Applications of Higher-Order Model Checking to Program Verification

Hiroshi Unno
University of Tsukuba
(Joint work with Naoki Kobayashi, Ryosuke Sato, Tachio Terauchi, and Takuya Kuwahara)

Success Story: Software Model Checkers for C

Prove Properties of Program Executions

Program: Specification:

$$P \models \Psi$$

Concurrency

Recursive Procedures

Heap Data Structures

Safety — SLAM, BLAST, MAGIC, …

Termination — TERMINATOR, …

Non-termination — TNT, T2, …

LTL, CTL, fair CTL, CTL* — T2, …

Challenge: How To Construct Software Model Checker for OCaml?

Prove Properties of Program Executions

Program:		Specification:
P	$\models$	Ψ

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures
- Objects & Dyn. Dispatch
- General References

Safety
Termination
Non-termination
LTL, CTL, fair CTL, CTL*

This Tutorial: Software Model Checker MoChi for OCaml based on HOMC

Prove Properties of Program Executions

Program: Specification:

$$P \models \Psi$$

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety

Termination

Non-termination

ω-regular properties

This Tutorial: Software Model Checker MoChi for OCaml based on HOMC

Prove Properties of Program Executions

Program: $P \models \Psi$ Specification:

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety

Termination

Non-termination

ω-regular properties

Tool Demonstration of MoCHi

- Web interface available from: http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/

Overall Flow of Safety Verification

OCaml Program

$\lambda^{\rightarrow}$ + recursion
+ algebraic data types
+ exceptions
+ integers + booleans

↓

Function Encoding [Sato+ '13]

↓

Higher-order Integer Program

$\lambda^{\rightarrow}$ + recursion
+ integers + booleans

↓

Predicate Abstraction + CEGAR [Kobayashi+ '11]

↓

Higher-order Boolean Program

$\lambda^{\rightarrow}$ + recursion
+ booleans

↓

Higher-Order Model Checking (TRecS, HorSat, C-SHORe, Preface,...)

Sound and complete!

Overall Flow of Safety Verification

OCaml Program

↓

Function Encoding [Sato+ ’13]

↓

Higher-order Integer Program

↓

Predicate Abstraction + CEGAR [Kobayashi+ ’11]

↓

Higher-order Boolean Program

↓

Higher-Order Model Checking (TRecS, HorSat, C-SHORe, Preface,…)

OCaml Program: $\lambda^{\rightarrow}$ + recursion + algebraic data types + exceptions + integers + booleans

Higher-order Integer Program: $\lambda^{\rightarrow}$ + recursion + integers + booleans

Higher-order Boolean Program: $\lambda^{\rightarrow}$ + recursion + booleans

Higher-Order Model Checking: Sound and complete!

Higher-Order Model Checking

- A generalization of ordinary model checking :
  - Model the target system as a recursion scheme and check if it satisfies the given specification

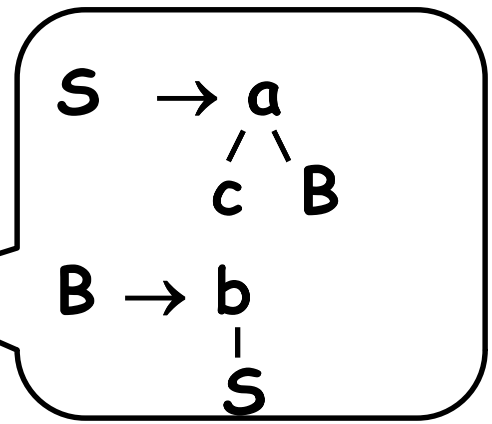
Model Checking	Verification Target
Finite state model checking	Simple loops
Pushdown model checking	First-order recursive functions
Higher-order model checking	Higher-order recursive functions

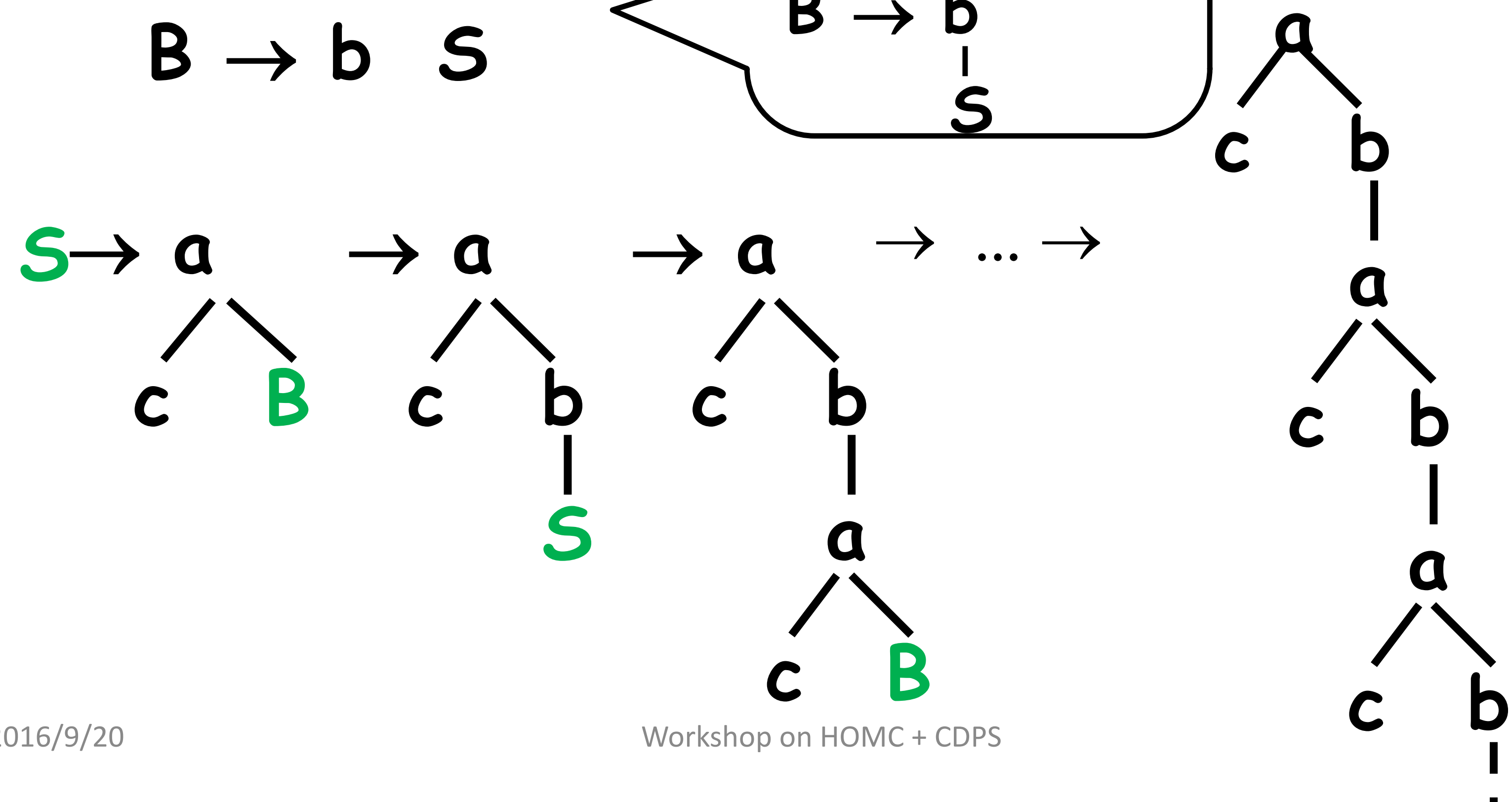
Higher-Order Recursion Scheme (HORS)

- Grammar for generating a possibly infinite tree

Order-0 scheme

S → a c B
B → b S





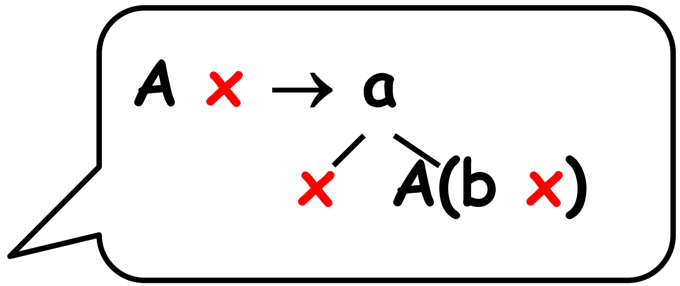
Higher-Order Recursion Scheme (HORS)

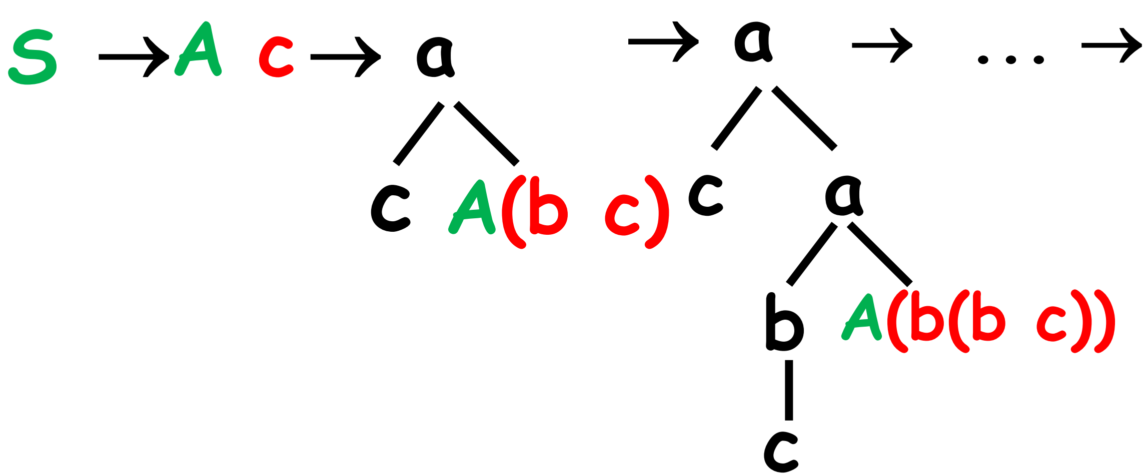
- Grammar for generating a possibly infinite tree

