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

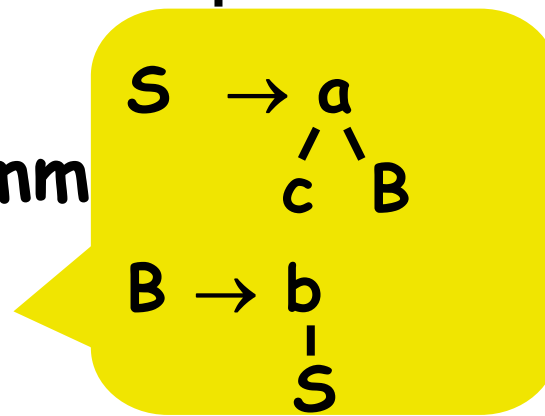
Higher-Order Recursion Scheme

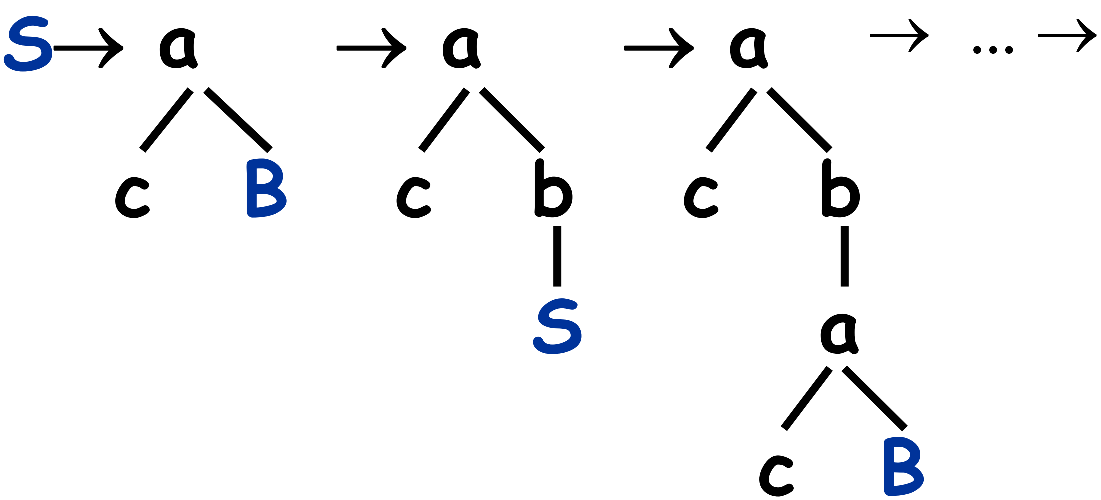
- Grammar for generating an infinite tree

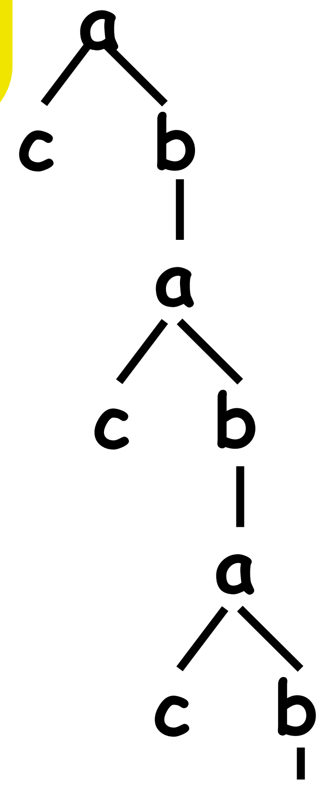
Order-0 scheme
(regular tree gramm

$S \rightarrow a\ c\ B$

$B \rightarrow b\ S$







Higher-Order Recursion Scheme

- Grammar for ite tree

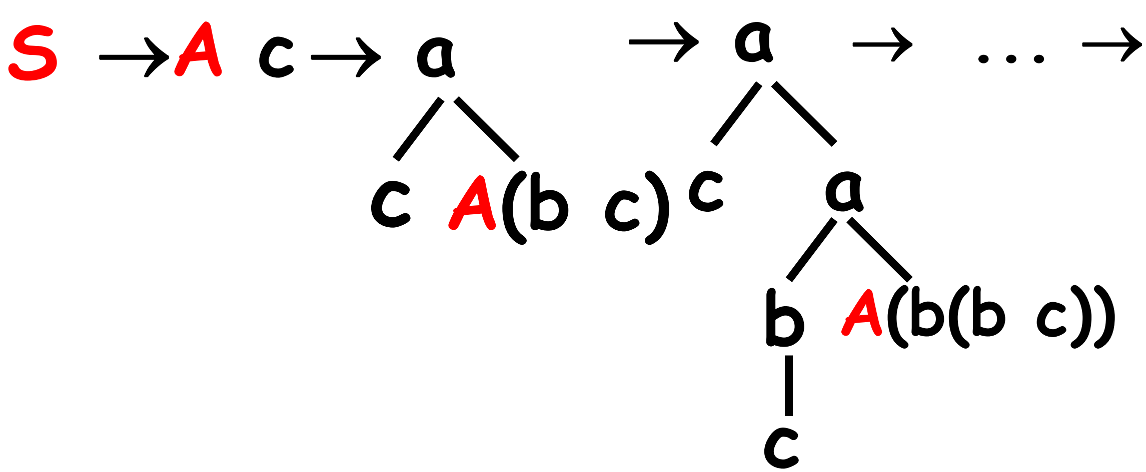
Tree whose paths are labeled by $a^{m+1}\ b^m\ c$

