Higher-Order Model Checking for Program Verification

Naoki Kobayashi
Tohoku University

In collaboration with:
Ryosuke Sato, Naoshi Tabuchi, and Hiroshi Unno
(Tohoku University)
Luke Ong (University of Oxford)
This Talk

♦ Overview of higher-order program verification based on higher-order model checking (or, the model checking of higher-order recursion schemes)
  - What is higher-order model checking?
  - How can program verification be reduced to higher-order model checking?
  - How can higher-order model checking problems be solved?

Goal: Software model checker for ML
Outline

♦ Higher-order recursion schemes and model checking

♦ From program verification to model checking of recursion schemes [K. POPL09]

♦ Dealing with infinite data
  - From recursion schemes to higher-order transducers [K., Tabuchi, and Unno, POPL10]
  - Predicate abstraction and CEGAR [K., Sato, Unno, 2010]

♦ Model checking algorithms for recursion schemes [K. PPDP09]

♦ Conclusion
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-1 scheme**

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

Tree whose paths are labeled by \(a^{m+1} b^m c\)
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 \( \text{Tree}(G) \)?

\[ n\text{-EXPTIME-complete [Ong, LICS06]} \]
(for order-n recursion scheme)

\[ \frac{2^p(x)}{2} \text{ for } n \]

E.g.
- Does every finite path end with “c”?  
- Does “a” occur eventually whenever “b” occurs?
TRecS [K., PPDP09]
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 Buchi automata with a trivial acceptance condition.

Some examples are given below. More examples are available here.

Example 1

```plaintext
BEGIN
/// Prenux rules of a recursion scheme. Non-terminal must start with a capital letter. ///
S -> F C. /// The non-terminal of the first rule is interpreted as the start symbol. ///
F x -> a x (F (b x)). /// Unbounded names "a", "b" in this rule are interpreted as terminals. ///
END

BEGIN
/// Transition rules of a Buchi tree automaton (where all the states are final). ///
q0 a -> q0 q0. /// The first state is interpreted as the initial state. ///
q0 b -> q1.
```
Why Recursion Schemes?

♦ Expressive:
  - Subsumes many other MSO-decidable tree classes
    (regular, algebraic, Caucal hierarchy, HPDS, ...)

♦ High-level (c.f. higher-order PDS):
  - Recursion schemes
    ≈
    Simply-typed λ-calculus
    + recursion
    + tree constructors (but not destructors)
    + finite data domains such as booleans
      (via Church encoding, true=λx.λy.x, false=λx.λy.y)
    + infinite data with a restricted set of primitives

Suitable models for higher-order programs
Outline

♦ Higher-order recursion schemes and model checking

♦ From program verification to model checking of recursion schemes [K. POPL09]

♦ Dealing with infinite data
  - From recursion schemes to higher-order transducers [K., Tabuchi, and Unno, POPL10]
  - Predicate abstraction and CEGAR [K., Sato, Unno, 2010]

♦ Model checking algorithms for recursion schemes [K. PPDP09]

♦ Conclusion
From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

Higher-order program + specification → Program Transformation → Rec. scheme (describing all event sequences or outputs) + Tree automaton, recognizing valid event sequences or outputs

Model Checking

Sound, complete, and automatic for:
- Simply-typed $\lambda$-calculus + recursion + finite base types (e.g. booleans)
- A large class of verification problems:
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 $r^*c$?
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?

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

F x k → + (c k) (r(F x k))
S → F d ★

CPS Transformation!

Is each path of the tree labeled by r*c?
Dealing with Multiple Resources

```ml
let f(x, y) =  
  if * then  
    close(x); close(y)  
  else  
    read(x); write(y);  
    f(x, y)  
in  
let x = newrc() in  
let y = newwc() in  
  f(x, y)
```

recursion scheme generating a tree that represents resource-wise access sequence

access to x

access to y
Dealing with Multiple Resources

\[
\text{let } f(x, y) = \\
\text{if } * \text{ then close}(x); \text{ close}(y) \\
\text{else read}(x); \text{ write}(y); \text{ f}(x, y) \\
\text{in} \\
\text{let } x = \text{new}^{r*c}() \text{ in} \\
\text{let } y = \text{new}^{w*c}() \text{ in} \\
\text{f}(x, y)
\]
Dealing with Multiple Resources

