Higher-Order Model Checking and Program Verification

Naoki Kobayashi
University of Tokyo

What's This Talk About?

- Introduction to
  - higher-order model checking
    (model checking of higher-order recursion schemes)
  - and its applications to
    automated verification of
    higher-order functional programs
    (e.g. "software model checker" for ML)
- Discussion on potential applications to
  automated verification of code generators

Tool demonstration: MoCHi

(a software model checker for a subset of OCaml)

Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- "Standard" applications to program verification
- Applications to verification of code generators
- Conclusion

Higher-Order Recursion Scheme (HORS)

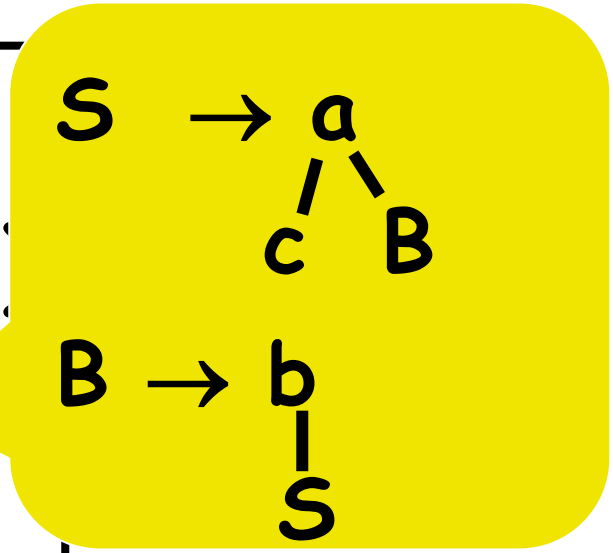
- Grammar for generating an infinite tree

Order-0 HORS

(regular tree grammar)

$S \to a\ c\ B$

$B \to b\ S$



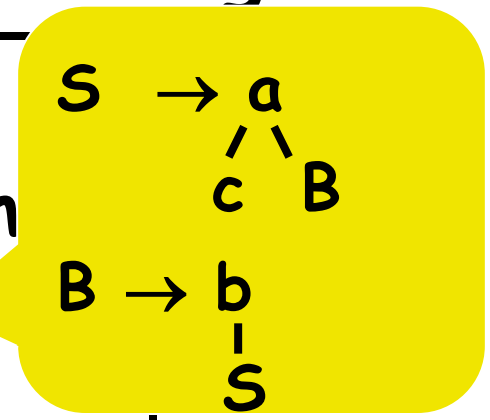
Higher-Order Recursion Scheme (HORS)

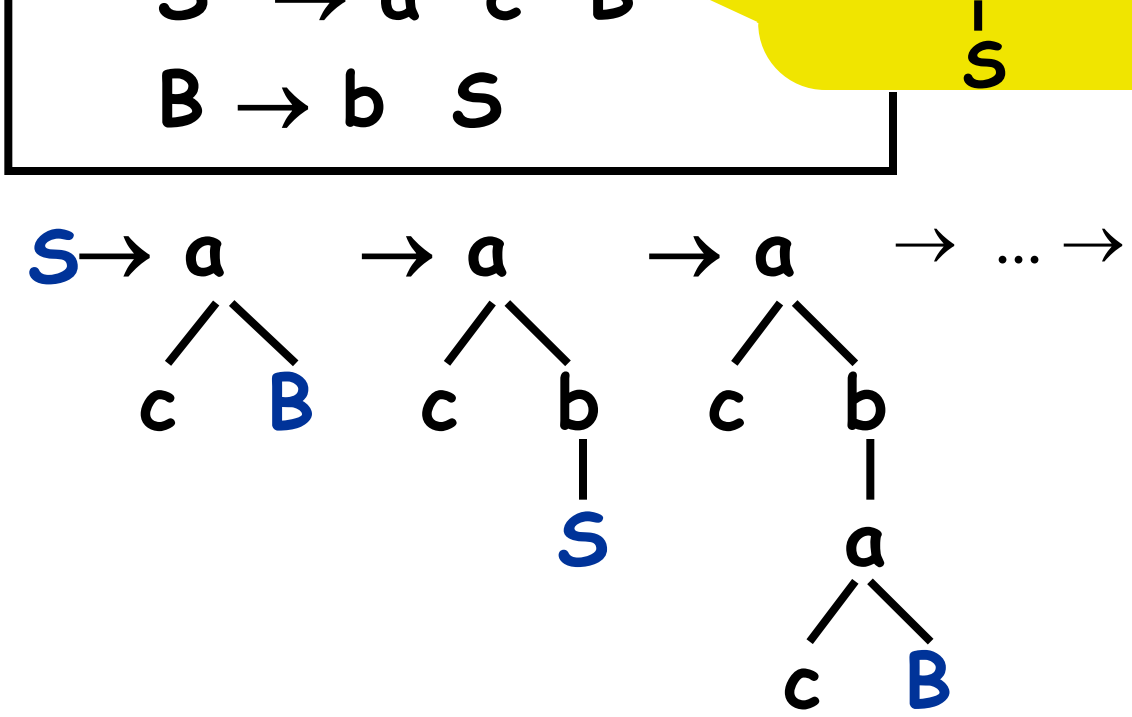
- Grammar for generating an infinite tree