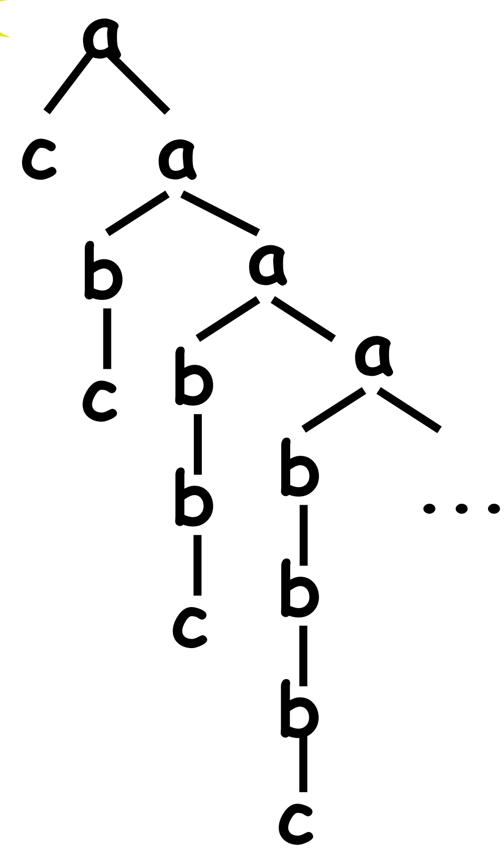
Order-1 schem

$$S \to A\ c$$

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

S: o, A: o→ o





Model Checking Recursion Schemes

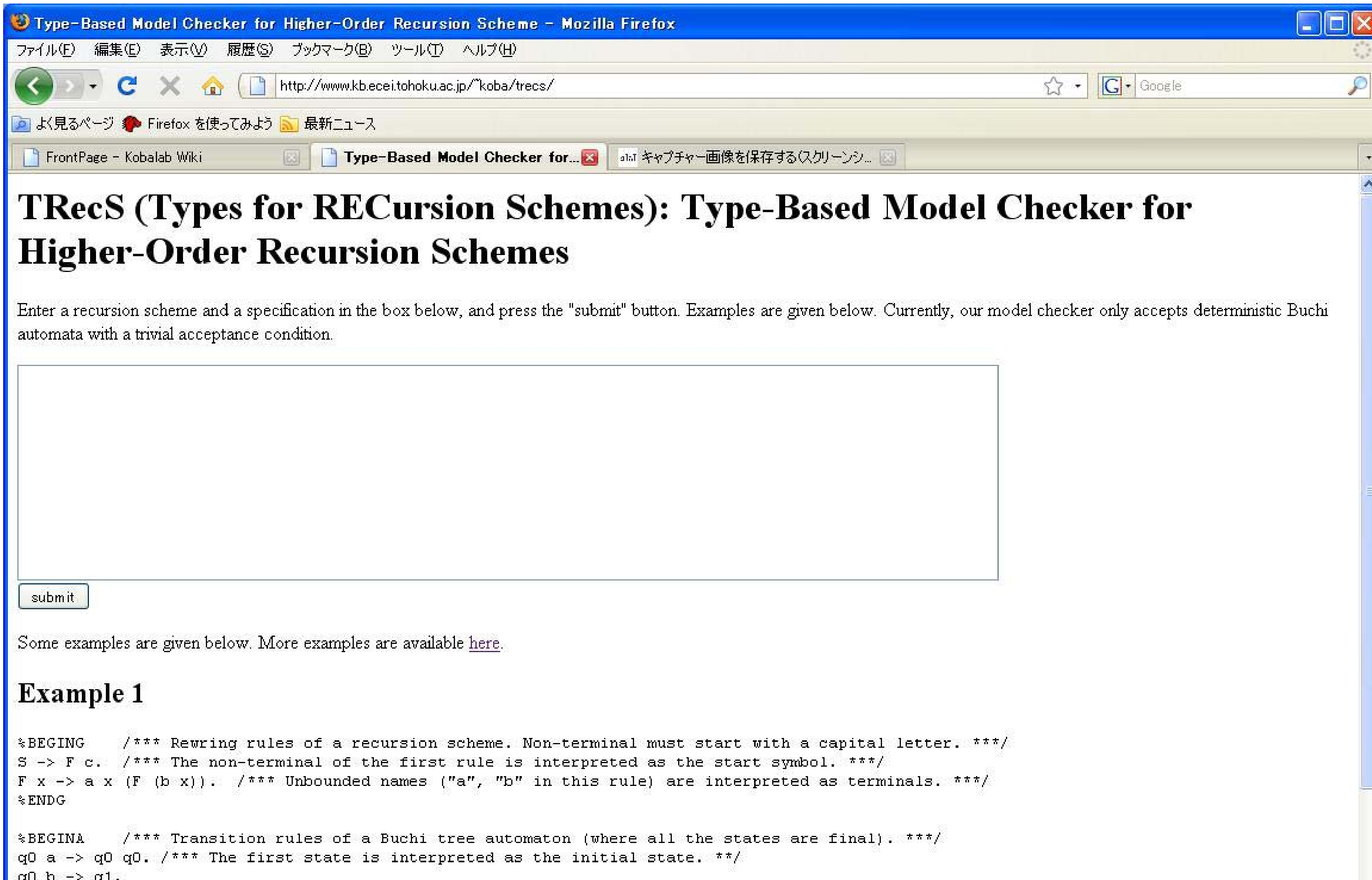
Given
  G: higher-order recursion scheme
  A: alternating parity tree automaton (APT)
    (a formula of modal μ-calculus or MSO),
does A accept Tree(G)?

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

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

TRecS [K., PPDP09]

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



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

    $\approx$

    Simply-typed λ-calculus
    \+ recursion
    \+ tree constructors (but not destructors)
    \+ finite data domains such as booleans
    (via Church encoding, true=$\lambda x.\lambda y.x$, false=$\lambda x.\lambda y.y$)
    \+ 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:

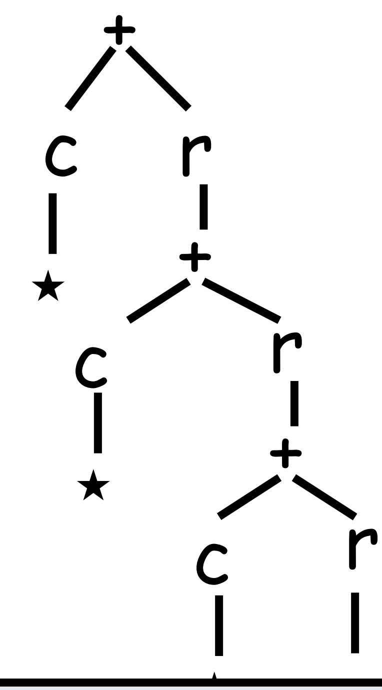
- Simply-typed λ-calculus + recursion + finite base types (e.g. booleans)
- A large class of verification problems:

