Higher-Order Model Checking  
and Applications to Program Verification

Naoki Kobayashi  
Tohoku University

In collaboration with  
Luke Ong (University of Oxford),  
Ryosuke Sato, Naoshi Tabuchi, Takeshi Tsukada, Hiroshi Unno  
(Tohoku University)
Program Verification Techniques

♦ Finite state/pushdown model checking
  - Applicable to first-order procedures (pushdown model checking), but not to higher-order programs

♦ Type-based program analysis
  - Applicable to higher-order programs
  - Sound but imprecise

♦ Dependent types/theorem proving
  - Requires human intervention

Sound and precise verification technique for higher-order programs (e.g. ML/Java programs)?
This Talk

♦ New program verification method based on **higher-order model checking**

- Sound, **complete**, and automatic for
  - A large class of higher-order programs
  - A large class of verification problems

- Built on recent/new advances in
  - Type theories
  - Automata/formal language theories
    (esp. **higher-order recursion schemes**)
  - Model checking
Outline

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Ongoing work
♦ Discussion
Higher-Order Recursion Scheme

*Grammar for generating an infinite tree*

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

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

\[ S \rightarrow a \]
\[ a \]
\[ a \]
\[ a \]
\[ a \]
\[ a \]

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

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

Grammar for generating an infinite tree

Order-1 scheme

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

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

\[ S \rightarrow A \ c \rightarrow a \rightarrow a \rightarrow \ldots \rightarrow \]

\[ \begin{array}{c}
\text{c} \\
A(b \ c) \\
\text{c}
\end{array} \rightarrow \\
\begin{array}{c}
\text{c} \\
A(b(b \ c)) \\
\text{c}
\end{array} \rightarrow \\
\begin{array}{c}
\text{c} \\
\text{b} \\
A(b(b(b \ c))) \\
\text{c}
\end{array} \rightarrow \\
\begin{array}{c}
\text{c} \\
\text{b} \\
A(b(b(b(b \ c)))) \\
\text{c}
\end{array} \rightarrow \\
\ldots \]
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 eventually whenever “b” occurs?

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

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Ongoing work
♦ Discussion
From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

Higher-order program + specification \rightarrow Program Transformation

Rec. scheme (describing all event sequences or outputs) + Tree automaton, recognizing valid event sequences or outputs \rightarrow Model Checking
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

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

\[
\begin{align*}
F \times k &\rightarrow + (c \ k) (r(F \times k)) \\
S &\rightarrow F \ d \ \star
\end{align*}
\]

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

\[
\text{let } f(x) = \begin{cases} 
close(x) & \text{if } \ast \\
read(x); f(x) & \text{else}
\end{cases} \\
in 
let y = \text{open "foo"} \\
in 
f(y)
\]

\[
F \times k \rightarrow + (c \ k) (r(F \times k))
\]

\[
S \rightarrow F \ d \star
\]

\[
\text{CPS Transformation!}
\]

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

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

\[
\begin{align*}
F \times k &\rightarrow + (c \ k) (r(F \times k)) \\
S &\rightarrow F d \star
\end{align*}
\]

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)
```

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);
 in
  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

