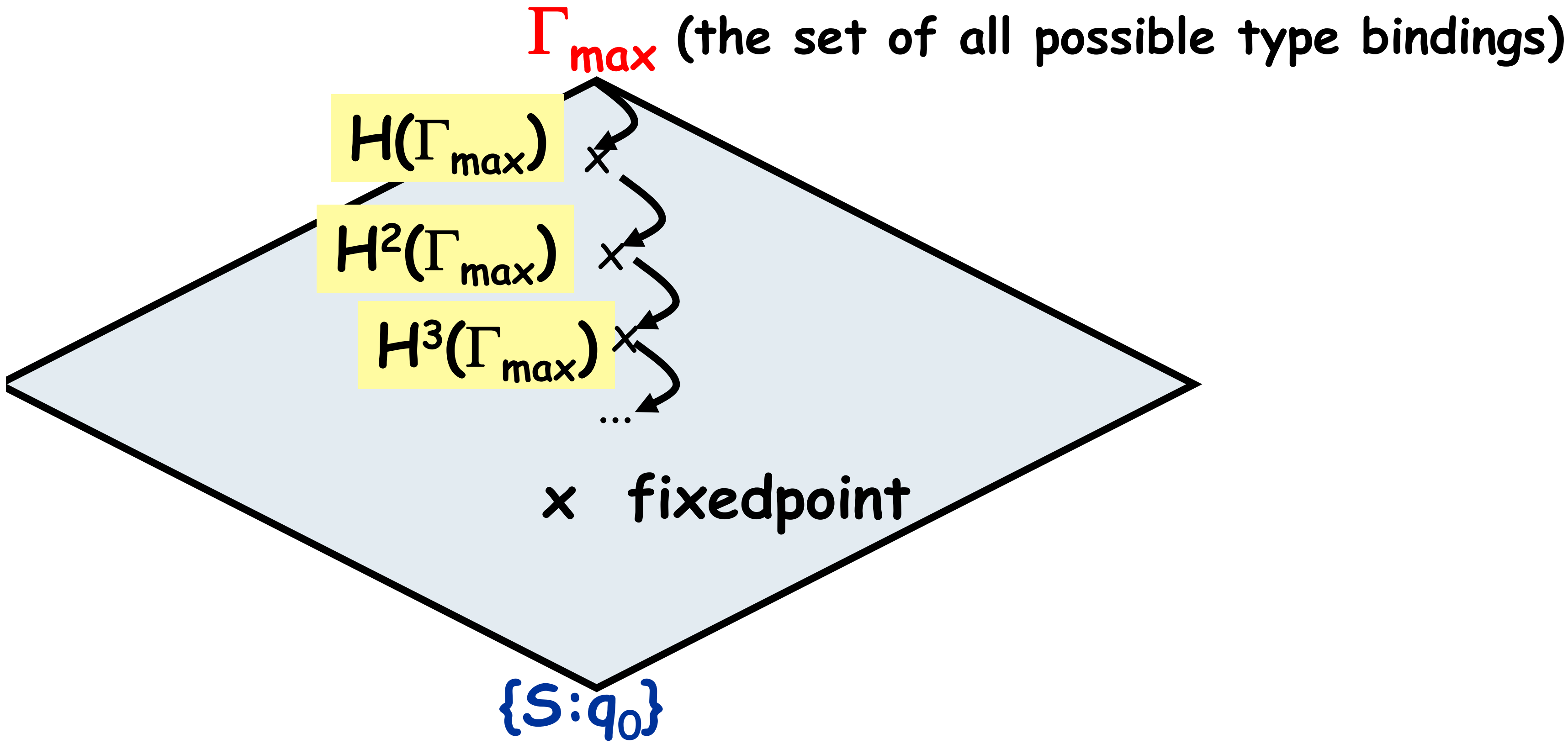
Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint Γ_{gfp} of H
   $(H(\Gamma) = \{ F_j{:}\tau \in \Gamma \mid \Gamma \vdash t_j{:}\tau \})$
2. Check whether $S{:}q_0 \in \Gamma_{gfp}$



Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint Γ_{gfp} of H
   $(H(\Gamma) = \{ F_j:\tau \in \Gamma \mid \Gamma \vdash t_j:\tau \})$
2. Check whether $S:q_0 \in \Gamma_{gfp}$



Example

- **Recursion scheme:**

  $S \to F\ c \qquad F \to \lambda x.a\ x\ (F\ (b\ x))$

  $(S{:}o,\ F{:}\ o \to o)$

- **Automaton:**

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$

  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$

$\Gamma_{max} = \{S{:}q_0,\ S{:}q_1,\ F{:}\ \top \to q_0,\ F{:}\ q_0 \to q_0,\ F{:}\ q_1 \to q_0,\ F{:}\ q_0 \wedge q_1 \to q_0,$
$F{:}\ \top \to q_1,\ F{:}\ q_0 \to q_1,\ F{:}\ q_1 \to q_1,\ F{:}\ q_0 \wedge q_1 \to q_1\}$

$$H(\Gamma_{max}) = \{\, S{:}\tau \in \Gamma_{max} \mid \Gamma_{max} \vdash F\ c{:}\tau \,\} \cup \{\, F{:}\tau \in \Gamma_{max} \mid \Gamma_{max} \vdash \lambda x.a\ x\ (F(b\ x)) : \tau \,\}$$

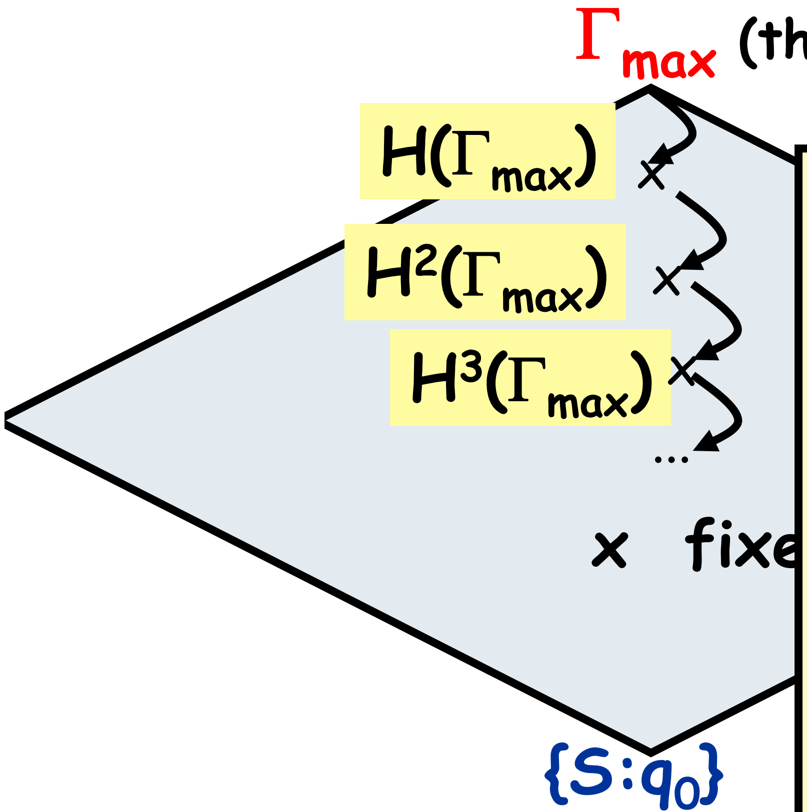
$$= \{S{:}q_0,\ S{:}q_1,\ F{:}\ q_0 \to q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\}$$