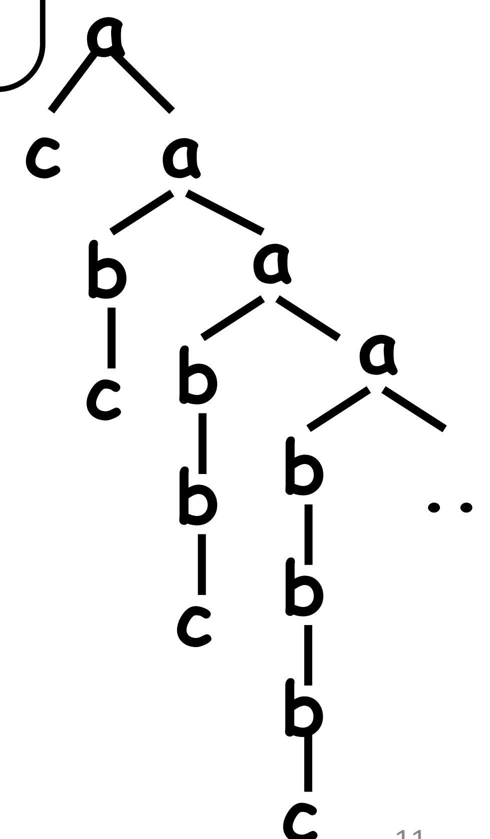
Order-1 scheme

S → A c

A x → a x (A (b x))







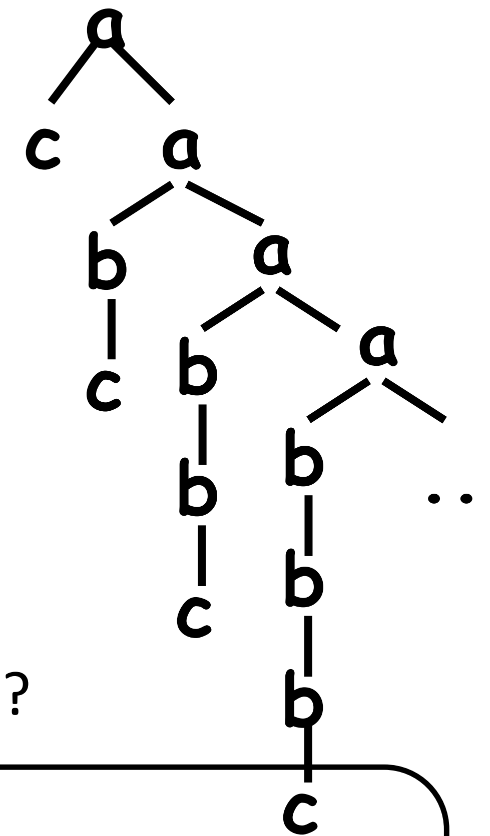
Higher-Order Model Checking

Given

G: **a recursion scheme**

A: **a tree automaton,**

$Tree(G) \in L(A)$**?**



e.g.

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

- Decidable but n-EXPTIME-complete (for order-n recursion scheme) [Ong '06]
- Practical higher-order model checkers have been developed [Kobayashi '09,…]

HORS as a Programming Language

Recursion schemes

$\approx$

Simply-typed λ-calculus

+ recursion

+ tree constructors (but no destructors)

(+ finite data domains such as booleans)

From Program Verification to Higher-Order Model Checking [Kobayashi ’09]

Higher-order boolean program + specification (on events or output)

→ **Program Transformation** →

HORS (describing all event sequences or outputs) + Tree automaton recognizing valid event sequences or outputs

→ **Model Checking**

Example: Fr… Higher-…

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

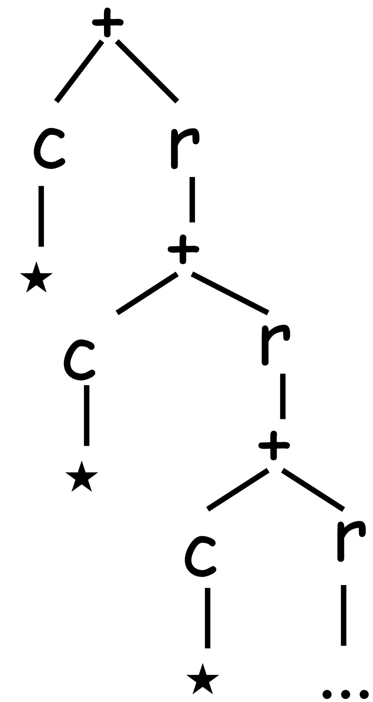
```
let rec f(x) =
   if * then close(x)
   else (read(x); f(x))
in
let y = open "foo"
in
      f (y)
```

→

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

$S \rightarrow F\ d\ \star$

CPS Transformation!