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)
```

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

$S \rightarrow F\ d\ \star$



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

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

From Program ... ing:

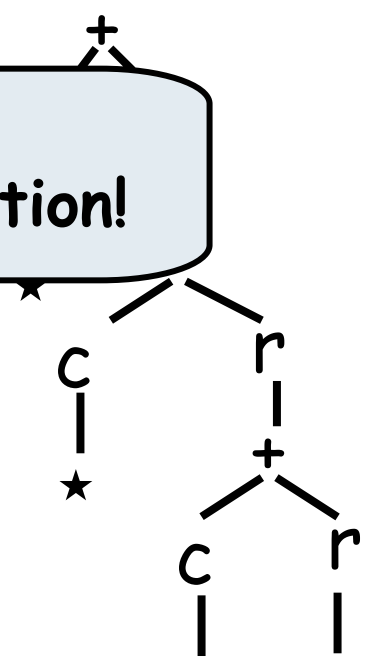
continuation parameter, expressing how "foo" is accessed after the call returns

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

$$S \to F\ d\ \star$$

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

CPS Transformation!



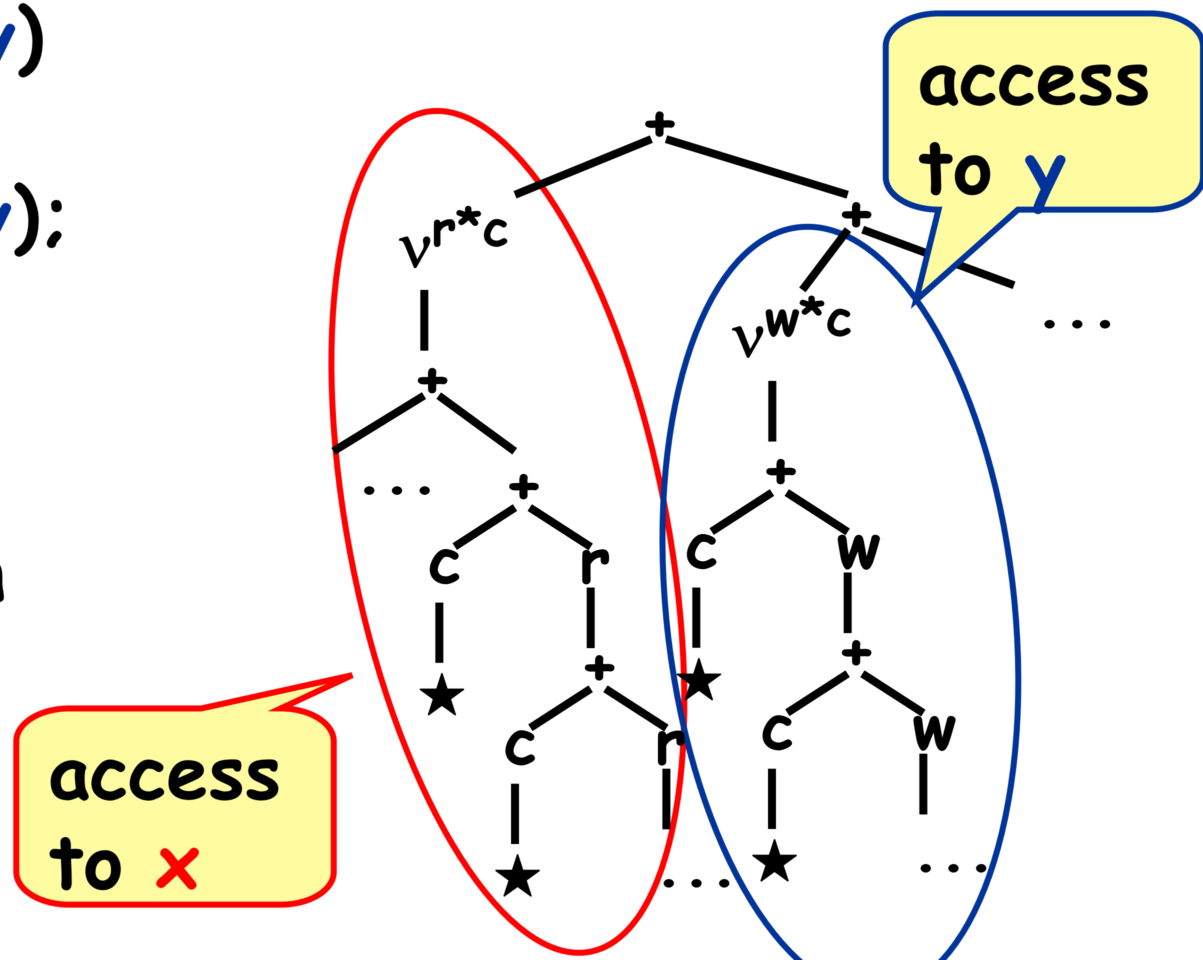
Is the file "foo" accessed according to read* close?

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

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 = new^{r*c} () in
let y = new^{w*c} () in
    f (x, y)
```

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



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 = new^{r*c} () in
let y = new^{w*c} () in
   f (x, y)
```

$\Longrightarrow$

$S \to \mathrm{new}^{r^*c}$
$\quad (\lambda x.\ \mathrm{new}^{w^*c}(\lambda y.\ F\ x\ y \star))$

$\mathrm{new}^{r^*c}\ k \to +\ (\nu^{r^*c}\ (k\ I))\ (k\ K)$

$I\ a\ k \to a\ k \qquad K\ a\ k \to k$

$\mathrm{close}\ x\ k \to x\ c\ k$

...

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 = new^{r*c} () in
let y = new^{w*c} () in
   f (x, y)
```

$$S \to \mathrm{new}^{r^*c}\ (\lambda x.\ \mathrm{new}^{w^*c}(\lambda y.\ F\ x\ y\ \star\ ))$$

