Types and Recursion Schemes for Higher-Order Program Verification

Naoki Kobayashi
Tohoku University

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

This Talk

- Type-theoretic approach to model checking of recursion schemes
  - Simpler proofs of decidability/complexity of model checking
  - A practical algorithm for model checking (c.f. TRecS: a type-based recursion scheme model checker)
- Applications to program verification
  - A sound, complete, and automated verification method for higher-order functional programs

Plan of the Talk

- **Part 1**
  - From program verification to model checking recursion schemes [K. POPL09]
  - From model checking to type checking: Simple case (safety properties) [K. POPL09]
  - Model checking (=type checking) algorithm [K. PPDP09]
- **Part 2**
  - From model checking to type checking: General case [K. and Ong, LICS09]
  - Towards a software model checker for higher-order languages [K., Tabuchi and Unno, POPL10][Tsukada and K. FoSSaCS10]
  - Remaining challenges

Plan of the Talk

- Part 1
  - From program verification to model checking recursion schemes [K. POPL09]
  - From model checking to type checking: Simple case (safety properties) [K. POPL09]
  - Model checking (=type checking) algorithm [K. PPDP09]
- Part 2
  - From model checking to type checking: General case [K. and Ong, LICS09]
  - Towards a software model checker for higher-order languages
  - Remaining challenges

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 techniques for higher-order programs (e.g. ML/Java programs)?

From Program Verification to Model Checking Recursion Schemes

[K. POPL 2009]

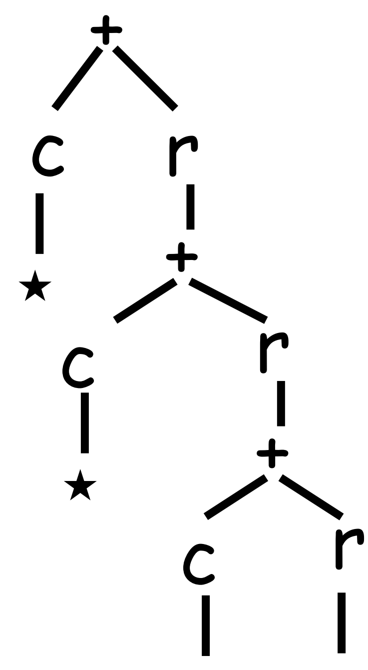
Higher-order program + specification → **Program Transformation** → Rec. scheme (describing all event sequences or outputs) + Tree automaton, recognizing valid event sequences or outputs → **Model Checking**

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?

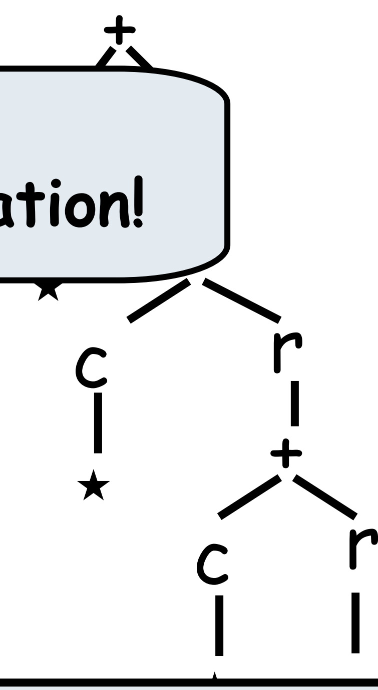
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \star$

CPS Transformation!



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

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

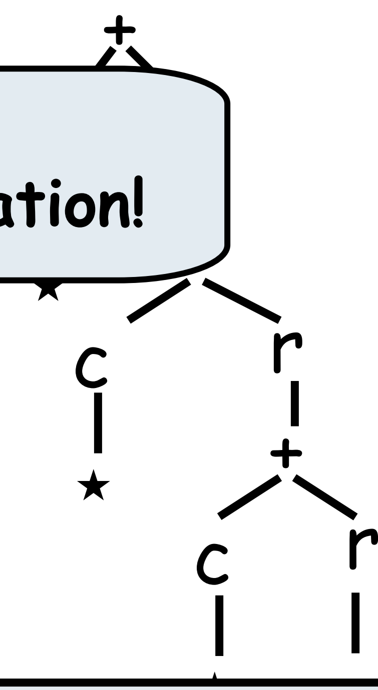
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \star$

CPS Transformation!



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

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

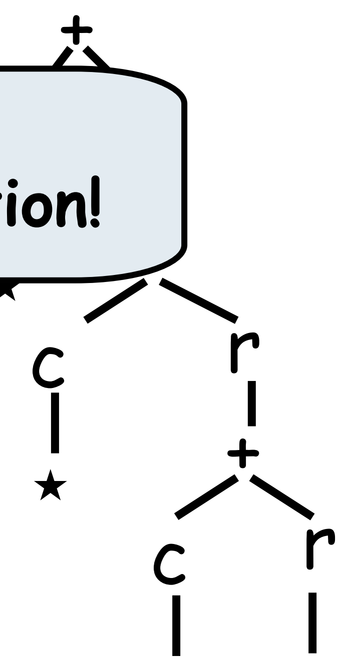
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \star$

CPS Transformation!



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

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

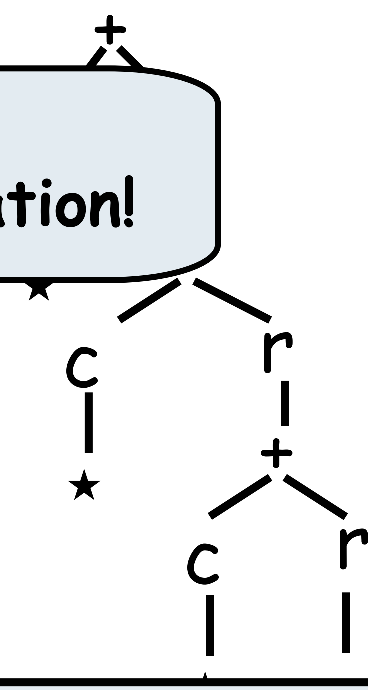
From Program Verification to Model Checking: Example

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

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

$S \rightarrow F\ d\ \star$

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)
```

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

$S \rightarrow F\ d\ \star$

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?

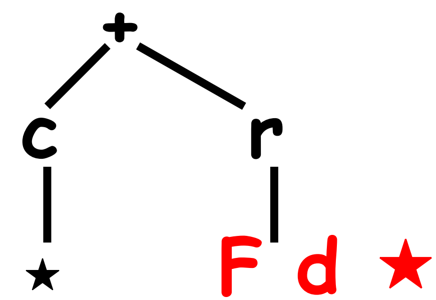
Is each path of the tree labeled by r*c?

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)
```

