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 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 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 grammar)

\[ S \rightarrow a \ c \ B \]
\[ B \rightarrow b \ S \]

\[ S \rightarrow a \]
\[ C \rightarrow c \ B \]
\[ B \rightarrow b \ S \]
Higher-Order Recursion Scheme

Grammar for generating an infinite tree

Order-0 scheme  
(regular tree grammar)

\[ S \rightarrow a \quad c \quad B \]
\[ B \rightarrow b \quad S \]
Higher-Order Recursion Scheme

Grammar for generating an infinite tree

Order-0 scheme
(regular tree grammar)

\[ S \rightarrow a \ S \]
\[ S \rightarrow b \ B \]
\[ B \rightarrow b \ S \]
\[ B \rightarrow c \ B \]

\[ S \rightarrow a \ c \ B \]
\[ S \rightarrow b \ S \]
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

\[
S \rightarrow A \, c \\
A \rightarrow \lambda x. \, a \, x \, (A \, (b \, x))
\]

\[
S: \, o, \quad A: \, o \rightarrow o
\]

\[
S \rightarrow A \, c \rightarrow a \\
\quad \triangleleft \quad c \, A(b \, c)
\]
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

\[
S \rightarrow A \ c \rightarrow a \rightarrow a \\
\text{c \ A(b c) \ c} \rightarrow \text{a} \rightarrow \text{a} \\
\text{b \ A(b(b c))} \rightarrow \text{b} \rightarrow \text{c}
\]
Higher-Order Recursion Scheme

Grammar for generating an infinite tree

Order-1 scheme

\[
\begin{align*}
S & \rightarrow A \ c \\
A & \rightarrow \lambda x. \ a \ x \ (A \ (b \ x))
\end{align*}
\]

\[
S: \ o, \ A: \ o \rightarrow o
\]

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

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

\[ S: o, \ A: o \rightarrow o \]

Higher-order recursion schemes

\[ \approx \]

Call-by-name simply-typed \( \lambda \)-calculus

+ recursion, tree constructors
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 \( \text{Tree}(G) \)?

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

\[
\begin{align*}
S & \rightarrow A \ c \\
A & \rightarrow \lambda x. \ a \ x \ (A \ (b \ x))
\end{align*}
\]

\[S: \text{o, } A: \text{o} \rightarrow \text{o}\]

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 \( \mu \)-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]
(for order-\( n \) recursion scheme)

\[ \underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{p(x)} \]
(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)
```

Is the file "foo" accessed according to read* close?
let \( f(x) = \)
  
  \[
  \text{if } * \text{ then close(x) else read(x); } f(x) \]
  
  in

  let \( y = \text{open "foo"} \)
  
  in

  \( f(y) \)

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)

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)
```

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

\[
\begin{align*}
\text{let } f(x) &= \begin{cases} 
\text{close}(x) & \text{if } * \\
\text{read}(x); f(x) & \text{else}
\end{cases} \\
\text{in } f(y)
\end{align*}
\]

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

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

CPS Transformation!
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 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 $\lambda$-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]

- Higher-order functional program
  - Predicate abstraction
    - New predicates
      - Error path
        - Real error path?
          - Program is unsafe!
            - yes
              - Error path
                - property not satisfied
                  - property satisfied
                    - Program is safe!
    - Higher-order boolean program
      - Higher-order model checking
        - property not satisfied
## Comparison with Traditional Approach (Software Model Checking)

<table>
<thead>
<tr>
<th>Program Classes</th>
<th>Verification Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs with while-loops</td>
<td>Finite state model checking</td>
</tr>
<tr>
<td>Programs with 1\textsuperscript{st}-order recursion</td>
<td>Pushdown model checking</td>
</tr>
<tr>
<td>Higher-order functional programs</td>
<td>Higher-order model checking</td>
</tr>
</tbody>
</table>

\{ 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

\[ a(a(a(\ldots(a(e))\ldots))) \]

\[ 2^n \]