$$\boxed{\mathrm{new}^{r^*c}\ \ k \to +\ (\nu^{r^*c}\ (k\ I))\ \ (k\ K)}$$

$$I\ a\ k \to a\ k \qquad K\ a\ k \to k$$

$$\mathrm{close}\ x\ k \to x\ c\ k$$

Non-deterministically choose whether or not to keep track of the new resource

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 = new^{r*c} () in
let y = new^{w*c} () in
   f (x, y)
```

$\Longrightarrow$

$S \to \text{new}^{r*c}$

$\quad (\lambda x.\ \text{new}^{w*c}(\lambda y.\ F\ x\ y\ \star\ )$

$\text{new}^{r*c}\ \ k \to +\ (\nu^{r*c}\ (k\ I))\ \ (k\ K)$

$I\ a\ k \to a\ k \qquad\qquad K\ a\ k \to k$

$\boxed{\text{close}\ x\ k \to x\ c\ k}$

...

x is bound to
I or K

$\text{close}\ I\ k$
$\to I\ c\ k$
$\to c\ k$

$\text{close}\ K\ k$
$\to K\ c\ k$
$\to k$

Dealing with Multiple Resources

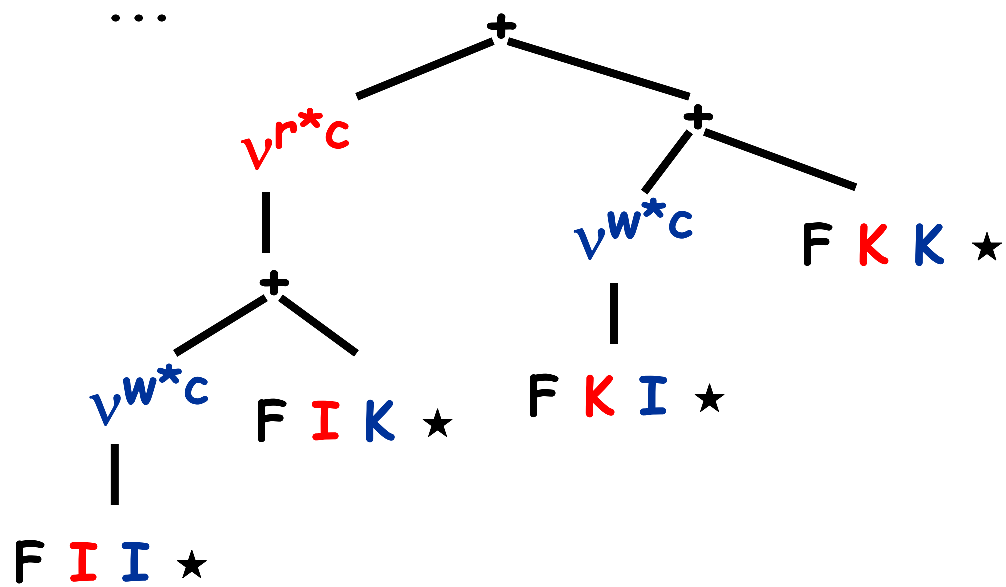
```
let f(x, y) =
  if * then
    close(x); close(y)
  else
    read(x); write(y);
    f(x, y)
in
let x = newr*c () in
let y = neww*c () in
     f (x, y)
```

$\Longrightarrow$

$S \to \mathbf{new}^{r^*c}$
$\quad (\lambda x.\ \mathbf{new}^{w^*c}(\lambda y.\ F\ x\ y\ \star\ ))$

$\mathbf{new}^{r^*c}\ k \to +\ (\nu^{r^*c}\ (k\ I))\ \ (k\ K)$

...



Dealing with Multiple Resources

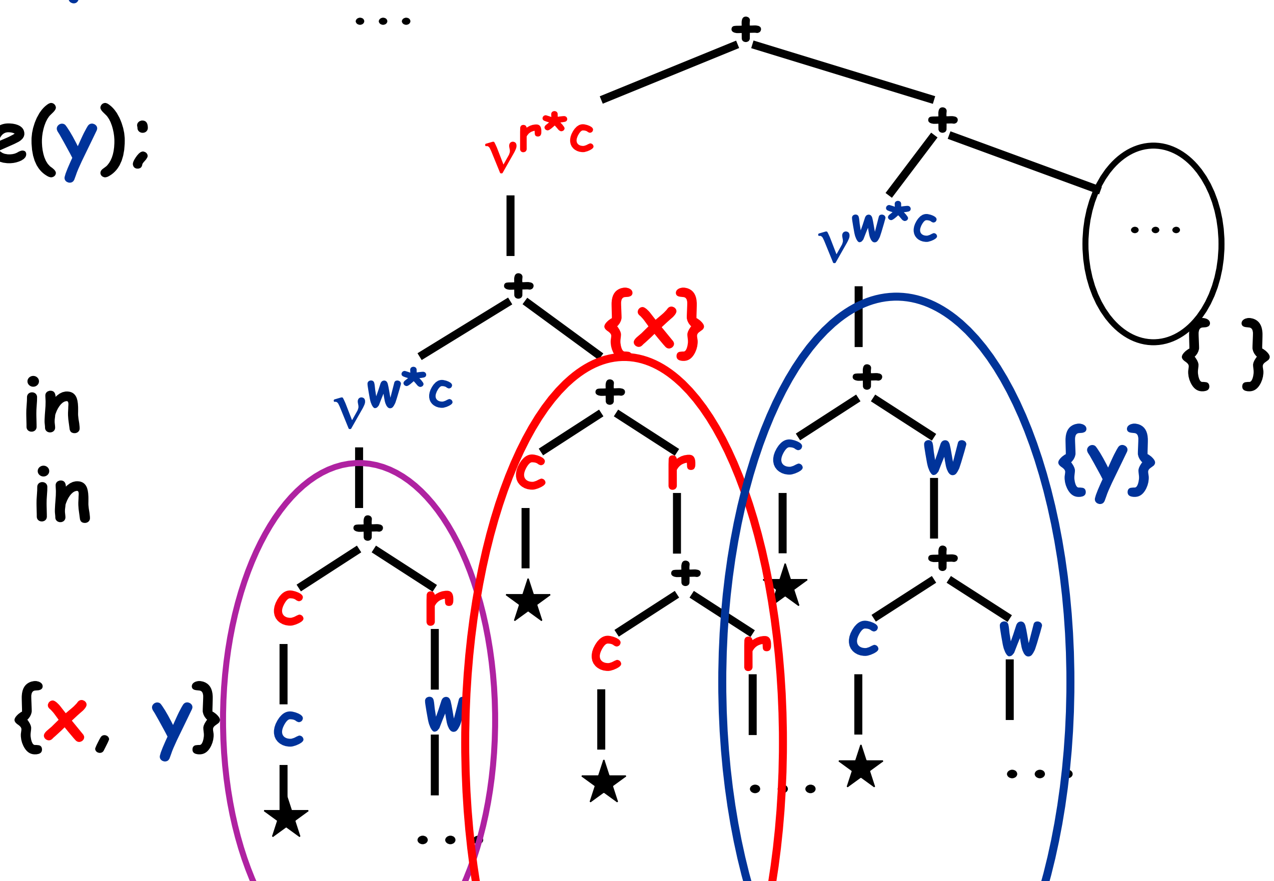
```
let f(x, y) =
  if * then
    close(x); close(y)
  else
    read(x); write(y);
    f(x, y)
in
let x = new^{r*c} () in
let y = new^{w*c} () in
  f (x, y)
```

$\Longrightarrow$

$S \to \mathrm{new}^{r^*c}$
$(\lambda x.\ \mathrm{new}^{w^*c}(\lambda y.\ F\ x\ y\ \star\ ))$

$\mathrm{new}^{r^*c}\ \ k \to +\ (\nu^{r^*c}\ (k\ I))\ \ (k\ K)$

...



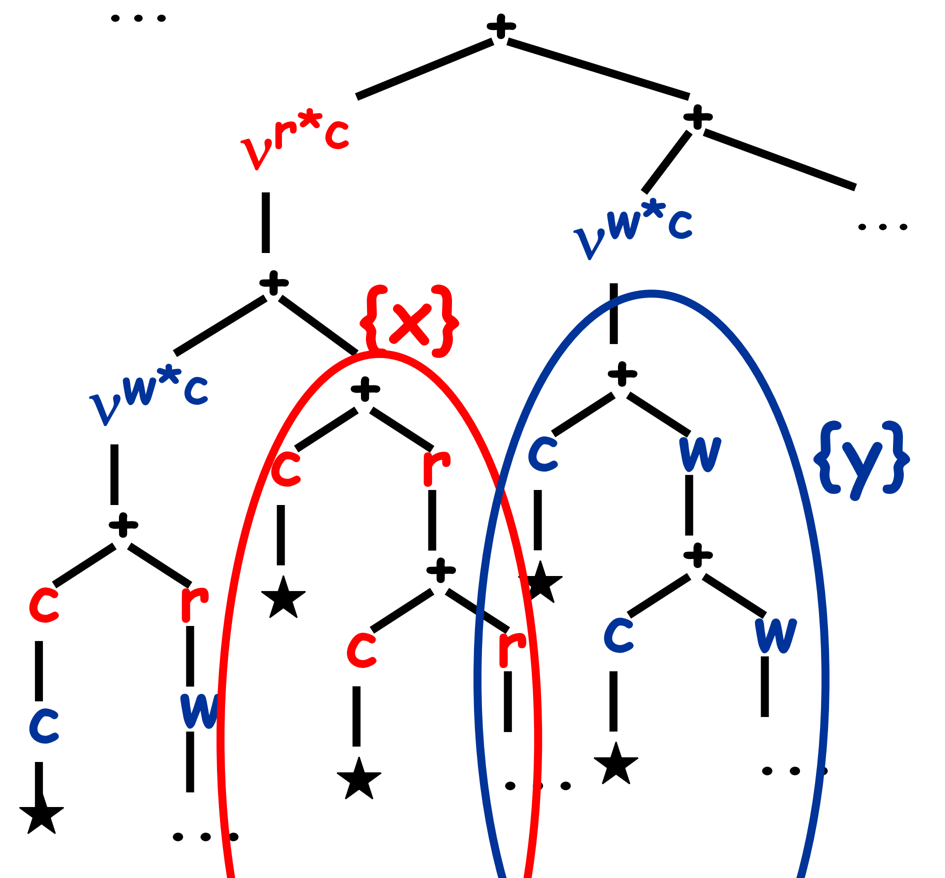
Dealing with Multiple Resources

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

$\Longrightarrow$

$$S \to \mathrm{new}^{r^*c}$$
$$(\lambda x.\ \mathrm{new}^{w^*c}(\lambda y.\ F\ x\ y\ \star\ ))$$
$$\mathrm{new}^{r^*c}\ \ k \to +\ (\nu^{r^*c}\ (k\ I))\ \ (k\ K)$$

...



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 (+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)

Program Classes	Verification Methods
Programs with while-loops	Finite state model checking
Programs with 1st-order recursion	Pushdown model checking
Higher-order functional programs	Recursion scheme model checking

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 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)