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

$$S \rightarrow F\ d\ \star$$



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

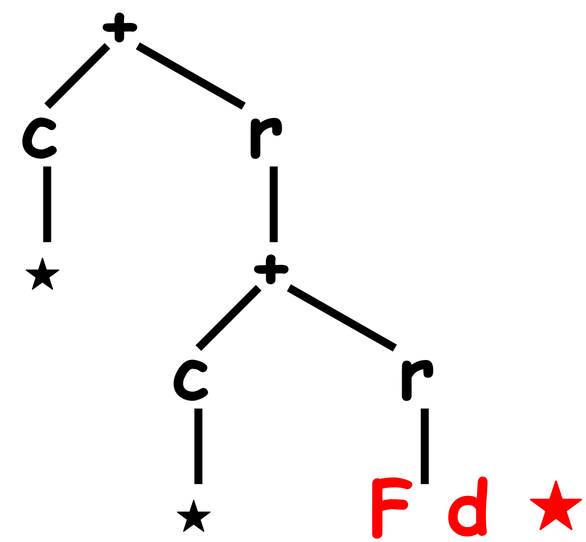
Is each path of the tree labeled by r*c?

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)
```

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

$S \rightarrow F\ d\ \star$



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

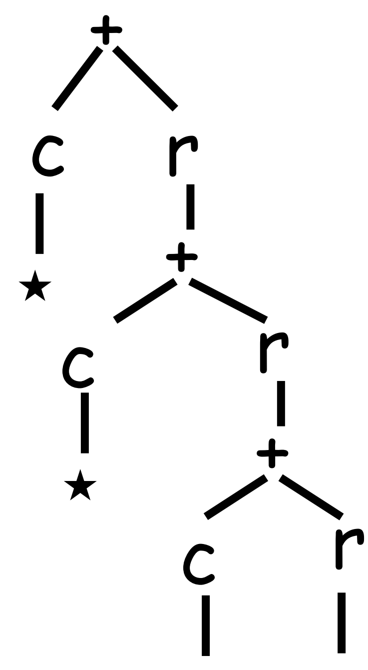
Is each path of the tree labeled by r*c?

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)
```

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

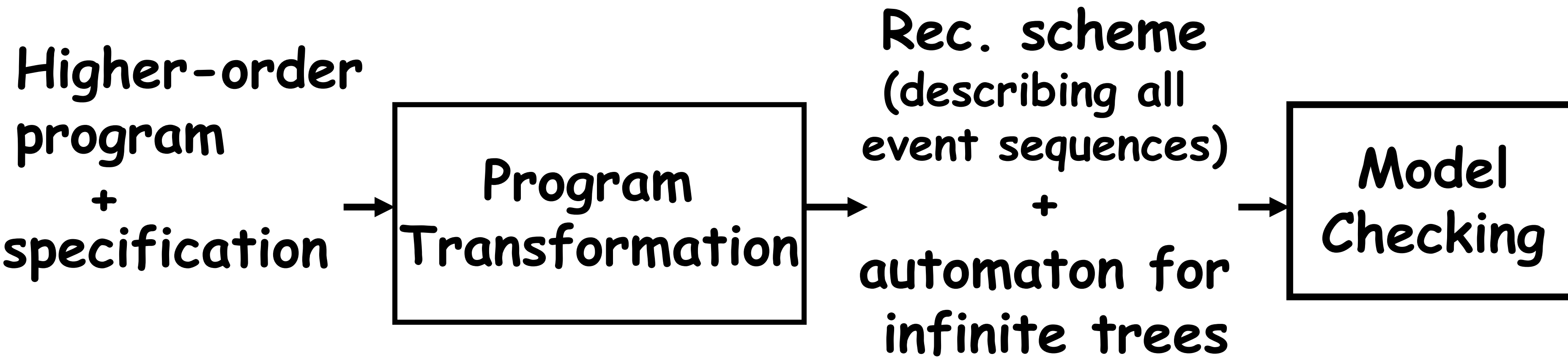
$S \to F\ d\ \star$



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 Recursion Schemes



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

Plan of the Talk

- Part 1
  - From program verification to model checking recursion schemes [K. POPL09]
  - From model checking to type checking: Simple case (safety properties) [K. POPL09]
  - Model checking (=type checking) algorithm [K. PPDP09]
- Part 2
  - From model checking to type checking: General case [K. and Ong, LICS09]
  - Towards a software model checker for higher-order languages
  - Remaining challenges

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

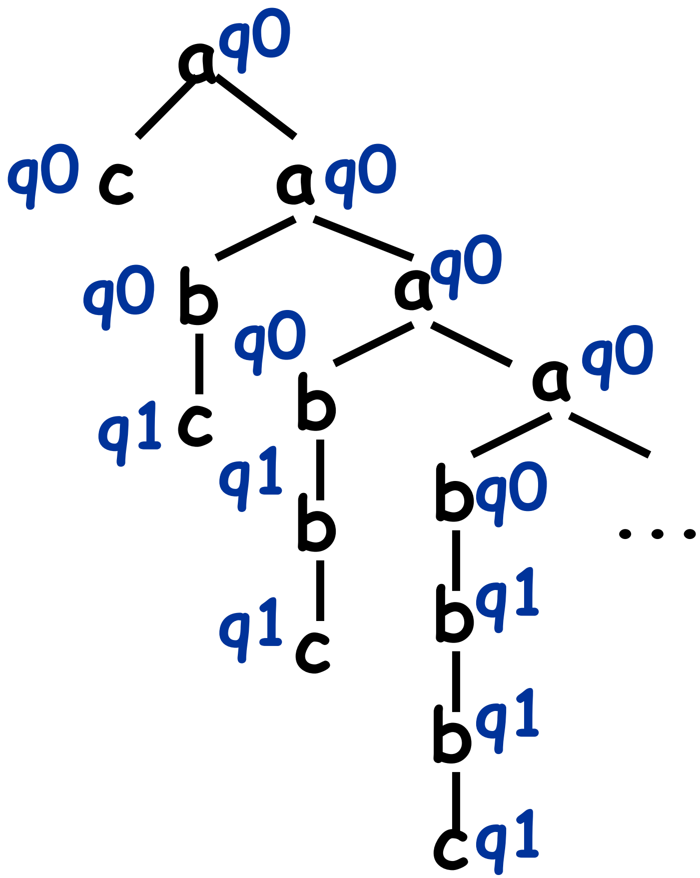
G: higher-order recursion scheme (without safety restriction)

A: trivial automaton [Aehlig CSL06] (Büchi tree automaton where all the states are accepting states)

does A accept Tree(G)?

