Higher-Order Model Checking and Applications to Program Verification

Naoki Kobayashi
Tohoku University

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

Program Verification Techniques

- Finite state/pushdown model checking
  - Applicable to first-order procedures (pushdown model checking), but not to higher-order programs
- Type-based program analysis
  - Applicable to higher-order programs
  - Sound but imprecise
- Dependent types/theorem proving
  - Requires human intervention

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

This Talk

- New program verification method based on higher-order model checking
  [POPL 2009/2010, LICS 2009, ICALP 2009, PPDP 2009]
  - Sound, complete, and automatic for
    - A large class of higher-order programs
    - A large class of verification problems
  - Built on recent/new advances in
    - Type theories
    - Automata/formal language theories (esp. higher-order recursion schemes)
    - Model checking

Outline

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

Higher-Order Recursion Scheme

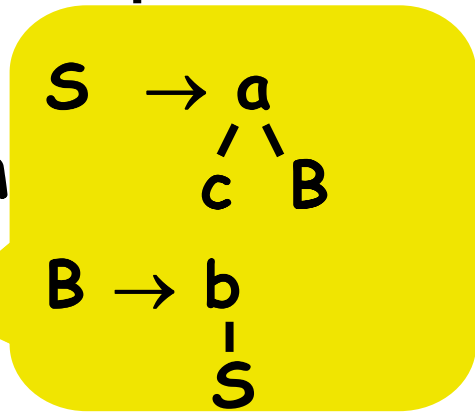
- Grammar for generating an infinite tree

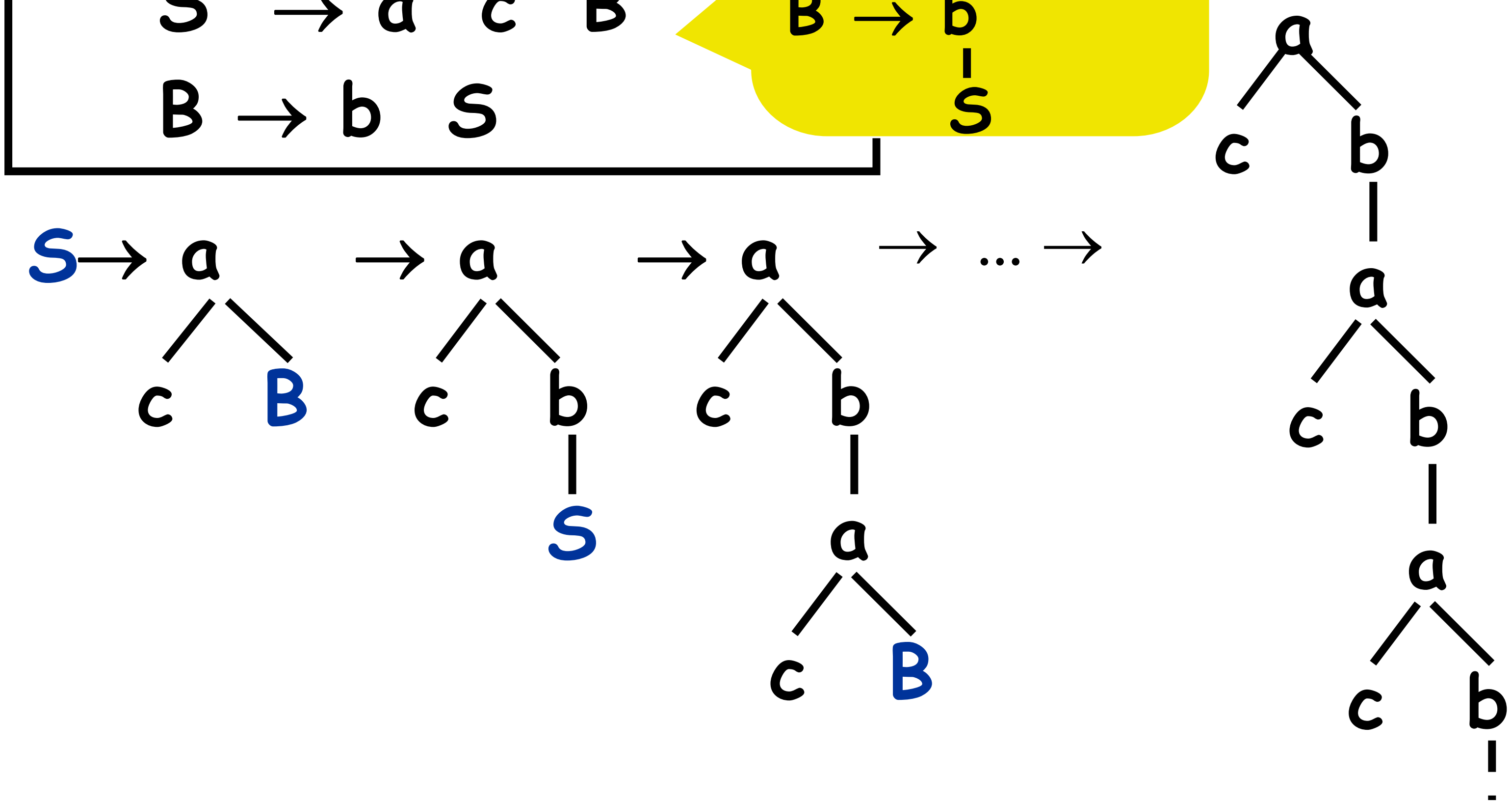
Order-0 scheme

(regular tree gramm

$S \to a \; c \; B$

$B \to b \; S$





Higher-Order Recursion Scheme

♦ Grammar for … nite tree

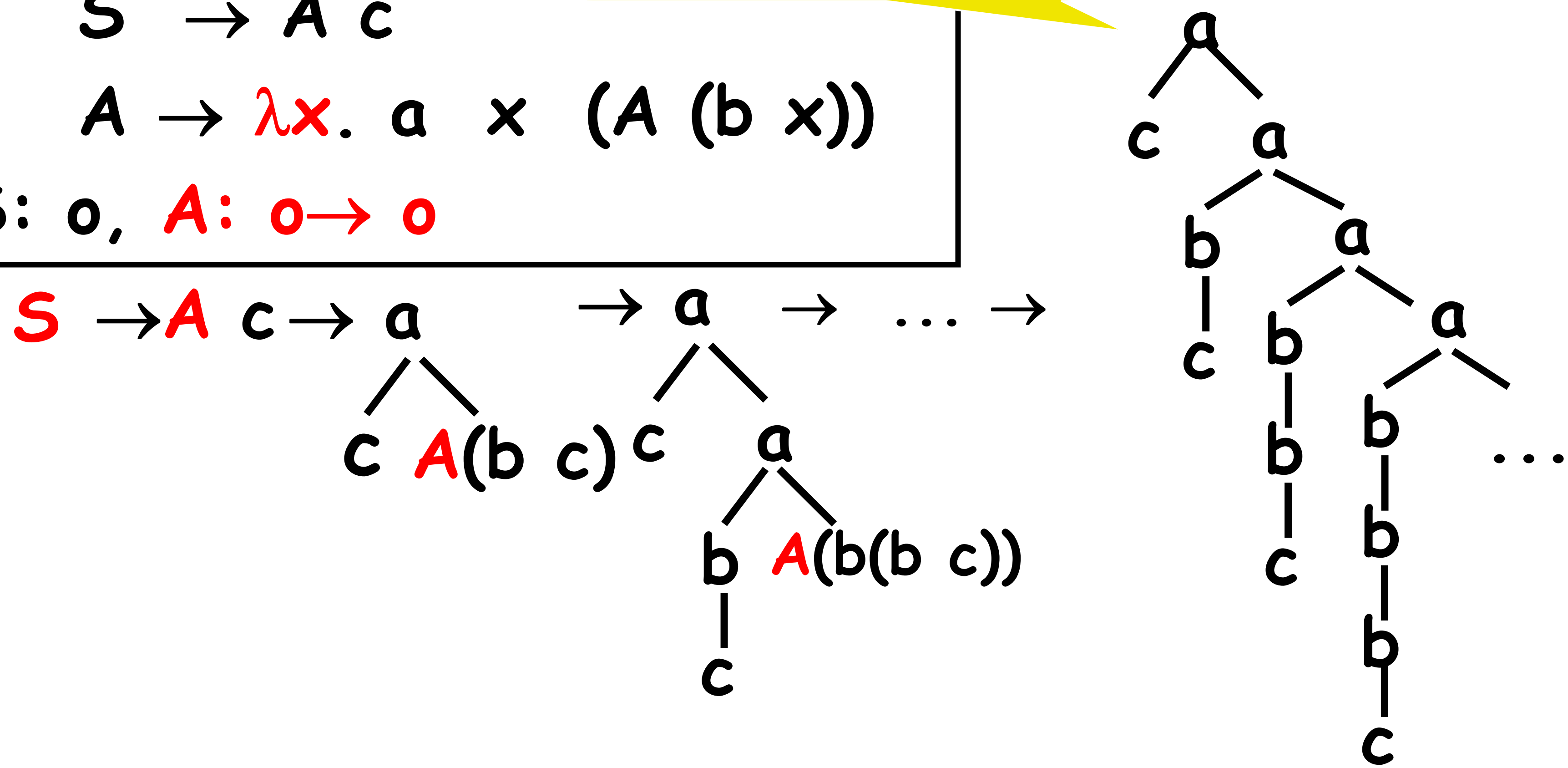
Tree whose paths are labeled by $a^{m+1} b^m c$

Order-1 scheme…

$S \rightarrow A\ c$

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

$S: o,\ A: o \rightarrow o$



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]
(for order-n recursion scheme)

Outline

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

From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

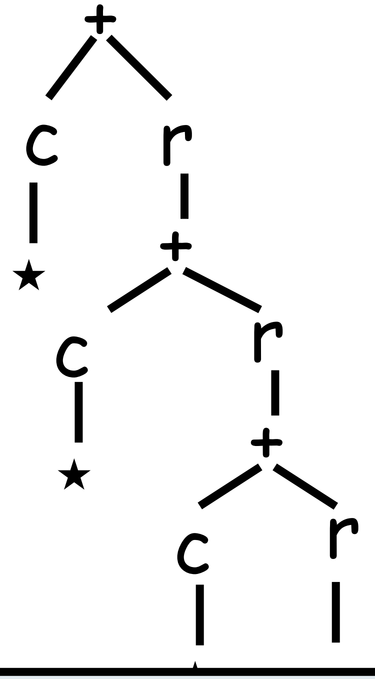
Higher-order program + specification → **Program Transformation** → Rec. scheme (describing all event sequences or outputs) + Tree automaton, recognizing valid event sequences or outputs → **Model Checking**

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?