\[
\text{let } f(x, y) = \\
\quad \text{if } * \text{ then } \\
\quad \quad \text{close}(x); \text{close}(y) \\
\quad \text{else } \\
\quad \quad \text{read}(x); \text{write}(y); \text{f}(x, y) \\
\quad \text{in } \\
\text{let } x = \text{new}^{r*c}() \text{ in } \\
\text{let } y = \text{new}^{w*c}() \text{ in } \\
\quad \text{f}(x, y)
\]

\[
S \rightarrow \text{new}^{r*c} \\
\quad (\lambda x. \text{new}^{w*c}(\lambda y. F x y *)) \\
\text{new}^{r*c} k \rightarrow + (\nu^{r*c}(k I)) (k K) \\
I a k \rightarrow a k \\
K a k \rightarrow k \\
\text{close } x k \rightarrow x c k
\]

Non-deterministically choose whether or not to keep track of the new resource
Dealing with Multiple Resources

\[
\text{let } f(x, y) = \\
\text{if } * \text{ then close}(x); \text{ close}(y) \\
\text{else read}(x); \text{ write}(y); \text{ f}(x, y) \\
\text{in} \\
\text{let } x = \text{new}_{r*c}(\lambda x. \text{new}_{w*c}(\lambda y. F \times y * )) \\
\text{let } y = \text{new}_{w*c}(\lambda x. \text{new}_{r*c}(\lambda y. F \times y * )) \\
\text{in} \\
\]
Dealing with Multiple Resources

let f(x, y) = 
  if * then 
    close(x); close(y) 
  else 
    read(x); write(y); 
  f(x, y) 
in 
let x = newr*c () in 
let y = neww*c () in 
f(x, y)
Dealing with Multiple Resources

let f(x, y) = 
  if * then 
    close(x); close(y) 
  else 
    read(x); write(y); 
  f(x, y) 

in 
let x = newr*c () in 
let y = neww*c () in 
  f(x, y)
Dealing with Multiple Resources

let \( f(x, y) = \)
if * then
  close(x); close(y)
else
  read(x); write(y);
  \( f(x, y) \)
in
let \( x = \text{new}^{r*c}() \) in
let \( y = \text{new}^{w*c}() \) in
\( f(x, y) \)
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 (+dynamic creation of resources)
- A large class of verification problems:
  resource usage verification [Igarashi&K. POPL2002], reachability, flow analysis, strictness 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>

\[
\text{infinite state model checking}
\]
Outline

- Higher-order recursion schemes and model checking
- From program verification to model checking of recursion schemes [K. POPL09]
- Dealing with infinite data
  - From recursion schemes to higher-order transducers [K., Tabuchi, and Unno, POPL10]
  - Predicate abstraction and CEGAR [K., Sato, Unno, 2010]
- Model checking algorithms for recursion schemes [K. PPDP09]
- Conclusion
Goal

♦ Automated verification of higher-order tree-processing programs

fun revApp x y = revAcc y (revAcc x [])
and revAcc x z =
   case x of (* list destructor *)
      [ ] => z
   | elem::y => revAcc y (elem::z)

Given two sequences \(x \in a^*b^*\) and \(y \in b^*c^*\), does \(\text{revApp} \ x \ y\) return an element of \(c^*b^*a^*\)?
Our Approach

♦ Extension of our previous verification method, by higher-order, multi-tree transducers (HMTT)
  - Sound, complete, automatic for a certain class of higher-order programs manipulating lists and trees

Previous approach [K. POPL09]:
Functional program → Recursion scheme → Model checking

New approach:
Functional program → HMTT → Recursion scheme with finite data → Model checking

More suitable for modeling higher-order programs with algebraic data types
Higher-Order, Multi-Tree Transducers (HMTT)