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

\[ S = \text{Twice(Twice(\ldots(Twice a)\ldots)) e} \]

\[ \text{Twice } f \ x = f(f(x)) \]

\[ n \]
Compressed Data as Recursion Schemes

\[ a(a(a(\ldots)(a(e))\ldots))) \]

\[ n \]

\[ 2 \]

\[ 2 \]

\[ \ldots \]

\[ 2 \]

compression

\[ S = ((\text{Twice Twice}) \ldots \text{Twice}) \ a \ e \]

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

\[ n \]
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:
\[ w_0 = b, \ w_1 = a, \ w_2 = w_1 w_0 = ab, \ w_3 = w_2 w_1 = aba, \ldots, \]
\[ w_n = w_{n-1} w_{n-2} \]

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

\[ S = \text{Twice}(\text{Twice}(\ldots(\text{Twice Next})\ldots)) \]
\[ \text{Fst} \ b \ a \ e \]
\[ \text{Next} \ k \ u \ v = k \ v \ (\text{Concat} \ v \ u) \]
\[ \text{Concat} \ f \ g \ x = f(g(x)) \]
\[ \text{Twice} \ f \ x = f(f(x)) \]

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

- **tree** $T = \text{Tree}(G)$
- **transducer** $f$
  - e.g. counting “ab”:
  - $a/\varepsilon$
  - $b/\varepsilon$
  - $b/1$
- **decompress** $f(T) = \text{Tree}(G')$

- **grammar** $G$
- **decompress** $T = \text{Tree}(G)$
- **grammar** $G'$
- **higher-order model checking** + $\alpha$

- **tree** $f(T)$
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]
    \[
    \underbrace{2 \times 2 \times \ldots \times 2}_{n \text{ times}}^{p(x)}
    \]
  - 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 \( \mu \)-calculus or MSO),

does \( A \) accept \( \text{Tree}(G) \)?

\text{n-EXPTIME-complete [Ong, LICS06] (for order-}\text{-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 \( \mu \)-calculus model checking)
Trivial tree automaton for infinite trees

\[ \delta(q_0, a) = q_0 \]
\[ \delta(q_0, b) = q_1 \]
\[ \delta(q_1, b) = q_1 \]
\[ \delta(q_0, c) = \varepsilon \]
\[ \delta(q_1, c) = \varepsilon \]

"a" does not occur below "b"
Trivial tree automaton
for infinite trees

\[ \delta(q_0, a) = q_0 q_0 \]
\[ \delta(q_0, b) = q_1 \]
\[ \delta(q_1, b) = q_1 \]
\[ \delta(q_0, c) = \varepsilon \]
\[ \delta(q_1, c) = \varepsilon \]

“a” does not occur below “b”
Trivial tree automaton for infinite trees

δ(q₀, a) = q₀ q₀
δ(q₀, b) = q₁
δ(q₁, b) = q₁
δ(q₀, c) = ε
δ(q₁, c) = ε

“a” does not occur below “b”
Trivial tree automaton for infinite trees

\[ \delta(q_0, a) = q_0 \]
\[ \delta(q_0, b) = q_1 \]
\[ \delta(q_0, c) = \varepsilon \]
\[ \delta(q_1, b) = q_1 \]
\[ \delta(q_1, c) = \varepsilon \]

"a" does not occur below "b"
Trivial tree automaton for infinite trees

\[ \delta(q_0, a) = q_0 \]
\[ \delta(q_0, b) = q_1 \]
\[ \delta(q_0, c) = \varepsilon \]
\[ \delta(q_1, b) = q_1 \]
\[ \delta(q_1, c) = \varepsilon \]

“a” does not occur below “b”
Trivial tree automaton for infinite trees

\[
\delta(q_0, a) = q_0 \quad \delta(q_0, b) = q_1 \\
\delta(q_0, c) = \varepsilon \\
\delta(q_1, b) = q_1 \\
\delta(q_1, c) = \varepsilon
\]

"a" does not occur below "b"
Types for Recursion Schemes

- Automaton state as the type of trees
  - $q$: trees accepted from state $q$
  - $q_1 \land q_2$: trees accepted from both $q_1$ and $q_2$

Is Tree($G$) accepted by $A$?