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

→

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

Example: Fr
Higher-

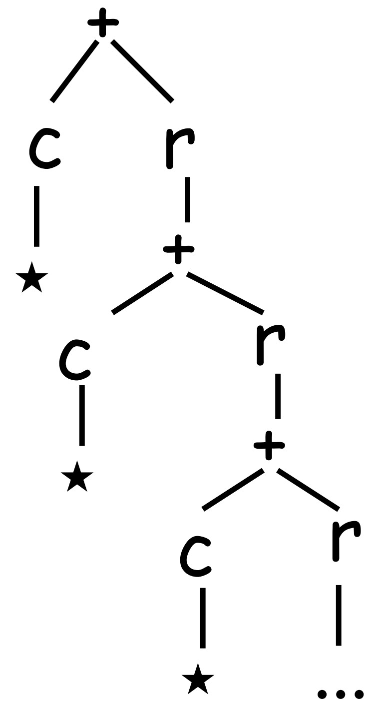
continuation parameter, expressing how “foo” is accessed after the call returns

```
let rec 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?

Example: Fr… Higher-…

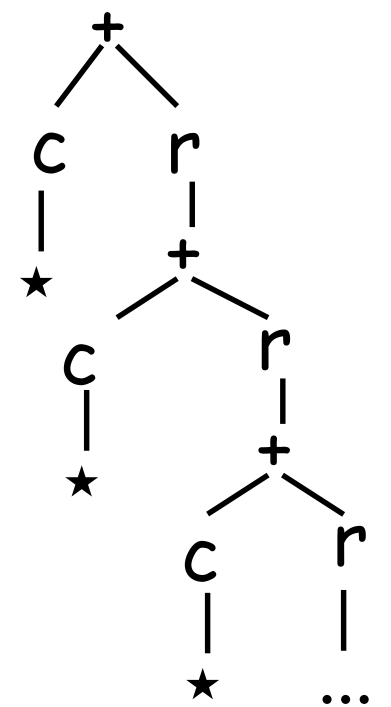
continuation parameter, expressing how "foo" is accessed after the call returns

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

CPS Transformation!

F x k → + (c k) (r (F x k))

S → F d ★



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

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

Example: Fr

Higher-

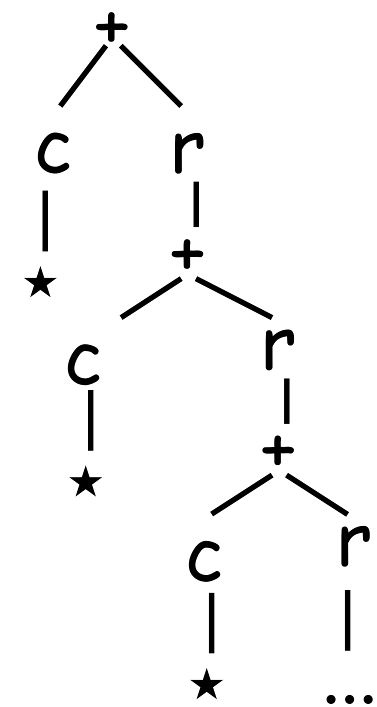
continuation parameter,
expressing how “foo” is accessed
after the call returns

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

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

$S \rightarrow F\ d\ \star$

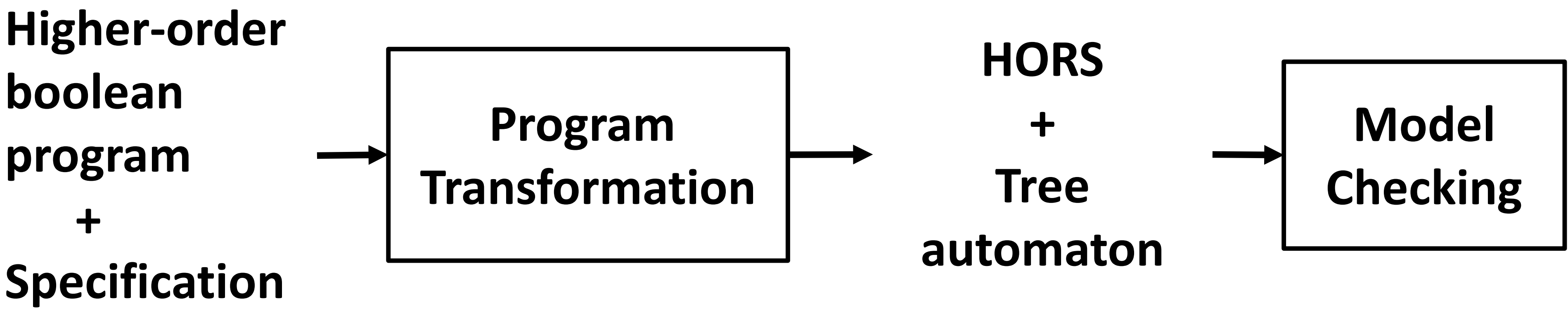
CPS
Transformation!



Is the file “foo” accessed according to read* close?

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

Program Verification based on Higher-Order Model Checking [Kobayashi '09]



Sound, complete, and automatic for:

- Simply-typed λ-calculus + recursion
  + tree constructors (but no destructors)
  + finite data domains (e.g. booleans)
  (but not for infinite data domains!)
- A large class of verification problems:
  resource usage verification, reachability, flow analysis, ...

Overall Flow of Safety Verification

OCaml Program

↓

Function Encoding [Sato+ '13]

↓

Higher-order Integer Program

↓

Predicate Abstraction
+ CEGAR [Kobayashi+ '11]

↓

Higher-order Boolean Program

↓

Higher-Order Model Checking
(TRecS, HorSat, C-SHORe, Preface,...)

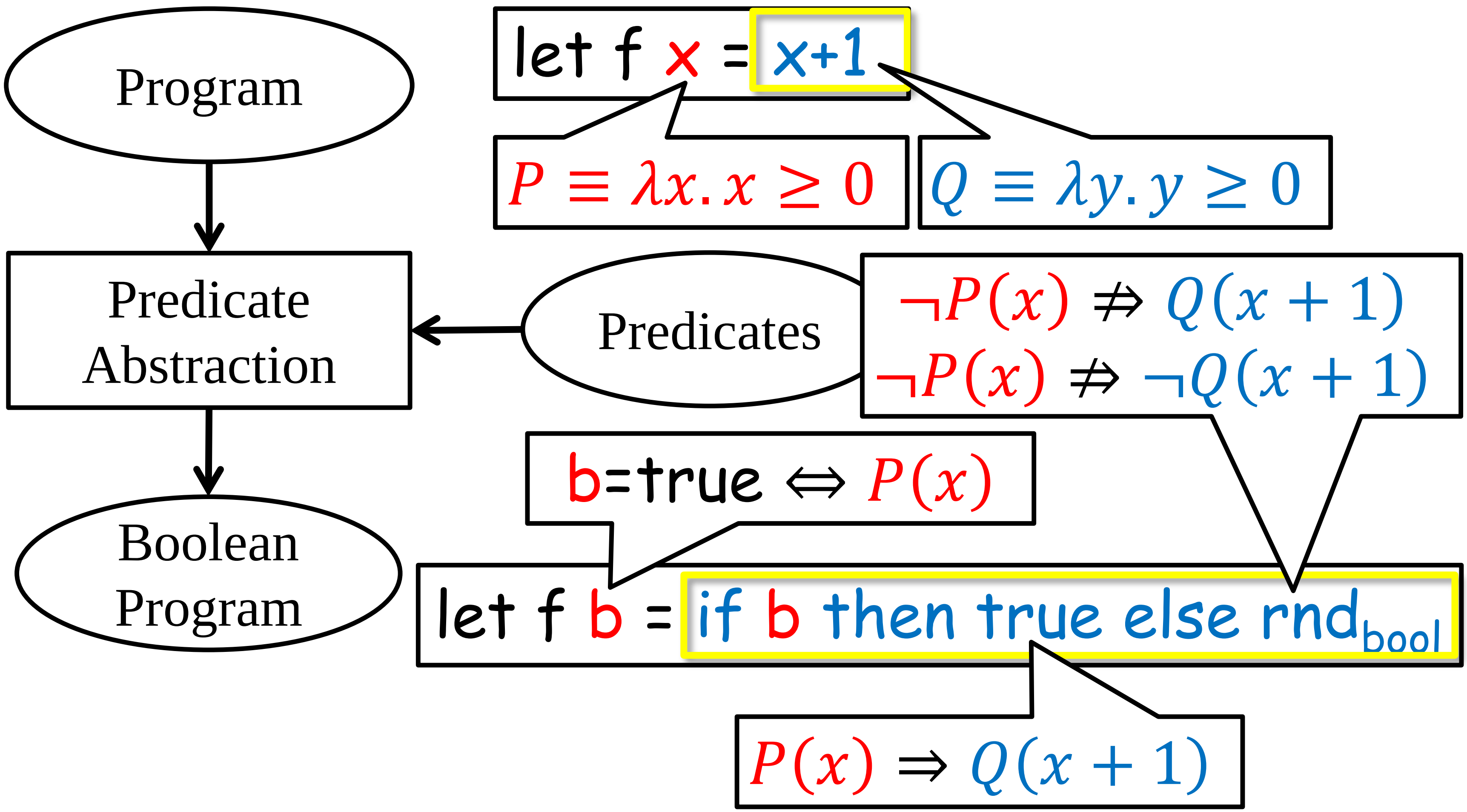
OCaml Program: $\lambda^{\rightarrow}$ + recursion
+ algebraic data types
+ exceptions
+ integers + booleans

Higher-order Integer Program: $\lambda^{\rightarrow}$ + recursion
+ integers + booleans

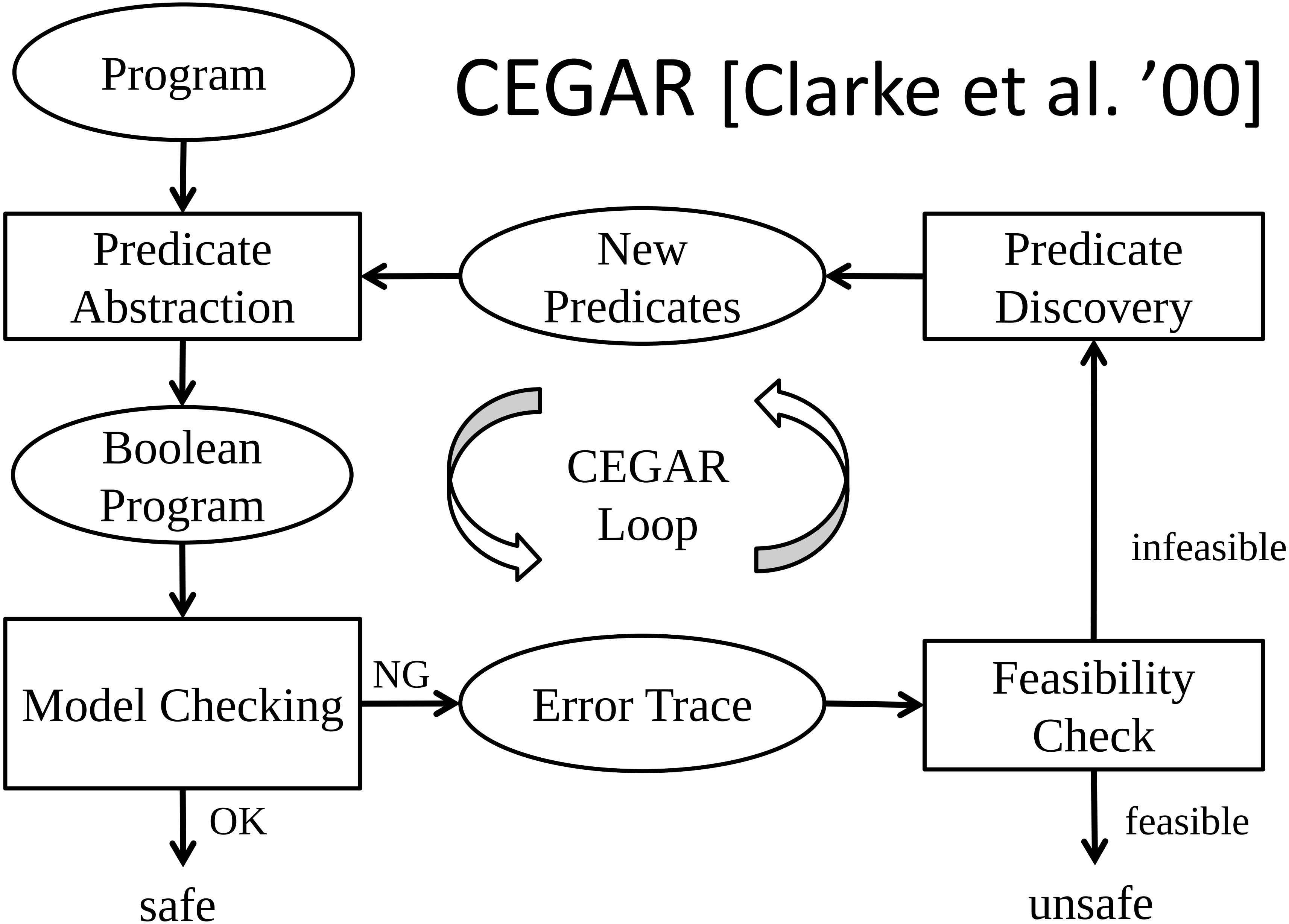
Higher-order Boolean Program: $\lambda^{\rightarrow}$ + recursion
+ booleans

Higher-Order Model Checking: Sound and complete!

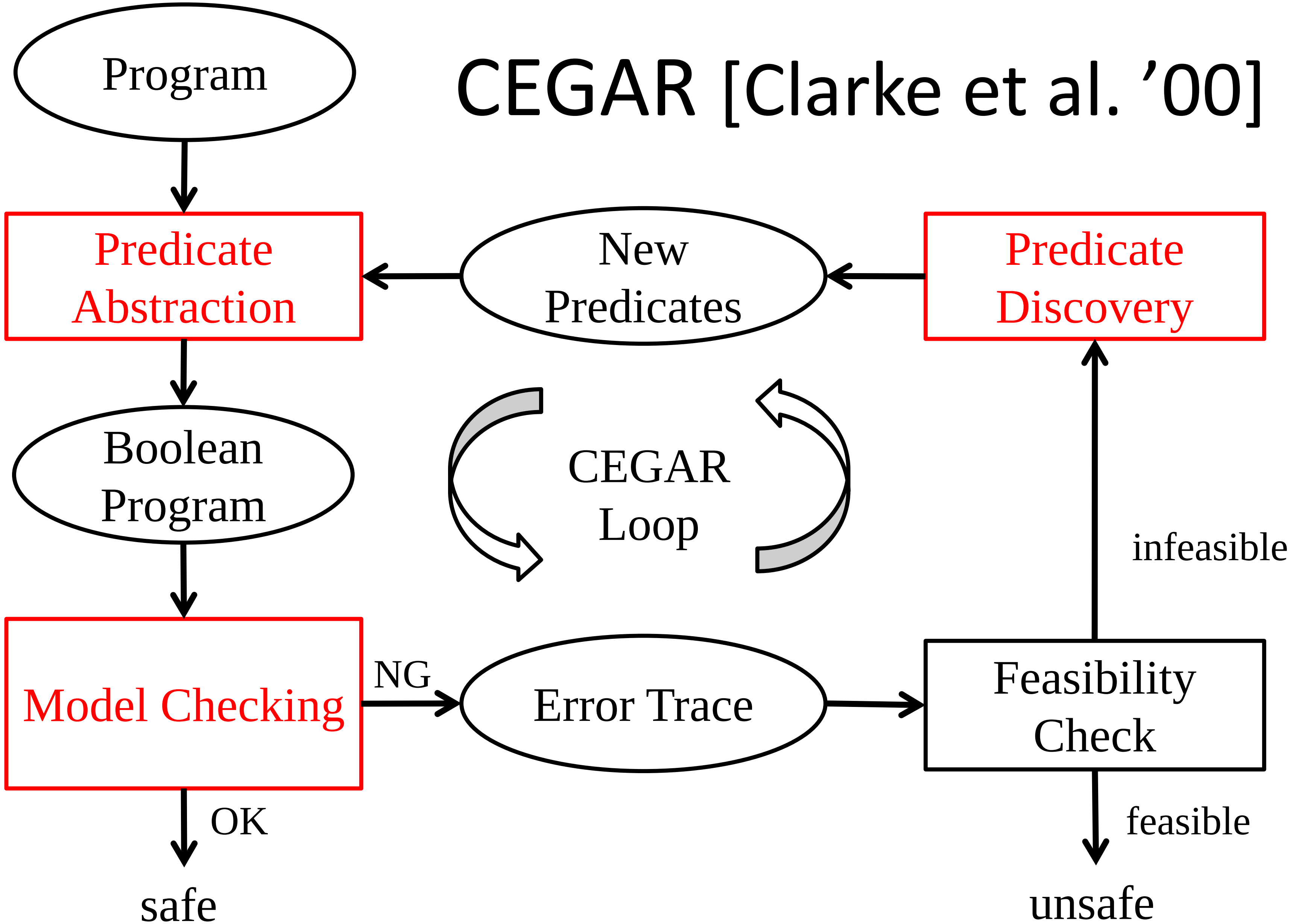
Predicate Abstraction [Graf & Saidi '97]



CEGAR [Clarke et al. ’00]



CEGAR [Clarke et al. '00]



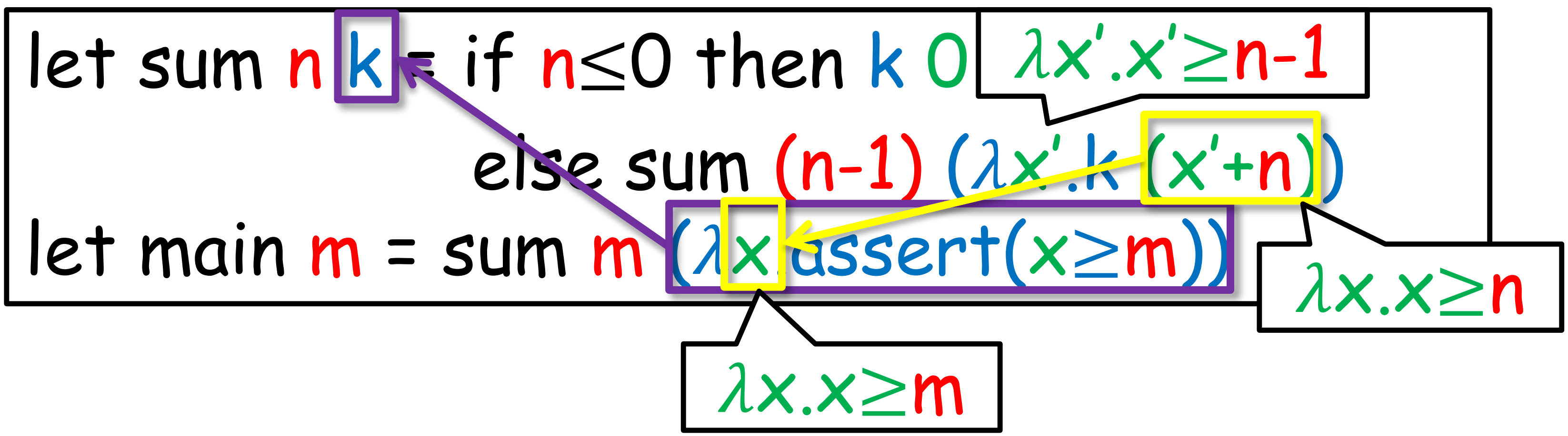
Challenges in Higher-Order Setting

- Model Checking
  - How to precisely analyze higher-order control flows?

  ⇒ Higher-order model checking!
- Predicate Abstraction
  - How to ensure consistency of abstraction?
- Predicate Discovery
  - How to find new predicates that can eliminate an infeasible error trace from the abstraction?

Challenges in Higher-Order Setting

- Predicate Abstraction
  - How to ensure consistency of abstraction?



Our Solution: Abstraction Types

- Specify which predicates should be used for abstraction of each expression
- $\mathrm{int}[P_1, \ldots, P_n]$
  Int. exps. that should be abstracted by $P_1, \ldots, P_n$
  e.g., $3 : \mathrm{int}[\lambda x. x > 0, even?] \rightsquigarrow (\mathrm{true}, \mathrm{false})$
- $(x : \mathrm{int}[P_1, \ldots, P_n]) \to \mathrm{int}[Q_1, \ldots, Q_m]$
  Assuming that argument x is abstracted by $P_1, \ldots, P_n$,
  abstract the return value by $Q_1, \ldots, Q_m$

Example: Abstraction Types