The general case (full modal μ-calculus model checking) is discussed in Part 2

(Trivial) tree automaton for infinite trees

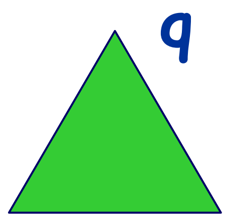


$$\delta(q0,\ a) = q0\ q0$$
$$\delta(q0,\ b) = q1$$
$$\delta(q1,\ b) = q1$$
$$\delta(q0,\ c) = \varepsilon$$
$$\delta(q1,\ c) = \varepsilon$$

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∧q2: trees accepted from both q1 and q2

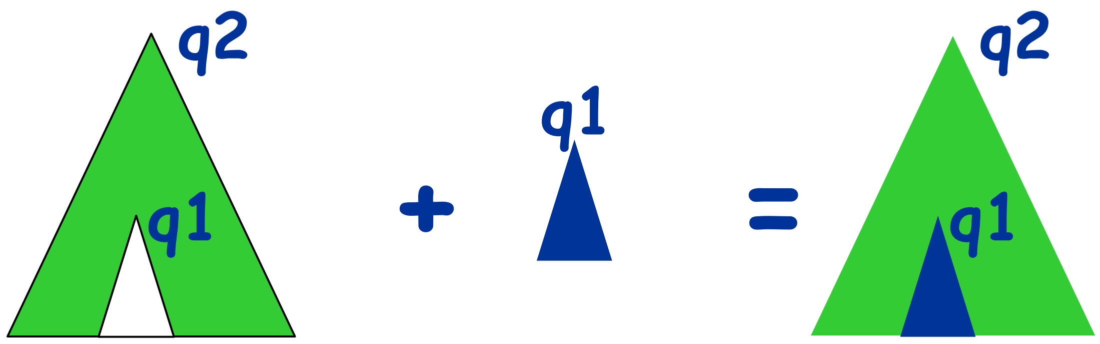
Does A accept Tree(G)?

⇓

Does Tree(G) have type q_0?