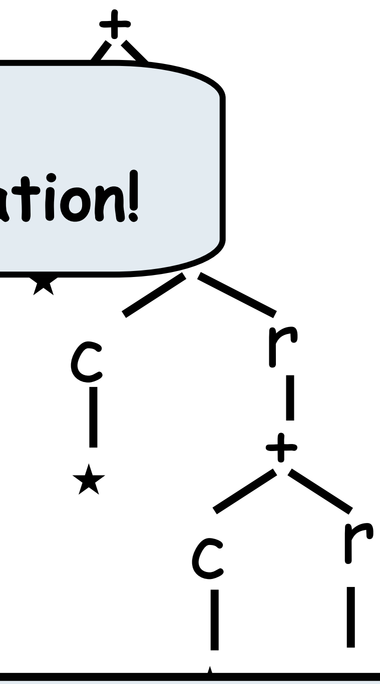
From Program Verification to Model Checking: Example

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

F x k → + (c k) (r(F x k))

S → F d ★

CPS Transformation!



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

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

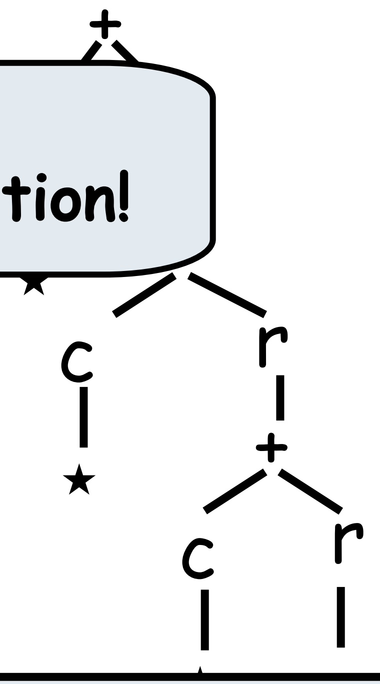
From Program Verification to Model Checking: Example

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

F x k → + (c k) (r(F x k))

S → F d ★

CPS Transformation!



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

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

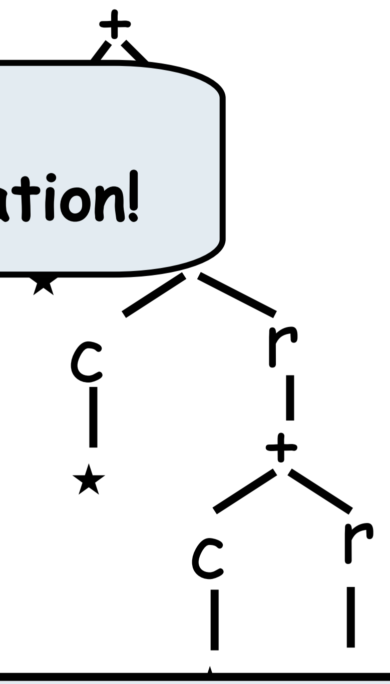
From Program Verification to Model Checking: Example

```
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?

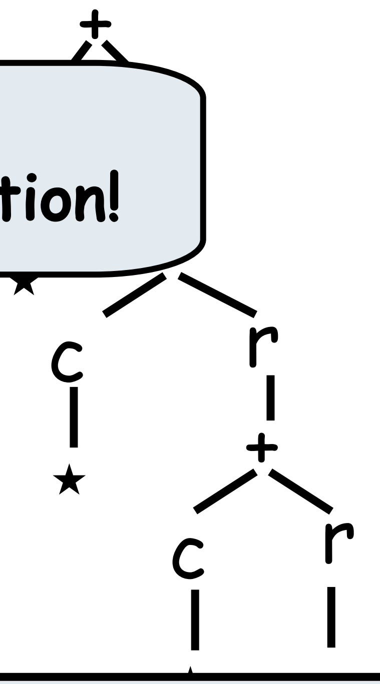
From Program Verification to Model Checking: Example

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

F x k → + (c k) (r(F x k))

S → F d ★

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)
```

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

$S \to F\ d\ \star$

S

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)
```

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

$S \rightarrow F\ d\ \star$

$F\ d\ \star$

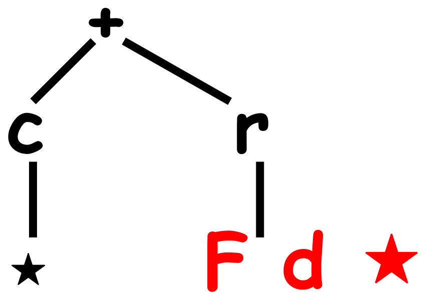
Is the file "foo" accessed according to read* close?

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 → + (c k) (r(F x k))
S → F d ★



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

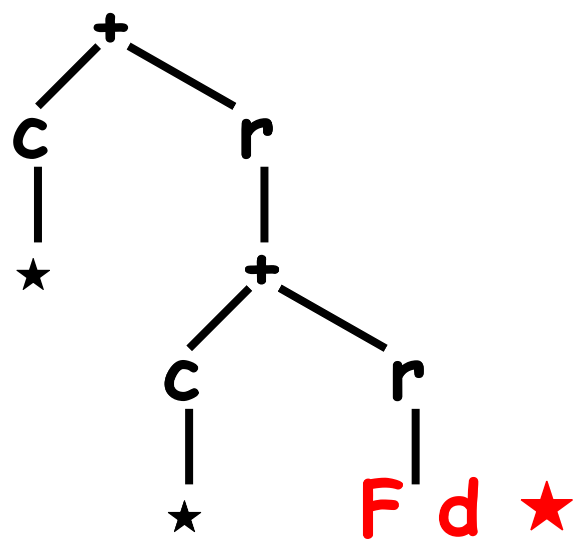
Is each path of the tree labeled by r*c?

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)
```

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

$S \to F\ d\ \star$



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

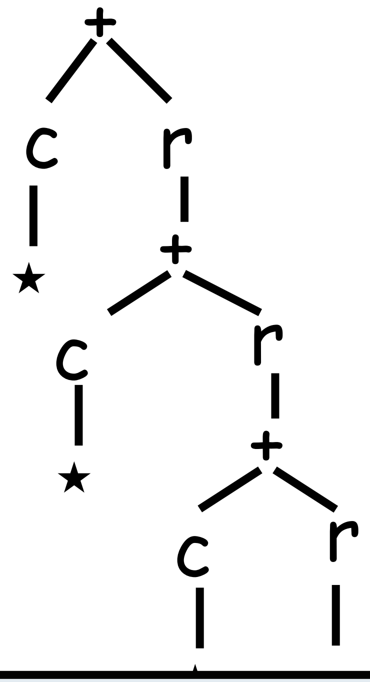
Is each path of the tree labeled by r*c?