♦ Two kinds of trees
  - input trees, which can only be destructed
  - output trees, which can only be constructed
♦ Higher-order (recursive) functions
♦ Multiple inputs (unlike ordinary transducers)

\[
\text{RevApp } x \ y = \text{RevAcc } y \ \text{(RevAcc } x \ \text{nil}).
\]
\[
\text{RevAcc } x \ z =
\]
\[
\text{case } x \ \text{of}
\]
\[
\text{nil } \Rightarrow \ z
\]
\[
\text{a}(y) \Rightarrow \text{RevAcc } y \ \text{(a } z)\]
\[
\text{b}(y) \Rightarrow \text{RevAcc } y \ \text{(b } z)\]
\[
\text{c}(y) \Rightarrow \text{RevAcc } y \ \text{(c } z).
\]
Given:

\[ \text{HMTT} \; H: \ i \rightarrow \ldots \rightarrow i \rightarrow o \]

input specification: \( L_1, \ldots, L_m \) (regular tree languages)

output specification: \( L \)

does

\[ H : L_1 \rightarrow \ldots \rightarrow L_m \rightarrow L \]

(i.e. \( \forall t_1 \in L_1, \ldots, t_m \in L_m . (H \; t_1 \; \ldots \; t_m) \in L \))

hold?

Example. RevApp: \( a*b* \rightarrow b*c* \rightarrow c*b*a* \) ?

RevApp \( x \; y = \text{RevAcc} \; y \; (\text{RevAcc} \; x \; \text{nil}) \).

RevAcc \( x \; z = \text{case} \; x \; \text{of} \; \text{nil} \; \Rightarrow \; z \)

| \; a(y) \; \Rightarrow \; \text{RevAcc} \; y \; (a \; z) 
| \; b(y) \; \Rightarrow \; \text{RevAcc} \; y \; (b \; z) 
| \; c(y) \; \Rightarrow \; \text{RevAcc} \; y \; (c \; z) .
HMTT verification method

HMTT verification problem:

\[ H(L_1, \ldots, L_m) \subseteq L? \]

1. Construct a recursion scheme (with finite data) \( G \) that approximates \( H(L_1, \ldots, L_m) \)
2. Use a higher-order model checker to decide:

\[ \text{Lang}(G) \subseteq L \]
From HMTT to Recursion Scheme

Ideas:
- Construct a recursion scheme (with finite data) approximating the output language.
- By abstracting input trees by states of tree automaton.

\[
H: a^*b^*c^* \to c^*b^*a^*?
\]

\[
H(x) = \text{Rev } x \ z
\]

\[
\text{Rev } x \ z =
\text{case } x \text{ of}
\]

\[
\begin{align*}
\text{nil} & \Rightarrow z \\
| a(y) & \Rightarrow \text{Rev } y \ (a \ z) \\
| b(y) & \Rightarrow \text{Rev } y \ (b \ z) \\
| c(y) & \Rightarrow \text{Rev } y \ (c \ z).
\end{align*}
\]

Abstraction of nil or c(y), with \( \alpha(y) = q_2 \)

Ranges over \( \{q_0, q_1, q_2\} \)

\[
\begin{align*}
\text{Rev } x \ z & \Rightarrow \\
\text{case } x \text{ of}
\end{align*}
\]

\[
\begin{align*}
q_0 & \Rightarrow + (\text{Rev } q_0 \ (a \ z)) \\
& \quad (\text{Rev } q_1 \ (b \ z)) \\
& \quad (\text{Rev } q_2 \ (c \ z)) \\
q_2 & \Rightarrow + (\text{Rev } q_2 \ (c \ z)) \ z
\end{align*}
\]
Soundness and (In)completeness

♦ Soundness:
   If the verification succeeds, HMTT satisfies the input/output specification.

♦ Completeness for linear HMTT:
   If a linear HMTT satisfies the specification, the verification succeeds.

♦ Incompleteness:
   HMTT verification is undecidable
   (so that the verification may fail even if the given HMTT satisfies the spec.)
From (deterministic) HTT [Engelfreit] to linear HMTT

♦ HTT

\[
F\ e = e \\
F\ (a(x)) = a\ (F\ x)\ (F\ x) \\
F\ (b(x)) = b\ (F\ x)\ (F\ x)
\]

♦ linear HMTT

\[
F\ x = H\ x\ (\lambda y.\ y) \\
H\ z\ k = \\
\text{case}\ z\ \text{of} \\
a(x) \Rightarrow H\ x\ (\lambda y.\ k\ (a\ y\ y)) \\
b(x) \Rightarrow H\ x\ (\lambda y.\ k\ (b\ y\ y))
\]
### Experiments