```plaintext
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

```
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 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
- A large class of verification problems:
  resource usage verification [Igarashi&K. POPL2002], reachability, flow analysis, …
Comparison with Traditional Approach (Control Flow Analysis)

♦ Control flow analysis

Higher-order program → Flow Analysis
Control flow graph (finite state or pushdown machines) → verification

♦ Our approach

Higher-order program → Program Transformation
Recursion scheme → verification

Only information about infinite data domains is approximated!
## Comparison with Traditional Approach

(Software Model Checking)

<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>Recursion scheme model checking</td>
</tr>
</tbody>
</table>
Outline

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Ongoing work
♦ Discussion
Goal

Construct a type system $TS(A)$ s.t.

- Tree($G$) is accepted by tree automaton $A$
- if and only if
- $G$ is typable in $TS(A)$

Model Checking as Type Checking  
(c.f. [Naik & Palsberg, ESOP2005])
Why Type-Theoretic Characterization?

♦ **Simpler** decidability proof of model checking recursion schemes
  - Previous proofs [Ong, 2006][Hague et. al, 2008] made heavy use of game semantics

♦ **More efficient** model checking algorithm
  - Known algorithms [Ong, 2006][Hague et. al, 2008] always require n-EXPTIME
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 $\text{Tree}(G)$?

$n$-EXPTIME-complete [Ong, LICS06]
(for order-$n$ recursion scheme)
Model Checking Problem

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 $\text{Tree}(G)$?

See [K.&Ong, LICS09] for the general case (full modal $\mu$-calculus model checking)
(Trivial) tree automaton for infinite trees

In every path, "a" cannot occur after "b"

\[ \delta(q_0, a) = q_0 \quad \delta(q_0, b) = q_1 \]
\[ \delta(q_1, b) = q_1 \quad \delta(q_0, c) = \varepsilon \]
\[ \delta(q_1, c) = \varepsilon \]
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 $\text{Tree}(G)$ accepted by $A$?

Does $\text{Tree}(G)$ 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 \(q_3\)
\[ \delta(q, a) = q_1 \ldots q_n \]

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

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

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

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

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

\[ \vdash \{ F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n \} : \Gamma \]
Soundness and Completeness
[K., POPL2009]

Let

$G$: Rec. scheme with initial non-terminal $S$
$A$: Trivial automaton with initial state $q_0$
$TS(A)$: Intersection type system derived from $A$

Then,

$Tree(G)$ is accepted by $A$
if and only if
$S$ has type $q_0$ in $TS(A)$
Outline

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
  - A naive algorithm
  - A practical algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Ongoing work
♦ Discussion
Typing

\[ \delta(q, a) = q_1 \ldots q_n \]
\[ \vdash a : q_1 \rightarrow \cdots \rightarrow q_n \rightarrow q \]

\[ \Gamma, \mathbf{x}: \tau_1, \ldots, \mathbf{x}: \tau_n \vdash t: \tau \]
\[ \Gamma \vdash \lambda \mathbf{x}.t: \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau \]

\[ \Gamma \vdash t_1: \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau \]
\[ \Gamma \vdash t_2: \tau_i (i=1, \ldots n) \]
\[ \Gamma \vdash t_1 \ t_2: \tau \]

\[ \Gamma \vdash t_j : \tau \text{ (for every } F_j: \tau \in \Gamma) \]
\[ \vdash \{ F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n \} : \Gamma \]
Naïve Type Checking Algorithm

S has type \( q_0 \)

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

(i) \( \Gamma \vdash t_j : \tau \)
for each \( F_j : \tau \in \Gamma \)
(ii) \( S : q_0 \in \Gamma \)
for some \( \Gamma \)

\[ S : q_0 \in \text{gfp}(H) = \bigcap_k H^k(\Gamma_{\text{max}}) \]
where
\[ H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma \vdash t_j : \tau \} \]
\[ \Gamma_{\text{max}} = \{ F : \tau \mid \tau :: \text{sort}(F) \} \]

Filter out invalid type bindings

All the possible type bindings
E.g. for \( F : o \rightarrow o \),
\( \{ F : T \rightarrow q_0, F : q_0 \rightarrow q_0, F : q_1 \rightarrow q_0, F : q_0 \land q_1 \rightarrow q_0, \ldots \} \)
Naïve Algorithm Does NOT Work

S has type $q_0$

\[ S : q_0 \in \text{gfp}(H) = \bigcap_k H^k(\Gamma_{\text{max}}) \]

where \( H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma |- t_j : \tau \} \)

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

This is huge!

<table>
<thead>
<tr>
<th>sort</th>
<th># of types ($Q={q_0, q_1, q_2, q_3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>4 ($q_0, q_1, q_2, q_3$)</td>
</tr>
<tr>
<td>o $\rightarrow$ o</td>
<td>$2^4 \times 4 = 64$ ((\wedge S \rightarrow q), with (S \in 2^Q), (q \in Q))</td>
</tr>
<tr>
<td>(o $\rightarrow$ o) $\rightarrow$ o</td>
<td>$2^{64} \times 4 = 2^{66}$</td>
</tr>
<tr>
<td>((o $\rightarrow$ o) $\rightarrow$ o) $\rightarrow$ o</td>
<td>$2^{2^{66}}$</td>
</tr>
<tr>
<td></td>
<td>1000000000000000000000000000000000</td>
</tr>
</tbody>
</table>
Outline

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
  - A naive algorithm
  - A practical algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Ongoing work