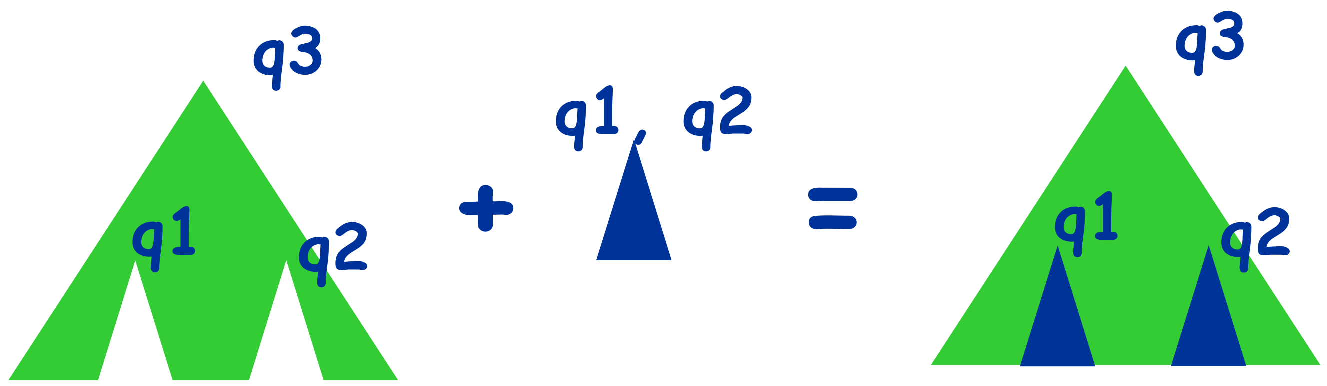
Types for Recursion Schemes

- 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 $q3$

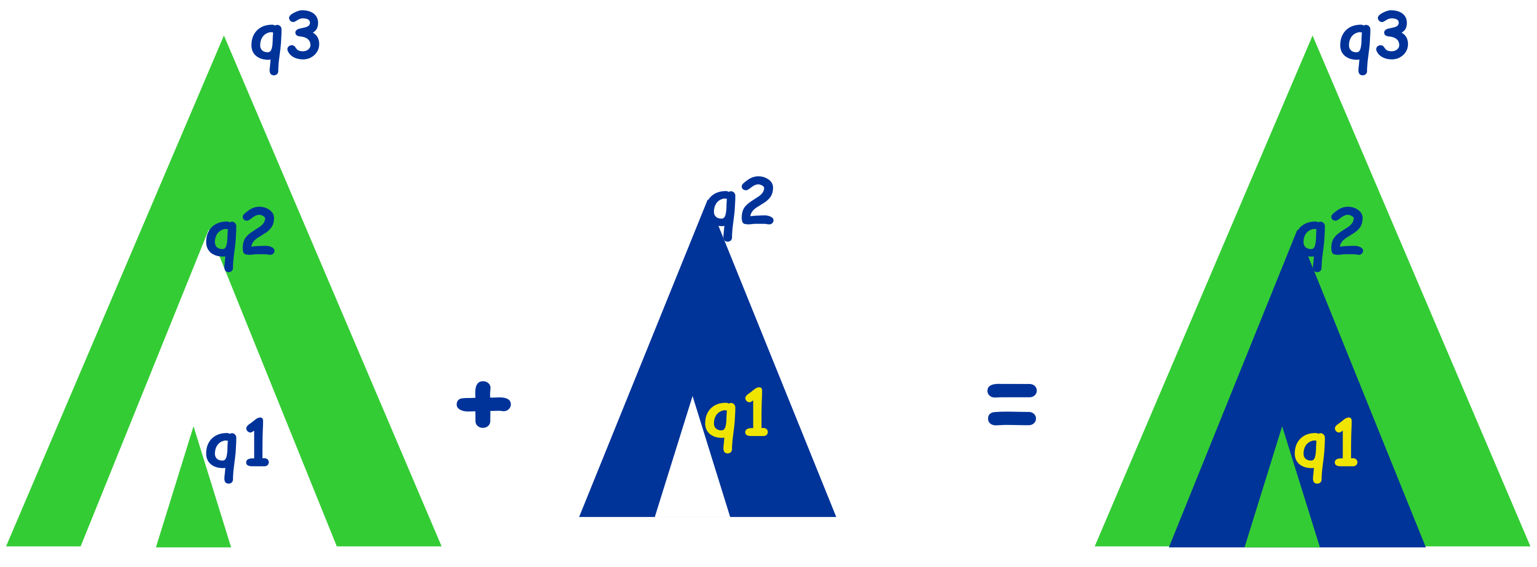


Types for Recursion Schemes

- 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 $q3$

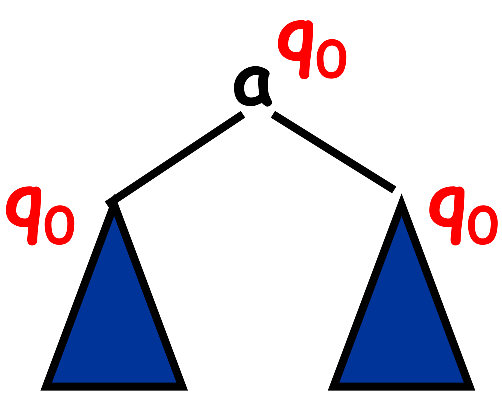


Example

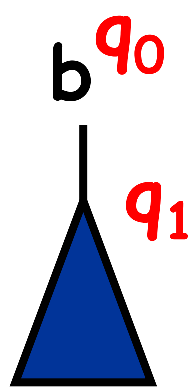
Automaton:

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

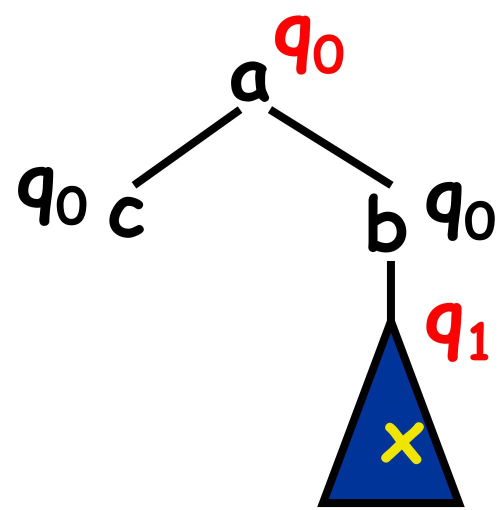
a: $q_0 \to q_0 \to q_0$



b: $q_1 \to q_0$



$\lambda x.a\ c\ (b\ x)$: $q_1 \to q_0$



Typing

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

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

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

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

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

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*)

Plan of the Talk

- Part 1
  - From program verification to model checking recursion schemes [K. POPL09]
  - From model checking to type checking: Simple case (safety properties) [K. POPL09]
  - Model checking (=type checking) algorithm
    - Naive algorithm
    - Practical algorithm
- Part 2
  - From model checking to type checking: General case [K. and Ong, LICS09]
  - Summary of our recent results
  - Ongoing and future work

Typing

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

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

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

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

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

Naïve Type Checking Algorithm

S has type q_0

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

(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 $F : o \rightarrow o$,
$\{F : \top \rightarrow q0,\ F : q0 \rightarrow q0,$
$F : q1 \rightarrow q0,$
$F : q0 \wedge q1 \rightarrow q0, \ldots\}$

Example

- **Recursion scheme:**

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

  $(S{:}o,\ F{:}\ o \to o)$

- **Automaton:**

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

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

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

$\Gamma_1 = \{\ S{:}\tau \in \Gamma_{max} \mid \Gamma_{max} \vdash F\ c{:}\tau\ \}$
$\cup\ \{\ F{:}\tau \in \Gamma_{max} \mid \Gamma_{max} \vdash \lambda x.a\ x\ (F(b\ x))\ {:}\tau\ \}$
$= \{S{:}q_0,\ S{:}q_1,\ F{:}\ q_0 \to q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\}$

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

$\Gamma_3 = \{S{:}q_0,\ F{:}\ q_0 \wedge q_1 \to q_0\} = \Gamma_2$

Naïve Algorithm Does NOT Work

S has type q_0

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

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

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

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 $S \in 2^Q$, $q \in Q$)
$(o \to o) \to o$	$2^{64} \times 4 = 2^{66}$
$((o \to o) \to o) \to o$	$2^{2^{66}} \times 4 > 10^{1000000000000000000000}$

Plan of the Talk

- Part 1
  - From program verification to model checking recursion schemes [K. POPL09]
  - From model checking to type checking: Simple case (safety properties) [K. POPL09]
  - Model checking (=type checking) algorithm
    - Naive algorithm
    - Practical algorithm
- Part 2
  - From model checking to type checking: General case [K. and Ong, LICS09]
  - Summary of our recent results
  - Ongoing and future work

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:

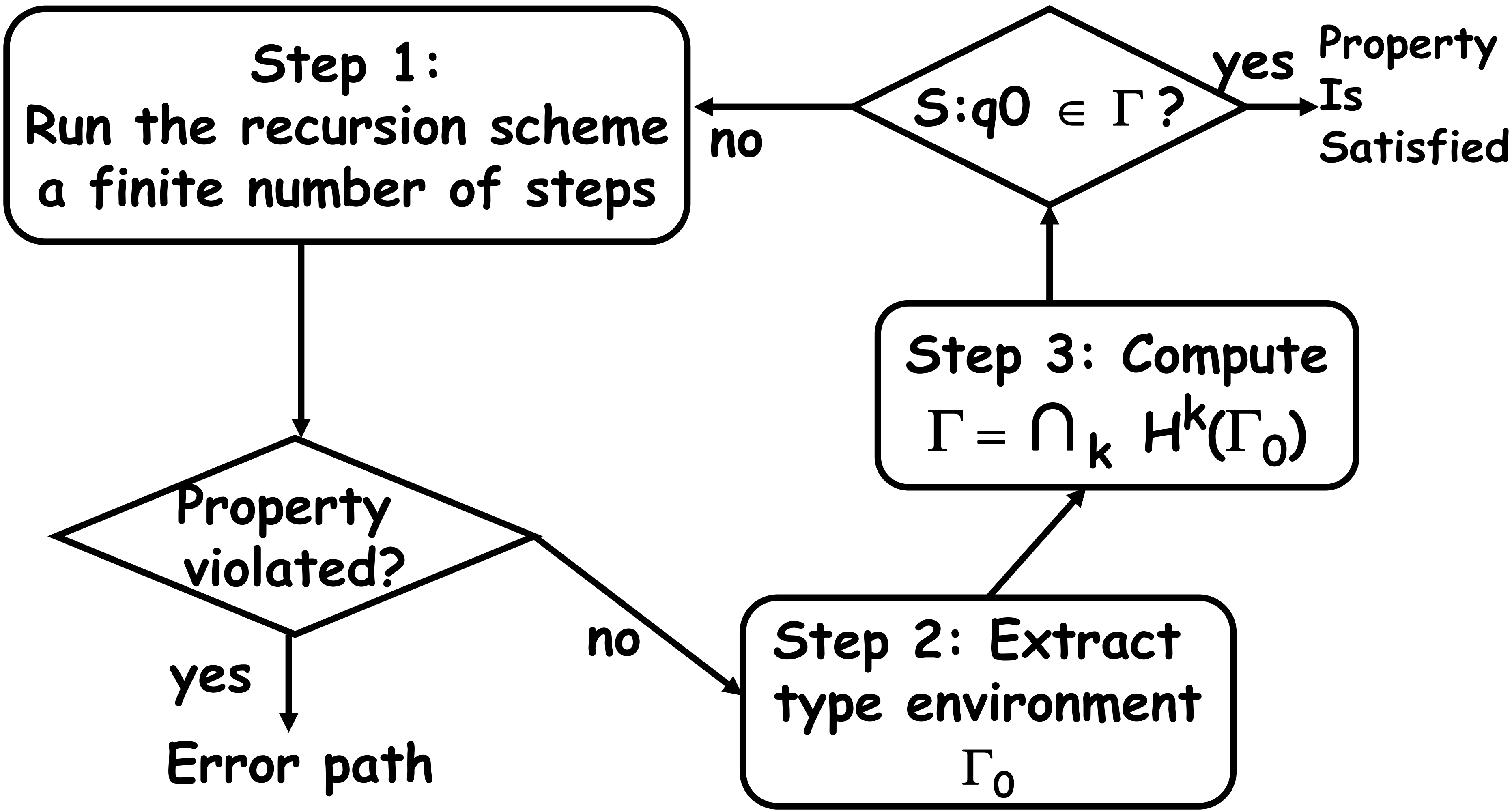
(i) How can we find an appropriate Γ_0?