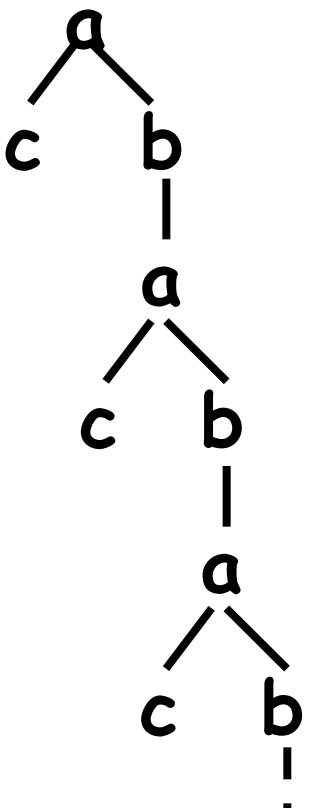
Order-0 HORS
(regular tree gramm

$S \to a\ c\ B$

$B \to b\ S$







Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

Order-1 HORS

$S \rightarrow A\ c$

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

S: o, A: o → o

Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

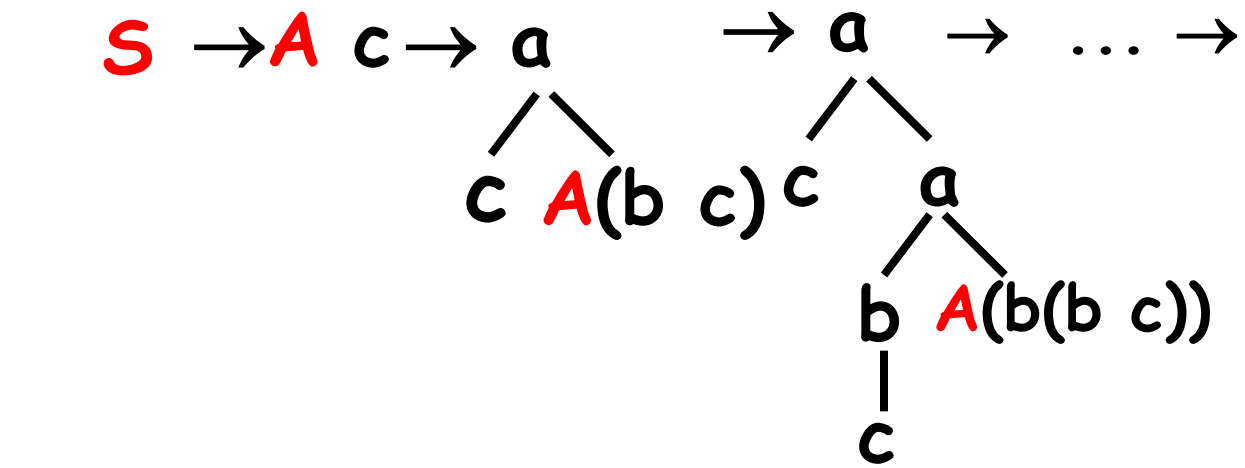
Order-1 HORS

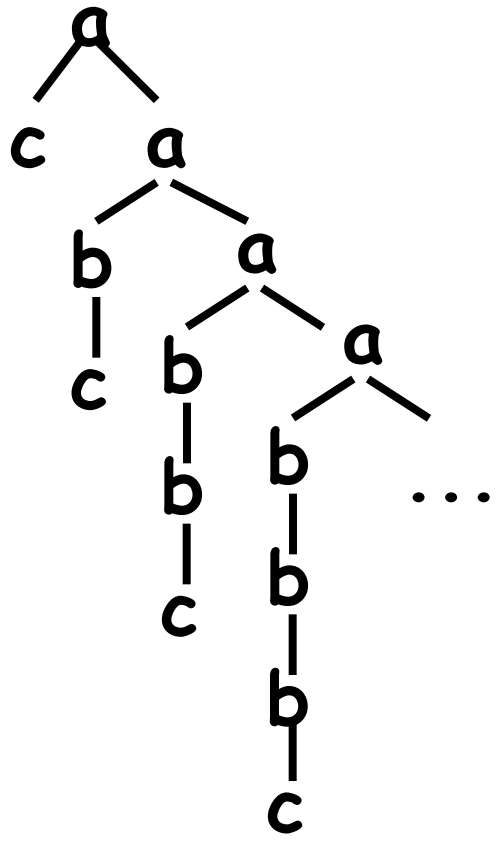
$S \rightarrow A\ c$

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

S: o, A: o→ o

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





Higher-Order Recursion Scheme (HORS)

- Grammar for generating an infinite tree

Order-1 HORS

$S \rightarrow A\ c$

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

S: o, A: o→ o

HORS

≈

Call-by-name simply-typed λ-calculus

+

recursion, tree constructors

Higher-Order Model Checking

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

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

Higher-Order Model Checking

Order-1 HORS

$S \rightarrow A\ c$

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

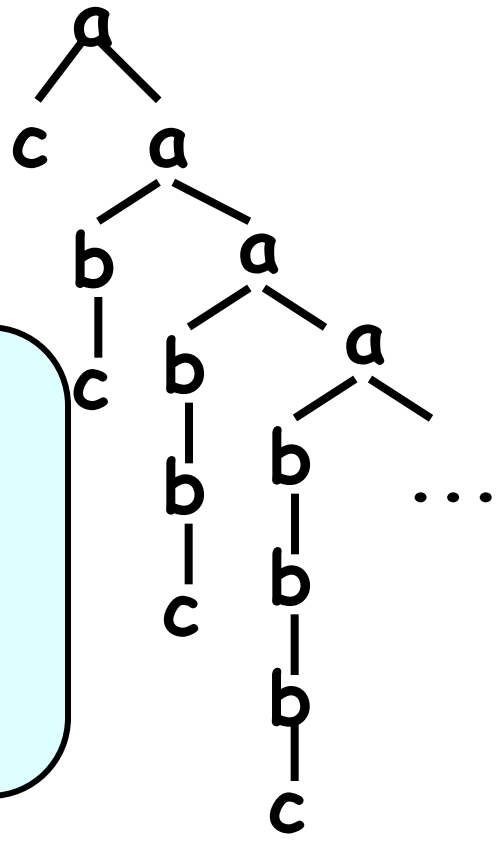
S: o, A: o→ o

Q1. Does every finite path end with "c"?