Does Tree($G$) have type $q_0$?
Types for Recursion Schemes

♦ Automaton state as the type of trees

- \( q_1 \rightarrow q_2 \): functions that take a tree of type \( q_1 \) and return a tree of \( q_2 \)
Types for Recursion Schemes

- Automaton state as the type of trees
  - $q_1 \land q_2 \rightarrow q_3$: functions that take a tree of type $q_1 \land q_2$ and return a tree of type $q_3$
Types for Recursion Schemes

♦ Automaton state as the type of trees

\[(q_1 \rightarrow q_2) \rightarrow q_3:\]

functions that take a function of type \(q_1 \rightarrow q_2\) and return a tree of type \(q_3\)
Typing

\[ \delta(q, a) = q_1 \ldots q_n \]

\[ \vdash a : q_1 \to \ldots \to q_n \to q \]

\[ \Gamma, x : \tau_1, \ldots, x : \tau_n \vdash t : \tau \]

\[ \Gamma \vdash \lambda x. t : \tau_1 \land \ldots \land \tau_n \to \tau \]

\[ \Gamma \vdash t_1 : \tau_1 \land \ldots \land \tau_n \to \tau \]

\[ \Gamma \vdash t_2 : \tau_i \ (i = 1, \ldots, n) \]

\[ \Gamma \vdash t_1 \ t_2 : \tau \]

\[ \Gamma \vdash t_k : \tau \ (\text{for every } F_k : \tau \in \Gamma) \]

\[ \vdash \{F_1 \to t_1, \ldots, F_n \to t_n\} : \Gamma \]
\[ \delta(q, a) = q_1 \ldots q_n \]
\[ \vdash a : q_1 \to \ldots \to q_n \to q \]
\[ \Gamma, x : \tau_1, \ldots, x : \tau_n \vdash t : \tau \]
\[ \vdash \lambda x. t : \tau_1 \land \ldots \land \tau_n \to \tau \]
\[ \Gamma \vdash t_2 : \tau_i \ (i=1, \ldots n) \]
\[ \Gamma \vdash t_1 \ t_2 : \tau \]
\[ \Gamma \vdash t_k : \tau \ (\text{for every } F_k : \tau \in \Gamma) \]
\[ \vdash \{ F_1 \to t_1, \ldots, F_n \to t_n \} : \Gamma \]
Typing

\[ \delta(q, a) = q_1...q_n \]

\[ \vdash a : q_1 \rightarrow ... \rightarrow q_n \rightarrow q \]

\[ \Gamma, x: \tau_1, ..., x: \tau_n \vdash t: \tau \]

\[ \vdash \lambda x.t : \tau_1 \wedge ... \wedge \tau_n \rightarrow \tau \]

\[ \Gamma \vdash t_1 : \tau_1 \wedge ... \wedge \tau_n \rightarrow \tau \]

\[ \Gamma \vdash t_2 : \tau_i (i=1,..n) \]

\[ \Gamma \vdash t_1 t_2 : \tau \]

\[ \Gamma \vdash t_k : \tau \text{ (for every } F_k : \tau \in \Gamma) \]

\[ \vdash \{F_1 \rightarrow t_1, ..., F_n \rightarrow t_n\} : \Gamma \]
Soundness and Completeness

\[ G = \{ F_1 \rightarrow t_1, \ldots, F_m \rightarrow t_m \} \text{ (with } S=F_1) \]
\[ A: \text{Trivial automaton with initial state } q_0 \]
\[ \text{TS}(A): \text{Intersection type system for } A \]

Tree(G) is accepted by A if and only if
S has type \( q_0 \) in TS(A),
i.e. \( \exists \Gamma. (S: q_0 \in \Gamma \land \vdash \{ F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n \} : \Gamma) \)
if and only if
\( \exists \Gamma. (S: q_0 \in \Gamma \land \forall (F_k: \tau) \in \Gamma. \Gamma \vdash t_k : \tau) \)
Soundness and Completeness

[K., POPL2009]

Tree(G) is accepted by A
if and only if
S has type q₀ in TS(A),
i.e. \( \exists \Gamma. (S: q₀ \in \Gamma \land \vdash \{ F₁ \to t₁, \ldots, Fₙ \to tₙ \}: \Gamma) \)
if and only if
\( \exists \Gamma. (S: q₀ \in \Gamma \land \forall (F_k : \tau) \in \Gamma. \Gamma \vdash t_k : \tau) \)
if and only if
\( \exists \Gamma. (S: q₀ \in \Gamma \land \Gamma = H(\Gamma)) \)
for \( H(\Gamma) = \{ F_k : \tau \in \Gamma \mid \Gamma \vdash t_k : \tau \} \)