$$H^2(\Gamma_{max}) = \{S{:}q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\}$$

$$H^3(\Gamma_{max}) = \{S{:}q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\} = H^2(\Gamma_{max})$$

Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint Γ_{gfp} of H
   $(H(\Gamma) = \{ F_j{:}\tau \in \Gamma \mid \Gamma \vdash t_j{:}\tau \})$
2. Check whether $S{:}q_0 \in \Gamma_{gfp}$



Drawbacks:
- Huge cost for computing H
- Huge number of iterations

(both as huge as $|\Gamma_{max}|$ =

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

)

A: largest arity
Q: automaton size

How large is Γ_{max}?

Γ_{max}: the set of all possible type bindings for non-terminals

sort	# of types for each sort ($Q=\{q_0,q_1,q_2,q_3\}$)
o (trees)	4 (q_0,q_1,q_2,q_3)
$o \to o$	$2^4 \times 4 = 64$ ($\wedge S \to q$, with $S \in 2^Q$, $q \in Q$)
$(o \to o) \to o$	$2^{64} \times 4 = 2^{66}$
$((o \to o) \to o) \to o$	$2^{2^{66}} \times 4 > 10^{1000000000000000000}$

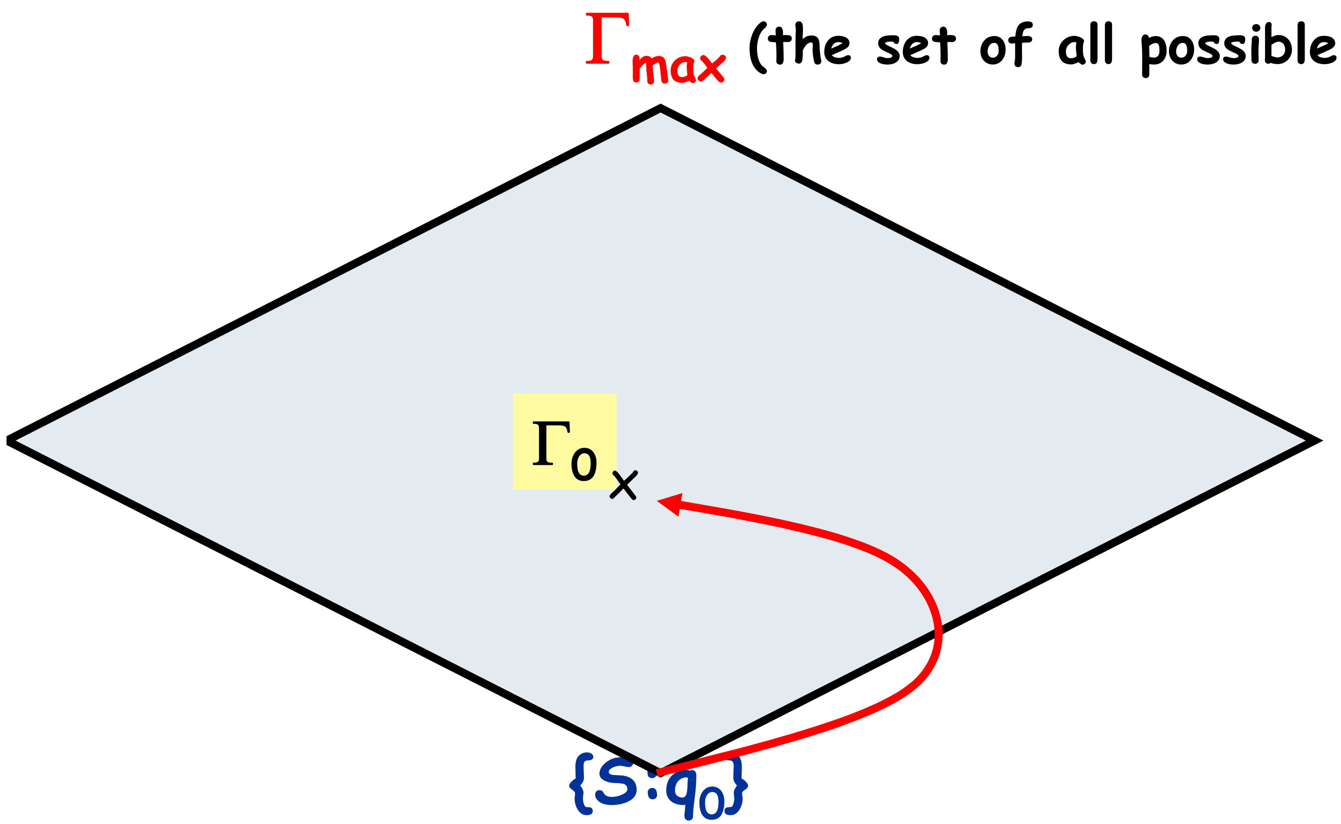
$$|\Gamma_{max}| = O(|G| \times \underbrace{2^{2^{\cdot^{\cdot^{2^{(A|Q|)^{1+\varepsilon}}}}}}}_{n})$$

Outline

- What is higher-order model checking?
- Applications
  - program verification:
    "software model checker for ML"
  - data compression
- Algorithms for higher-order model checking
  - from model checking to typing
  - practical algorithms
- Future directions