```
let sum n k = if n≤0 then k 0
              else sum (n-1) (λx'.k (x'+n))
let main m = sum m (λx.assert(x≥m))
```

λx'.x'≥n-1

λx.x≥n

λx.x≥m

$$\text{sum}: (n:\text{int}[\,]) \rightarrow (\text{int}[\lambda x.x \geq n] \rightarrow \star) \rightarrow \star$$

no predicates for n

predicate for abstracting the 1st argument of k

Unit type

Example: Predicate Abstraction

```
let sum n k = if n≤0 then k 0
                 else sum (n-1) (λx'.k (x'+n))
let main m = sum m (λx.assert(x≥m))
```

λx'.x'≥n-1

n>0

λx.x≥n

```
sum: (n:int[]) → (int[λx.x≥n] → ⋆) → ⋆
```

```
let sum () k =
  if * then k true
        else sum () (λb'.k (if b' then true else rnd_bool))
let main () = sum () (λb. assert(b))
```

x'≥n-1 ∧ n>0 ⇒ x'+n≥n

Successfully model checked!

Type-Directed Predicate Abstraction

HO Int Expression

Abstraction Type

$$\Gamma \vdash M : \tau \rightsquigarrow t$$

Abstraction Type Environment

HO Bool Expression

$$\frac{\Gamma \vdash M : \tau' \to \tau \rightsquigarrow s \qquad \Gamma \vdash N : \tau' \rightsquigarrow t}{\Gamma \vdash M\,N : \tau \rightsquigarrow s\,t}$$

Predicate Abstraction Rule for Function Applications

Challenges in Higher-Order Setting

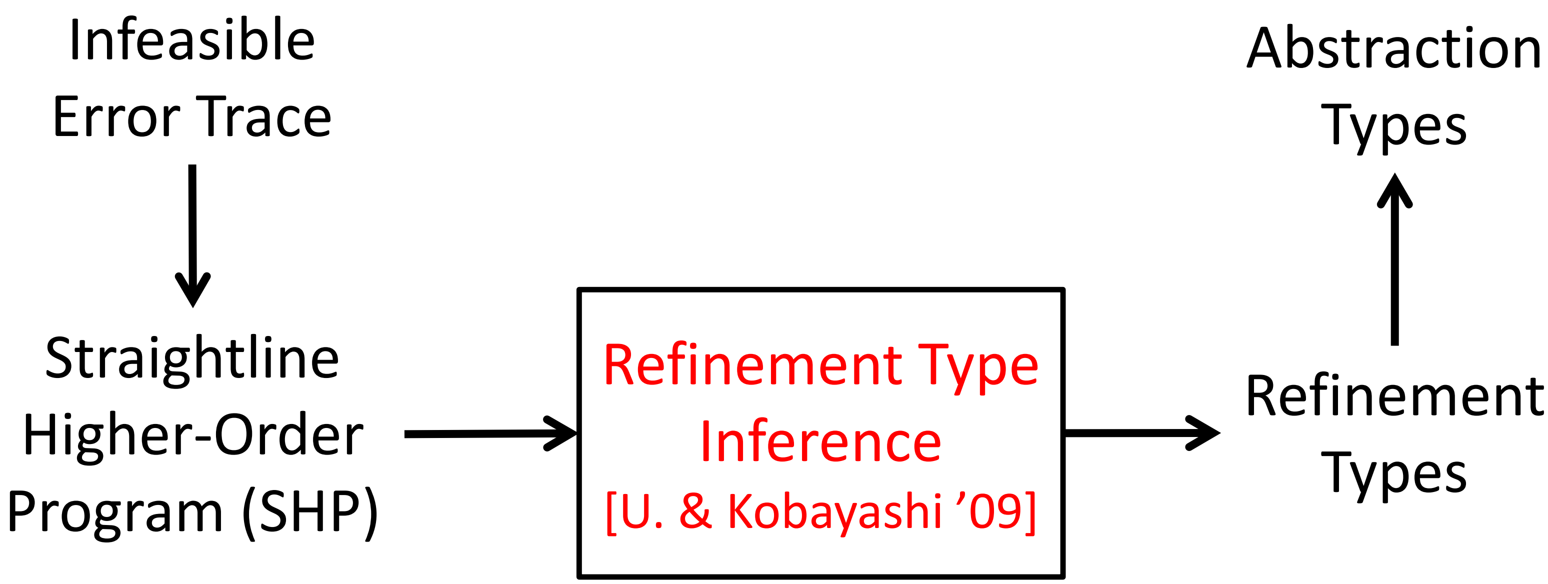
- Predicate Discovery
  - How to find new predicates that can eliminate an infeasible error trace from the abstraction?

Challenges in Higher-Order Setting

- Predicate Discovery
  - How to find abstraction types that can eliminate an infeasible error trace from the abstraction?

Our Solution

- Reduction to refinement type inference of a straightline higher-order program (SHP)



Refinement Types [Xi & Pfenning '98, '99]

- $\{x : \mathrm{int} \mid x \geq 0\}$

  Non-negative integers

- $(x : \mathrm{int}) \rightarrow \{r : \mathrm{int} \mid r \geq x\}$

  Functions that take an integer x and return an integer r not less than x

FOL formulas (e.g. QFLIA) for type refinement

Soundness of refinement type system $\vdash_{Ref}$:

P is safe (i.e., $P \not\longrightarrow^{*}$ assert false)

if P is well-typed (i.e., $\exists \Gamma. \Gamma \vdash_{Ref} P$)

Example: Abstraction Type Finding (1/2)

```
let sum n k = if n≤0 then k 0
                   else sum (n-1) (λx'.k (x'+n))
let main m = sum m (λx.assert(x≥m))
```

Infeasible error trace:

```
main m → sum m (λx.assert(x≥m))
→ if m≤0 then (λx.assert(x ≥m)) 0 else …
→ (λx.assert(x≥m)) 0
→ assert(0≥m)
→ fail
```

m≤0

0<m

Example: Abstraction Type Finding (2/2)