```
RevApp x y = RevAcc y (RevAcc x nil).
RevAcc x z =
  case x of nil => z
    | a(y) => RevAcc y (a z)
    | b(y) => RevAcc y (b z)
    | c(y) => RevAcc y (c z).
```

HMTT Verification Problem

Given:

HMTT H: $\overbrace{i \to \dots \to i}^{m} \to o$

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

output specification: L

does

$H : L_1 \to \dots \to L_m \to L$

(i.e. $\forall t_1 \in L_1, \dots, t_m \in L_m . (H\ t_1\ \dots\ t_m) \in L$)

hold?

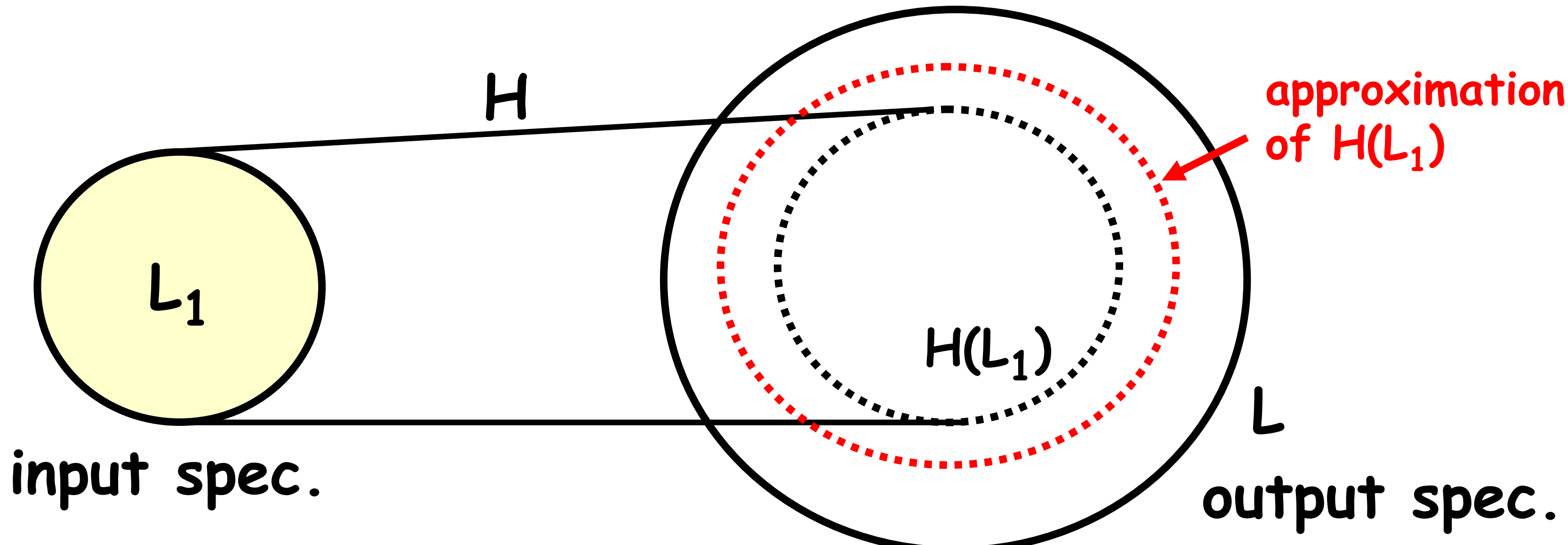
Example. RevApp: $a^*b^* \to b^*c^* \to c^*b^*a^*$?