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)
```

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

$S \rightarrow F\ d\ \star$



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

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

From Program Verification to Model Checking Recursion Schemes

Higher-order program + specification → **Program Transformation** → Rec. scheme (describing all event sequences) + automaton for infinite trees → **Model Checking**

Sound, complete, and automatic for:

- A large class of higher-order programs: simply-typed λ-calculus + recursion + finite base types
- A large class of verification problems: resource usage verification [Igarashi&K. POPL2002], reachability, flow analysis, ...

Comparison with Traditional Approach (Control Flow Analysis)

- **Control flow analysis**

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

- **Our approach**

Higher-order program → **Program Transformation** → Recursion scheme → **verification**

Only information about infinite data domains is approximated!

Comparison with Traditional Approach (Software Model Checking)

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

infinite state model checking

Outline

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

Goal

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

Tree(G) is accepted by tree automaton *A*

if and only if

G is typable in TS(*A*)

Model Checking as Type Checking
(c.f. [Naik & Palsberg, ESOP2005])

Why Type-Theoretic Characterization?

- **Simpler** decidability proof of model checking recursion schemes
  - Previous proofs [Ong, 2006][Hague et. al, 2008] made heavy use of game semantics
- **More efficient** model checking algorithm
  - Known algorithms [Ong, 2006][Hague et. al, 2008] **always** require n-EXPTIME

Model Checking Problem

Given
  G: higher-order recursion scheme
    (without safety restriction)
  A: alternating parity tree automaton (APT)
    (a formula of modal μ-calculus or MSO),
does A accept Tree(G)?

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

Model Checking Problem