```
let sum n k = if n≤0 then k 0
                else sum (n-1) (λx'.k (x'+n))
let main m = sum m (λx.assert(x≥m))
```

main m $\rightarrow^*$ if m≤0... $\rightarrow^*_{m\leq 0}$ assert(0≥m) $\rightarrow_{0<m}$ fail

Straightline Higher-Order Program (SHP):

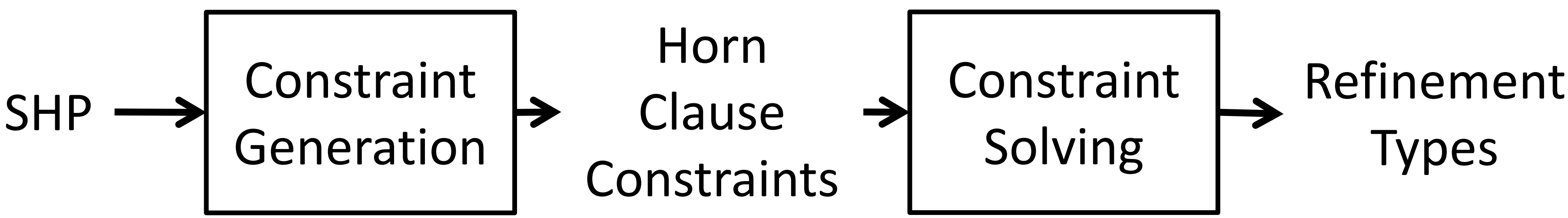
```
let sum n k = assume(n≤0); k 0
let main m = sum m (λx.assume(x<m); fail)
```

[U. & Kobayashi '09]

Abstraction Type:

sum: (n:int[]) → (int[λx.x≥n] → ⋆) → ⋆

Refinement Type Inference [U. & Kobayashi '09]



Example: Constraint Generation

Straightline Higher-Order Program (SHP):

```
let sum n k = assume(n≤0); k 0
let main m = sum m (λx.assume(x<m); fail)
```

Refinement Type Templates:

$$\text{sum}: (n:\{n:\text{int} \mid P(n)\}) \to (\{x:\text{int} \mid Q(n,x)\} \to \star) \to \star$$

Horn Clause Constraints:

$$\top \Rightarrow P(m)$$

$$P(n) \land n \le 0 \land x=0 \Rightarrow Q(n,x)$$

$$P(m) \land Q(m,x) \land x<m \Rightarrow \bot$$

Example: Constraint Solving (1/2)

Horn Clause Constraints:

$$\top \Rightarrow P(m)$$
$$P(n) \wedge n \leq 0 \wedge x=0 \Rightarrow Q(n,x)$$
$$P(m) \wedge Q(m,x) \wedge x<m \Rightarrow \bot$$

↓

Horn Clause Constraints with P eliminated:

$$n \leq 0 \wedge x=0 \Rightarrow Q(n,x)$$
$$Q(n,x) \Rightarrow (n=m \Rightarrow x \geq m)$$

↓ Interpolating Prover

Solution: $Q(n,x) \equiv x \geq n$

Interpolating Prover

- Input: ϕ_1, ϕ_2 such that $\phi_1 \Rightarrow \phi_2$
- Output: an ***interpolant*** ϕ of ϕ_1, ϕ_2 such that:
  1. $\phi_1 \Rightarrow \phi$
  2. $\phi \Rightarrow \phi_2$
  3. $FV(\phi) \subseteq FV(\phi_1) \cap FV(\phi_2)$
- Example: x≥n is an interpolant of:

  n≤0 ∧ x=0 and n=m ⇒x≥m

Example: Constraint Solving (2/2)

Horn Clause Constraints:

$$\top \Rightarrow P(m)$$
$$P(n) \land n \leq 0 \land x = 0 \Rightarrow Q(n,x)$$
$$P(m) \land Q(m,x) \land x < m \Rightarrow \bot$$

↓ Substitute $Q(n,x)$ with $x \geq n$

Horn Clauses with P1 substituted:

$$\top \Rightarrow P(m)$$
$$P(n) \Rightarrow (n \leq 0 \land x = 0 \Rightarrow x \geq n)$$

↓ Interpolating Prover

Solution: $P(n) \equiv \top$

Example: Refinement Type Inference

Straightline Higher-Order Program (SHP):

let sum n k = assume(n≤0); k 0
let main m = sum m (λx.assume(x<m); fail)

Refinement Type Templates:

sum: (n:{n:int|P(n)}) →
  ({x:int|Q(n,x)} → ⋆) → ⋆

Refinement Types of SHP:

sum: (n:{n:int|T}) →
  ({x:int|x≥n} → ⋆) → ⋆

Overall Flow of Safety Verification

OCaml Program

↓

Function Encoding [Sato+ '13]

↓

Higher-order Integer Program

↓

Predicate Abstraction
+ CEGAR [Kobayashi+ '11]

↓

Higher-order Boolean Program

↓

Higher-Order Model Checking
(TRecS, HorSat, C-SHORe, Preface,...)

$\lambda^{\rightarrow}$ + recursion
+ algebraic data types
+ exceptions
+ integers + booleans

$\lambda^{\rightarrow}$ + recursion
+ integers + booleans

$\lambda^{\rightarrow}$ + recursion
+ booleans

Sound and complete!

Function Encoding of Lists

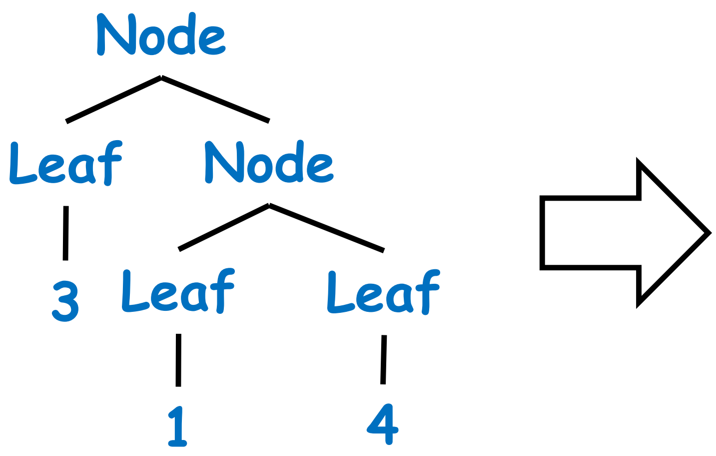
- Encode a list as a pair (len, f) such that:
  - len is the length of the list
  - f is a function from an index i to the i-th element
    - e.g., [3;1;4] is encoded as (3, f) where: f(0)=3, f(1)=1, f(2)=4, and undefined otherwise

```
let nil = (0, fun i -> ⊥)
let cons a (len, l) = (len + 1, fun i -> if i = 0 then a else l (i - 1))
let hd (len, l) = assert (len ≠ 0); l 0
let tl (len, l) = assert (len ≠ 0); (len - 1, fun i -> l (i + 1))
let is_nil (len, l) = len = 0
```

Function Encoding of Algebraic Data Structures

- Encode an algebraic data structure as a function from the path of a node to its label

```
type btree = Leaf of int | Node of btree * btree
```



A function f such that:

f [] = Node

f [1] = Leaf f [2] = Node

f [1;1] = 3

f [2;1] = Leaf f [2;2] = Leaf

f [2;1;1] = 1 f [2;2;1] = 4

Function Encoding of Exceptions

```
exception NotPos

let rec fact n =
  if n ≤ 0 then
    raise NotPos
  else
    try
      n × fact (n-1)
    with NotPos -> 1
```

CPS Trans.

```
type exc = NotPos

let rec fact n k exn =
  if n ≤ 0 then
    exn NotPos
  else
    fact (n-1)
      (fun r -> k (n × r))
      (fun NotPos -> k 1)
```

Summary: Safety Verification by MoChi

- For finite-data HO programs: sound, complete, and fully-automatic verification by reduction to HO model checking [Kobayashi '09]
- For infinite-data HO programs: sound and automatic (but incomplete) verification by a combination of:
  - HO model checking
  - predicate abstraction & discovery [Kobayashi+ '11, U.+ '09, '15]
  - program transformation [Sato+ '13]

  Necessarily incomplete but often more precise than other approaches

  Sometimes relatively complete modulo certain assumptions
  - relatively complete refinement type system [U.+ '13]
  - relatively complete predicate discovery [Terauchi & U. '15]

This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

OCaml Program: Specification:

$$P \models \Psi$$

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety

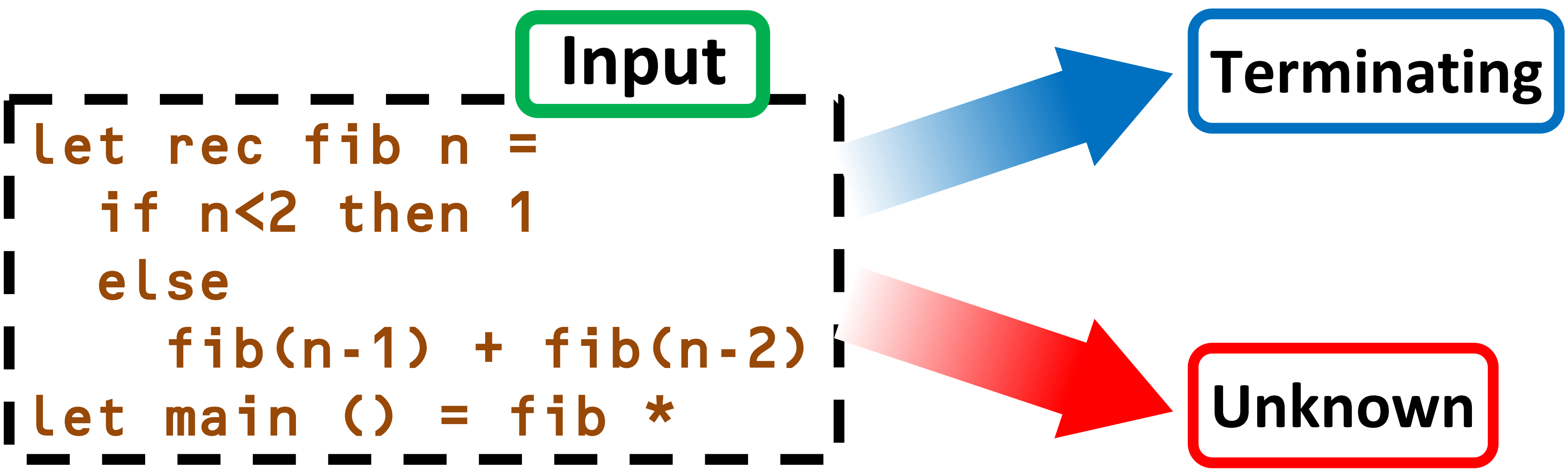
Termination

Non-termination

ω-regular properties

Termination Verification

- Automatically prove that a program terminates for every input (and non-determinism)



Tool Demonstration of MoChi

- Web interface available from: http://www.kb.is.s.u-tokyo.ac.jp/~kuwahara/termination/

1st Naïve Approach to Termination Verification of HO Functional Programs

- Abstract to a finite data HO program,
  and apply HO model checking
- Problem: many terminating programs are turned into non-terminating ones by abstraction

  e.g. f(x) = if x<0 then 1 else 1+f(x-1)   terminating
  → f($b_{x<0}$) = if $b_{x<0}$ then 1 else 1+f(*)   non-terminating

Termination Verification for Imperative Programs

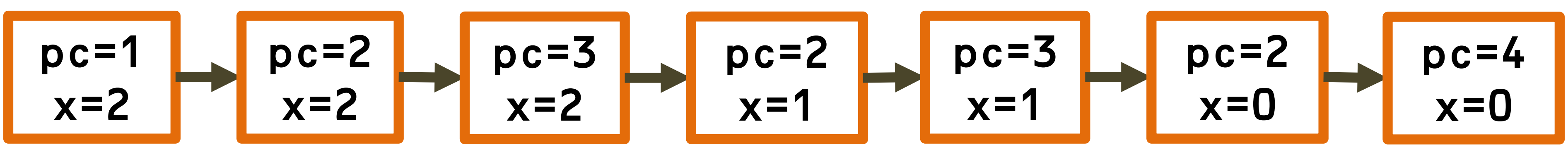
- Binary Reachability Analysis [Cook+ ’06]
  - Theorem [Podelski & Rybalchenko ’04]:
    P is terminating iff
    T^+ is disjunctively well-founded (dwf)
    - T: the transition relation of P
    - dwf: a finite union of well-founded relations

Example: Binary Reachability Analysis