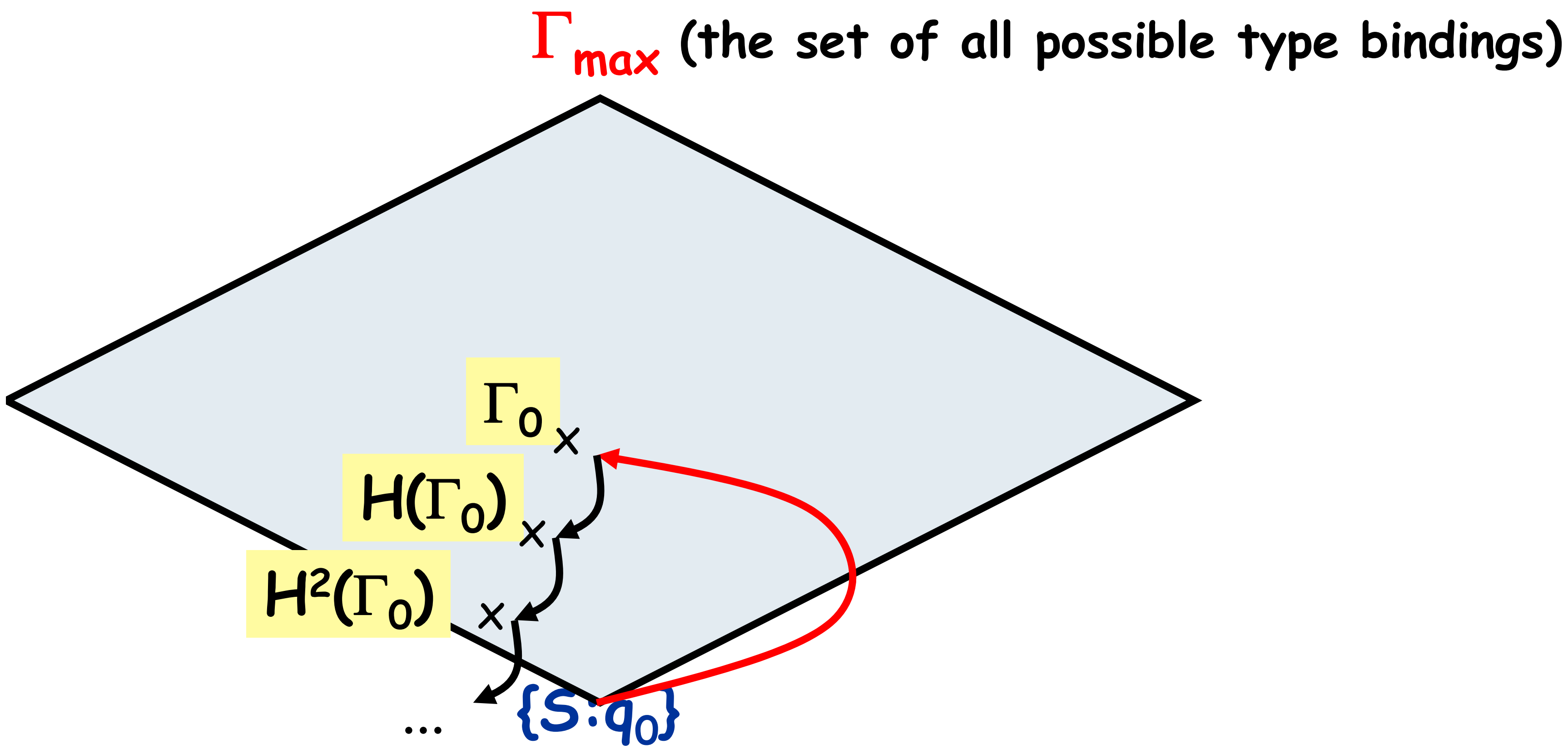
Practical Algorithms [K. PPDP09] [K.FoSSaCS11]

1. Guess a type environment Γ_0
2. Compute greatest fixedpoint Γ smaller than Γ_0
3. Check whether $S{:}q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.



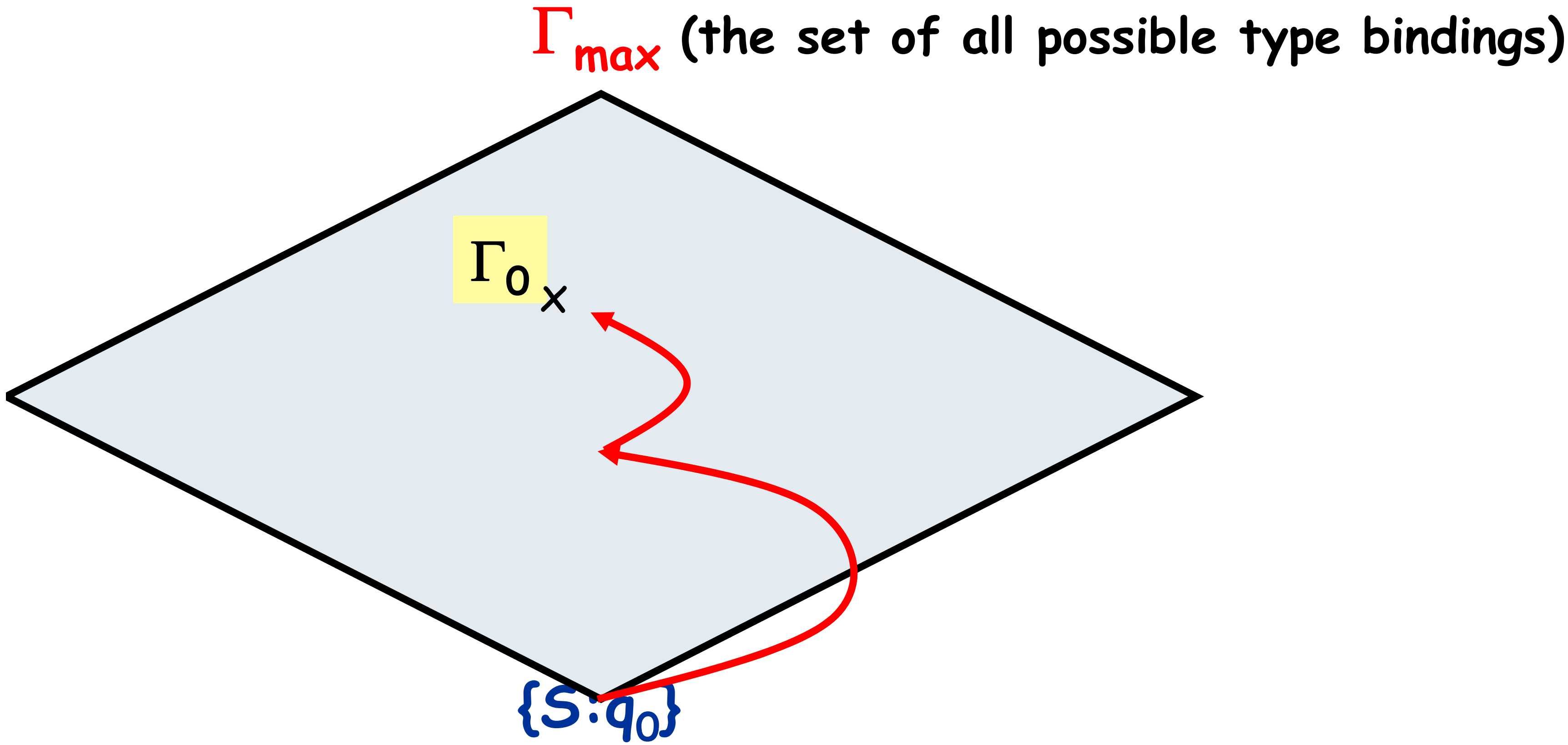
Practical Algorithms [K. PPDP09] [K.FoSSaCS11]