Function to filter out invalid type bindings
Type checking (=model checking) problem

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

\( \Gamma_{\text{max}} \) (the set of all type bindings)
Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint $\Gamma_{gfp}$ of $H$
   \[ (H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma |- t_j : \tau \}) \]
2. Check whether \( S : q_0 \in \Gamma_{gfp} \)
Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint $\Gamma_{gfp}$ of $H$
   
   \[ H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma |- t_j : \tau \} \]

2. Check whether $S : q_0 \in \Gamma_{gfp}$

\[ \Gamma_{\text{max}} \] (the set of all possible type bindings)

\[ H(\Gamma_{\text{max}}) \]

\[ H^2(\Gamma_{\text{max}}) \]

\[ \{ S : q_0 \} \]
Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint \( \Gamma_{gfp} \) of \( H \)
   \( (H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma |- t_j : \tau \}) \)

2. Check whether \( S : q_0 \in \Gamma_{gfp} \)

\( \Gamma_{max} \) (the set of all possible type bindings)
Example

Recursion scheme:

\[ S \rightarrow F \quad F \rightarrow \lambda x. a \times (F (b \ x)) \]

\((S: o, F: o \rightarrow o)\)

Automaton:

\[ \delta(q_0, a) = q_0, q_0 \quad \delta(q_0, b) = q_1 \]

\[ \delta(q_0, c) = \delta(q_1, c) = \varepsilon \]

\[ \Gamma_{\text{max}} = \{ S:q_0, S:q_1, F: T \rightarrow q_0, F: q_0 \rightarrow q_0, F: q_1 \rightarrow q_0, F: q_0 \land q_1 \rightarrow q_0, \]

\[ F: T \rightarrow q_1, F: q_0 \rightarrow q_1, F: q_1 \rightarrow q_1, F: q_0 \land q_1 \rightarrow q_1 \} \]

\[ H(\Gamma_{\text{max}}) = \{ S: \tau \in \Gamma_{\text{max}} | \Gamma_{\text{max}} \vdash \neg F \quad c : \tau \} \]

\[ \cup \{ F: \tau \in \Gamma_{\text{max}} | \Gamma_{\text{max}} \vdash \neg \lambda x. a \times (F(b \ x)) : \tau \} \]

\[ = \{ S:q_0, S:q_1, F: q_0 \rightarrow q_0, F: q_0 \land q_1 \rightarrow q_0 \} \]

\[ H^2(\Gamma_{\text{max}}) = \{ S:q_0, F: q_0 \land q_1 \rightarrow q_0 \} \]

\[ H^3(\Gamma_{\text{max}}) = \{ S:q_0, F: q_0 \land q_1 \rightarrow q_0 \} = H^2(\Gamma_{\text{max}}) \]
Naive Algorithm [K. POPL09]

1. Compute the greatest fixedpoint $\Gamma_{\text{gfp}}$ of $H$
   
   $H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma \vdash t_j : \tau \}$

2. Check whether $S : q_0 \in \Gamma_{\text{gfp}}$

Drawbacks:

- Huge cost for computing $H$
- Huge number of iterations

(both as huge as $|\Gamma_{\text{max}}| = O(|G| \times (AQ)^{1+\epsilon})$)

$n$: number of iterations

$A$: largest arity

$Q$: automaton size
How large is $\Gamma_{\text{max}}$?