```
1:  x = *;
2:  while(x>0){
3:      x--;
4:  }
```

$$T^{+} \subseteq \{(s, s') \mid s.pc < s'.pc\} \cup \{(s, s') \mid s.pc > s'.pc\} \cup \{(s, s') \mid s.x > s'.x \geqq 0\}$$

Terminating!



2^{nd} Naïve Approach to Termination Verification of HO Functional Programs

- Check that $\rightarrow^+$ is dwf by [Cook+ '06]
  $\rightarrow$: the one-step reduction relation of the HO program P
- Problem: [Cook+ '06] needs to reason about change in calling context / call stack
  – Theorem [Berardi+'14, Yokoyama'14]:
    [Cook+ '06] can only prove termination of primitive recursive functions (when usable wf relations have height at most ω)

2nd Naïve Approach to Termination

```
let rec ack m n =
  if m = 0 then n + 1
  else if n = 0 then ack (m-1) 1
  else ack (m-1) (ack m (n-1))
let main m n = if m > 0 && n > 0 then ack m n
```

Terminates but transition relation is quite complex

- Theorem [Berardi+'14, Yokoyama'14]:
  [Cook+ '06] can only prove termination of primitive recursive functions (when usable wf relations have height at most ω)

Our Solution: Binary Reachability Analysis Generalized to HO [Kuwahara+ '14]

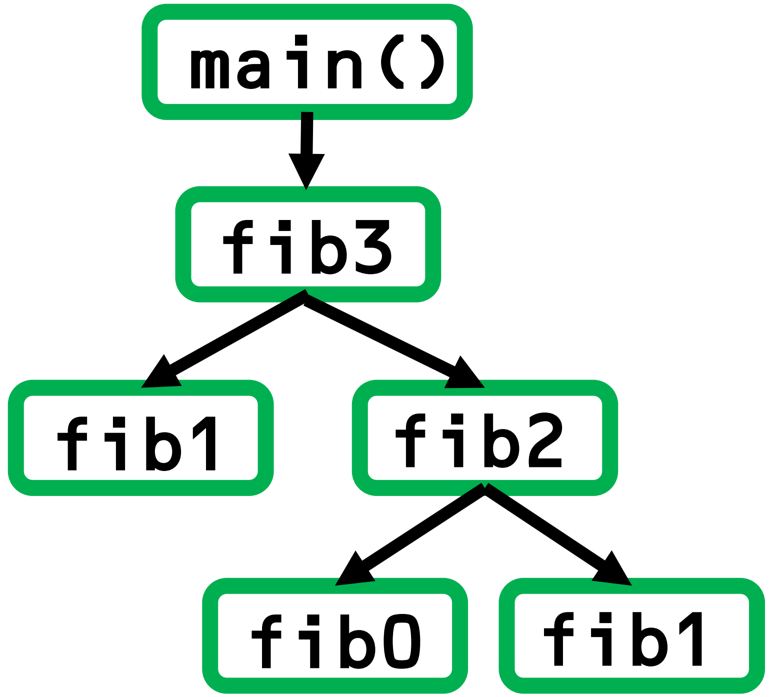
- Theorem [Kuwahara+ '14]:
  HO functional program P is terminating iff $Call_P^+$ is dwf
  - The calling relation $Call_P$ of P:
    $\{(f\,\tilde{v}, g\,\widetilde{w}) \mid g\,\widetilde{w} \text{ is called from } f\,\tilde{v} \text{ in an execution of } P\}$
  - $Call_P^+ = \{(f\,\tilde{v}, g\,\widetilde{w}) \mid main() \to^* E[f\,\tilde{v}], f\,\tilde{v} \to^+ E'[g\,\widetilde{w}]\}$

Example: Generalized Binary Reachability Analysis

```
let rec fib n =
  if n<2 then 1
  else fib (n-1)
     + fib (n-2)
let main()=fib(rand())
```

Call={(fib(n),fib(n-1))|n>1}
∪{(fib(n),fib(n-2))|n>1}
⊆{(fib m,fib n) | m>n≥0}

(Tree representation)



Reduce Binary Reachability to Plain Reachability

- Goal: check $Call_P \subseteq W$ for some dwf W
- Approach: reduction to a safety verification problem by program transformation
  - To each function f, add an extra argument to record the argument of an ancestor call to f
  - Assert that W holds when f is called

```
fib n =
   if n<2 then n
   else fib(n-1)+fib(n-2)
main() = fib(rand())