YES!

Q2. Does "a" occur below "b"?

NO!



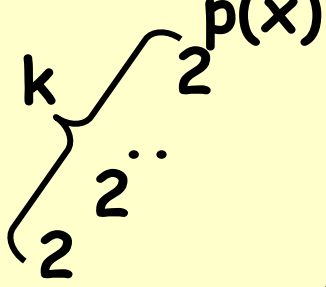
Higher-Order Model Checking

Given
- G: HORS
- A: alternating parity tree automaton (APT) (a formula of modal μ-calculus or MSO),

does A accept Tree(G)?

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

k-EXPTIME-complete [Ong, LICS06] (for order-k HORS)



TRecS [K. PPDP09]

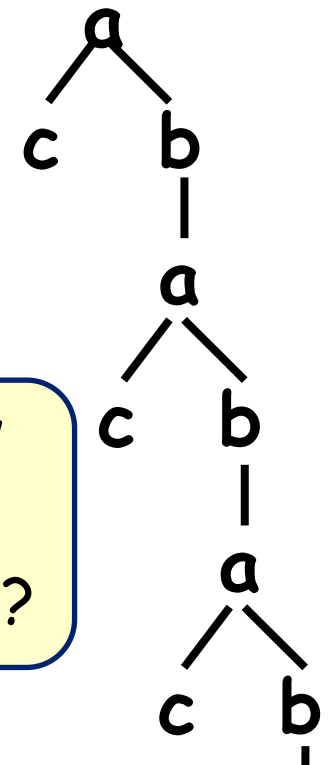
http://www-kb.is.s.u-tokyo.ac.jp/~koba/trecs/

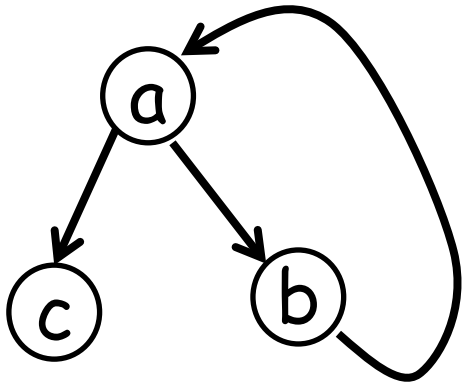
- The first practical model checker for HORS
- Does not immediately suffer from k-EXPTIME bottleneck
- A more recent model checker (HorSat2) can scale up to grammars consisting of 100,000 rules, depending on input

HO Model Checking as Generalization of Finite State/Pushdown Model Checking

- order-0 ≈ finite state model checking
- order-1 ≈ pushdown model checking

infinite tree ≈ transition system





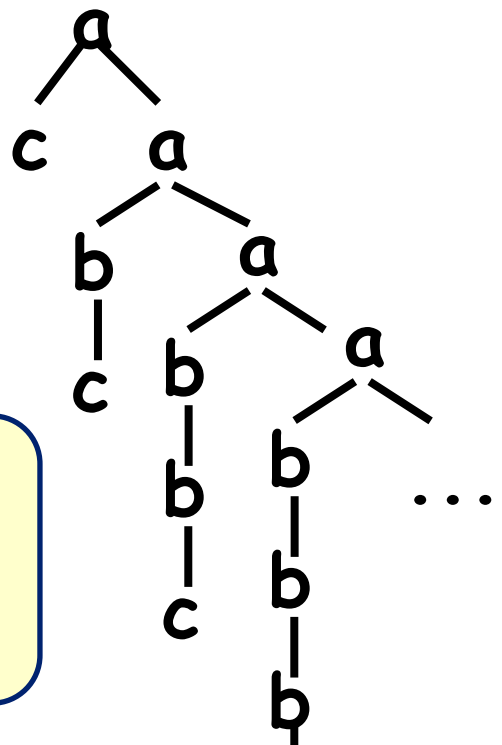
Does "a" occur below "b"?

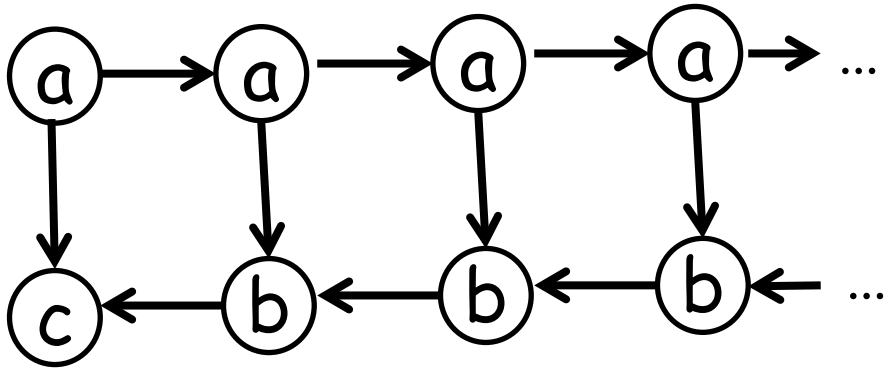
Is there a transition sequence in which "a" occurs after "b"?