$\Gamma_{\text{max}}$: the set of all possible type bindings for non-terminals

<table>
<thead>
<tr>
<th>sort</th>
<th># of types for each sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>o (trees)</td>
<td>$4 \ (q_0,q_1,q_2,q_3)$</td>
</tr>
<tr>
<td>o → o</td>
<td>$2^4 \times 4 = 64 \ \ (\land S \rightarrow q, \text{ with } S \in 2^Q, q \in Q)$</td>
</tr>
<tr>
<td>(o→o) → o</td>
<td>$2^{64} \times 4 = 2^{66}$</td>
</tr>
<tr>
<td>(((o→o) → o) → o) → o</td>
<td>$2^{66} \times 4 &gt; 10$</td>
</tr>
</tbody>
</table>

\[ |\Gamma_{\text{max}}| = O(|G| \times \left( \prod_{i=1}^{n} \frac{2^{(A|Q|)^{1+\varepsilon}}}{2} \right) ) \]
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
1. Guess a type environment $\Gamma_0$
2. Compute greatest fixedpoint $\Gamma$ smaller than $\Gamma_0$
3. Check whether $S : q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.

$\Gamma_{\text{max}}$ (the set of all possible type bindings)
1. Guess a type environment $\Gamma_0$
2. Compute greatest fixedpoint $\Gamma$ smaller than $\Gamma_0$
3. Check whether $S:q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.

$\Gamma_{\text{max}}$ (the set of all possible type bindings)
1. Guess a type environment $\Gamma_0$
2. Compute greatest fixedpoint $\Gamma$ smaller than $\Gamma_0$
3. Check whether $S:q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.

$\Gamma_{\text{max}}$ (the set of all possible type bindings)
1. Guess a type environment $\Gamma_0$
2. Compute greatest fixedpoint $\Gamma$ smaller than $\Gamma_0$
3. Check whether $S:q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.
How to guess $\Gamma_0$?

♦ 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:

\[
S \rightarrow F \ c \quad F \rightarrow \lambda x. a \ x \ (F \ (b \ x))
\]

♦ Automaton:

\[
\begin{align*}
\delta(q_0, a) &= q_0 \ q_0 \\
\delta(q_0, b) &= q_1 \\
\delta(q_0, c) &= \delta(q_1, c) = \varepsilon
\end{align*}
\]
Example

Recursion scheme:

\[ S \rightarrow F \ c \quad F \rightarrow \lambda x. a \times (F \ (b \ x)) \]

Automaton:

\[ \delta(q_0, a) = q_0 \quad \delta(q_0, b) = q_1 \]
\[ \delta(q_0, c) = \delta(q_1, c) = \varepsilon \]

\[ S \rightarrow F \ c \rightarrow a \rightarrow a \]

\[ \Gamma_0: \]
\[ S: q_0 \]
Example

Recursion scheme:
\[ S \rightarrow F \, c \quad F \rightarrow \lambda x. a \times (F \,(b \, x)) \]

Automaton:
\[ \delta(q_0, a) = q_0 \quad \delta(q_0, b) = q_1 \]
\[ \delta(q_0, c) = \delta(q_1, c) = \varepsilon \]

\[ \Gamma_0 : \]
\[ S: q_0 \quad F: \ ? \rightarrow q_0 \]
Example

Recursion scheme:

\[ S \rightarrow F \, c \quad F \rightarrow \lambda x. a \times (F\, (b \, x)) \]

Automaton:

\[ \delta(q_0, a) = q_0 \quad \delta(q_0, b) = q_1 \]
\[ \delta(q_0, c) = \delta(q_1, c) = \varepsilon \]

\[ \Gamma_0 : \]
\[ S: q_0 \]
\[ F: q_0 \land q_1 \]
\[ \rightarrow q_0 \]
Example

♦ Recursion scheme:

\[ S \rightarrow F \ c \quad F \rightarrow \lambda x. a \ x \ (F \ (b \ x)) \]

♦ Automaton:

\[ \delta(q_0, a) = q_0 \quad q_0 \quad \delta(q_0, b) = q_1 \]
\[ \delta(q_0, c) = \delta(q_1, c) = \varepsilon \]

\[ S^{q_0} \rightarrow F \ c^{q_0} \rightarrow a^{q_0} \rightarrow a^{q_0} \]

\[ \Gamma_0 : \]

\[ S : q_0 \]
\[ F : q_0 \land q_1 \rightarrow q_0 \]

\[ F : q_0 \rightarrow q_0 \]
Example

- Recursion scheme:
  \[ S \rightarrow F \, c \quad F \rightarrow \lambda x. a \times (F \, (b \, x)) \]