Given

G: higher-order recursion scheme

(without safety restriction)

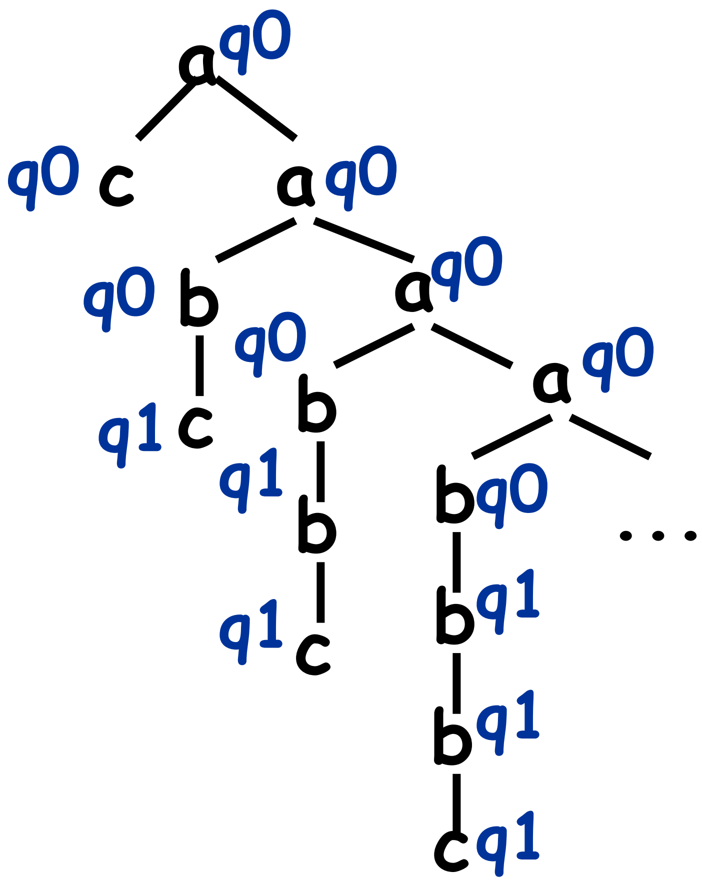
A: trivial automaton [Aehlig CSL06]

(Büchi tree automaton where all the states are accepting states)

does A accept Tree(G)?

See [K.&Ong, LICS09] for the general case (full modal μ-calculus model checking)

(Trivial) tree automaton for infinite trees

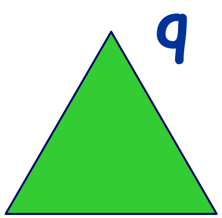


$\delta(q0, a) = q0\ q0$
$\delta(q0, b) = q1$
$\delta(q1, b) = q1$
$\delta(q0, c) = \varepsilon$
$\delta(q1, c) = \varepsilon$

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

Types for Recursion Schemes

- Automaton state as the type of trees
  - q: trees accepted from state q



  - $q1 \wedge q2$: trees accepted from both q1 and q2

$\boxed{\text{Is Tree(G) accepted by A?}}$

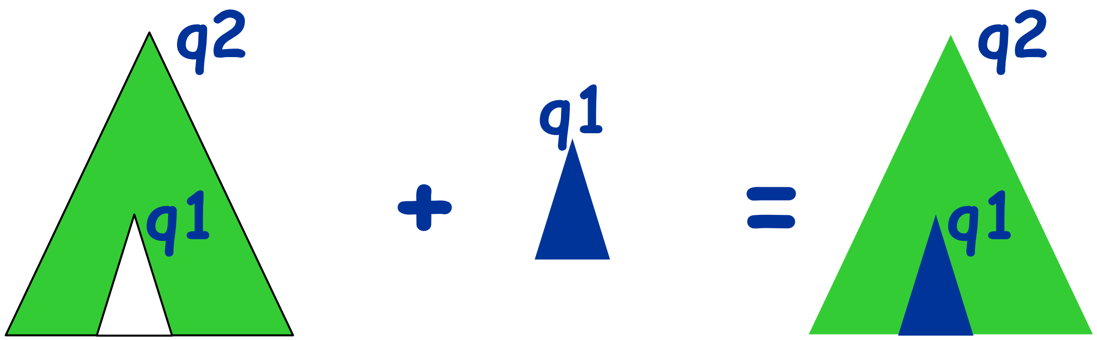
$\Downarrow$

$\boxed{\text{Does Tree(G) have type } q_0\text{?}}$

Types for Recursion Schemes

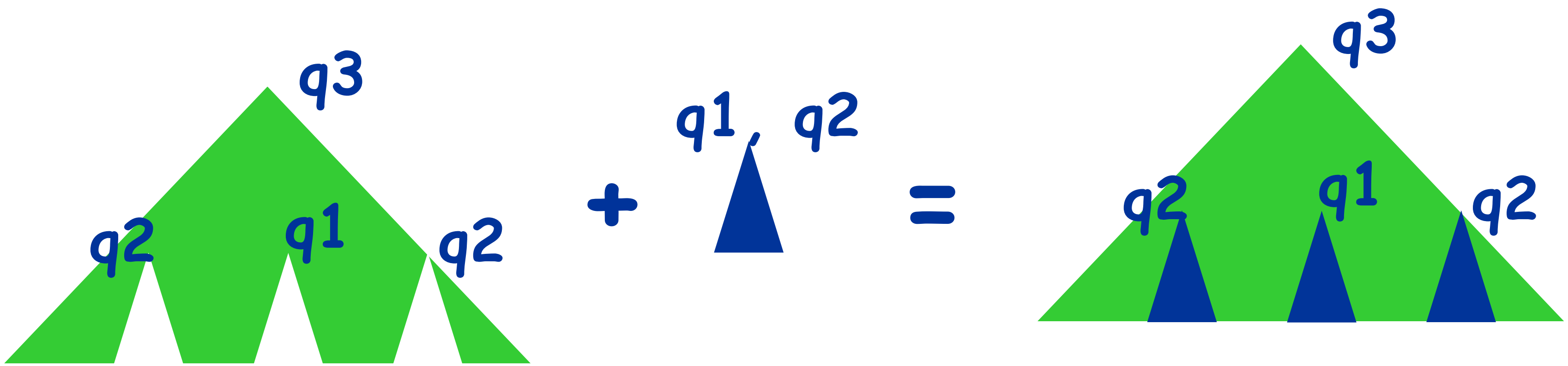
- Automaton state as the type of trees

  - $q1 \to q2$: functions that take a tree of type q1 and return a tree of q2



Types for Recursion Schemes

- Automaton state as the type of trees
  - $q1 \wedge q2 \rightarrow q3$:
    functions that take a tree of type $q1 \wedge q2$ and return a tree of type q3

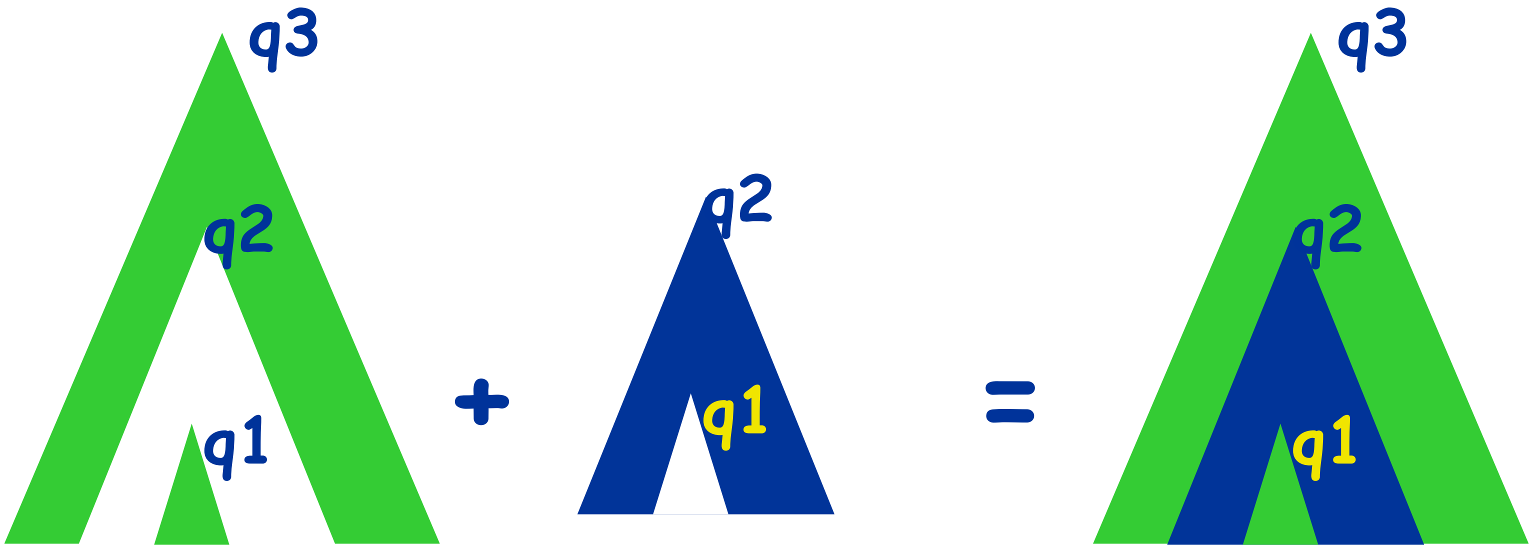


Types for Recursion Schemes

- Automaton state as the type of trees

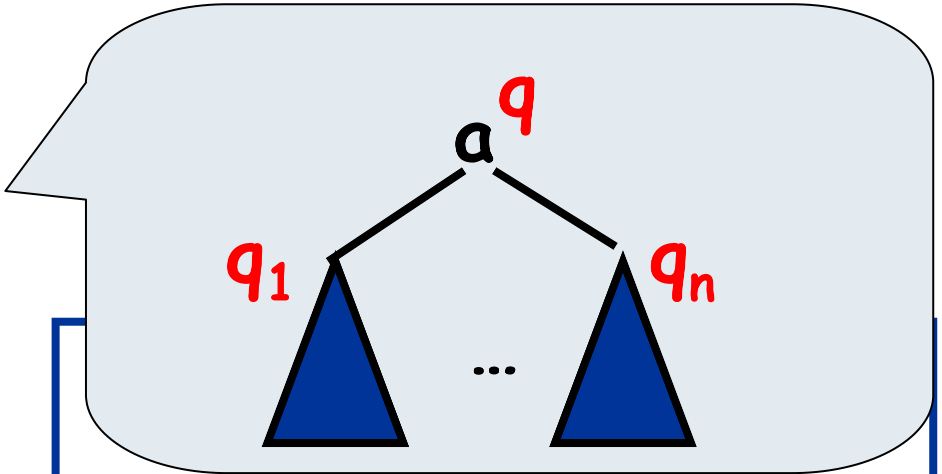
  $(q1 \to q2) \to q3$:

  functions that take a function of type $q1 \to q2$ and return a tree of type $q3$



Typing

$$\frac{\delta(q,\ a) = q_1 \ldots q_n}{\vdash a : q_1 \rightarrow \ldots \rightarrow q_n \rightarrow q}$$



$$\frac{\Gamma, x{:}\tau_1, \ldots, x{:}\tau_n \ \vdash t{:}\tau}{\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau}$$

$$\frac{\Gamma \vdash t_2{:}\tau_i\ (i=1,..n)}{\Gamma \vdash t_1\ t_2{:}\tau}$$

$$\frac{\Gamma \vdash t_k : \tau \text{ (for every } F_k{:}\tau \in \Gamma)}{\vdash \{F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n\} : \Gamma}$$

Soundness and Completeness [K., POPL2009]

Let