1. Guess a type environment Γ_0
2. Compute greatest fixedpoint Γ smaller than Γ_0
3. Check whether $S{:}q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.



Practical Algorithms [K. PPDP09] [K.FoSSaCS11]

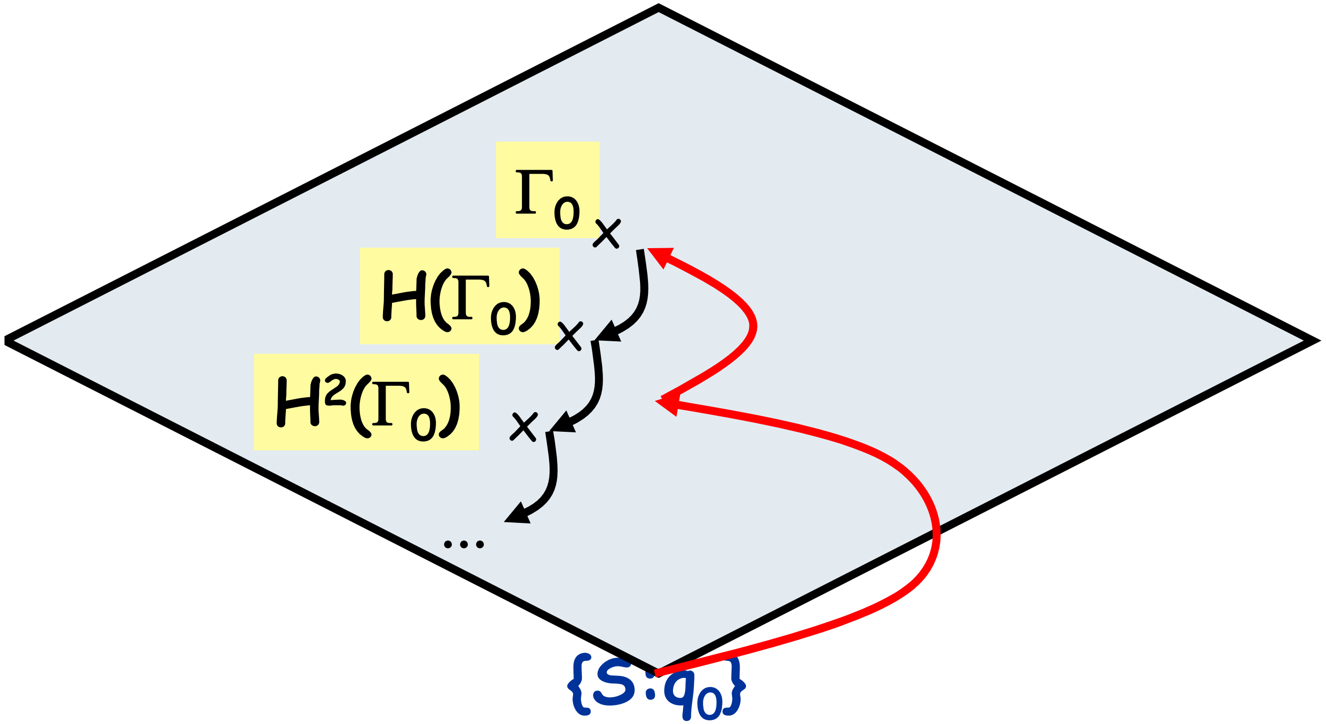
1. Guess a type environment Γ_0
2. Compute greatest fixedpoint Γ smaller than Γ_0
3. Check whether $S{:}q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.



Practical Algorithms [K. PPDP09] [K.FoSSaCS11]

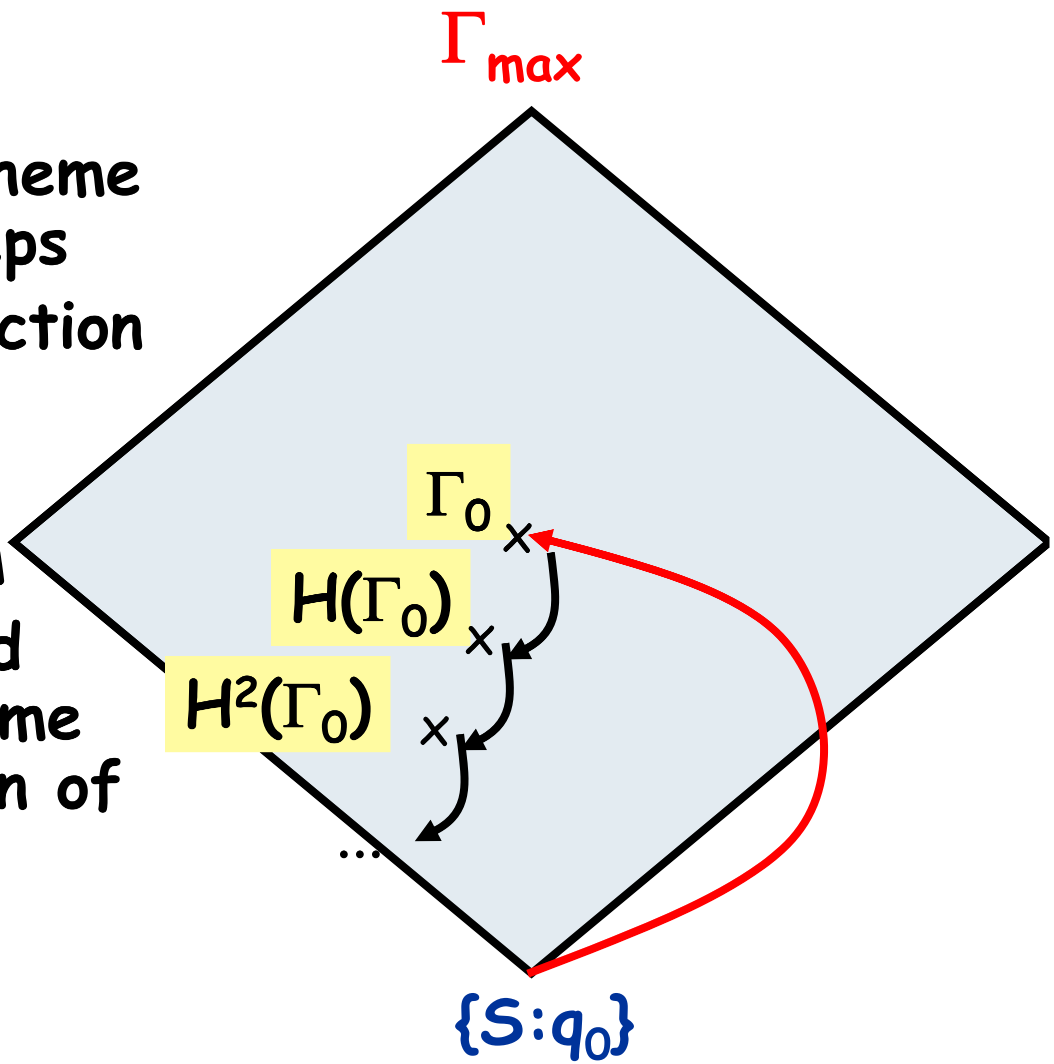
1. Guess a type environment Γ_0
2. Compute greatest fixedpoint Γ smaller than Γ_0
3. Check whether $S{:}q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.