"Run" 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

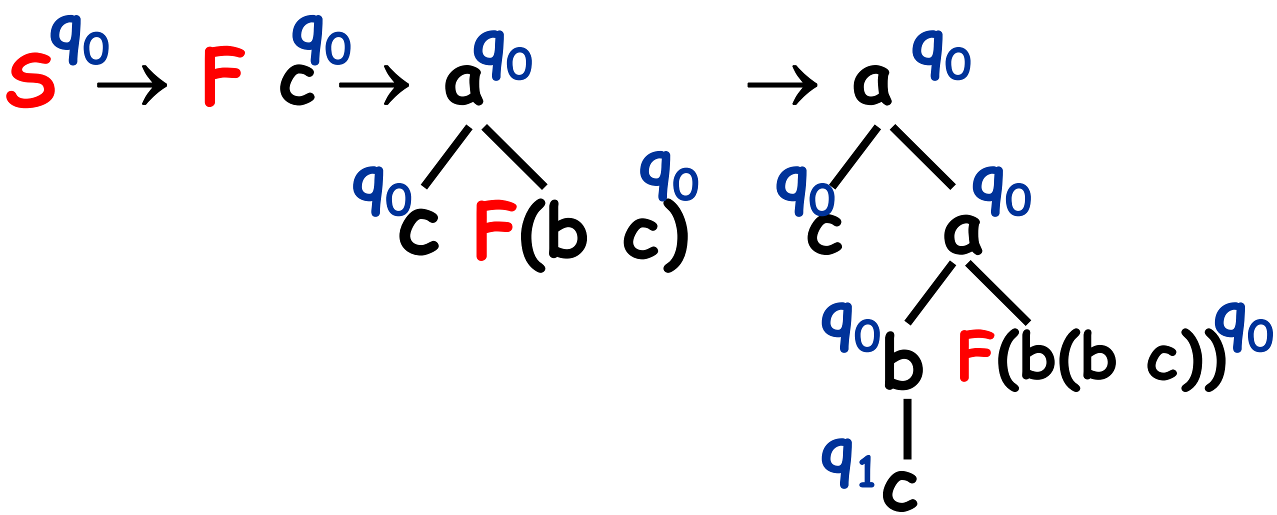
- **Recursion scheme:**

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

- **Automaton:**

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

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



Example

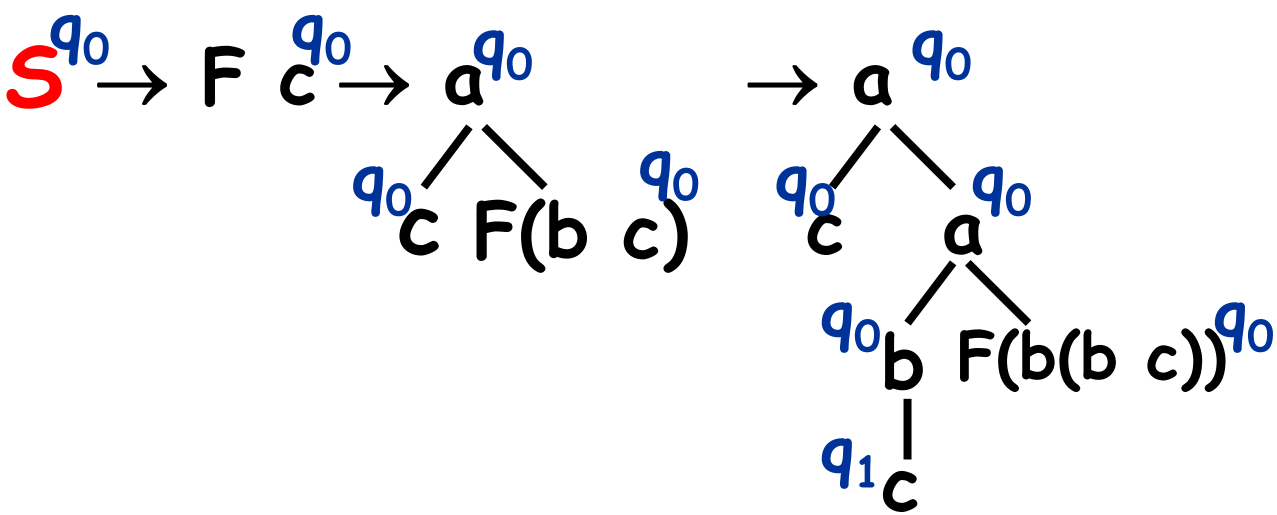
- **Recursion scheme:**

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

- **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

Example

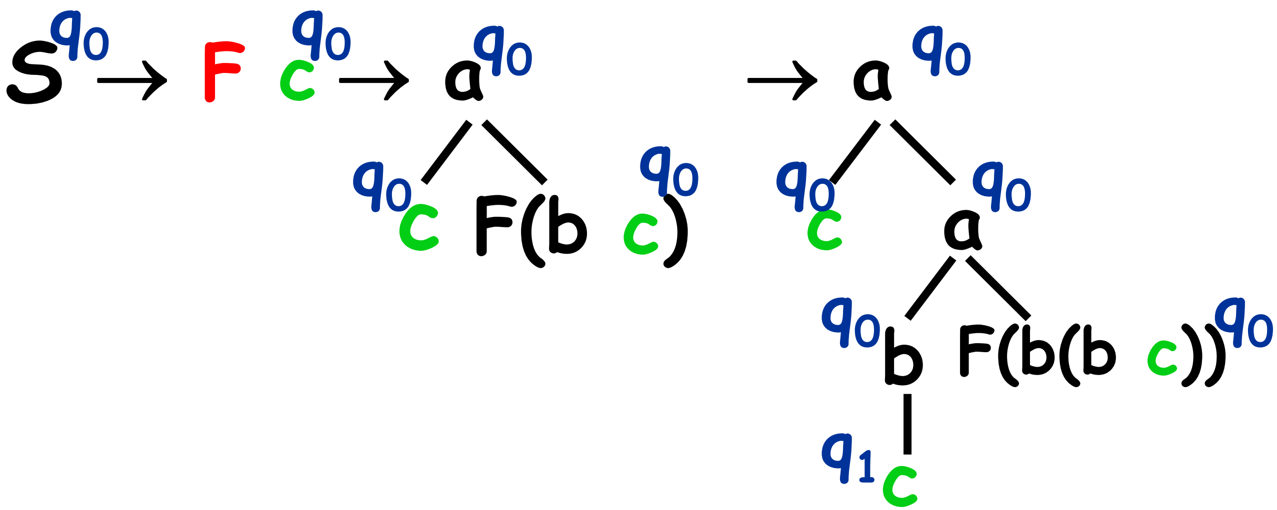
- **Recursion scheme:**

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

- **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

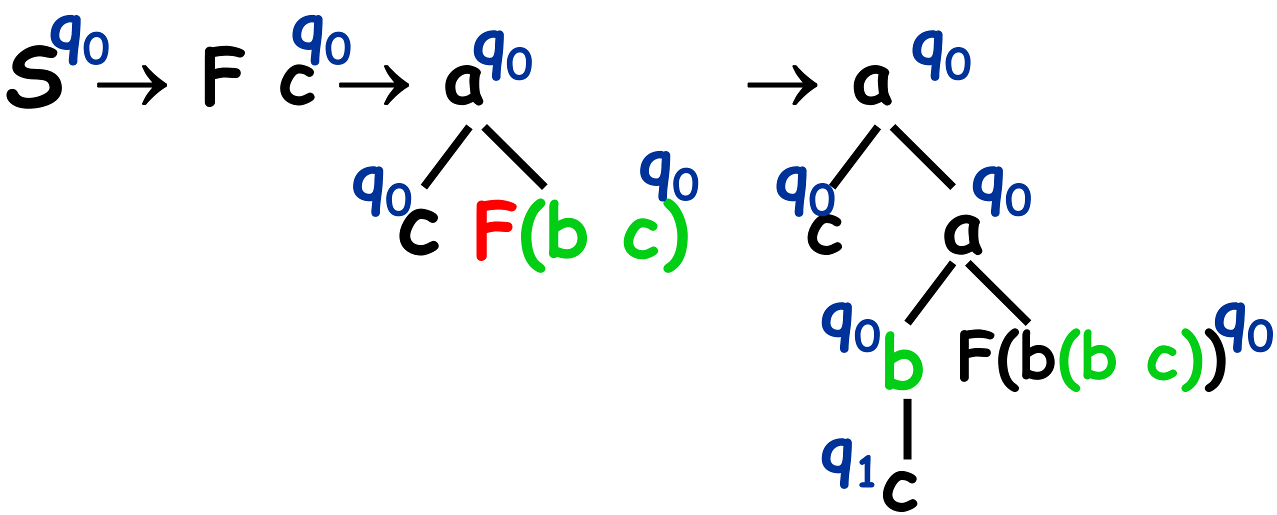
- **Recursion scheme:**

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

- **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

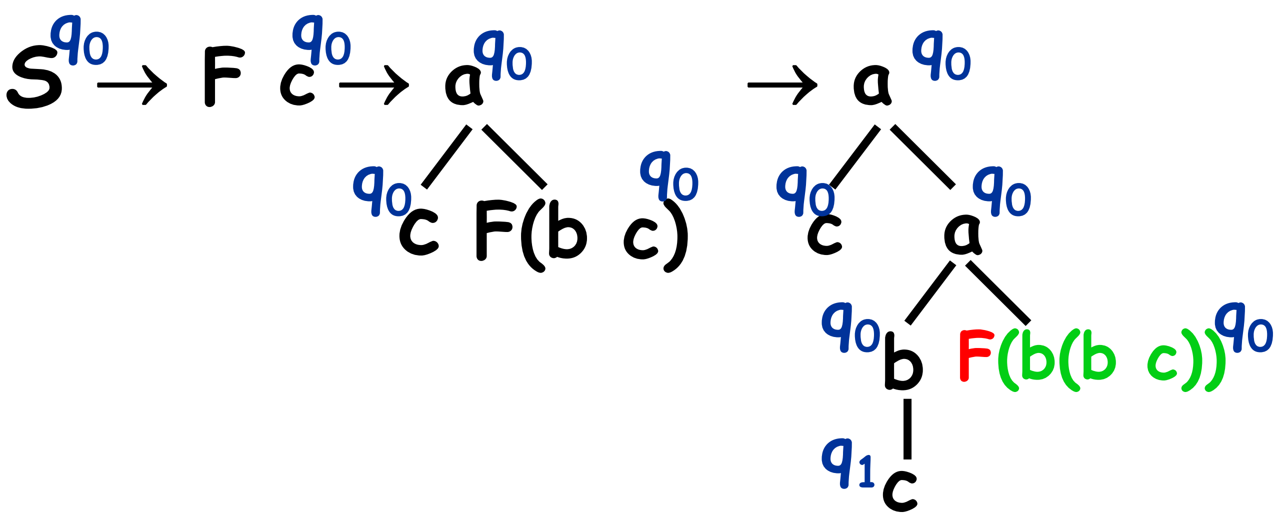
- **Recursion scheme:**

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

- **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 :