HO Model Checking as Generalization of Finite State/Pushdown Model Checking

- order-0 ≈ finite state model checking
- order-1 ≈ pushdown model checking

infinite tree ≈ (infinite-state) transition system





Does "a" occur below "b"?

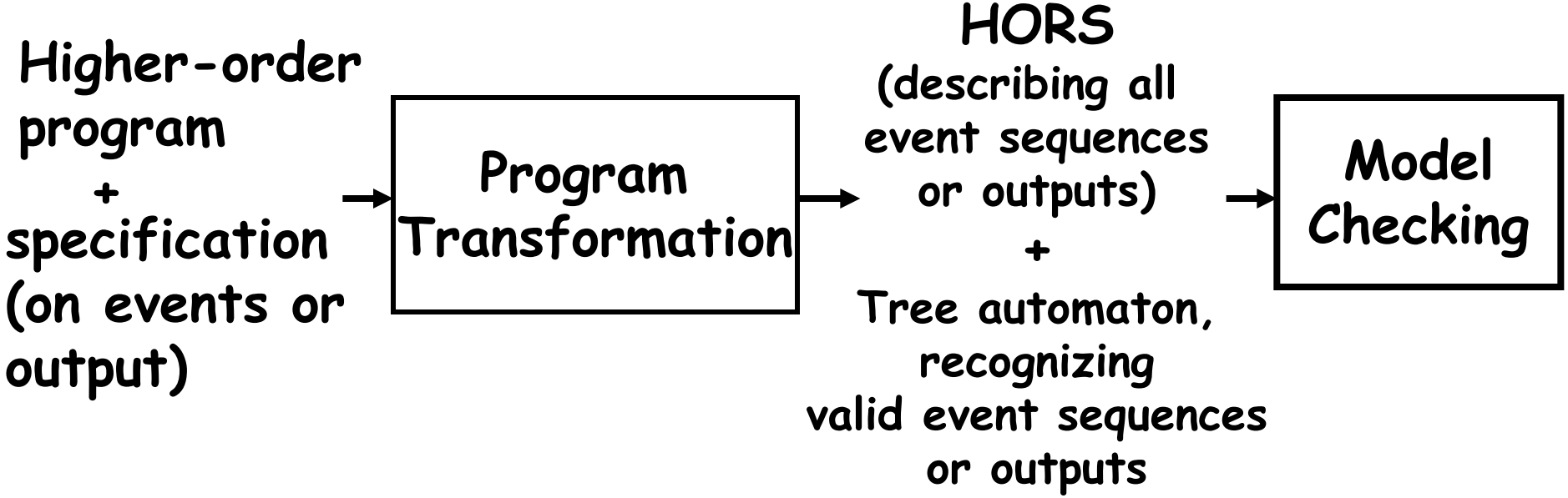
Is there a transition sequence in which "a" occurs after "b"?

Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- "Standard" applications to program verification
- Applications to verification of code generators
- Conclusion

From Program Verification to HO Model Checking

[K. POPL 2009]