<table>
<thead>
<tr>
<th></th>
<th>order</th>
<th>rules</th>
<th>states</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeAdr</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>RemoveB</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>HomRep</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>Gapid</td>
<td>3</td>
<td>12</td>
<td>30</td>
<td>87</td>
</tr>
<tr>
<td>Xml_rep1</td>
<td>3</td>
<td>8</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>Xhtml_id</td>
<td>1</td>
<td>1</td>
<td>50</td>
<td>86</td>
</tr>
<tr>
<td>Xhtml_div</td>
<td>1</td>
<td>2</td>
<td>50</td>
<td>39</td>
</tr>
<tr>
<td>Xhtml_a</td>
<td>1</td>
<td>2</td>
<td>50</td>
<td>243</td>
</tr>
</tbody>
</table>

*Cannot be verified by previous methods*

*Much faster than state-of the art for HTT [Tozawa05]*

*Comparable to state-of the art for MTT [Frisch&Hosoya]*

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

HomRep \( n \ s = F \ n \ (\text{Hom} \ (\text{Concat} \ B \ B) \ A) \ (\text{I2Str} \ s) \).

\( F \ n \ h \ s = \begin{cases} \text{zero} => & (\text{Str2O} \ s) \\ \text{succ}(m) => & F \ m \ h \ (h \ s) \end{cases} \)

\begin{align*}
A \ xa \ xb \ z &= xa \ z. \\
B \ xa \ xb \ z &= xb \ z. \\
\text{Empty} \ xa \ xb \ z &= z. \\
\text{Concat} \ s1 \ s2 \ xa \ xb \ z &= \\
& s1 \ xa \ xb \ (s2 \ xa \ xb \ z). \\
\text{Hom} \ sa \ sb \ s \ xa \ xb \ z &= \\
& s \ (sa \ xa \ xb) \ (sb \ xa \ xb) \ z. \\
\text{I2Str} \ x &= \begin{cases} e => & \text{Empty} \\
& a(y) => \text{Concat} \ A \ (\text{I2Str} \ y) \\
& b(y) => \text{Concat} \ B \ (\text{I2Str} \ y). \end{cases} \\
\text{Str2O} \ s &= s \ a \ b \ e.
\end{align*}

String operations

Conversion between internal string representation and trees
Forward vs Backward Inference

Forward inference

\[ H(L_I) \]

approximation of \( H(L_I) \) (exact for linear HMTT)

input spec. \( L_I \)

Backward inference

\[ H^{-1}(L_O) \]
(exact for HTT)

\[ H(L_1) \]

output spec. \( L_O \)
Dealing with Infinite Data Domains

♦ Higher-order multi-tree transducers (HMTT) to deal with algebraic data
  - HMTT verification method
    [K., Tabuchi & Unno, POPL 2010]
  - Extension to deal with arbitrary tree-processing programs
    [Unno, Tabuchi & K., APLAS 2010]

♦ Predicate abstraction and CEGAR (c.f. BLAST, SLAM, ...)

[Contributed by Hidetaka Tabuchi and Mayuko Kato]
Limitation of HMTT

Strict classification of trees into input/output trees (constructed trees cannot be destructed again)

```haskell
fun rev x =
    case x of
        nil => nil
        | a(y) => app (rev y) (a nil)
        | b(y) => app (rev y) (b nil)

and app x y =
    case x of
        nil => y
        | a(z) => a (app z y)
        | b(z) => b (app z y)
```
Extended HMTT [Unno et al. APLAS 2010]

Allow coercion from output to input trees:

```
fun rev x =
  case x of
    nil => nil
    | a(y) => app (coerce (rev y)) (a nil)
    | b(y) => app (coerce (rev y)) (b nil)

and app x y =
  case x of
    nil => y
    | a(z) => a (app z y)
    | b(z) => b (app z y)
```
Extended HMTT [Unno et al. APLAS 2010]

♦ Allow coercion from output to input trees and requires it be annotated with invariant:

fun rev x = (* rev: a*b* -> b*a* *)
    case x of
        nil => nil
        | a(y) => app (coerceb*a* (rev y)) (a nil)
        | b(y) => app (coerceb* (rev y)) (b nil)

and app x y =
    case x of
        nil => y
        | a(z) => a (app z y)
        | b(z) => b (app z y)
Verification method for Extended HMTT