Γ_{max} (the set of all possible type bindings)



How to guess Γ_0?

- PPDP09 algorithm
  - Reduce a recursion scheme a finite number of steps
  - Observe how each function is used and express it as types
- FoSSaCS11 algorithm
  - Like PPDP09, but avoid reductions by using game semantic interpretation of types



Example

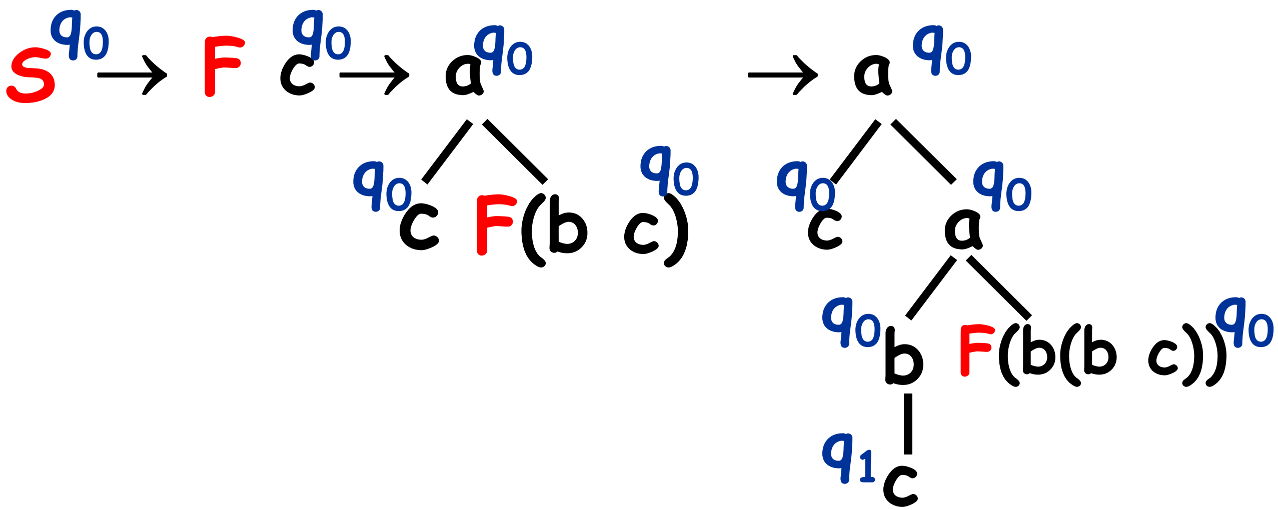
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0,\ a) = q_0\ q_0 \qquad \delta(q_0,\ b) = q_1$

  $\delta(q_0,\ c) = \delta(q_1,\ c) = \varepsilon$



Example

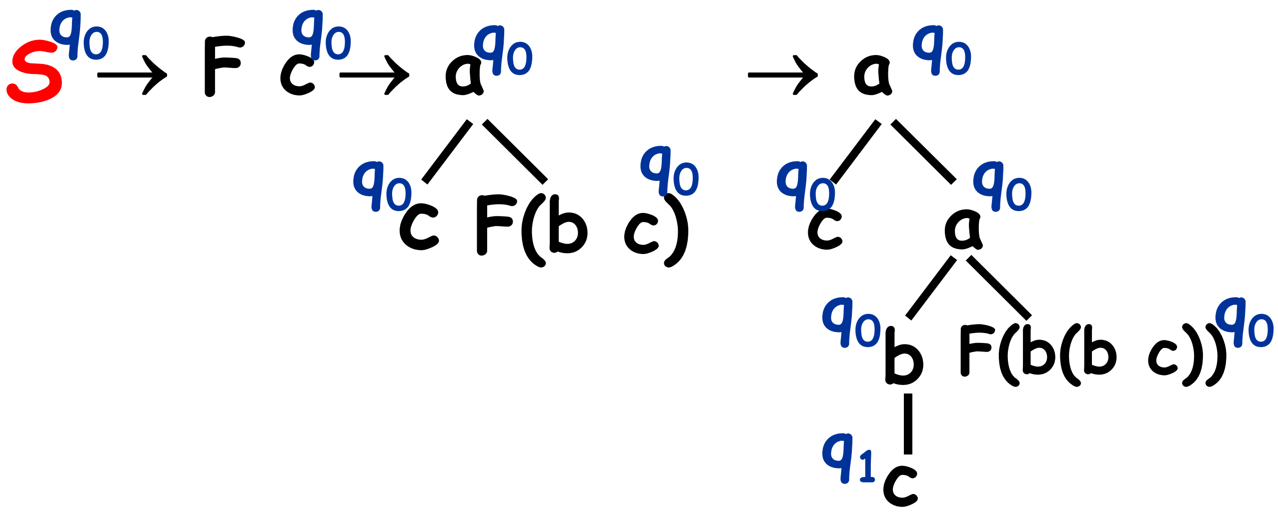
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0,\ a) = q_0\ q_0 \qquad \delta(q_0,\ b) = q_1$