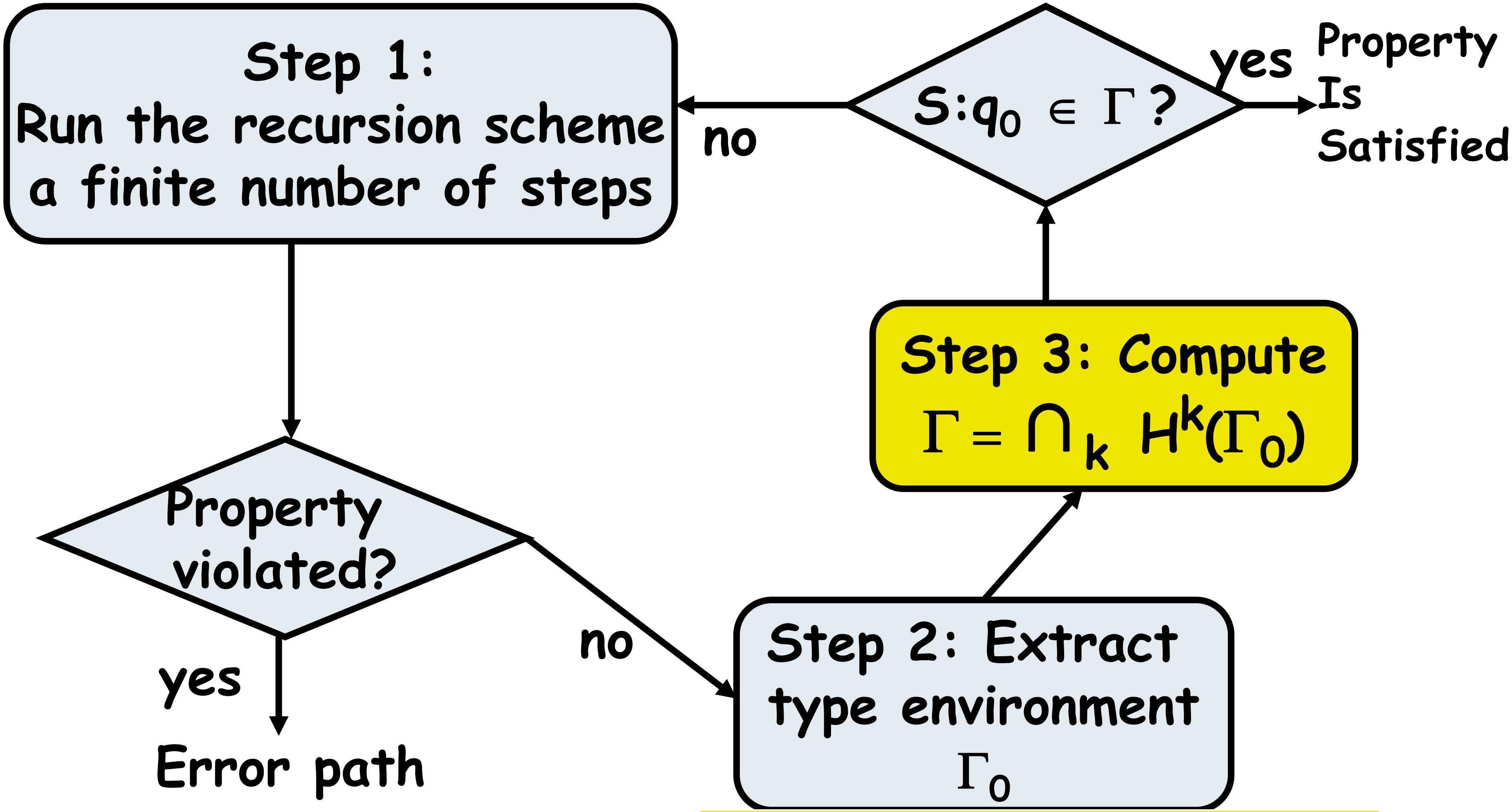
S: q_0

F: $q_0 \wedge q_1 \rightarrow q_0$

F: $q_0 \rightarrow q_0$

F: $\top \rightarrow q_0$

Example



$S: q_0$ $\quad F: q_0 \wedge q_1 \rightarrow q_0$

$F: q_0 \rightarrow q_0$ $\quad F: T \rightarrow q_0$

Example:
Filtering out invalid judgments

- **Recursion scheme:**

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

- **Automaton:**