```
RevApp x y = RevAcc y (RevAcc x nil).
RevAcc x z = case x of nil => z
                     | a(y) => RevAcc y (a z)
                     | b(y) => RevAcc y (b z)
                     | c(y) => 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:

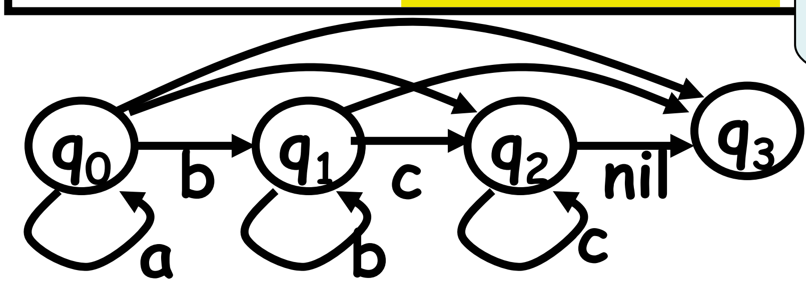
$$\mathrm{Lang}(G) \stackrel{?}{\subseteq} L$$

From HMTT to Recursion Scheme

Ideas:
- Construct a recursion schem
  approximating the output la
- by abstracting input trees b   tomaton

Ranges over {q0,q1,q2}

```
H: a*b*c* → c*b*a*?
H x = Rev x z
Rev x z =
  case x of nil => z
   | a(y) => Rev y (a z)
   | b(y) => Rev y (b z)
   | c(y) => Rev y (c z).
```




```
Rev x z →
  case x of
    q0 => + (Rev q0 (a z))
            (Rev q1 (b z))
            (Rev q2 (c z))
            z
            Rev q1 (b z))
            Rev q2 (c z))
            z
  | q2 =>  + (Rev q2 (c z)) z
```

Abstraction of nil or c(y) with α(y)=q2

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.)

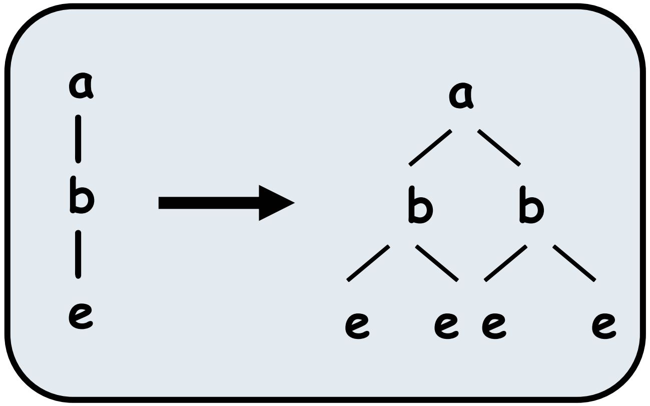
From (deterministic) HTT [Engelfreit] to linear HMTT

- **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 (λy.y)
H z k =
    case z of
       a(x) => H x (λy.k (a y y))
     | b(x) => H x (λy.k (b y y))
```

Experiments

	order	rules	states	Time (msec)
MergeAdr	1	3	6	2
RemoveB	2	4	7	1
HomRep	4	10	4	29
Gapid [Tozawa]	3	12	30	87
Xml_rep1 [Tozawa]	3	8	23	3
Xhtml_id	1	1	50	86
Xhtml_div	1	2	50	39
Xhtml_a	1	2	50	243

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)

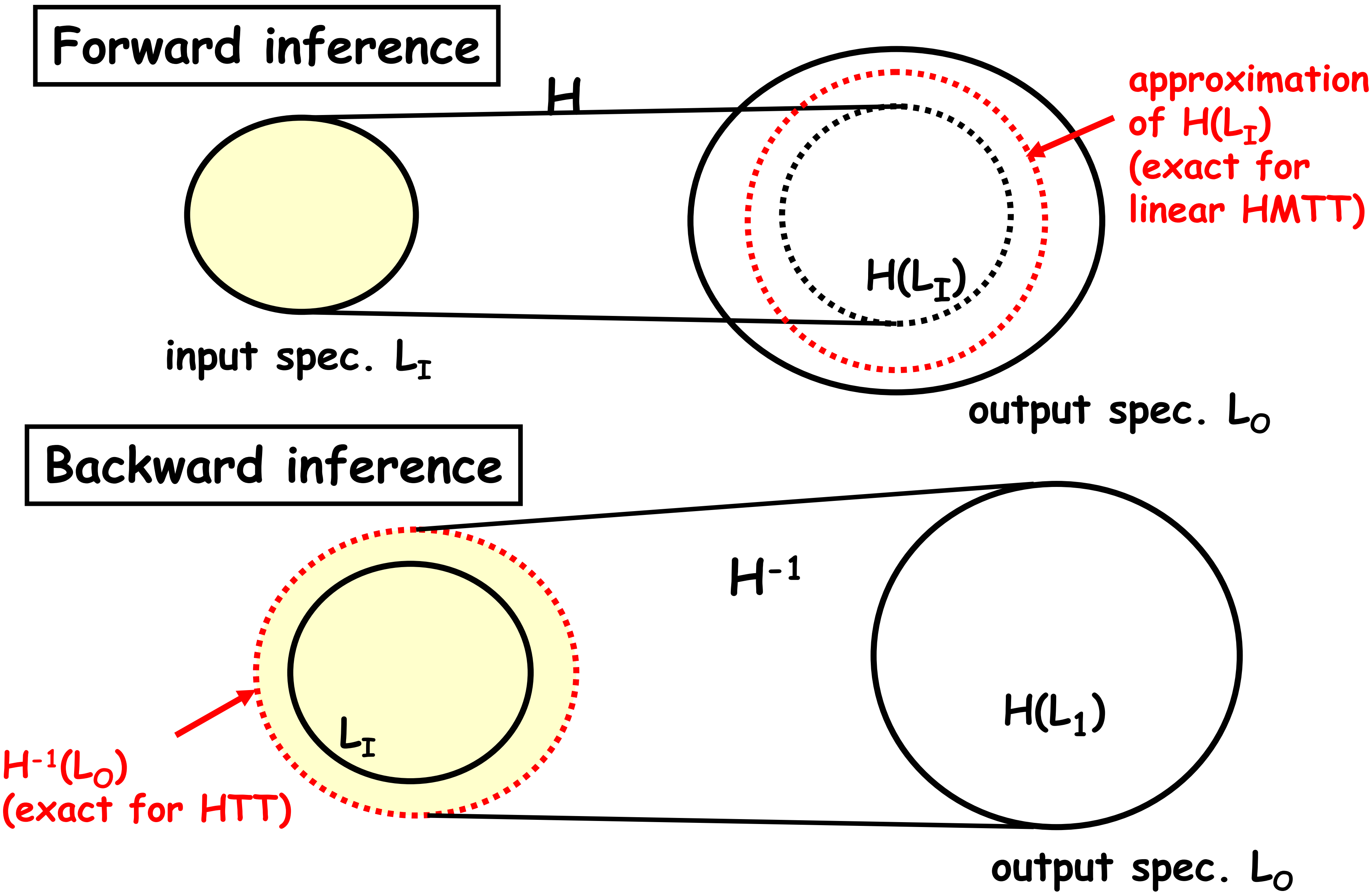
HomRep