W = {(m,n) | m>n≥0}
```

⇒

```
fib m n =
  assert(m>n≥0);
  let m'= if * then m else n in
   if n<2 then n
   else fib m' (n-1)+fib m' (n-2)
main() = fib ⊥ (rand())
```

This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

OCaml Program: Specification:

$$P \models \Psi$$

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety

Termination

Non-termination

ω-regular properties

Automata-Theoretic Approach [Vardi'91]

- Input:
  - Program P
  - ω-regular temporal property Ψ

1. Construct ω-automaton $A_{\neg\Psi}$ (with a fairness acceptance condition) that recognizes $L(\neg\Psi)$
2. Construct product program $P \times A_{\neg\Psi}$
3. Verify that $P \times A_{\neg\Psi}$ is fair terminating (i.e., no infinite execution trace that is fair)

Theorem: $P \vDash \Psi$ iff $P \times A_{\neg\Psi}$ is fair terminating

Definition: Fair Termination of P

- Fairness Constraint: $C = \{(A_1, B_1), \ldots, (A_n, B_n)\}$
- Infinite sequence π is **fair** wrt C if $\forall (A, B) \in C$,
  - A occurs only finitely often in π or
  - B occurs infinitely often in π
- P is **fair terminating** wrt C if P has no infinite execution trace that is fair wrt C

Fair Termination Verification for Imperative Programs [Cook+ '07]

- Theorem:
  P is fair terminating wrt C iff $T^{+\upharpoonright C}$ is dwf
  - T: transition relation of P
  - fair transitive closure $R^{+\upharpoonright C}$ of R is defined by:
    $$R^{+\upharpoonright C} = \left\{ (s_1, s_n) \;\middle|\; \begin{array}{l} \forall 1 \leq i < n. (s_i, s_{i+1}) \in R, \\ s_1 \cdots s_n \text{ is fair wrt } C, n \geq 2 \end{array} \right\}$$
    (Intuitively means the subset of R^+ that is fair wrt C)
    - Finite sequence $s_1 \cdots s_n$ is **fair** wrt C if $\forall (A, B) \in C$, A does not occur in $s_1 \cdots s_n$ or B occurs in $s_1 \cdots s_n$

1st Naïve Approach to Fair Termination Verification of HO Functional Programs

- Check that $\rightarrow^{+\restriction C}$ is dwf
  - $\rightarrow$: the one-step reduction relation of the HO program P
- Suffers from the same problem as the 1st naïve approach to plain termination verification of HO functional programs:
  - [Cook+ '07] needs to reason about change in calling context / call stack

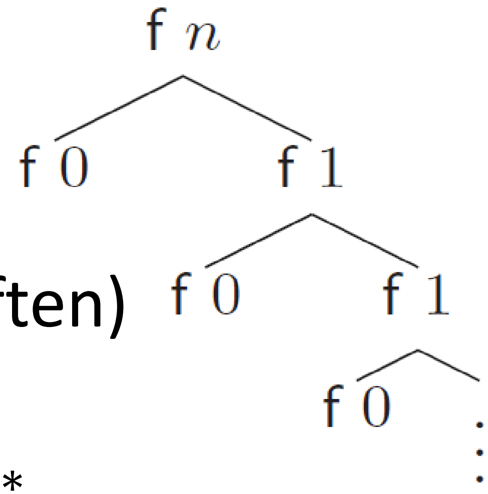
2^{nd} Naïve Approach to Fair Termination Verification of HO Functional Programs

- Check that $Call_P^{+\restriction C}$ is dwf
- Unsound: There is a case that $Call_P^{+\restriction C}$ is dwf but P is not fair-terminating wrt C
  - For example,

    f x = if x ≤ 0 then () else (f 0; f 1)

    $$C = \{(\text{true}, f\ 0)\}$$

    (fair wrt C iff f 0 is called infinitely often)



$f\ 2 \to^* f\ 0; f\ 1 \to^* f\ 1 \to^* f\ 0; f\ 1 \to^*$ ...

Our Solution: Fair-Termination Analysis Generalized to HO Programs [Murase+ '16]

- Check disjunctive well-foundedness of $\rhd_P^C$:
  $\{(f\,\tilde{v}, g\,\widetilde{w}) \mid main() \to^* E[f\,\tilde{v}], f\,\tilde{v} \to^{+\restriction C} E'[g\,\widetilde{w}]\}$
  - Note that $\rhd_P^C$ is $Call_P^+$ but $\to^+$ replaced by $\to^{+\restriction C}$
- Theorem:
  P is fair-terminating wrt C iff $\rhd_P^C$ is dwf

How to Check that $\rhd_P^C$ is dwf?

- By reduction to a safety verification problem via program transformation similar to the one for binary reachability analysis
  (see our POPL'16 paper [Murase+ '16] for details)

Summary: Plain and Fair Termination Verification by MoCHi

- Naïve combination of HO model checking and predicate abstraction into HO Boolean programs is too imprecise
- Generalize binary reachability analysis to the HO setting by introducing the calling relations $Call_P$ and $\rhd_P^C$

This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

OCaml Program: Specification:

$$P \models \Psi$$

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety

Termination

Non-termination

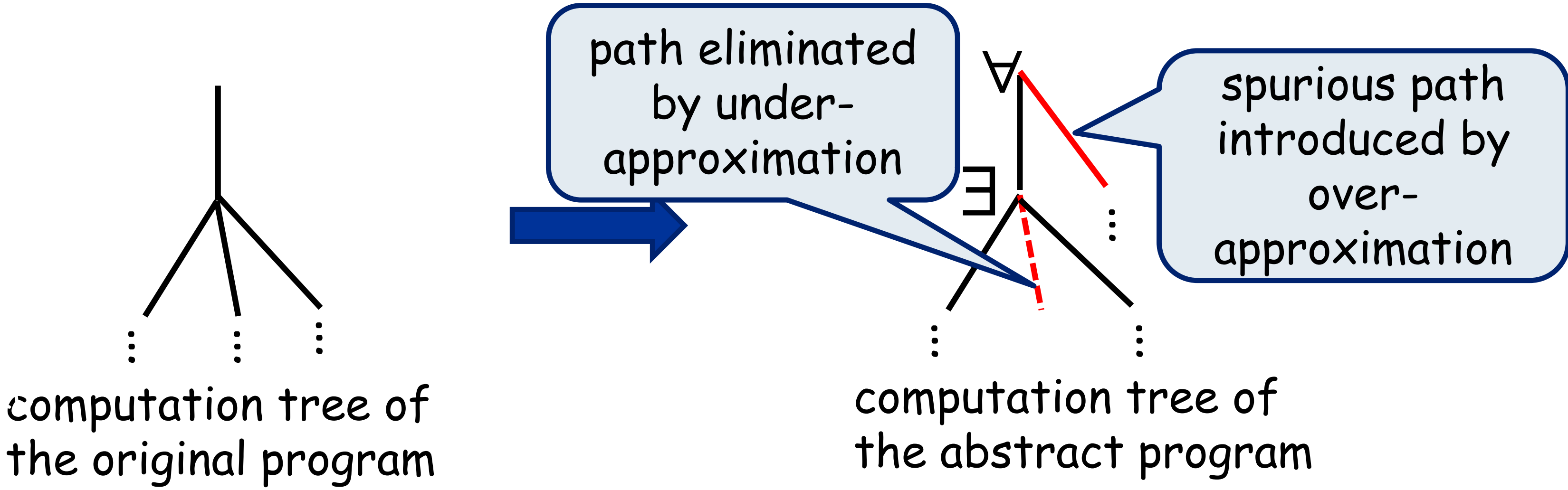
ω-regular properties

Verifying Non-Termination (or Disproving Termination) of HO programs

- Goal: prove that a program is non-terminating for some input (or for some non-deterministic choice)
  - complementary to termination verification

Our approach [Kuwahara+ '15]

- combine over- and under-approximation
  - over-approximate deterministic branches, and check that all the branches are non-terminating
  - under-approximate non-deterministic branches, and check that one of the branches is non-terminating



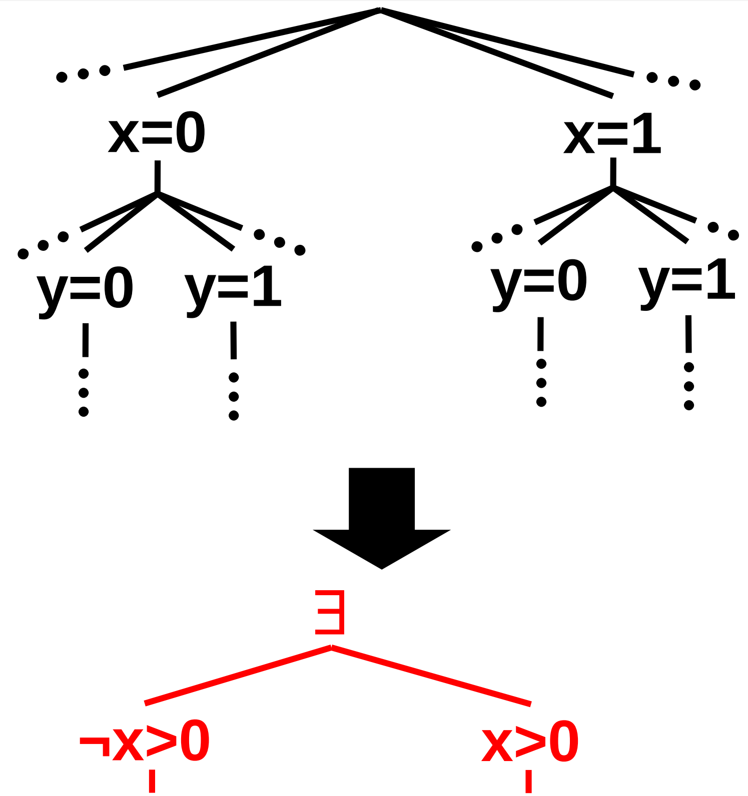
Our Approach: Combination of Under-/Over-approximation

pred: x>0

```
let x=* in
let y=* in
   f(x+y)
```

Only one of the branches needs to be non-terminating

```
∃( ...
   /* case ¬x>0 */
 , ...
   /* case x>0 */
)
```



Our Approach: Combination of Under-/Over-approximation

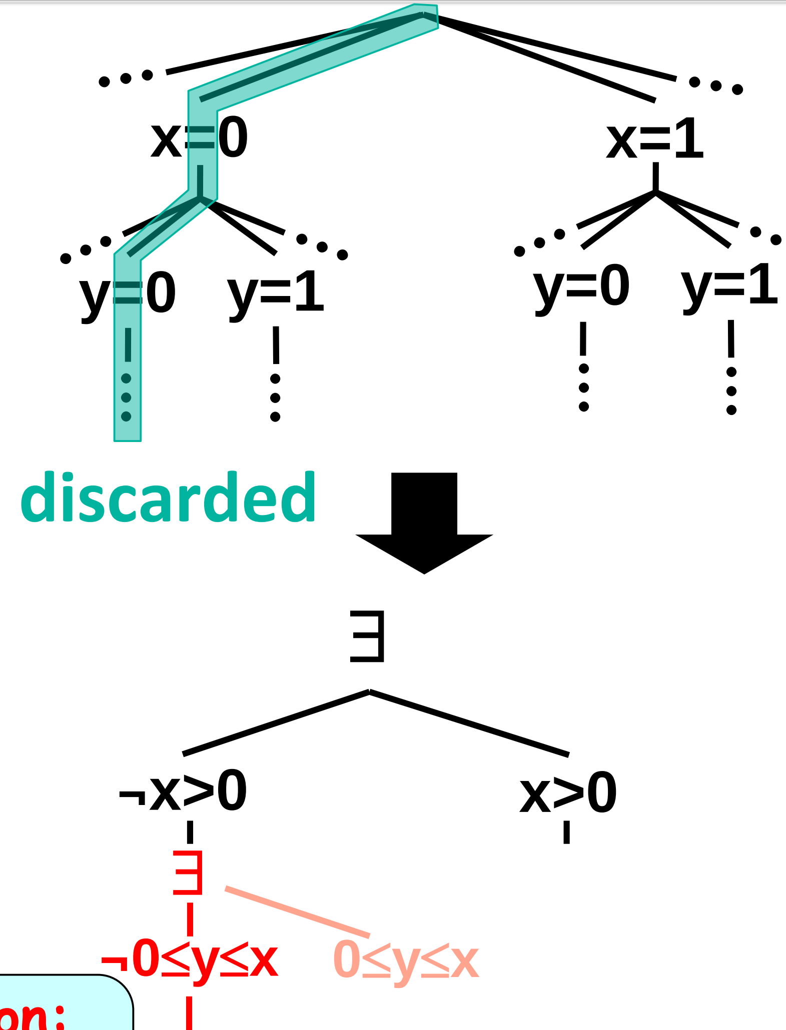
```
let x=* in
let y=* in
   f(x+y)
```

pred: x>0

pred: 0≤y≤x

```
∃( /* case ¬x>0 */
  ∃(...
     /* case ¬0≤y≤x */ )
, ...
)
```

Under-approximation: case for $\neg x>0 \wedge 0 \le y \le x$ is discarded

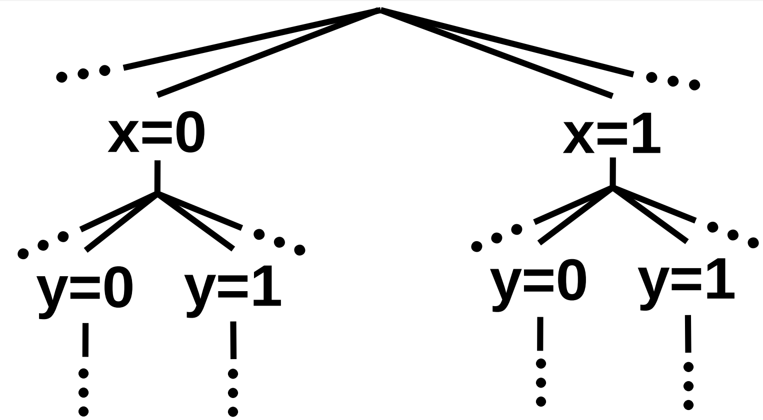


Our Approach: Combination of Under-/Over-approximation