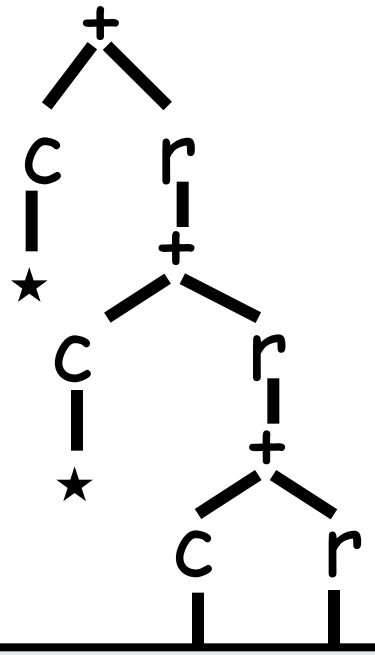


From Program Verification to Model Checking: Example

```
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

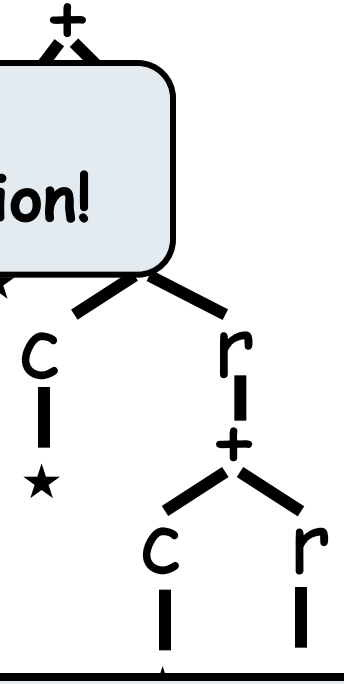
continuation parameter, expressing how "foo" is accessed after the call returns

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

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

$S \rightarrow F\ d\ \bigstar$

CPS Transformation!



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

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

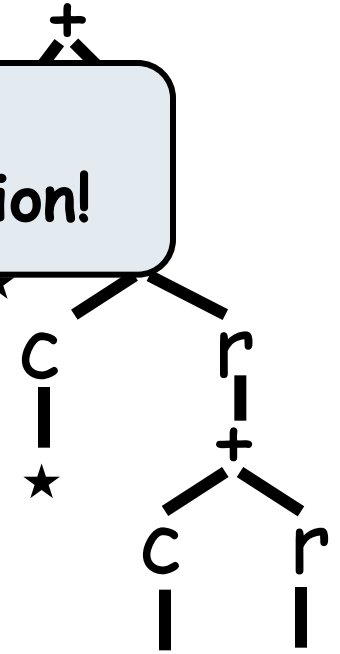
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \bigstar$

CPS Transformation!



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

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

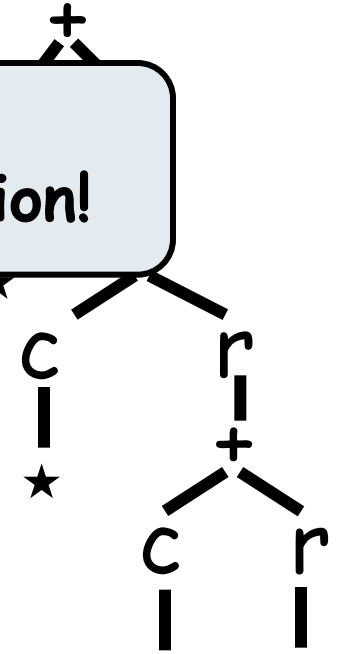
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \star$

CPS Transformation!



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

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

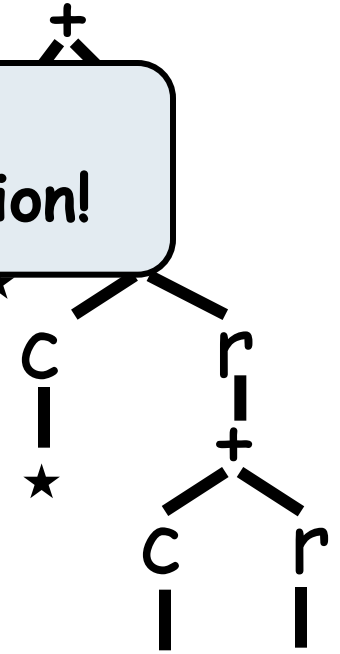
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \bigstar$

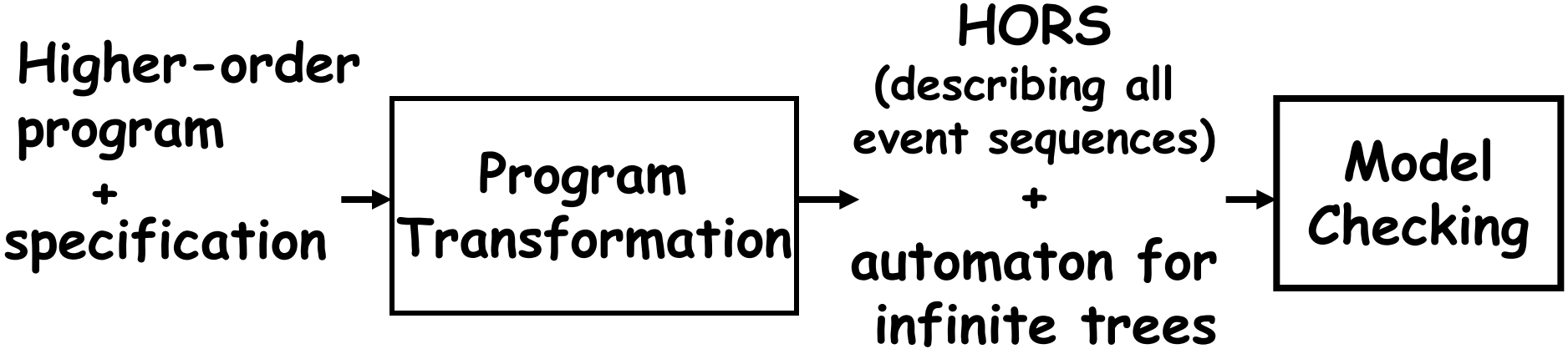
CPS Transformation!



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

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

From Program Verification to HO Model Checking

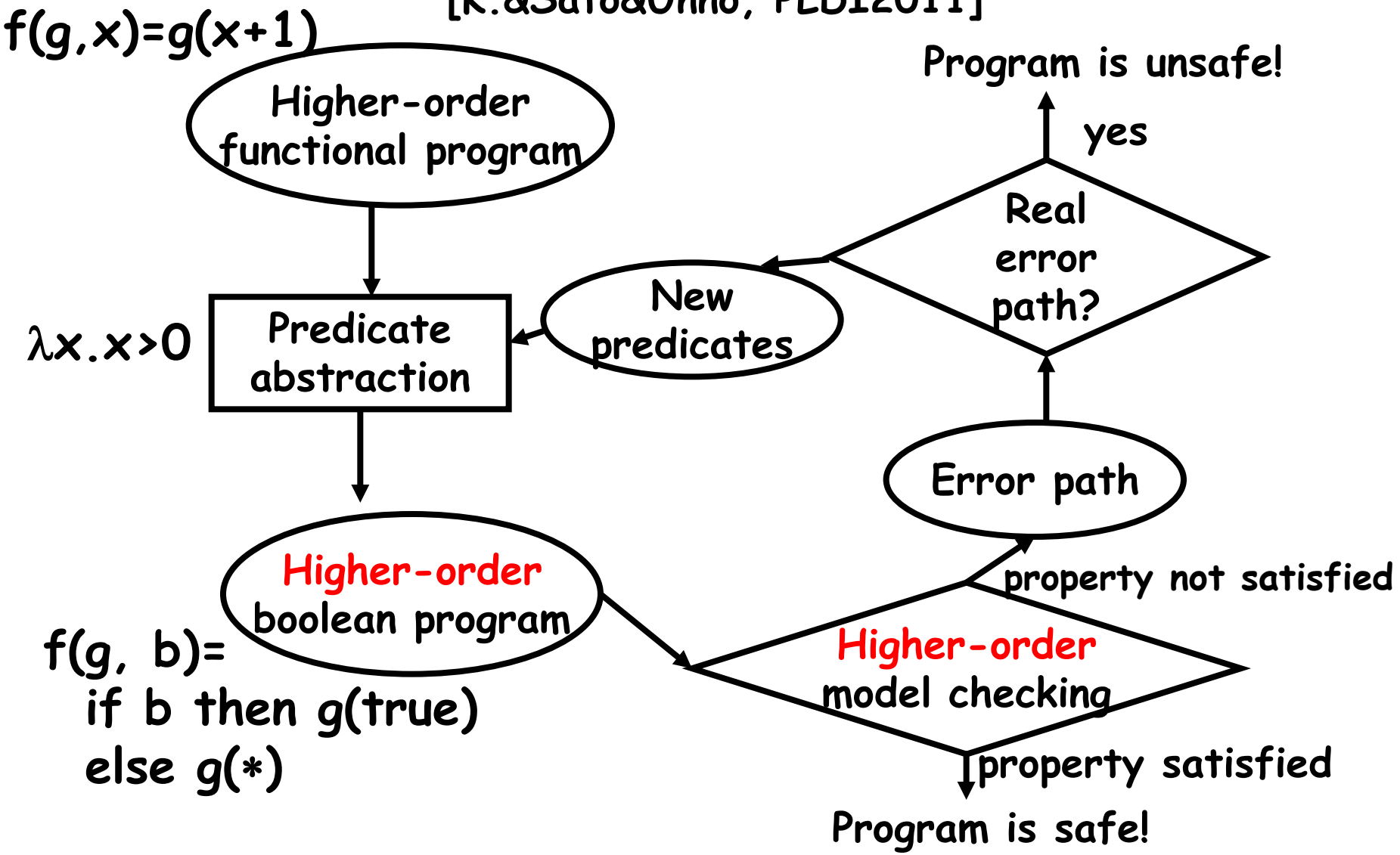


Sound, complete, and automatic for:

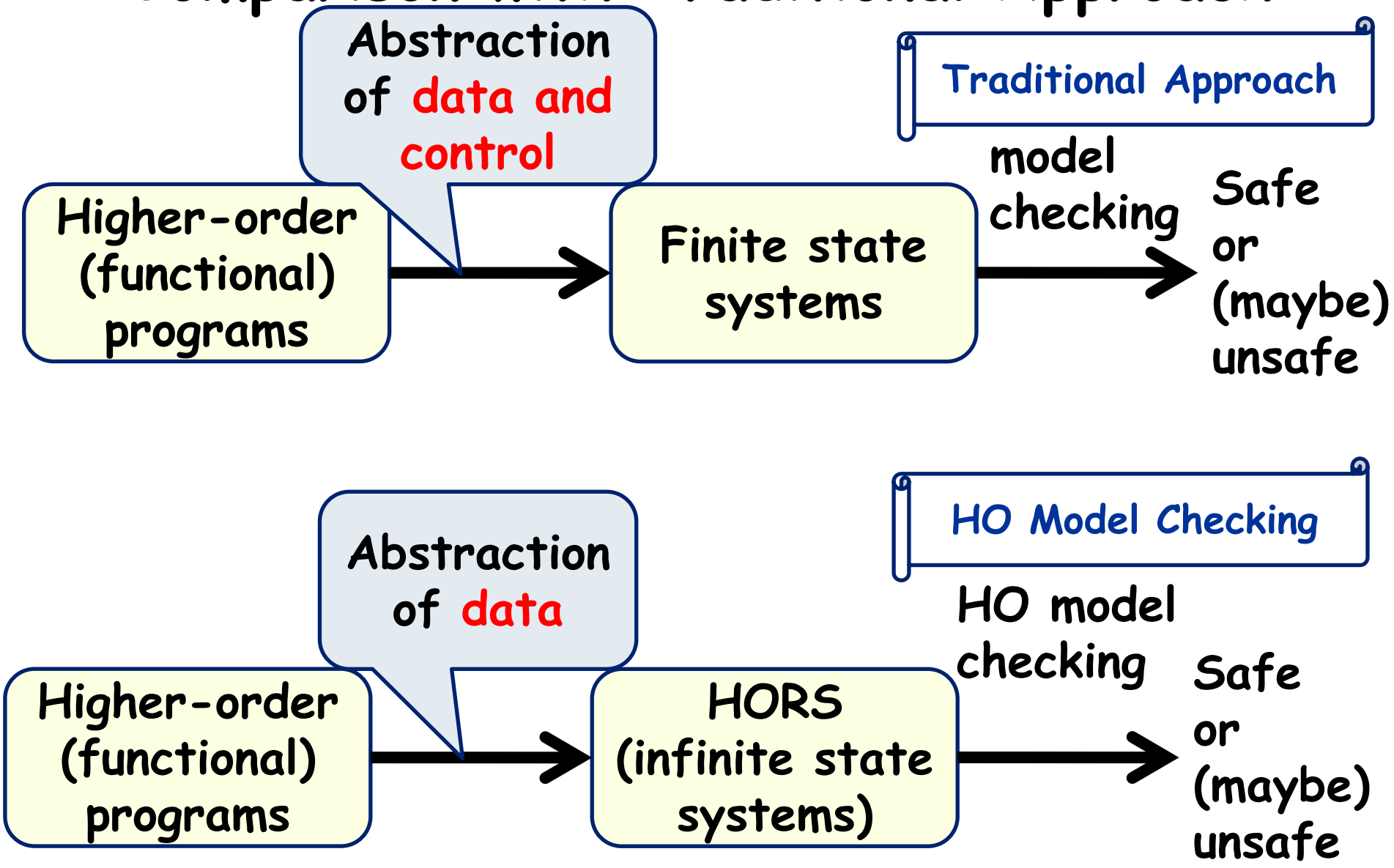
- A large class of higher-order programs:
  simply-typed λ-calculus + recursion
  + finite base types (e.g. booleans) + exceptions + ...
- A large class of verification problems:
  resource usage verification (or typestate checking),
  reachability, flow analysis, strictness analysis, ...

Predicate Abstraction and CEGAR for Higher-Order Model Checking

[K.&Sato&Unno, PLDI2011]



Comparison with Traditional Approach



Applications to Program Verification/Analysis

- For functional programs:
  - Lack of assertion failures, uncaught exceptions, etc. [K+ PLDI2011][Sato+ PEPM2013]...