G: Rec. scheme with initial non-terminal S

A: Trivial automaton with initial state q_0

TS(A): Intersection type system derived from A

Then,

Tree(G) is accepted by A

if and only if

S has type q_0 in TS(A)

Outline

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

Typing

$$\frac{\delta(q,\ a) = q_1 \ldots q_n}{\vdash a : q_1 \rightarrow \ldots \rightarrow q_n \rightarrow q}$$

$$\Gamma, x{:}\tau \ \vdash x : \tau$$

$$\frac{\Gamma, x{:}\tau_1, \ldots, x{:}\tau_n \ \vdash t{:}\tau}{\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau}$$

$$\frac{\Gamma \vdash t_1{:}\ \tau_1 \wedge \ldots \wedge \tau_n \rightarrow \tau \qquad \Gamma \vdash t_2{:}\tau_i\ (i=1,\ldots n)}{\Gamma \vdash t_1\ t_2{:}\tau}$$

$$\frac{\Gamma \vdash t_j : \tau \text{ (for every } F_j{:}\tau \in \Gamma)}{\vdash \{F_1 \rightarrow t_1, \ldots, F_n \rightarrow t_n\} : \Gamma}$$

Naïve Type Checking Algorithm

S has type q_0

Recursion Scheme:
$\{F_1 \to t_1, \ldots, F_m \to t_m\}$

(i) $\Gamma \vdash t_j : \tau$
for each $F_j : \tau \in \Gamma$

(ii) $S : q_0 \in \Gamma$

for some Γ

Filter out invalid type bindings

$S : q_0 \in \mathrm{gfp}(H) = \bigcap_k H^k(\Gamma_{max})$

where

$H(\Gamma) = \{ F_j : \tau \in \Gamma \mid \Gamma \vdash t_j : \tau \}$

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

All the possible type bindings

E.g. for $F : o \to o$,
$\{F : T \to q0,\ F : q0 \to q0,$
$F : q1 \to q0,$
$F : q0 \wedge q1 \to q0, \ldots\}$

Naïve Algorithm Does NOT Work

S has type q_0

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

where $H(\Gamma) = \{\, F_j{:}\tau \in \Gamma \mid \Gamma \vdash t_j{:}\tau \,\}$

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

This is huge!

sort	# of types ($Q=\{q_0,q_1,q_2,q_3\}$)
o	4 (q_0,q_1,q_2,q_3)
$o \to o$	$2^4 \times 4 = 64$ ($\wedge S \to q$, with $S \in 2^Q$, $q \in Q$)
$(o\to o) \to o$	$2^{64} \times 4 = 2^{66}$
$((o\to o) \to o) \to o$	$2^{2^{66}} \times 4 > 10^{1000000000000000000}$

Outline

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

More Efficient Algorithm?

S has type q_0

$\Leftarrow$