```
HomRep n s = F n (Hom (Concat B B) A) (I2Str s).
F n h s =  case n of zero => (Str2O s)
                   | succ(m) => F m h (h s).
A xa xb z = xa z.
B xa xb z = xb z.
Empty xa xb z = z.
Concat s1 s2 xa xb z =
          s1 xa xb (s2 xa xb z).
Hom sa sb s xa xb z=
          s (sa xa xb) (sb xa xb) z.
I2Str x = case x of e => Empty
                 |  a(y) => Concat A (I2Str y)
                 | b(y) => Concat B (I2Str y).
Str2O s = s a b e.
```

String operations

Conversion between internal string representation and trees

Forward vs Backward Inference



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, …)

Limitation of HMTT

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

```
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:

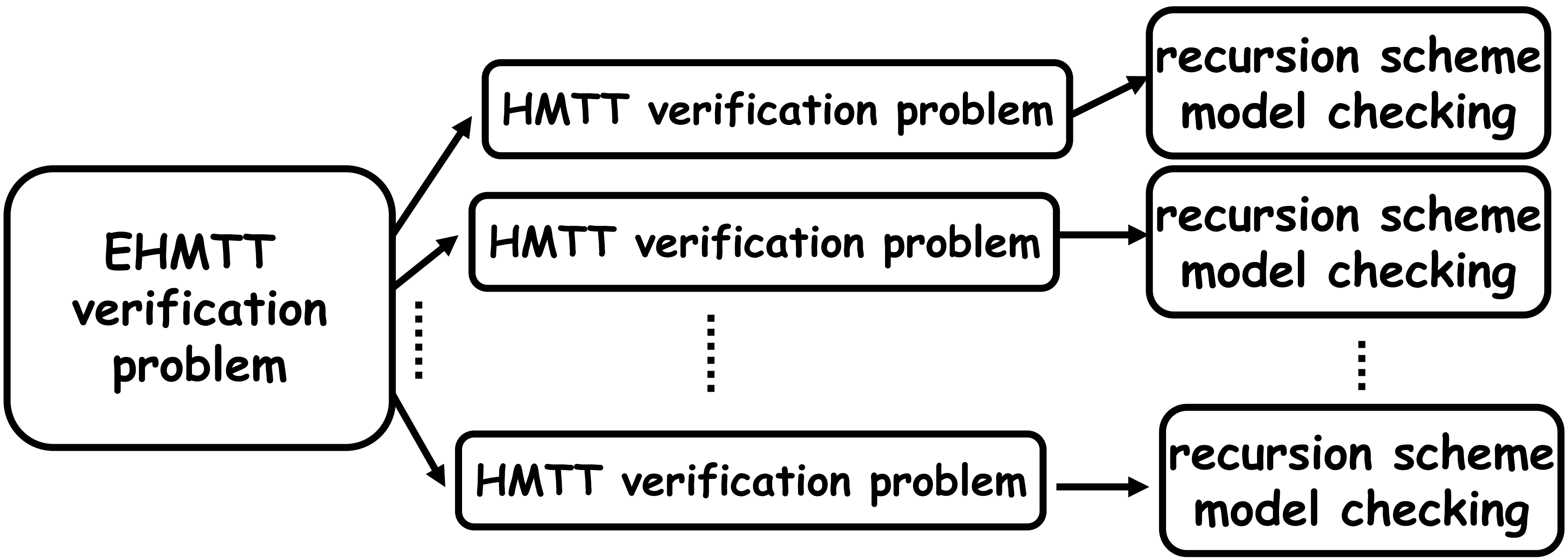
```
fun rev x =
    case x of
      nil    => nil
    | a(y) => app (coerce (rev y)) (a nil)
    | b(y) => app (coerce (rev y)) (b nil)
and app x y =
    case x of
      nil => y
    | a(z) => a (app z y)
    | b(z) => b (app z y)
```

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 (coerce^{b*a*} (rev y)) (a nil)
     | b(y) => app (coerce^{b*} (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



Reduction from EHMTT to HMTT verification

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

```
fun rev x =  (*  rev: a*b* -> b*a*   *)
    case x of
        nil    => nil
      | a(y) => app (coerce^{b*a*} (rev y)) (a nil)
      | b(y) => app (coerce^{b*} (rev y)) (b nil)
```

```
fun revO x =  (*  revO: a*b* -> b*a*   *)
    case x of
        nil    => nil
      | a(y) => app gen^{b*a*} (a nil)
      | b(y) => app gen^{b*} (b nil)
```

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)