EHMTT verification problem

HMTT verification problem

HMTT verification problem

recursion scheme model checking

recursion scheme model checking

recursion scheme model checking
Reduction from EHMTT to HMTT verification

1. Assuming coercion annotations to be correct, verify that the output is correct.

\[
\text{fun rev } x = \begin{cases}
\text{nil} & \Rightarrow \text{nil} \\
\text{a(y)} & \Rightarrow \text{app (coerce}^{b*a*} (\text{rev y})) (\text{a nil}) \\
\text{b(y)} & \Rightarrow \text{app (coerce}^{b*} (\text{rev y})) (\text{b nil})
\end{cases}
\]

\[
\text{fun revO } x = \begin{cases}
\text{nil} & \Rightarrow \text{nil} \\
\text{a(y)} & \Rightarrow \text{app gen}^{b*a*} (\text{a nil}) \\
\text{b(y)} & \Rightarrow \text{app gen}^{b*} (\text{b nil})
\end{cases}
\]
Reduction from EHMTT to HMTT verification

1. Assuming coercion annotations to be correct, verify that the output is correct.

2. Verify the soundness of each annotation (assuming the other coercions to be correct)

```plaintext
fun rev x = (* rev: a*b* -> b*a* *)
case x of nil => nil
| a(y) => app (coerceb*a* (rev y)) (a nil)
| ...
```

```plaintext
fun revC x = (* approximate trees passed to coerceb*a* *)
case x of nil => empty
| a(y) => (revO y)
```
Reduction from EHMTT to HMTT verification

1. Assuming coercion annotations to be correct, verify that the output is correct.

2. Verify the soundness of each annotation (assuming the other coercions to be correct)

\[
\text{fun rev} \ x = (* \ rev: \ a^*b^* \rightarrow b^*a^* \ *)
\]
\[
\text{case } x \text{ of } \text{nil} \Rightarrow \text{nil}
\]
\[
| \ a(y) \Rightarrow \text{app (coerce}^{b^*a^*} (\text{rev } y)) (a \text{ nil})
\]
\[
| \ ... 
\]

\[
\text{fun revC} \ x = (* \ \text{approximate trees passed to } \text{coerce}^{b^*a^*} *)
\]
\[
\text{case } x \text{ of } \text{nil} \Rightarrow \text{empty}
\]
\[
| \ a(y) \Rightarrow \text{union (revO } y) (\text{revC } y)
\]
Reduction from EHMTT to HMTT verification

1. Assuming coercion annotations to be correct, verify that the output is correct.

2. Verify the soundness of each annotation (assuming the other coercions to be correct)

```haskell
fun rev x = (* rev: a*b* -> b*a* *)
  case x of nil => nil
           | a(y) => app (coerce b*a* (rev y)) (a nil)
           | ...,

fun revC x = (* approximate trees passed to coerce b*a* *)
  case x of nil => empty
           | a(y) => union (union (revO y) (revC y))
                     (appC gen b*a* (a nil))
```
Correctness Issues

♦ Soundness:
A verified (EHMTT) program satisfies the input/output specification

♦ Incompleteness:
There is a program that is correct but cannot be verified by our method

Sources of incompleteness:
- Incompleteness of HMTT verification
- Coercion annotations may not be strong enough.
  (c.f. loop invariant annotations for Hoare logic)
## Experiments