  - Tree-processing (e.g. XML processing) programs [K+ POPL10][Ong&Ramsay POPL11]...
  - Termination/non-termination [Kuwahara+ ESOP14, CAV15]
  - Temporal properties [Murase+ POPL16]
  - Exact flow analysis [Tobita+ FLOPS12]

- For multi-threaded programs:
  - Pairwise reachability analysis [Yasukata+ CONCUR14]

Outline

- What is higher-order model checking?
  - higher-order recursion schemes
  - model checking problem
- "Standard" applications to program verification
- Applications to verification of code generators (ongoing work with Igarashi)
- Conclusion

Simple language for cogen

- M ::= x | v | λx.M | $M_1 M_2$ | fix(f,λx.M)
  | if M_1 then M_2 else M_3
  | gensym() (* symbol generation *)
  | abs(M_1, M_2) (* code for abstraction *)
  | app(M_1, M_2) (* code for application *)
  | op(M_1, M_2) (* code for a primitive *)

ordinary constructs for FP

primitives for constructing code

Example:

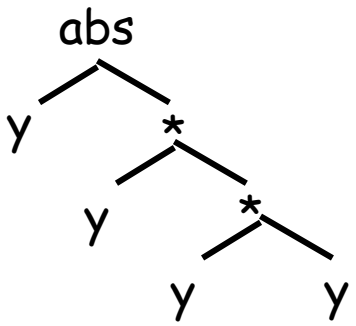
```
let power n x = if n=0 then one
                else *(x, power (n-1) x)
let powergen n = let x = gensym() in
                     abs(x, power n x)
powergen 3 ->*  abs(y, *(y, *(y, *(y, one))))
                (i.e., λy.y*y*y*1)
```

Expected properties for a code generator

- It generates only programs that:
  - are closed
    (i.e. "no undefined variables" error)
  - are well-typed
  - do not fail (e.g. due to assertion failure)
  - return expected values
  - ...

Code generator verification by higher-order model checking?

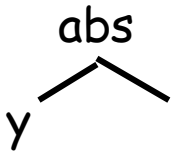
- Model a generated program as a tree



- Code generator is then modeled as a (higher-order) tree grammar *G*
  - let powergen n =
    let x = gensym() in abs(x, power n x)

  => Powergen -> Gensym (λx. abs x (Power x r))

- Model a property on generated programs as a tree automaton *A*
  - e.g. y occurs only below:



- Use HO model checking to check that all the trees generated by *G* are accepted by *A*

Example: Checking Closedness

source code M