  $\delta(q_0,\ c) = \delta(q_1,\ c) = \varepsilon$



Γ_0 :

S: q_0

Example

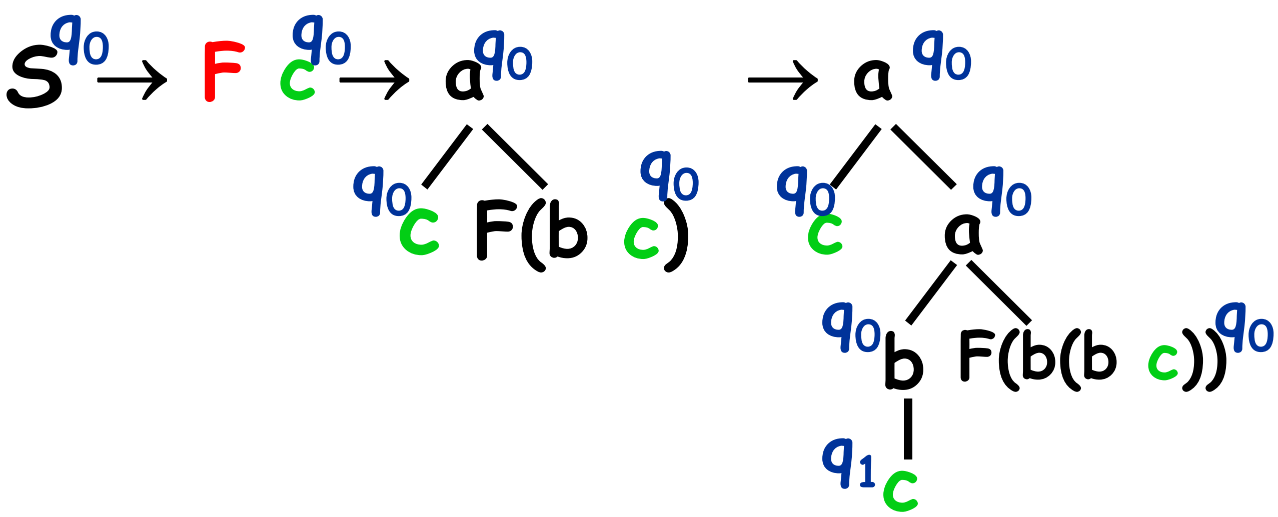
- **Recursion scheme:**

  $S \to F\ c \qquad F \to \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$

  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$



Γ_0 :

S: q_0

F: ? $\to q_0$

Example

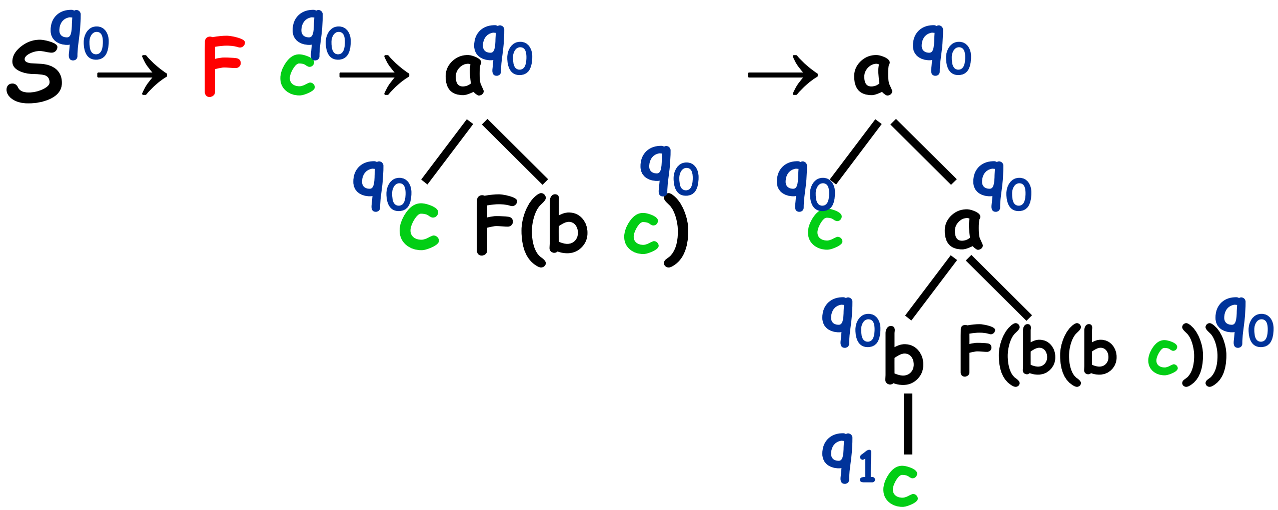
- **Recursion scheme:**

  $S \to F\ c \qquad F \to \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0,\ a) = q_0\ q_0 \qquad \delta(q_0,\ b) = q_1$

  $\delta(q_0,\ c) = \delta(q_1,\ c) = \varepsilon$



Γ_0 :

$S:\ q_0$

$F:\ q_0 \wedge q_1 \to q_0$

Example

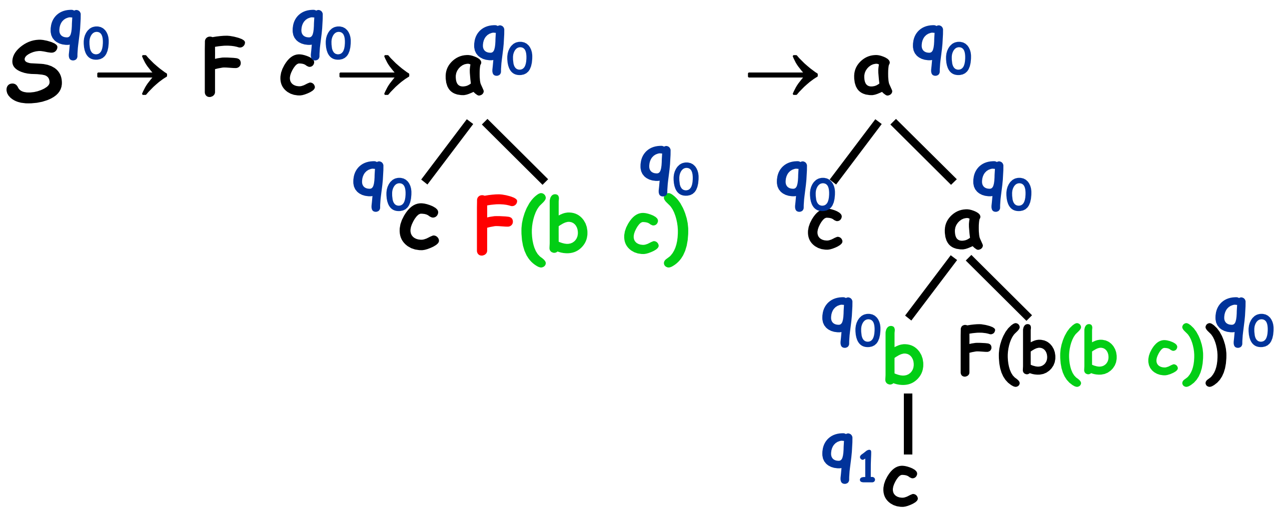
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0,\ a) = q_0\ q_0 \qquad \delta(q_0,\ b) = q_1$