$$S{:}q_0 \in \bigcap_k H^k(\overset{\Gamma_0}{\cancel{\Gamma_{max}}})$$

where

$$H(\Gamma) = \{ F_j{:}\tau \in \Gamma \mid \Gamma \vdash t_j{:}\tau \}$$

Challenges:

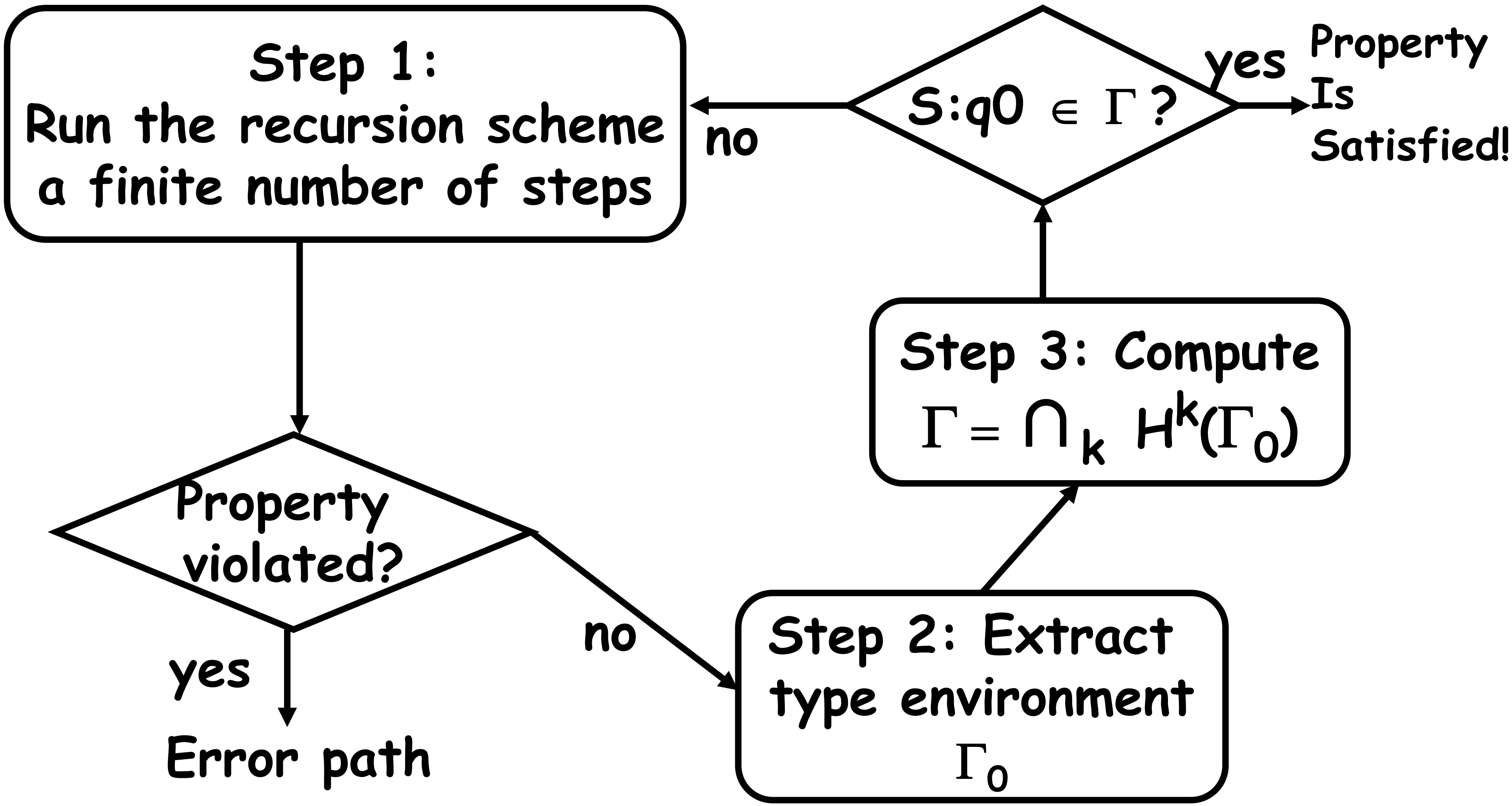
(i) How can we find an appropriate Γ_0 ?

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

(ii) How can we guarantee completeness?

Iteratively repeat (i) and type checking

Hybrid Type Checking Algorithm



Soundness and Completeness of the Hybrid Algorithm

Given:

- Recursion scheme *G*
- Deterministic trivial automaton *A*,

the algorithm eventually terminates, and:

(i) outputs an error path
 if Tree(*G*) is not accepted by *A*

(ii) outputs a type environment
 if Tree(*G*) is accepted by *A*

Example

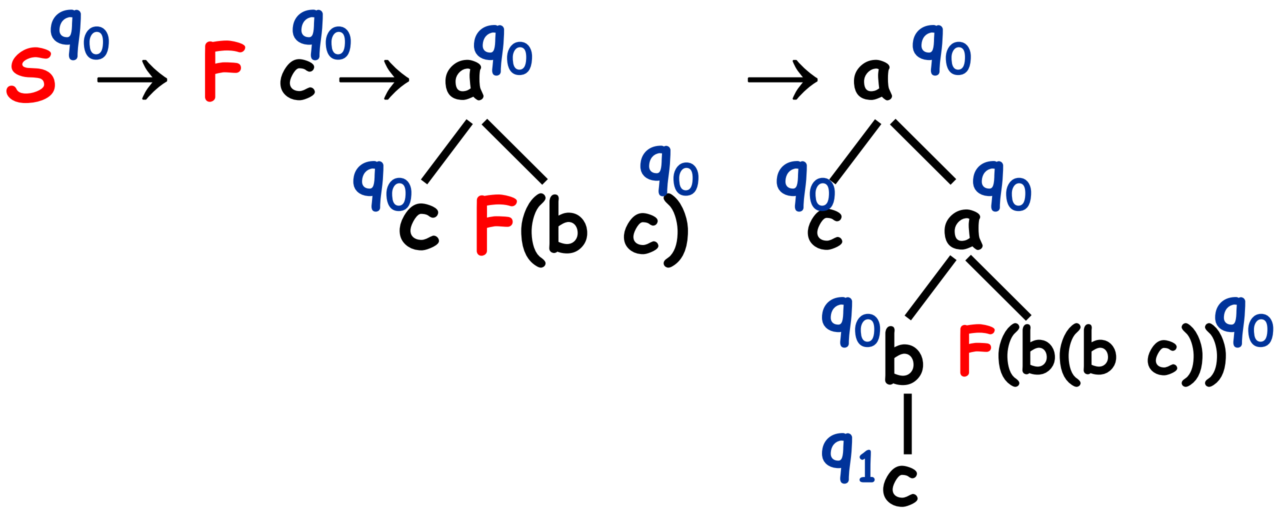
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0,\ a) = q_0\ q_0 \qquad \delta(q_0,\ b) = q_1$

  $\delta(q_0,\ c) = \delta(q_1,\ c) = \varepsilon$



Example

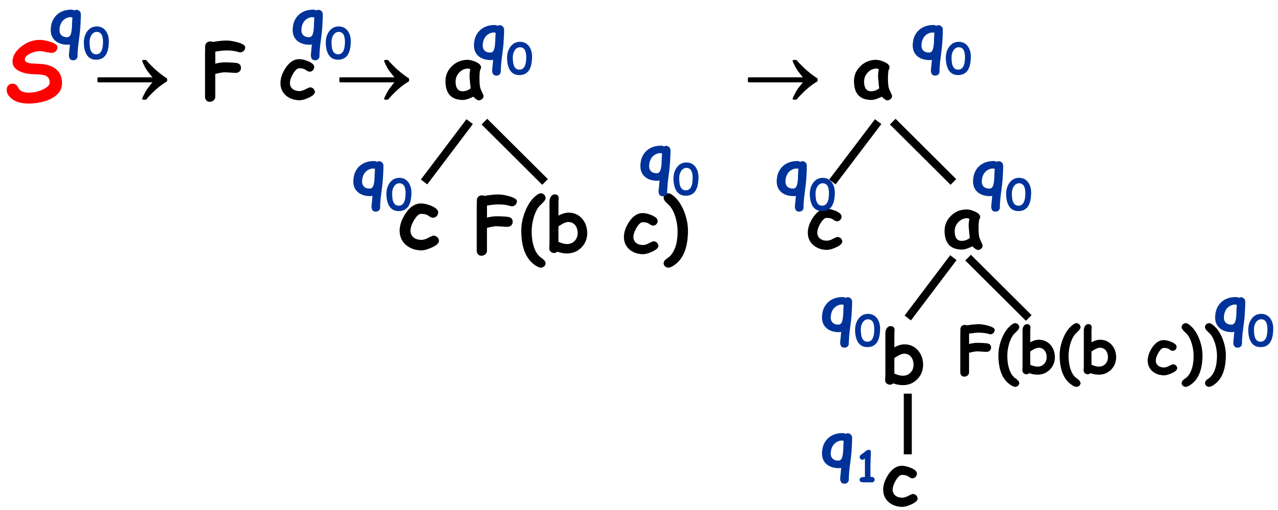
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0,\ a) = q_0\ q_0 \qquad \delta(q_0,\ b) = q_1$

  $\delta(q_0,\ c) = \delta(q_1,\ c) = \varepsilon$



Γ_0 :

S: q_0

Example

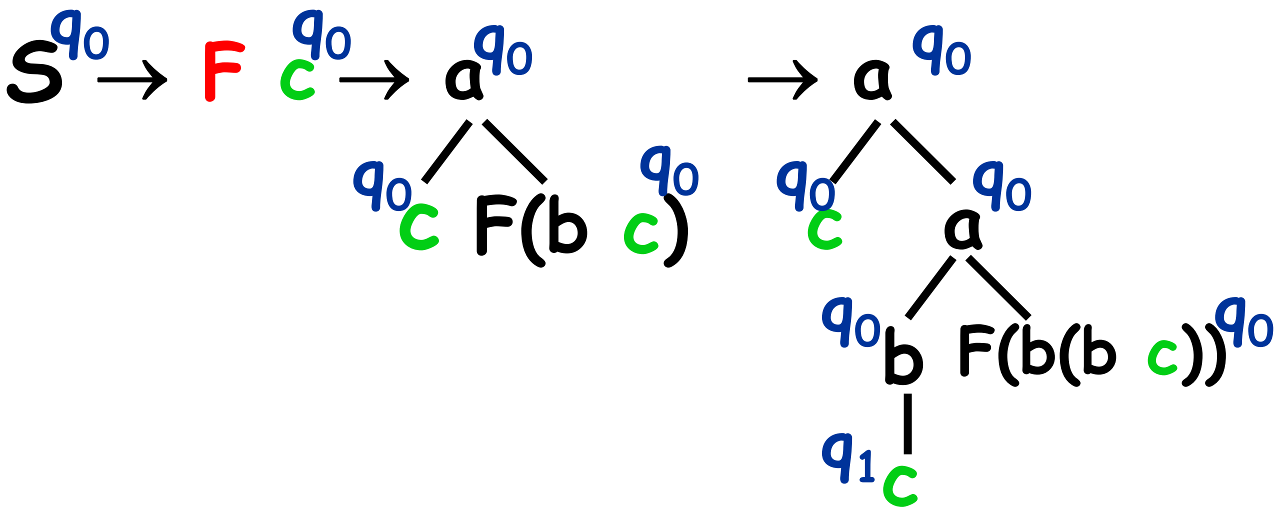
- **Recursion scheme:**

  $S \to F\ c \qquad F \to \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$

  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$



Γ_0 :

$S: q_0$

$F: q_0 \wedge q_1 \to q_0$

Example

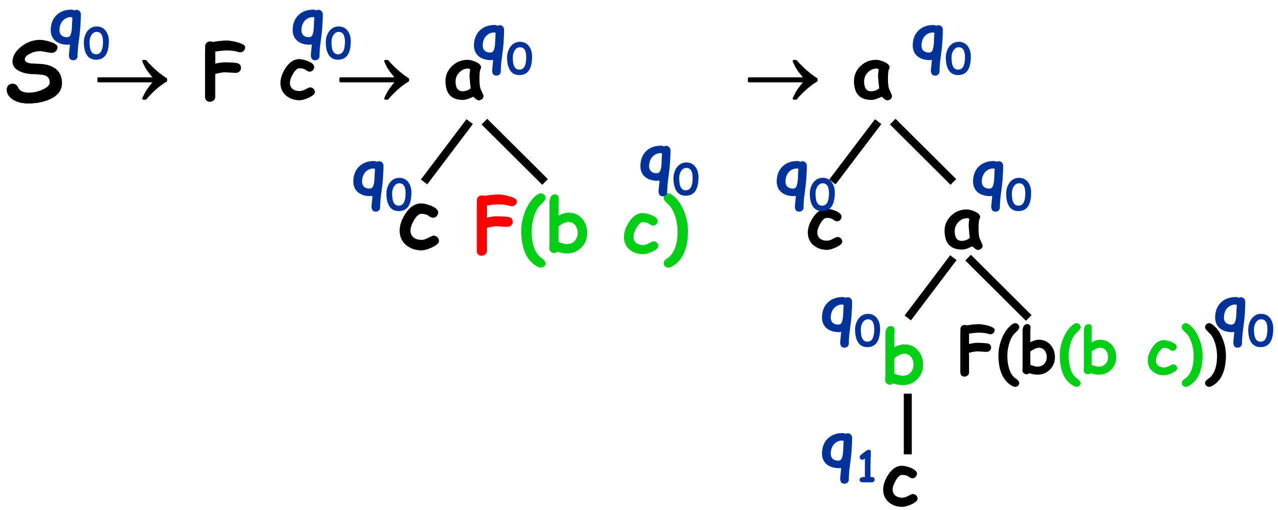
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$

  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$



Γ_0 :

$S: q_0$

$F: q_0 \wedge q_1 \rightarrow q_0$

$F: q_0 \rightarrow q_0$

Example

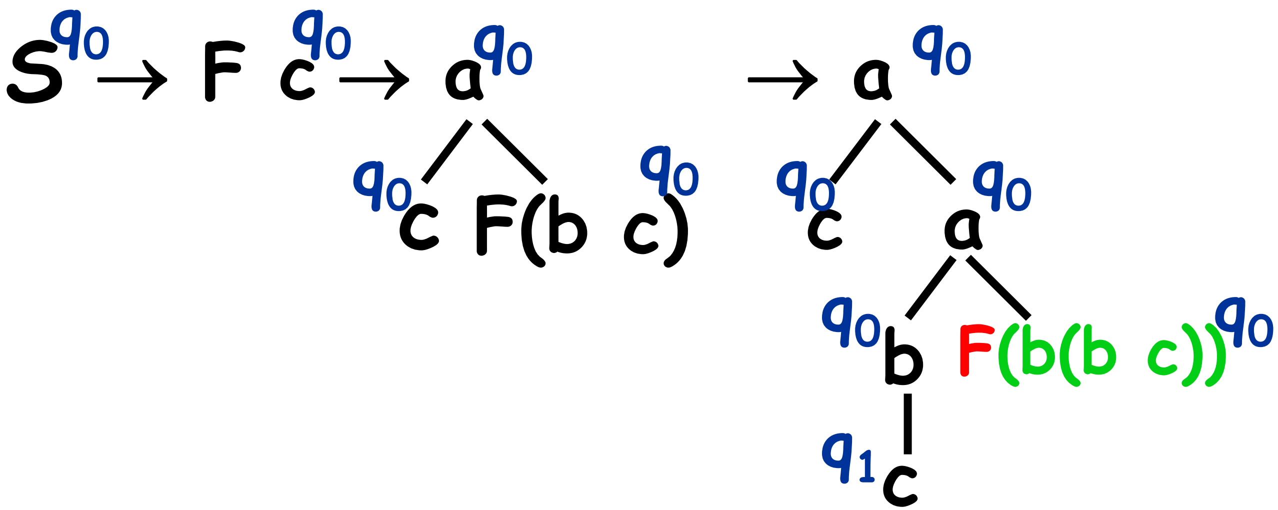
- **Recursion scheme:**

  $S \rightarrow F\ c \qquad F \rightarrow \lambda x.a\ x\ (F\ (b\ x))$

- **Automaton:**

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$

  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$



Γ_0 :

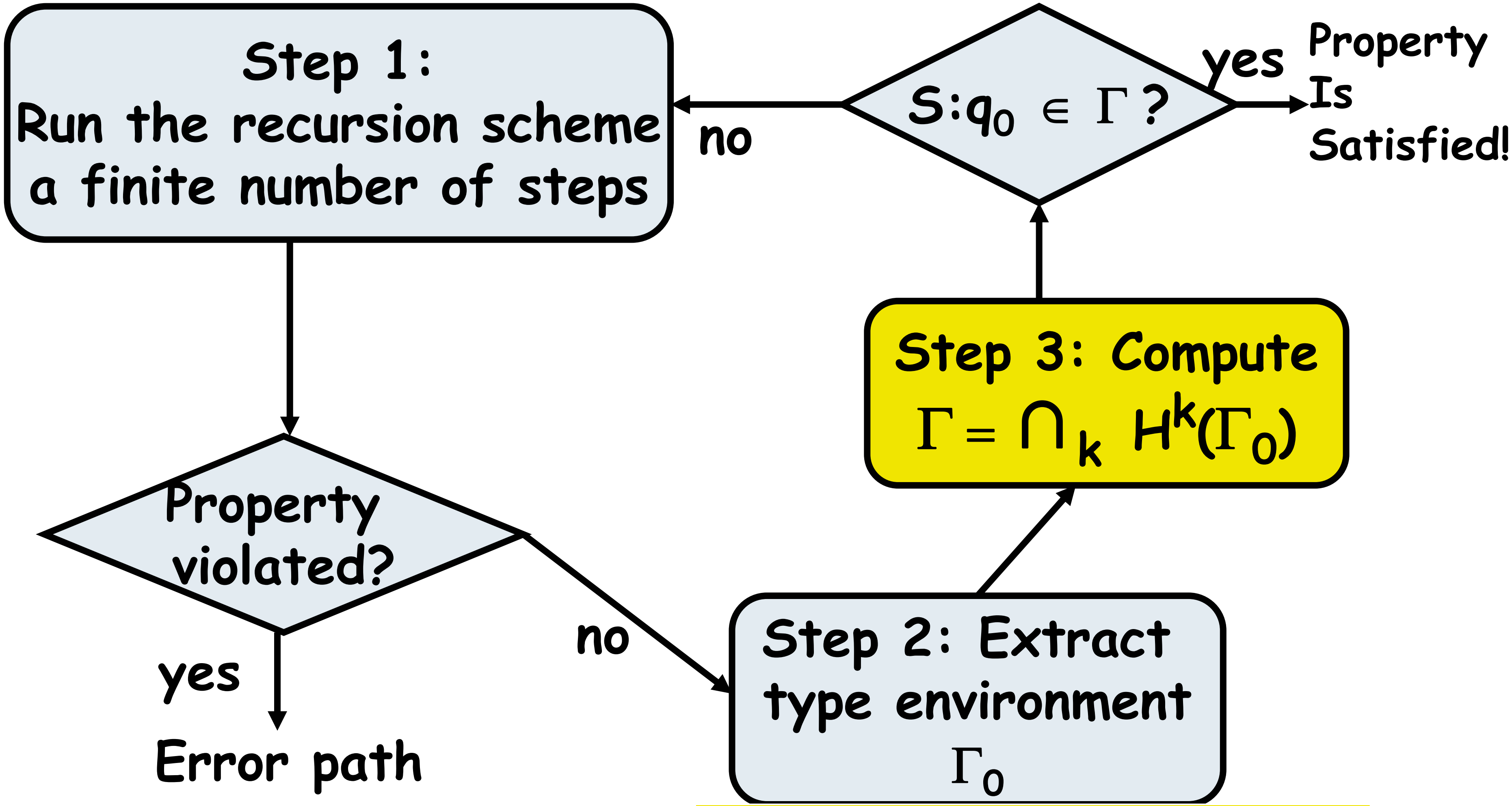
S: q_0

F: $q_0 \wedge q_1 \rightarrow q_0$

F: $q_0 \rightarrow q_0$

F: $\top \rightarrow q_0$

Example



S: q_0 F: $q_0 \wedge q_1 \to q_0$