```
let power n x =
 if n=0 then one
 else
   *(x, power (n-1) x)
let powergen n =
 let x = gensym() in
   abs(x, power n x)
```

grammar

```
Powergen -> Gensym C
C x -> abs x (Power x)
Power x -> one
Power x ->  * x (Power x)
Gensym k -> ...
   (* pass a symbol to k *)
```

Gensym passes a "fresh" symbol to C

Conditional branch has been replaced by non-deterministic rules; if necessary, use predicate abstraction to take into account the value of n

How can we represent the "fresh" symbol generation?

Example: Checking Closedness

source code M

```
let power n x =
 if n=0 then one
 else
   *(x, power (n-1) x)
let powergen n =
 let x = gensym() in
   abs(x, power n x)
```

grammar G

```
Powergen -> Gensym C
C x -> abs x (Power x)
Power x -> one
Power x ->  * x (Power x)
Gensym k -> k var
Gensym k -> k ig
```

Two names are sufficient for checking the closedness of generated programs (cf. [K, POPL09])

- var: the variable for which closedness should be checked
- ig: variables that should be ignored

By non-deterministically instantiating a fresh symbol to var or ig, we can check that every variable is bound.

Example: Checking Variable Usage

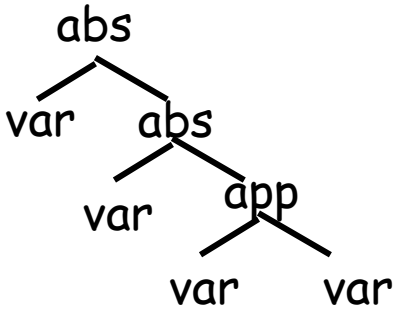
source code M

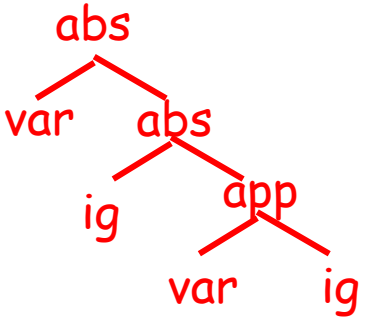
```
let x=gensym() in
let y=gensym() in
   abs(x, abs(y, app(y, x)))
```

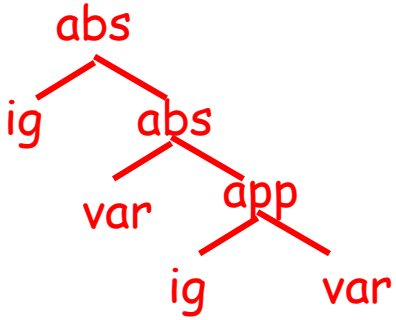
grammar G

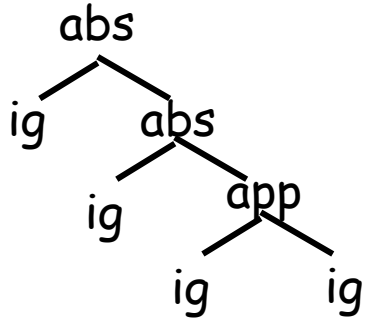
```
S -> Gensym C1
C1 x -> Gensym (C2 x)
C2 x y ->
    abs x (abs y (app y x))
Gensym k -> k var
Gensym k -> k ig
```

Trees generated by G:









M generates only closed programs ⇔
In all the trees generated by G, var occurs only inside (abs var ...)

Expected properties for a code generator

- It generates only programs that:
  - are closed
  - are well-typed
  - do not fail (e.g. due to assertion failure)
  - return expected values
  - ...

We can also check well-typedness, as long as the set of types used in the generated code is finite, and known statically

Example: Checking Well-Typedness

source code M