- Automaton:
  \[ \delta(q_0, a) = q_0 \quad \delta(q_0, b) = q_1 \]
  \[ \delta(q_0, c) = \delta(q_1, c) = \varepsilon \]

\[
\Gamma_0:
\begin{align*}
S &: q_0 \\
F &: q_0 \land q_1 \\
F &: q_0 \rightarrow q_0 \\
F &: T \rightarrow q_0
\end{align*}
\]
1. Guess a type environment $\Gamma_0$
2. Compute greatest fixedpoint $\Gamma$ smaller than $\Gamma_0$
3. Check whether $S:q_0 \in \Gamma$
4. Repeat 1-3 until the property is proved or refuted.

\[ \Gamma_0 = \{ S: q_0, F: q_0 \land q_1 \to q_0, F: q_0 \to q_0, F: T \to q_0 \} \]

\[ H(\Gamma_0) = \{ F_k: \tau \in \Gamma_0 | \Gamma_0 \vdash t_k: \tau \} \]
\[ = \{ S: q_0, F: q_0 \land q_1 \to q_0, F: q_0 \to q_0 \} \]

\[ H^2(\Gamma_0) = \{ S: q_0, F: q_0 \land q_1 \to q_0 \} \]

\[ H^3(\Gamma_0) = \{ S: q_0, F: q_0 \land q_1 \to q_0 \} \]
TRecS [K. PPDP09]
http://www.kb.ecei.tohoku.ac.jp/~koba/treces/

The first model checker for recursion schemes

Based on the PPDP09 algorithm, with certain additional optimizations
### Experiments

<table>
<thead>
<tr>
<th></th>
<th>order</th>
<th>rules</th>
<th>states</th>
<th>result</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twofiles</td>
<td>4</td>
<td></td>
<td></td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>FileWrong</td>
<td>4</td>
<td></td>
<td></td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>TwofilesE</td>
<td>4</td>
<td>12</td>
<td>3</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>FileOcamlC</td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>Yes</td>
<td>5</td>
</tr>
<tr>
<td>Lock</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>Yes</td>
<td>10</td>
</tr>
<tr>
<td>Order5</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>mc91</td>
<td>4</td>
<td>49</td>
<td>1</td>
<td>Yes</td>
<td>50</td>
</tr>
<tr>
<td>xhtml</td>
<td>2</td>
<td>64</td>
<td>50</td>
<td>Yes</td>
<td>884</td>
</tr>
</tbody>
</table>

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

Taken from the compiler of Objective Caml, consisting of about 60 lines of O’Caml code.
(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)
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 \( \text{PTIME} \) in the grammar size
  (linear time for safety properties)
+ Use PPDPO9 or FoSSaCS11 algorithm
+ Worst-case behavior shows an advantage of HO functions, rather than a disadvantage of HO model checking

**Bad News**
- \( \text{n-EXPTIME} \) complete
- Huge constant factor
Recursion schemes generating $a^{2m}c$

Order-1:

\[
S \rightarrow F_1 c, \quad F_1 x \rightarrow F_2(F_2 x), \ldots, \quad F_m x \rightarrow a(a x)
\]

Order-0:

\[
S \rightarrow a G_1, \quad G_1 \rightarrow a G_2, \ldots, \quad G_k \rightarrow c \quad (k=2^m)
\]

Exponential time algorithm for order-1

≈

Polynomial time algorithm for order-0
Recursion schemes generating $a^{2^m}c$

Order-1:
$S \rightarrow F_1 c, \ F_1 x \rightarrow F_2(F_2 x), \ldots, \ F_m x \rightarrow a(a x)$

Order-0:
$S \rightarrow a \ G_1, \ G_1 \rightarrow a \ G_2, \ldots, \ G_k \rightarrow c \ (k=2^m)$

$n$-EXPTIME algorithm for order-$n$
$\approx$
Polynomial time algorithm for order-0
Recursion schemes generating $a^{2^m} c$

**Order-1:**

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

**Order-0:**

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

(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 \(\approx\) finite state systems
- Order-1 rec. schemes \(\approx\) 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
  ⇒ applications to PCC (proof-carrying code)

- **Counterexample** when verification fails
  ⇒ 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 $\mu$-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