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

$\Downarrow$

```
fun revC x =  (*  approximate trees passed to coerce^{b*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)

```
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 (revO y) (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)

```
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

	Programs	#C	#Fun	Size	T_{Red}	T_{MC}
String processing	Reverse	2	3	32	1	4
	Isort	1	4	29	1	3
	Msort	4	8	131	2	224
	HomRep-Rev	1	12	90	1	31
XML transform.	Split	1	6	126	3	132
	Bib2Html	1	13	493	52	52
	XMarkQ1	1	12	454	29	168
	XMarkQ2	2	9	461	77	92
	Gapid-Html	1	17	374	2	112
Web applications	JWIG-guess	1	6	465	588	50
	JWIG-cal	2	12	475	72	73
Program transform.	MinCaml-K	8	19	605	5	647

Experiments on Buggy Programs

(milli-secs.)

Programs	#C	#Fun	Size	T_{Red}	T_{MC}
Split-e	1	6	126	3	27
JWIG-guess-e	1	6	465	586	49
JWIG-cal-e	2	12	475	2	55

Correctly Rejected!

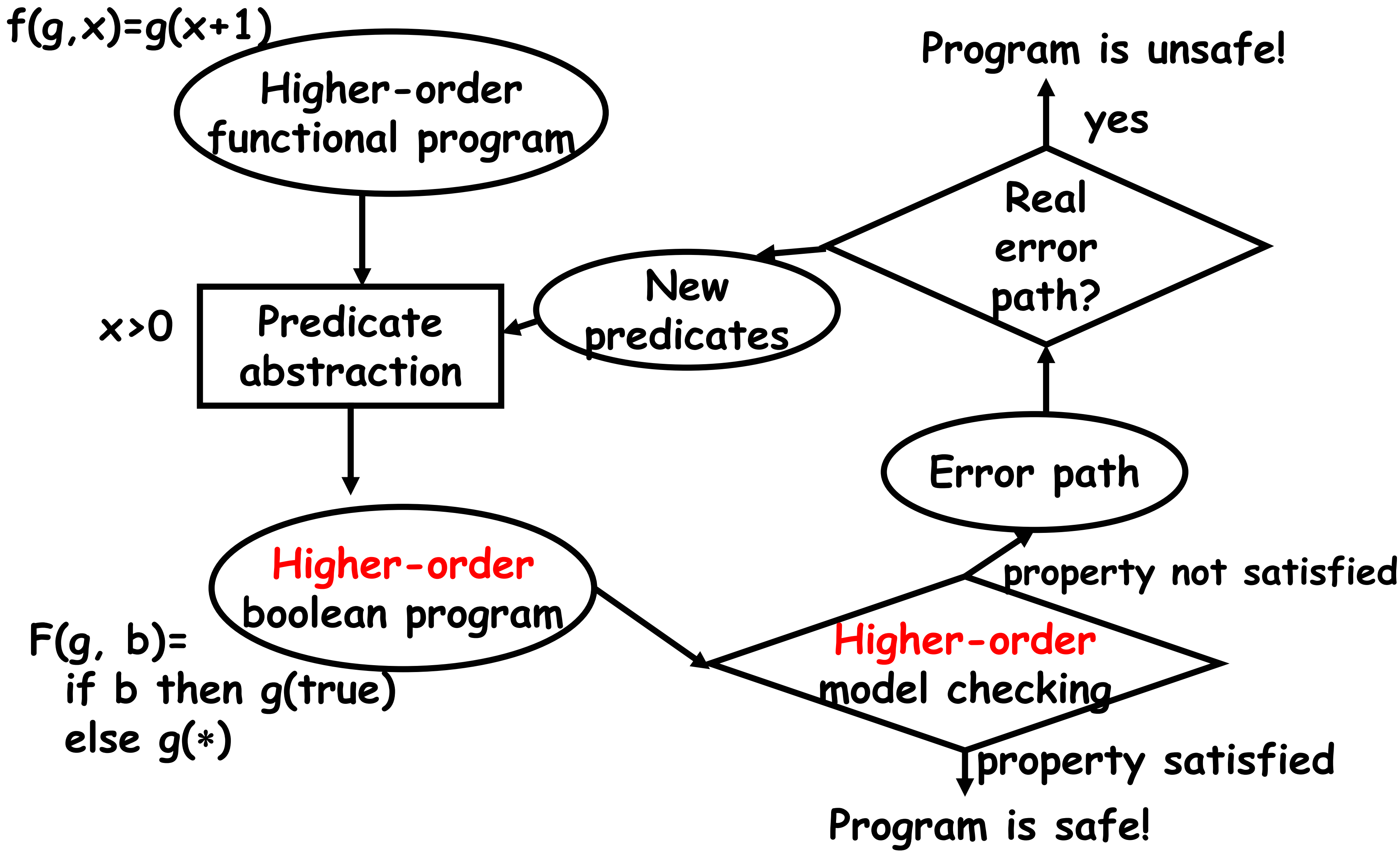
Related Work

- 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



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. $f: (x:\text{int}[\lambda x.x>0]) \to \text{int}[\lambda y.y>x]$

  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) = if x>0 then x+1 else ...
  => f'(b)= if b then true else ...

- CEGAR

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

Example (predicate abstraction)

```
let f x g = g(x+1) in
let h z y = assert(y>z) in
let k n = if n>=0 then f n (h n) else ( ) in
  k m
```

$$\Downarrow$$

f: $(x{:}\mathrm{int}[\,]) \rightarrow (\mathrm{int}[\lambda y.y>x] \rightarrow \star) \rightarrow \star$

h: $(x{:}\mathrm{int}[\,]) \rightarrow \mathrm{int}[\lambda y.y>x] \rightarrow \star$

k: $\mathrm{int}[\,] \rightarrow \star$

```
let f ( ) g = g(true) in
let h ( ) b = assert(b) in
let k ( ) = if * then f ( ) (h ( )) else ( ) in
  k( )
```

Experiments

	cycle	Time (sec)
mc91	2	0.07
ackermann	3	0.15
a-cppr	6	3.40
a-max	5	4.78
l-zipmap	4	0.20
l-zipunzip	3	0.12
repeat	3	0.15
a-max-e	2	0.13

Arrays encoded by:

```
let mk_array n i =
    assert(0<=i && i<n); 0
let update i n a x =
    a(i);
    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:

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

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

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:

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

Order-0:

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

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) $\models \varphi$)
- 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