```
let x=* in
let y=* in
   f(x+y)
```

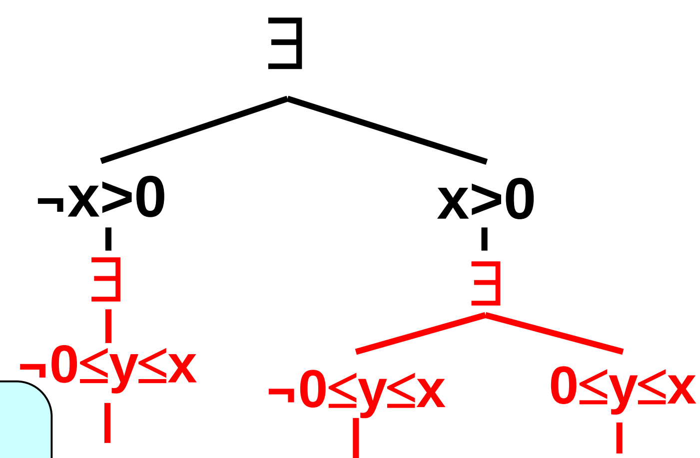
pred: x>0

pred: 0≤y≤x




```
∃( /* case ¬x>0 */
  ∃(...
     /* case ¬0≤y≤x */ )
, ...
)
```

Under-approximation: case for ¬x>0∧ 0≤y≤x is discarded

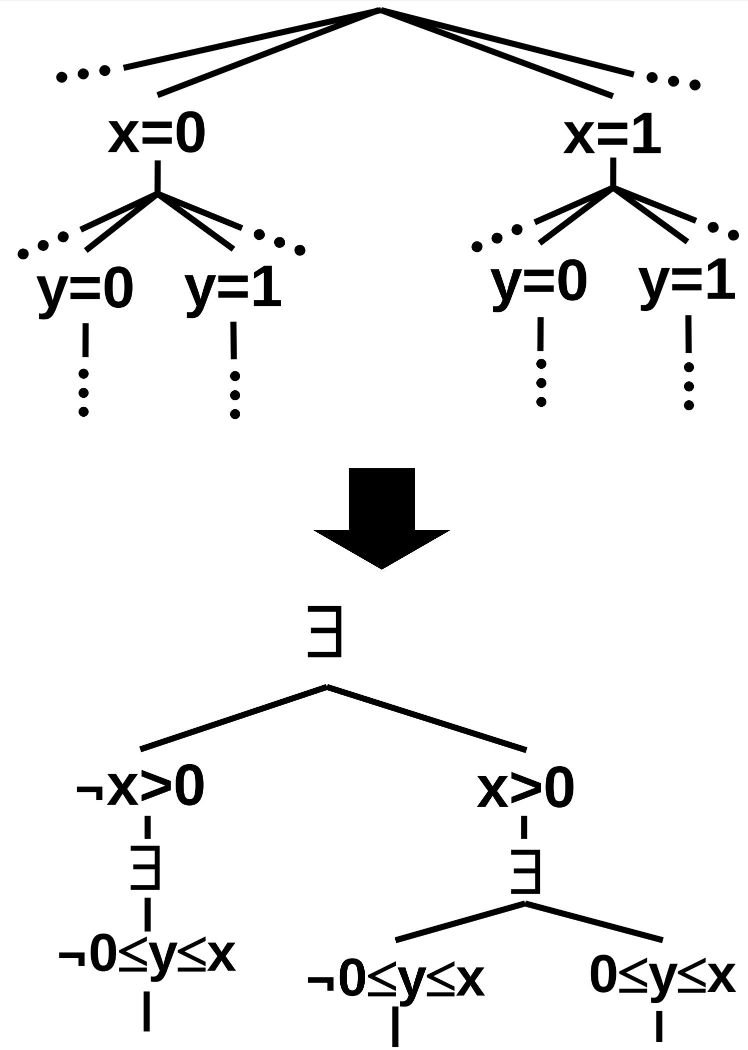


Our Approach: Combination of Under-/Over-approximation

```
let x=* in        pred: x>0
let y=* in        pred: 0≤y≤x
   f(x+y)         pred: x+y>0
```

⬇

```
∃( /* case ¬x>0 */
  ∃(/* case ¬0≤y≤x */
     ... )
, ...
)
```



Our Approach: Combination of Under-/Over-approximation

```
let x=* in
let y=* in
    f(x+y)
```

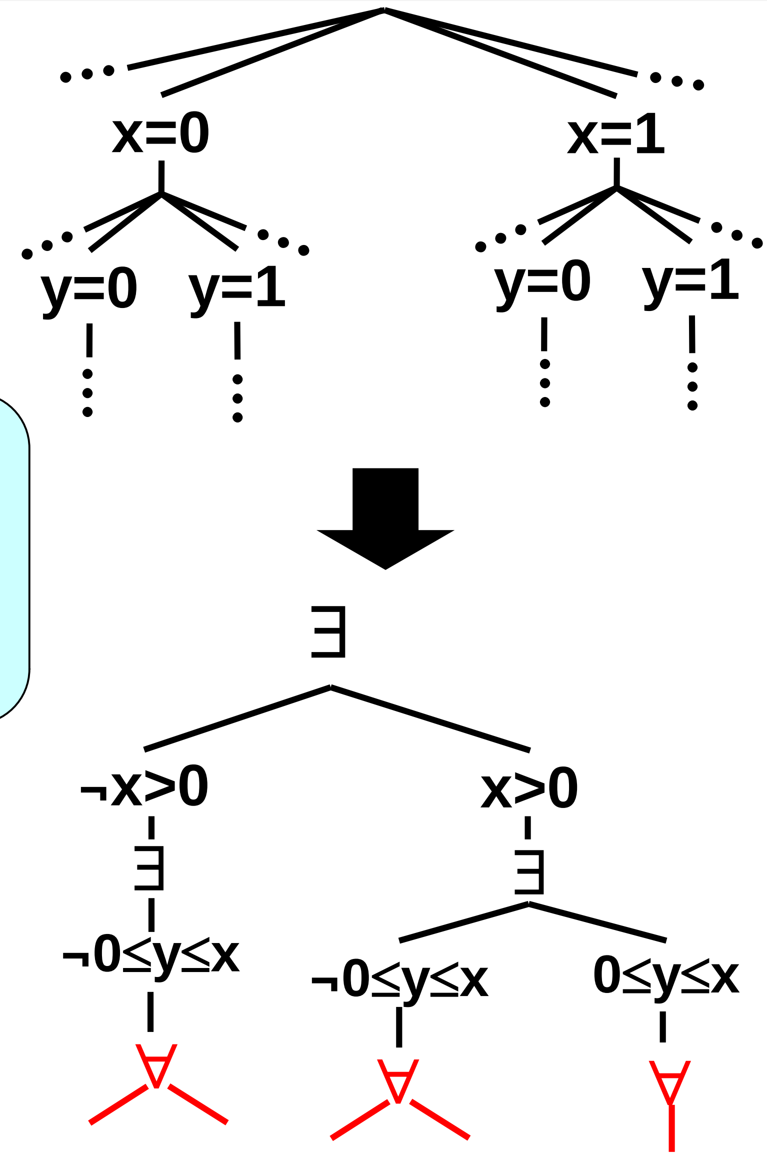
pred: x>0

pred: 0≤y≤x

pred: x+y>0

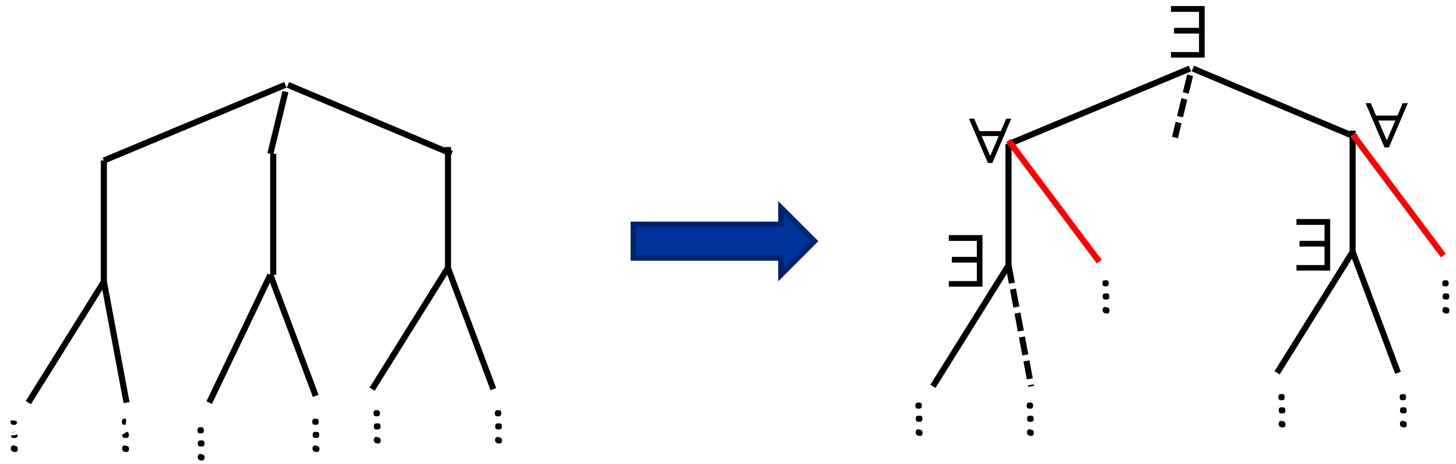
```
∃( /* case
  ∃(/* case
    ∀( f true /*case x+y>0 */,
       f false /*case ¬x+y>0 */)
  )
 , ...
)
```

Overapproximation: both branches should have an infinite path (since we don't know which branch is valid)



Summary: Non-Termination Verification by MoCHi

- Underapproximate non-deterministic computation, and check that one of the branches has a non-terminating path
- Overapproximate deterministic computation, and check that all the branches have non-terminating paths
- Check them by using HO model checking



Conclusions

- HO model checking alone is not enough to construct practical software model checkers for OCaml, Java, …
- It is often the case that software verification techniques developed for imperative programs cannot be reused in the HO setting
  - Types are useful for generalization to HO