F: $q_0 \to q_0$ F: $T \to q_0$

Example: Filtering out invalid judgments

- Recursion scheme:

  $S \to F\ c \qquad F \to \lambda x.a\ x\ (F\ (b\ x))$

- Automaton:

  $\delta(q_0, a) = q_0\ q_0 \qquad \delta(q_0, b) = q_1$
  $\delta(q_0, c) = \delta(q_1, c) = \varepsilon$

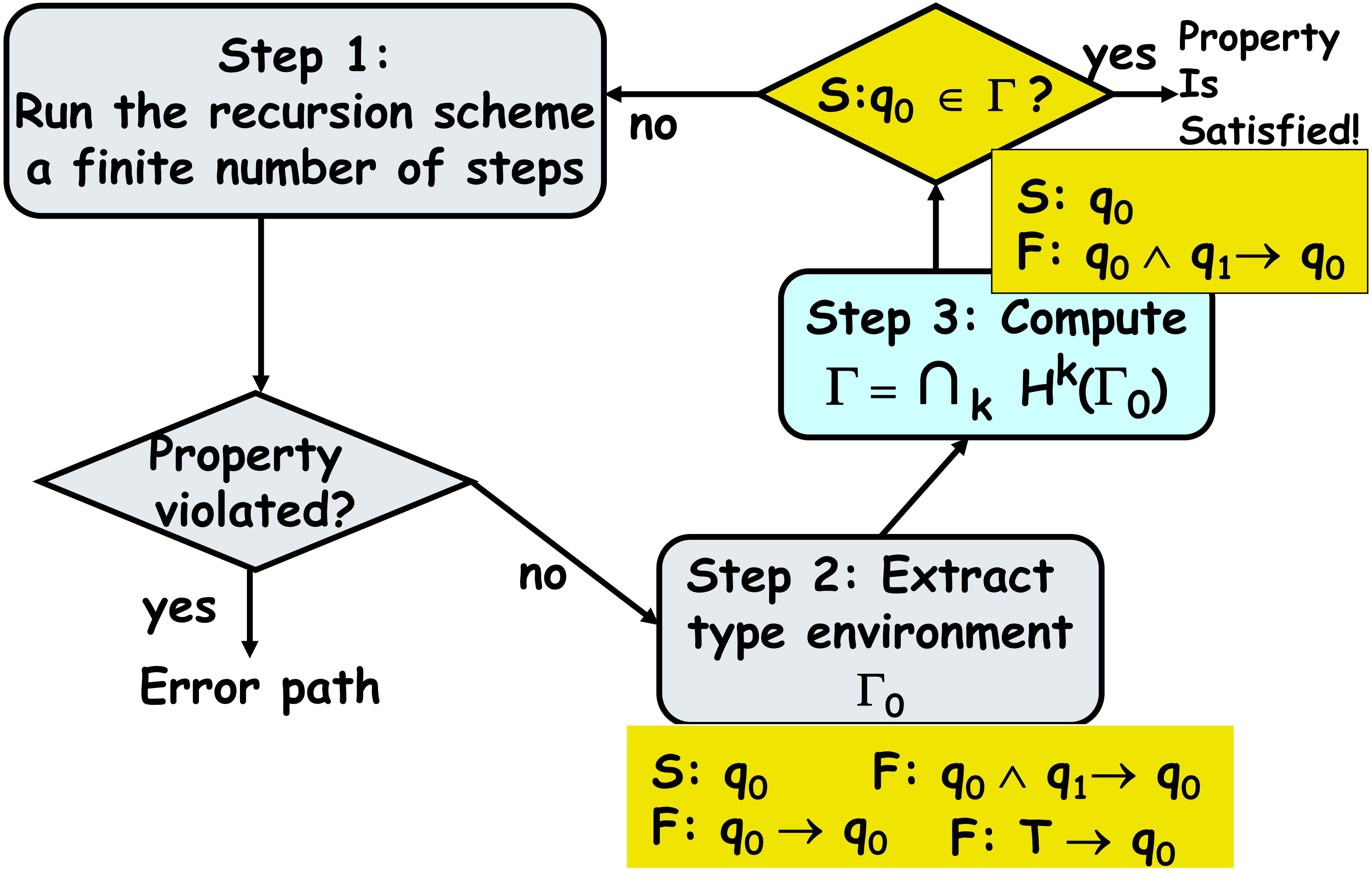
$\Gamma_0 = \{S: q_0,\ F: q_0 \wedge q_1 \to q_0,\ F: q_0 \to q_0\ ,\ F: \top \to q_0\}$

$\Gamma_1 = H(\Gamma_0) = \{\ F_k : \tau \in \Gamma_0 \mid \Gamma_0 \vdash t_k : \tau\ \}$
$\quad = \{S: q_0,\ F: q_0 \wedge q_1 \to q_0,\ F: q_0 \to q_0\ \}$

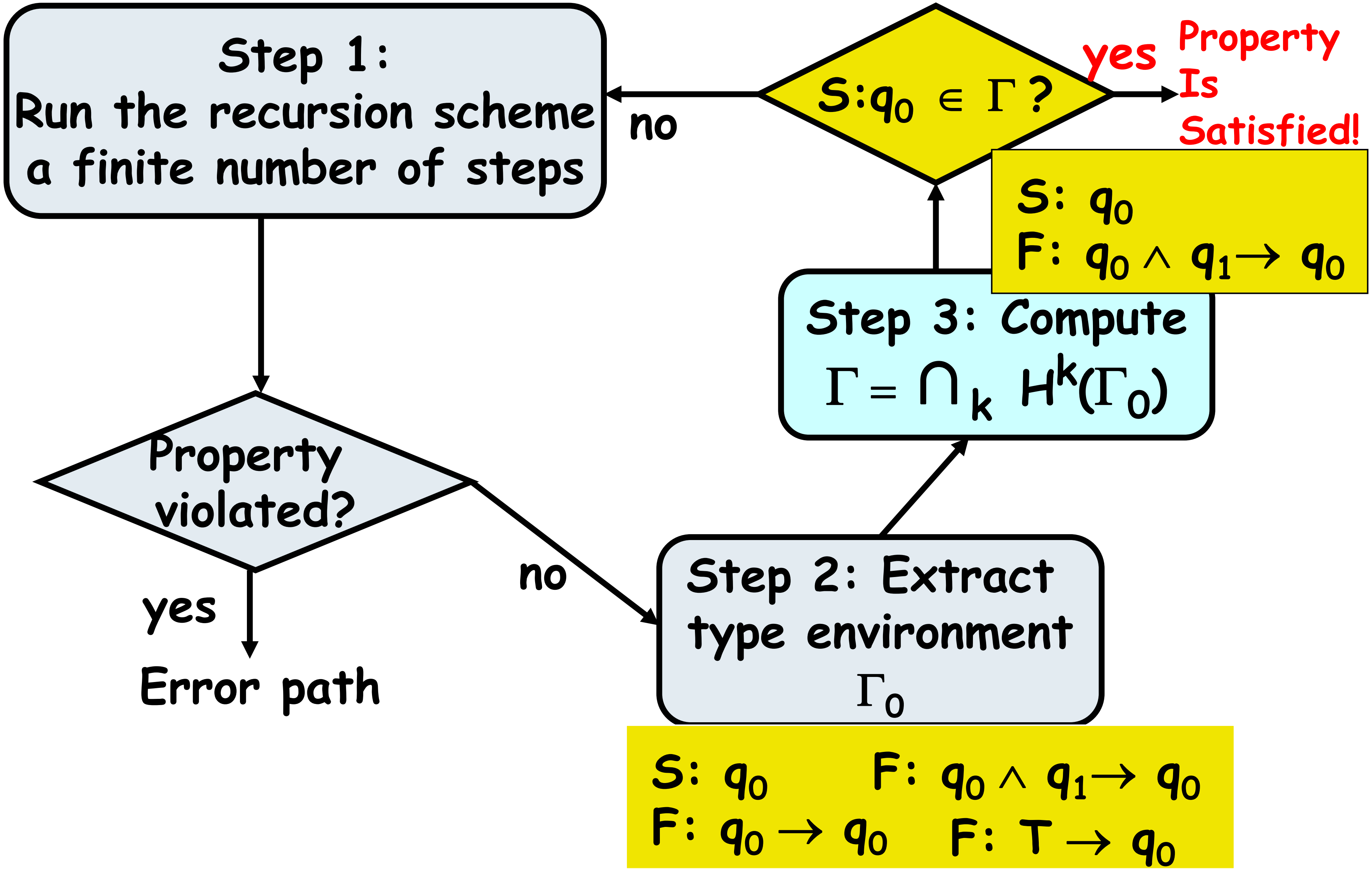
$\Gamma_2 = \{S: q_0,\ F: q_0 \wedge q_1 \to q_0\}$

$\Gamma_3 = \{S: q_0,\ F: q_0 \wedge q_1 \to q_0\}$

Example


Step 1:
Run the recursion scheme a finite number of steps
Property violated?
yes
Error path
no
Step 2: Extract type environment Γ_0
S: q_0
F: $q_0 \wedge q_1 \to q_0$
F: $q_0 \to q_0$
F: $T \to q_0$
Step 3: Compute $\Gamma = \cap_k H^k(\Gamma_0)$
S: q_0
F: $q_0 \wedge q_1 \to q_0$
$S{:}q_0 \in \Gamma$?
no
yes
Property Is Satisfied!

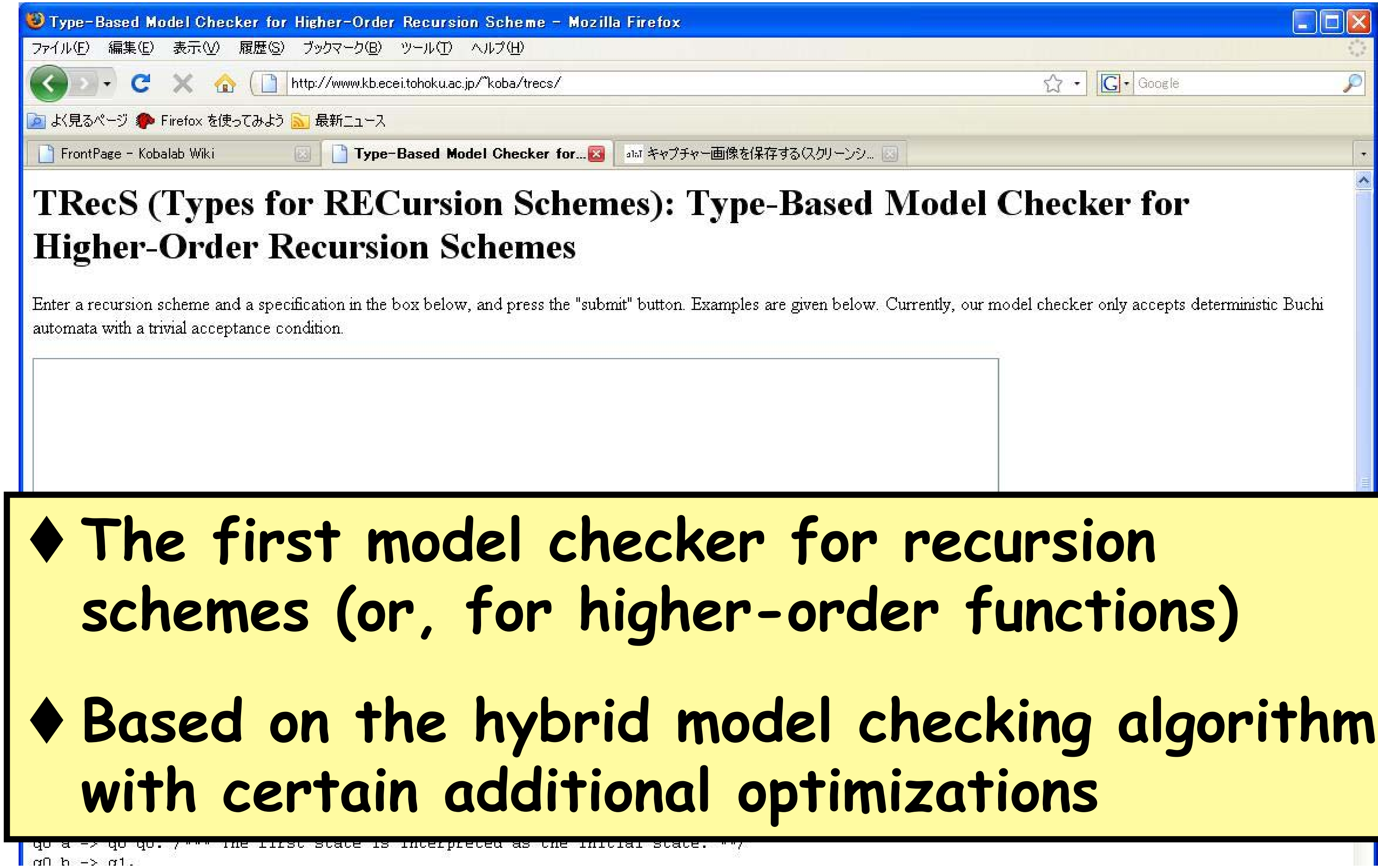
Example



Step 1:
Run the recursion scheme a finite number of steps
no
$S{:}q_0 \in \Gamma$?
yes
Property Is Satisfied!
S: q_0
F: $q_0 \wedge q_1 \rightarrow q_0$
Step 3: Compute
$\Gamma = \bigcap_k H^k(\Gamma_0)$
Property violated?
yes
Error path
no
Step 2: Extract type environment Γ_0
S: q_0
F: $q_0 \wedge q_1 \rightarrow q_0$
F: $q_0 \rightarrow q_0$
F: $T \rightarrow q_0$

TRecS

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



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

Experiments

	order	rules	states	result	Time (msec)
Twofiles	4	[illegible]	[illegible]	[illegible]	[illegible]
FileWrong	4	[illegible]	[illegible]	[illegible]	[illegible]
TwofilesE	4	12	[illegible]	Yes	2
FileOcamlC	4	23	4	Yes	5
Lock	4	11	3	Yes	10
Order5	5	9	4	Yes	2
m91	2	280	1	Yes	150
xhtml	1	2	50	Yes	263

Taken from the compiler of Objective Caml, consisting of about 60 lines of O'Caml code

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

(A simplified version of) FileOcamlC