<table>
<thead>
<tr>
<th>Programs</th>
<th>#C</th>
<th>#Fun</th>
<th>Size</th>
<th>T&lt;sub&gt;Red&lt;/sub&gt;</th>
<th>T&lt;sub&gt;MC&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse</td>
<td>2</td>
<td>3</td>
<td>32</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Isort</td>
<td>1</td>
<td>4</td>
<td>29</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Msort</td>
<td>4</td>
<td>8</td>
<td>131</td>
<td>2</td>
<td>224</td>
</tr>
<tr>
<td>HomRep-Rev</td>
<td>1</td>
<td>12</td>
<td>90</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>Split</td>
<td>1</td>
<td>6</td>
<td>126</td>
<td>3</td>
<td>132</td>
</tr>
<tr>
<td>Bib2Html</td>
<td>1</td>
<td>13</td>
<td>493</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>XMarkQ1</td>
<td>1</td>
<td>12</td>
<td>454</td>
<td>29</td>
<td>168</td>
</tr>
<tr>
<td>XMarkQ2</td>
<td>2</td>
<td>9</td>
<td>461</td>
<td>77</td>
<td>92</td>
</tr>
<tr>
<td>Gapid-Html</td>
<td>1</td>
<td>17</td>
<td>374</td>
<td>2</td>
<td>112</td>
</tr>
<tr>
<td>JWIG-guess</td>
<td>1</td>
<td>6</td>
<td>465</td>
<td>588</td>
<td>50</td>
</tr>
<tr>
<td>JWIG-cal</td>
<td>2</td>
<td>12</td>
<td>475</td>
<td>72</td>
<td>73</td>
</tr>
<tr>
<td>MinCaml-K</td>
<td>8</td>
<td>19</td>
<td>605</td>
<td>5</td>
<td>647</td>
</tr>
</tbody>
</table>

- **String processing**
- **XML transform**
- **Web applications**
- **Program transform**

(milli-sec.)
Experiments on Buggy Programs

Correctly
Rejected!

<table>
<thead>
<tr>
<th>Programs</th>
<th>#C</th>
<th>#Fun</th>
<th>Size</th>
<th>$T_{\text{Red}}$</th>
<th>$T_{\text{MC}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split-e</td>
<td>1</td>
<td>6</td>
<td>126</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>JWIG-guess-e</td>
<td>1</td>
<td>6</td>
<td>465</td>
<td>586</td>
<td>49</td>
</tr>
<tr>
<td>JWIG-cal-e</td>
<td>2</td>
<td>12</td>
<td>475</td>
<td>2</td>
<td>55</td>
</tr>
</tbody>
</table>
Related Work

♦ Ong and Ramsay, POPL2011
  - another approach to verification of tree processing programs via higher-order model checking
  - fully automated (c.f. coercion annotations in our approach)
  - abstracting input trees by patterns
  - counterexample-guided (pattern) abstraction refinement
Outline

♦ Higher-order recursion schemes and model checking

♦ From program verification to model checking of recursion schemes [K. POPL09]

♦ Dealing with infinite data
  - From recursion schemes to higher-order transducers [K., Tabuchi, and Unno, POPL10]
  - Predicate abstraction and CEGAR [K., Sato, Unno, 2010]

♦ Model checking algorithms for recursion schemes [K. PPDP09]

♦ Conclusion
Predicate Abstraction and CEGAR for Higher-Order Model Checking

\[ f(g, x) = g(x + 1) \]

Higher-order functional program

\[ x > 0 \]

Predicate abstraction

New predicates

Program is unsafe!

Real error path?

Error path

Property not satisfied

Property satisfied

Program is safe!

Higher-order model checking

Higher-order boolean program

\[ F(g, b) = \begin{cases} g(\text{true}) & \text{if } b \\ g(*) & \text{else} \end{cases} \]
What are challenges?

 Predicate abstraction

- How to choose predicates for each term, in such a way that the resulting HOBP is consistent?

  E.g. 
  
  ```
  fun f g x = ... g (x+1) ... 
  fun h y z = ... 
  fun main() = ... f (h 0) u ... 
  ```

  The same predicate should be used for z and u+1.

 CEGAR

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

♦ Predicate abstraction

Abstraction types to express abstraction interface:

E.g. \( f: (x:\text{int}[^\lambda x.x>0]) \rightarrow \text{int}[^\lambda y.y>x] \)

Assuming the argument \( n \) is abstracted using the predicate \( \lambda x.x>0 \), the abstraction of \( f \) should return the value of \( f(n) \) abstracted by \( \lambda y.y>n \).

\[
f(x) = \text{if } x>0 \text{ then } x+1 \text{ else } ...
\]

\[
\Rightarrow f'(b) = \text{if } b \text{ then } \text{true} \text{ else } ...
\]