```
let power n x =
 if n=0 then one
 else
   *(x, power (n-1) x)
let powergen n =
 let x = gensym() in
   abs(x, power n x)
```

grammar

```
Powergen -> Gensym C
C x -> abs x (Power x)
Power x -> one
Power x ->  * x (Power x)
Gensym k -> ...
   (* pass a symbol to k *)
```

Example: Checking Well-Typedness

source code M

```
let power n x =
 if n=0 then one
 else
   *(x, power (n-1) x)
let powergen n =
 let x = gensym() in
   abs(x, power n x)
```

grammar

```
Powergen -> Gensym C
C x -> abs x (Power x)
Power x -> one
Power x ->  * x (Power x)
Gensym k -> k var_int
```

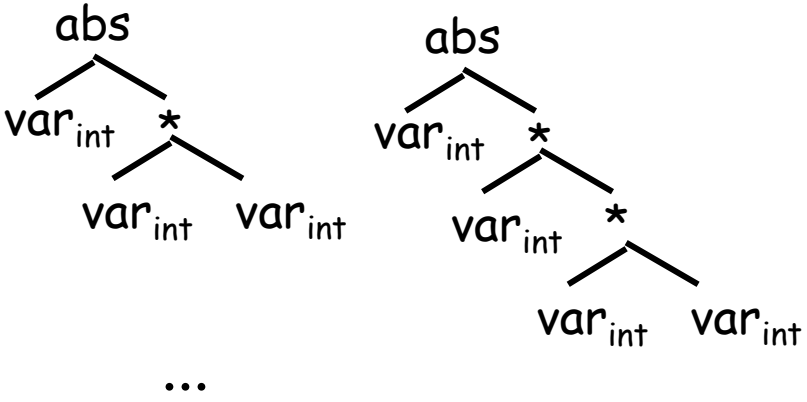
if the type of the generated symbol is known to be int

Example: Checking Well-Typedness

grammar G

Powergen -> Gensym C
C x -> abs x (Power x)
Power x -> one
Power x -> * x (Power x)
Gensym k -> k var_{int}

Trees generated by G:



Tree automaton A for accepting "well-typed" terms
(q_τ : the state for accepting terms of type τ)

q_{int} * -> q_{int} q_{int} $\quad$ q_{int} one -> .

$q_{\sigma\to\tau}$ abs -> $qvar_\sigma$ q_τ $\quad$ q_τ app -> $q_{\sigma\to\tau}$ q_σ (for each σ, τ)

$qvar_\sigma$ var_σ -> .

$$\frac{\vdash M: \sigma\to\tau \quad \vdash N: \sigma}{\vdash MN: \tau}$$

M generates only well-typed programs ⇔
All the trees generated by G are accepted by A

Verifying other properties

- Goa: check that all the generated programs:
  - are closed
  - are well-typed
  - do not fail (e.g. due to assertion failure)
  - return expected values
  - ...

- Design a type system for generated programs (possibly using recursive types, intersection types, etc.)
- Turn the type system into a tree automaton for accepting well-typed terms
- Apply HO model checking

Verification of Multi-Stage Programs?

- Translate a multi-stage program into a program of the single-stage, gensym language

  e.g. from (a variation of) λ° [Davies96] to gensym:

  $tr_0(\lambda x.M) = \lambda x.tr(M)$ $\quad tr_0(M_1M_2) = tr_0(M_1)\ tr_0(M_2)$

  $tr_0(x) = x$ $\quad tr_0(\text{next } M) = tr_1(M)$

  $tr_1(\lambda x.M) = \text{let } x=\text{gensym}() \text{ in } \text{abs}(x, tr_1(M))$ $\quad tr_1(x) = x$

  $tr_1(M_1M_2) = \text{app } (tr_1\ M_1)\ (tr_1\ M_2)$ $\quad tr_1(\text{prev } M) = tr_0(M)$

- Apply the verification method for the gensym language

Any benefit over typed multi-stage languages?

-- they already ensure well-typedness of generated code, etc...

+ more programs are accepted as well-typed

- difficult to support "run"

Conclusion

- HO model checking enables automated verification of functional programs
  - Various properties (including both safety and liveness properties) can be checked by an appropriate combination with abstraction and program transformation
- HO model checking may also be useful for verification of code generators