♦ Discussion
More Efficient Algorithm?

S has type $q_0$

$\iff S:q_0 \in \bigcap_k H^k(\Gamma) \cap \kappa$ $\Gamma_0$

where

$H(\Gamma) = \{ F_\tau : \tau \in \Gamma \mid \Gamma \vdash t_\tau : \tau \}$

Challenges:

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

Reduce the recursion scheme (finitely many steps), and extract type information

(ii) How can we guarantee completeness?

Iteratively repeat (i) and type checking
Hybrid Type Checking Algorithm

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

Property violated? yes Error path

Property violated? no

Step 2: Extract type environment $\Gamma_0$

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

$S:q0 \in \Gamma$?

Property Is Satisfied!
Soundness and Completeness of the Hybrid Algorithm

Given:
- Recursion scheme $G$
- Deterministic trivial automaton $A$, the algorithm eventually terminates, and:
  (i) outputs an error path if $\text{Tree}(G)$ is not accepted by $A$
  (ii) outputs a type environment if $\text{Tree}(G)$ is accepted by $A$
Example

Recursion scheme:

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

Automaton:

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

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

\[ \quad \quad c^{q_0} \ F(b \ c)^{q_0} \quad c^{q_0} \ a^{q_0} \]

\[ b \ F(b(b \ c))^{q_0} \]

\[ q_0 \]

\[ q_1 \]

\[ c \]
Example

♦ Recursion scheme:

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

♦ Automaton:

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

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

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

Recursion scheme:

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

Automaton:

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

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

♦ Recursion scheme:

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

♦ Automaton:

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

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

\[ \Gamma_0 : \]

\[ S: q_0 \]
\[ F: q_0 \wedge q_1 \rightarrow q_0 \]
\[ F: q_0 \rightarrow q_0 \]
Example

Recursion scheme:

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

Automaton:

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

\[
\begin{array}{c}
S \to F \ c \to a \\
q_0 q_0 a \\
q_0 c F(b \ c) \\
q_0 q_0 a \\
q_0 c a F(b(b \ c)) \\
q_0 q_1 c \\
q_0 q_0 q_0 \\
q_0 q_0 q_0 \\
q_0 q_0 q_0
\end{array}
\]

\[ \Gamma_0 : \]

\[ S : q_0 \]
\[ F : q_0 \land q_1 \to q_0 \]
\[ F : q_0 \to q_0 \]
\[ F : T \to q_0 \]
Example

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

Property violated? yes no

Step 2: Extract type environment

\[ \Gamma_0 \]

Step 3: Compute

\[ \Gamma = \bigcap_k H^k(\Gamma_0) \]

\[ S: q_0 \in \Gamma ? \]

no yes Property Is Satisfied!

S: \( q_0 \)
F: \( q_0 \land q_1 \rightarrow q_0 \)
F: \( q_0 \rightarrow q_0 \)
F: \( T \rightarrow q_0 \)
Example:
Filtering out invalid judgments

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

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

\[ \Gamma_0 = \{ S: q_0, \ F: q_0 \land q_1 \rightarrow q_0, \ F: q_0 \rightarrow q_0, \ F: T \rightarrow q_0 \} \]
\[ \Gamma_1 = H(\Gamma_0) = \{ F_k : \tau \in \Gamma_0 | \Gamma_0 \vdash t_k : \tau \} \]
\[ = \{ S: q_0, \ F: q_0 \land q_1 \rightarrow q_0, \ F: q_0 \rightarrow q_0 \} \]
\[ \Gamma_2 = \{ S: q_0, \ F: q_0 \land q_1 \rightarrow q_0 \} \]
\[ \Gamma_3 = \{ S: q_0, \ F: q_0 \land q_1 \rightarrow q_0 \} \]
Example

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

Property violated?

yes

Error path

no

Step 2: Extract type environment

\( \Gamma_0 \)

Step 3: Compute

\[ \Gamma = \bigcap_k H^k(\Gamma_0) \]

Property Is Satisfied!

yes

\( S: q_0 \in \Gamma ? \)

no

\( S: q_0 \)

\( F: q_0 \land q_1 \rightarrow q_0 \)

\( S: q_0 \quad F: q_0 \rightarrow q_0 \quad F: T \rightarrow q_0 \)
Example

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

Property violated?

yes

Error path

no

Step 2: Extract type environment

\[ \Gamma_0 \]

Step 3: Compute

\[ \Gamma = \bigcap_k H^k(\Gamma_0) \]

Property Is Satisfied!

\[ S: q_0 \in \Gamma ? \]

yes

\[ S: q_0 \]

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

no

\[ S: q_0 \]

\[ F: q_0 \rightarrow q_0 \]

\[ F: T \rightarrow q_0 \]
TRecS
http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/

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

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

- The first model checker for recursion schemes (or, for higher-order functions)
- Based on the hybrid model checking algorithm, with certain additional optimizations
## Experiments

<table>
<thead>
<tr>
<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></td>
</tr>
<tr>
<td>FileWrong</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TwofilesE</td>
<td>4</td>
<td>12</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>FileOcamC</strong></td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>Lock</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>Order5</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>m91</td>
<td>2</td>
<td>280</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>xhtml</td>
<td>1</td>
<td>2</td>
<td>50</td>
<td>Yes</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