♦ CEGAR

Reduction from abstraction type finding problem to a refinement type inference problem for SHP (straightline higher-order program).
Example (predicate abstraction)

\[
\begin{align*}
\text{let } f \ x \ g & = g(x+1) \text{ in} \\
\text{let } h \ z \ y & = \text{assert}(y>z) \text{ in} \\
\text{let } k \ n & = \text{if } n \geq 0 \text{ then } f \ n \ (h \ n) \text{ else } ( ) \text{ in} \\
\text{k } & m
\end{align*}
\]

\[
\begin{align*}
& \quad \begin{array}{l}
\text{f: } (x:\text{int}[]) \to (\text{int}[^{\lambda}y.y>x] \to \star) \to \star \\
\text{h: } (x:\text{int}[]) \to \text{int}[^{\lambda}y.y>x] \to \star \\
\text{k: } \text{int}[] \to \star
\end{array} \\
\text{let } f \ ( ) \ g & = g(\text{true}) \text{ in} \\
\text{let } h \ ( ) \ b & = \text{assert}(b) \text{ in} \\
\text{let } k \ ( ) & = \text{if } * \text{ then } f \ ( ) \ (h \ ( )) \text{ else } ( ) \text{ in} \\
\text{k( )}
\end{align*}
\]
## Experiments

<table>
<thead>
<tr>
<th></th>
<th>cycle</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc91</td>
<td>2</td>
<td>0.07</td>
</tr>
<tr>
<td>ackermann</td>
<td>3</td>
<td>0.15</td>
</tr>
<tr>
<td>a-cppr</td>
<td>6</td>
<td>3.40</td>
</tr>
<tr>
<td>a-max</td>
<td>5</td>
<td>4.78</td>
</tr>
<tr>
<td>l-zipmap</td>
<td>4</td>
<td>0.20</td>
</tr>
<tr>
<td>l-zipunzip</td>
<td>3</td>
<td>0.12</td>
</tr>
<tr>
<td>repeat</td>
<td>3</td>
<td>0.15</td>
</tr>
<tr>
<td>a-max-e</td>
<td>2</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Arrays encoded by:

```ocaml
let mk_array n i =
  assert(0<=i && i<n); 0
let update i n a x =
  let a' j = if i=j then x else a(i) in
  a'
```

(Environment: Intel(R) Xeon(R) 3Ghz with 8GB memory)
FAQ

Does it 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
- Use the hybrid algorithm
  - Programs with worst-case behavior show an advantage of higher-order programs, rather than disadvantage of HO model checking

**Bad News**
- n-EXPTIME completeness
  - Huge constant factor
  - Hybrid algorithm has bad worst-case complexity!
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:**

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

Exponential time algorithm for order-1

Approximately

Polynomial time algorithm for order-0
Does higher-order model checking scale?

**Good News**
+ Fixed-parameter PTIME
+ Use the hybrid algorithm
+ Programs with worst-case behavior show an advantage of higher-order programs, rather than disadvantage of HO model checking
+ There is a realistic, fixed-parameter PTIME algorithm! (see our forthcoming paper)

**Bad News**
- n-EXPTIME completeness
- Huge constant factor
- Hybrid algorithm has bad worst-case complexity!
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:
$$S \rightarrow a \ G_1, \ G_1 \rightarrow a \ G_2, \ldots, \ G_k \rightarrow c \ (k=2^m)$$

Polynomial time algorithm for order-1

>>

Polynomial time algorithm for order-0
FAQ

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

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

♦ New program verification technique based on model checking recursion schemes

- Many attractive features
  - Sound, complete, and fully automatic for certain classes of higher-order programs and verification problems

- Many interesting and challenging topics
Challenges

♦ A more efficient algorithm for higher-order model checking
♦ A software model checker for ML/Haskell
♦ Other applications of finite-state/pushdown automata that can be extended to higher-order pushdown automata (or recursion schemes)
  (e.g. extension of regular model checking?)
♦ Extension of the decidability of higher-order model checking (Tree(G) |= ϕ)
♦ An (incomplete) algorithm for model checking of recursion schemes with advanced (e.g. recursive) types
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
  Extension to deal with more advanced types
♦ Unno, Tabuchi & K., ..., APLAS 2010
  Extension of POPL10 work to deal with arbitrary tree-processing programs