  $\delta(q_0,\ c) = \delta(q_1,\ c) = \varepsilon$



Γ_0 :

S: q_0

F: $q_0 \wedge q_1 \rightarrow q_0$

F: $q_0 \rightarrow q_0$

Example

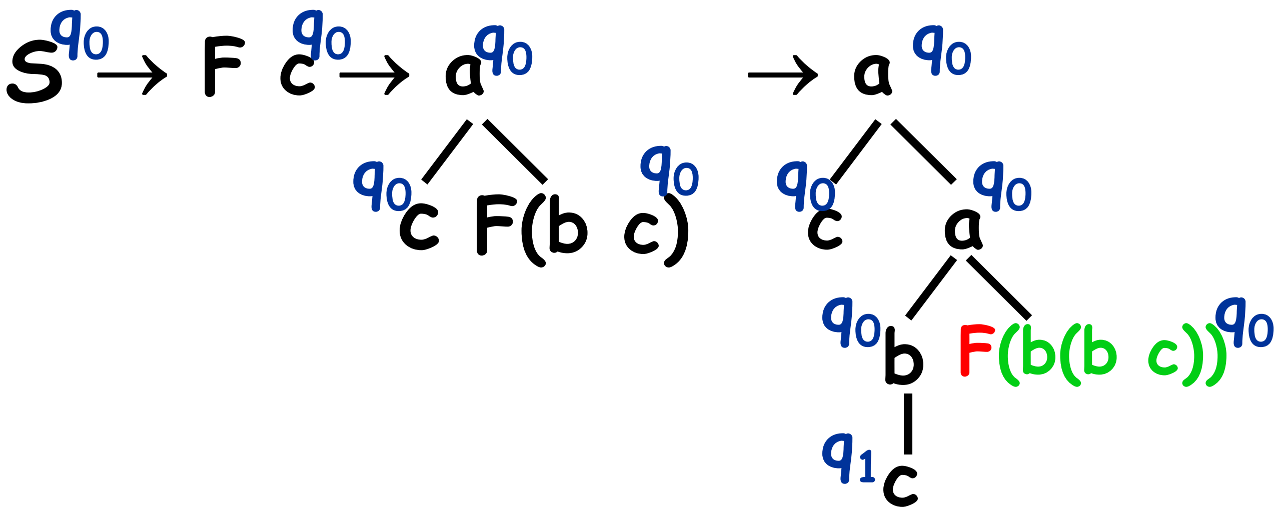
- **Recursion scheme:**

  $S \to F\ c \qquad F \to \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$

  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$



Γ_0 :

S: q_0

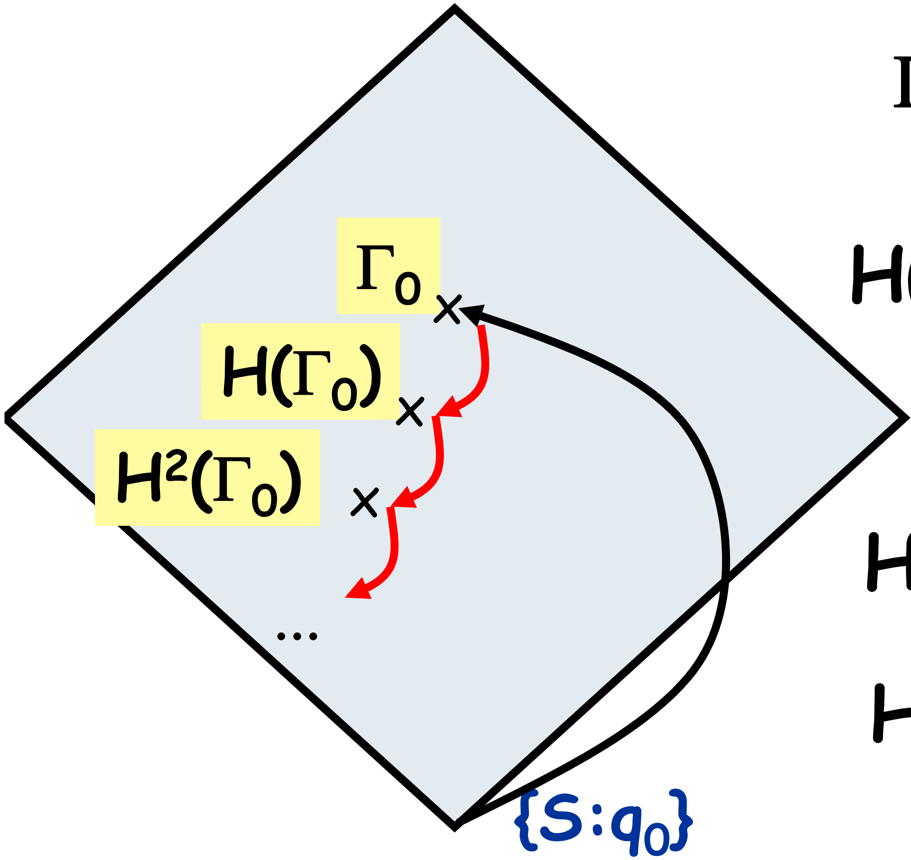
F: $q_0 \wedge q_1 \to q_0$

F: $q_0 \to q_0$

F: $\top \to q_0$

Practical Algorithms [K. PPDP09] [K.FoSSaCS11]

1. Guess a type environment Γ_0
2. Compute greatest fixedpoint Γ smaller than Γ_0
3. Check whether $S{:}q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.