## Experiments

<table>
<thead>
<tr>
<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>11</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>FileWrong</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>TwofilesE</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>FileOcamlC</td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>m91</td>
<td>2</td>
<td>280</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>xhtml</td>
<td>1</td>
<td>2</td>
<td>50</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

*Machine-generated code from McCurthy’s 91 function using predicate abstraction*

*Machine-generated code from a program manipulating Xhtml documents*
Outline

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Limitations and ongoing work
♦ Discussion
Recursion schemes as models of higher-order programs?

+ simply-typed $\lambda$-calculus
+ recursion
+ tree constructors
+ finite data domains (via Church encoding; $\text{true} = \lambda x.\lambda y.x$, $\text{false} = \lambda x.\lambda y.y$)
- infinite data domains (integers, lists, trees,...)
- advanced types (polymorphism, recursive types, object types, ...)
- imperative features/concurrency
Ongoing work
to overcome the limitation

♦ Predicate abstraction and CEGAR,
to deal with infinite data domains
(c.f. BLAST, SLAM, ...)

♦ From recursion schemes to transducers,
to deal with algebraic data types
(lists, trees, ...) [K., Tabuchi & Unno, POPL 2010]

♦ Infinite intersection types,
to deal with non-simply-typed programs
[Tsukada & K. FoSSaCS 2010]
Outline

♦ Higher-order recursion schemes
♦ From program verification to model checking recursion schemes
♦ From model checking to type checking
♦ Type checking (=model checking) algorithm
♦ TRecS: Type-based RECursion Scheme model checker
♦ Ongoing work
♦ Discussion
Advantages of our approach

(1) Sound, **complete and automatic** for a large class of higher-order programs
   - no false alarms!
   - no annotations
Advantages of our approach

(1) Sound, complete and automatic for a large class of higher-order programs
   - no false alarms!
   - no annotations

(2) Subsumes finite-state/pushdown model checking
   - Order-0 rec. schemes $\approx$ finite state systems
   - Order-1 rec. schemes $\approx$ pushdown systems
Advantages of our approach

(3) Take the best of model checking and types

- Types as certificates of successful verification
  ⇒ applications to PCC (proof-carrying code)

- Counterexample when verification fails
  ⇒ error diagnosis,
  CEGAR (counterexample-guided abstraction refinement)
Advantages of our approach

(4) Encourages structured programming

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

Main:
fp1 := open "r" "foo";
fp2 := open "w" "bar";
Loop:
c1 := read fp1;
if c1=eof then goto E;
write(c1, fp2);
goto Loop;
E:
close fp1;
close fp2;

V.S.

let copyfile fp1 fp2 =
  try write(read fp2, fp1);
  copyfile fp1 fp2
  with
    Eof -> close(fp1);close(fp2)
let main =
  let fp1 = open "r" file in
  let fp2 = open "w" file in
  copyfile fp1 fp2
Advantages of our approach

(4) Encourages structured programming

Our technique:
- No loss of precision for higher-order functions and recursion
- Performance penalty? -- Not necessarily!
  - n-EXPTIME in the specification size, but polynomial time in the program size
- Compact representation of large state space
e.g. recursion schemes generating $a^m(c)$
  $S \rightarrow F_1 \ c, \ F_1 \ x \rightarrow F_2(F_2 \ x), \ldots, \ F_n \ x \rightarrow a(a \ x)$
  vs
  $S \rightarrow a \ G_1, \ G_1 \rightarrow a \ G_2, \ldots, \ G_m \rightarrow c \ (m=2^n)$
Advantages of our approach

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

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

Property violated? yes no

Step 2: Extract type environment \( \Gamma_0 \)

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

Property satisfied?

yes

no
Challenges

♦ More efficient model checker
  - More language-theoretic properties of recursion schemes (e.g. pumping lemmas)
  - BDD-like state representation

♦ Software model checker for ML/Haskell

♦ Extension of the decidability of higher-order model checking (Tree(G) |= φ)

♦ Integration with testing (e.g. QuickCheck)
Conclusion

- New program verification technique based on model checking recursion schemes
  - Many attractive features
    - Sound and complete for higher-order programs
    - Take the best of model-checking and type-based techniques
  - Many interesting and challenging topics
References

♦ K., Types and higher-order recursion schemes for verification of higher-order programs, POPL09
  From program verification to model-checking, and typing

♦ K.&Ong, Complexity of model checking recursion schemes for fragments of the modal mu-calculus, ICALP09
  Complexity of model checking

♦ K.&Ong, A type system equivalent to modal mu-calculus model-checking of recursion schemes, LICS09
  From model-checking to type checking

♦ K., Model-checking higher-order functions, PPDP09
  Type checking (= model-checking) algorithm

♦ K., Tabuchi & Unno, Higher-order multi-parameter tree transducers and recursion schemes for program verification, POPL10
  Extension to transducers and its applications

♦ Tsukada & K., Untyped recursion schemes and infinite intersection types, FoSSaCS 10