```
let readloop fp =
  if * then () else readloop fp; read fp
let read_sect() =
  let fp = open "foo" in
  {readc=fun x -> readloop fp;
   closec = fun x -> close fp}
let loop s =
  if * then s.closec() else s.readc();loop s
let main() =
  let s = read_sect() in loop s
```

Experiments

	order	rules	states	result	Time (msec)
Twofiles	4	11	4	Yes	2
FileWrong	4	11	4	No	1
TwofilesE	4	12			
FileOcamlC	4	23			
				Yes	2
m91	2	280	1	Yes	150
xhtml	1	2	50	Yes	263

Machine-generated code from McCurthy's 91 function
ing predicate abstraction

Machine-generated code from a program manipulating Xhtml documents

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

Outline

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

Recursion schemes as models of higher-order programs?

+ simply-typed λ-calculus
+ recursion
+ tree constructors
+ finite data domains (via Church encoding; true = $\lambda x.\lambda y.x$, false=$\lambda x.\lambda y.y$)
- infinite data domains (integers, lists, trees,…)
- advanced types (polymorphism, recursive types, object types, …)
- imperative features/concurrency

Ongoing work to overcome the limitation

- Predicate abstraction and CEGAR, to deal with infinite data domains (c.f. BLAST, SLAM, ...)
- From recursion schemes to transducers, to deal with algebraic data types (lists, trees, ...) [K.,Tabuchi&Unno, POPL 2010]
- Infinite intersection types, to deal with non-simply-typed programs [Tsukada&K. FoSSaCS 2010]

Outline

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

Advantages of our approach

(1) Sound, complete and automatic for a large class of higher-order programs

- no false alarms!
- no annotations

Advantages of our approach

(1) Sound, complete and automatic for a large class of higher-order programs

- no false alarms!
- no annotations

(2) Subsumes finite-state/pushdown model checking

- Order-0 rec. schemes ≈ finite state systems
- Order-1 rec. schemes ≈ pushdown systems

Advantages of our approach

(3) Take the best of model checking and types

- Types as certificates of successful verification
  $\Rightarrow$ applications to PCC (proof-carrying code)

- Counterexample when verification fails
  $\Rightarrow$ error diagnosis,
  CEGAR (counterexample-guided abstraction refinement)

Advantages of our approach

(4) Encourages structured programming

Previous techniques:

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

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

v.s.

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

Advantages of our approach

(4) Encourages structured programming

Our technique:

- No loss of precision for higher-order functions and recursion
- Performance penalty? -- Not necessarily!
  - n-EXPTIME in the specification size, but polynomial time in the program size
  - Compact representation of large state space

    e.g. recursion schemes generating $a^m(c)$

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

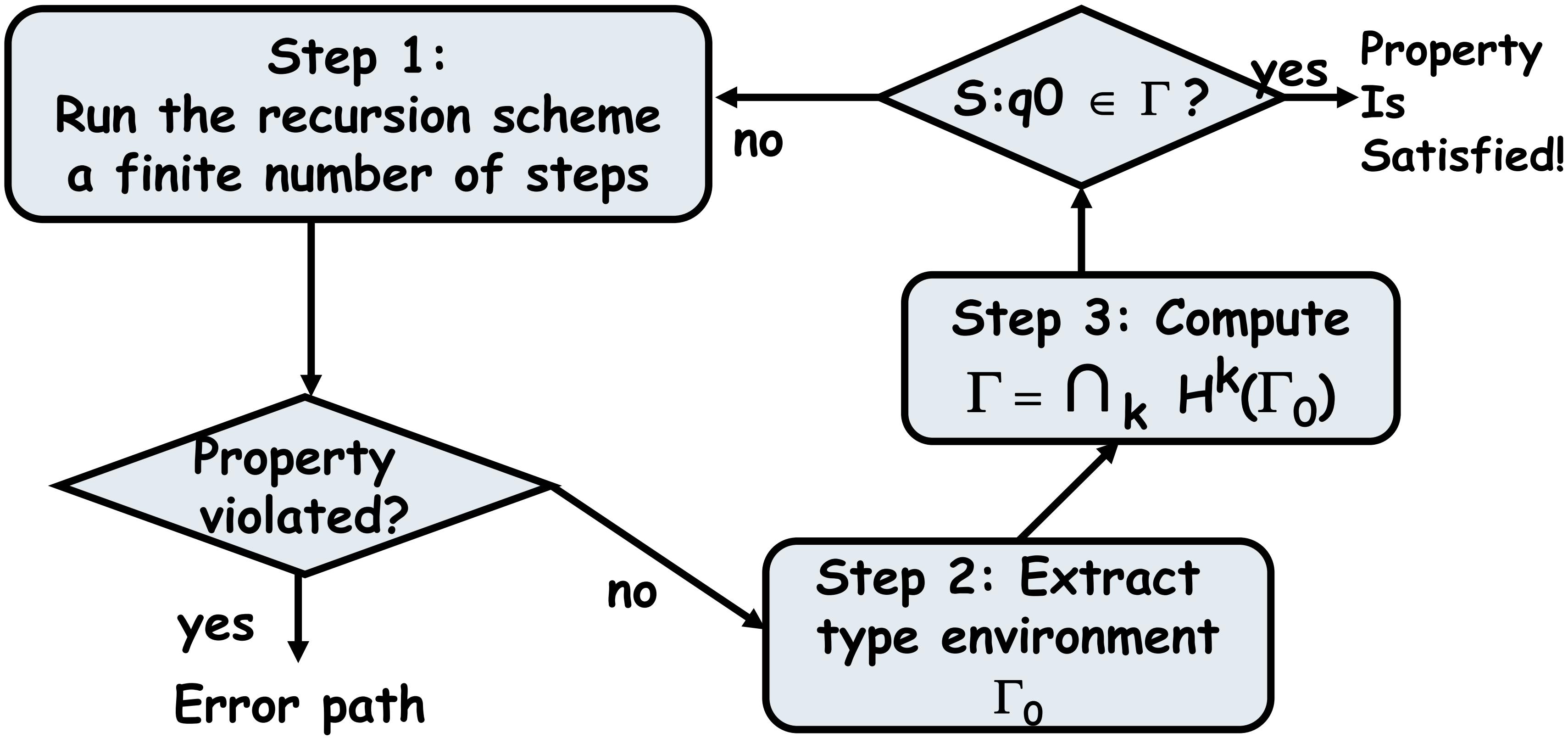
    vs

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

Advantages of our approach

(5) A good combination with testing:

Verification through testing



Challenges

- ♦ **More efficient model checker**
  - More language-theoretic properties of recursion schemes (e.g. pumping lemmas)
  - BDD-like state representation
- ♦ **Software model checker for ML/Haskell**
- ♦ **Extension of the decidability of higher-order model checking (Tree(G) $\models \varphi$)**
- ♦ **Integration with testing** (e.g. QuickCheck)

Conclusion

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

References

- K., Types and higher-order recursion schemes for verification of higher-order programs, POPL09
  From program verification to model-checking, and typing
- K.&Ong, Complexity of model checking recursion schemes for fragments of the modal mu-calculus, ICALP09
  Complexity of model checking
- K.&Ong, A type system equivalent to modal mu-calculus model-checking of recursion schemes, LICS09
  From model-checking to type checking
- K., Model-checking higher-order functions, PPDP09
  Type checking (= model-checking) algorithm
- K., Tabuchi & Unno, Higher-order multi-parameter tree transducers and recursion schemes for program verification, POPL10 Extension to transducers and its applications
- Tsukada & K., Untyped recursion schemes and infinite intersection types, FoSSaCS 10