$$\Gamma_0 = \{S{:}\ q_0,\ F{:}\ q_0 \wedge q_1 \to q_0,\ F{:}\ q_0 \to q_0\ ,\ F{:}\ \top \to q_0\}$$

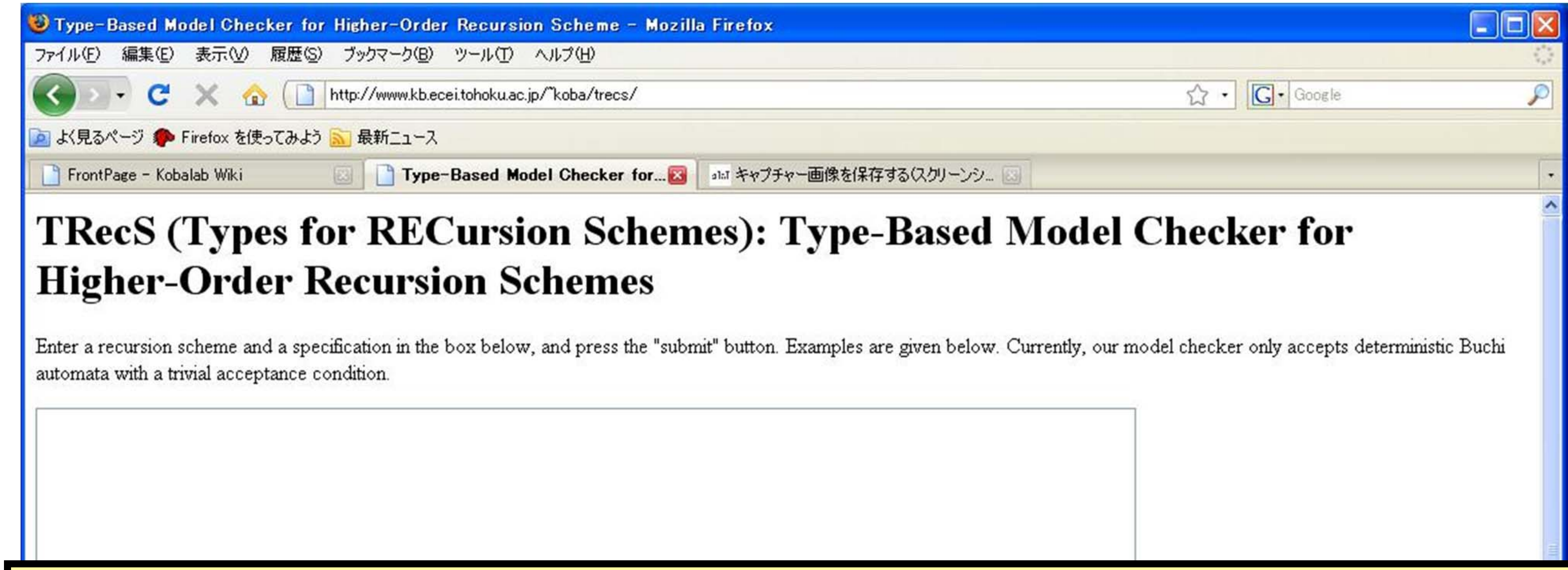
$$H(\Gamma_0) = \{\ F_k{:}\tau \in \Gamma_0 \mid \Gamma_0 \vdash t_k{:}\tau\ \}$$
$$= \{S{:}\ q_0,\ F{:}\ q_0 \wedge q_1 \to q_0,\ F{:}\ q_0 \to q_0\ \}$$

$$H^2(\Gamma_0) = \{S{:}\ q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\}$$

$$H^3(\Gamma_0) = \{S{:}\ q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\}$$

TRecS [K. PPDP09]

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



- ♦ The first model checker for recursion schemes
- ♦ Based on the PPDP09 algorithm, with certain additional optimizations

Experiments

	order	rules	states	result	Time (msec)
Twofiles	4				2
FileWrong	4				1
TwofilesE	4	12		Yes	2
FileOcamlC	4	23	4	Yes	5
Lock	4	11	3	Yes	10
Order5	5	9	4	Yes	2
mc91	4	49	1	Yes	50
xhtml	2	64	50	Yes	884

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
```

Algorithms for Higher-Order Model Checking: Summary

- Model checking can be reduced to type checking, which in turn becomes a fixedpoint problem
- Greatest fixedpoint is too costly to compute
- Practical algorithms *guess* a type environment and use it as a start point of fixedpoint computation
- FoSSaCS11 algorithm (for trivial automata model checking) is linear time in the size of grammar if other parameters (the size of types and automaton) are fixed

Outline

- What is higher-order model checking?
- Applications
  - program verification:
    “software model checker for ML”
  - data compression
- Algorithms for higher-order model checking
  - from model checking to typing
  - practical algorithms
- Discussions on FAQ and Future Directions

FAQ

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

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!

Does higher-order model checking scale?

Good News

+ Fixed-parameter PTIME in the grammar size (linear time for safety properties)

+ Use PPDP09 or FoSSaCS11 algorithm

+ Worst-case behavior shows an advantage of HO functions, rather than a disadvantage of HO model checking

Bad News

- n-EXPTIME complete

- Huge constant factor

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:

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

Exponential time algorithm for order-1

≈

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:

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

n-EXPTIME algorithm for order-n

$\approx$

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:

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

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

FAQ

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

Answer:
Don't know yet.
But there is a good hope it does!

Advantages of HO model checking for program verification

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

- no false alarms!
- no annotations

Advantages of HO model checking for program verification

(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 HO model checking for program verification

(3) Take the best of model checking and types

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

- Counterexample when verification fails
  $\Rightarrow$ error diagnosis,
  CEGAR (counterexample-guided abstraction refinement)

Advantages of HO model checking for program verification

(4) Encourages structured programming

Previous techniques:

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

Our technique:

- No loss of precision for higher-order functions and recursion
- Performance penalty? -- Not necessarily!

  If higher-order functions are properly used, there may be performance gain!

Remaining Challenges

- Refinement of HO model checkers
  - More efficiency
  - Support of full modal μ-calculus
- Software model checkers for full-scale programming languages
  - Refinement of predicate abstraction and CEGAR
  - Dealing with advanced types, references, etc.
- Extension of the decidability result?
  - Extension of models (recursion schemes)
  - Extension of properties
- Other applications (e.g. data compression)

Conclusion

- HO model checking problems can often be solved efficiently, despite the high worst-case complexity
(More justifications are needed, though.)
- Important and interesting applications:
  - automated program verification
  - data compression
- Only the first step from theory to practice; more efforts are required both in theoretical and practical communities