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

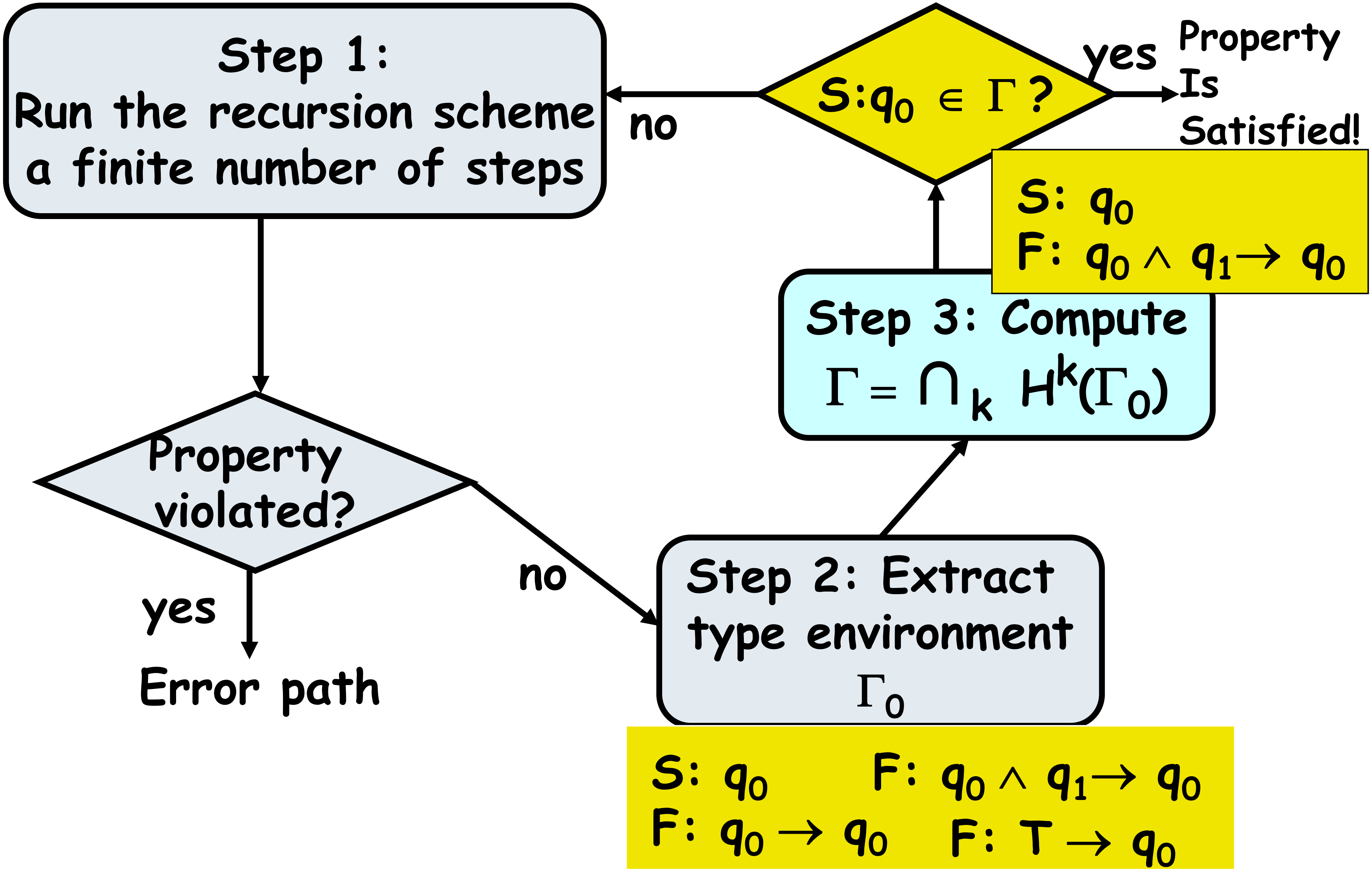
$\Gamma_0 = \{S: q_0,\ F: q_0 \wedge q_1 \to q_0,\ F: q_0 \to q_0\ ,\ F: \top \to q_0\}$

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

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

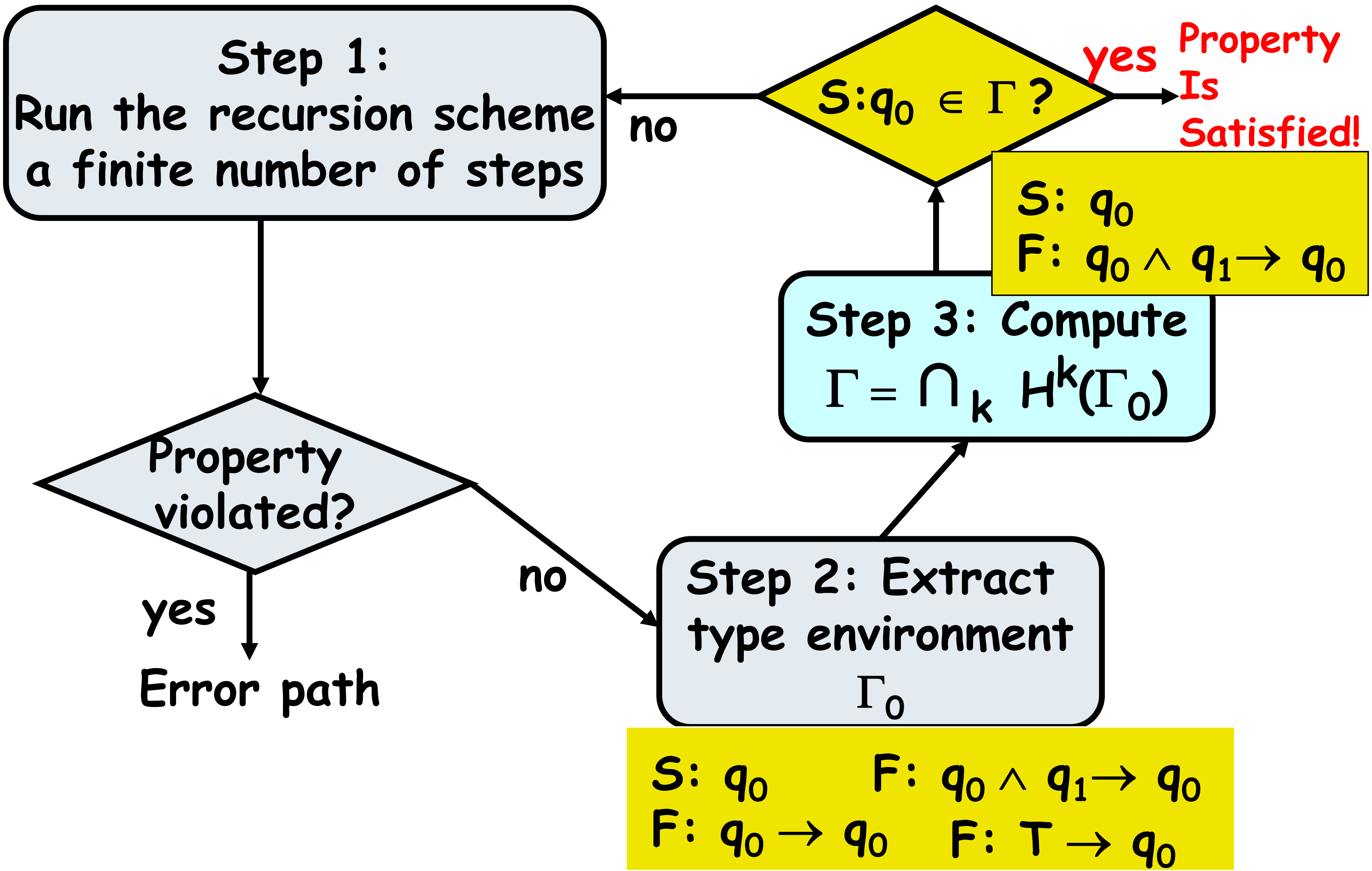
$\Gamma_3 = \{S: q_0,\ F: q_0 \wedge q_1 \to q_0\}$

Example



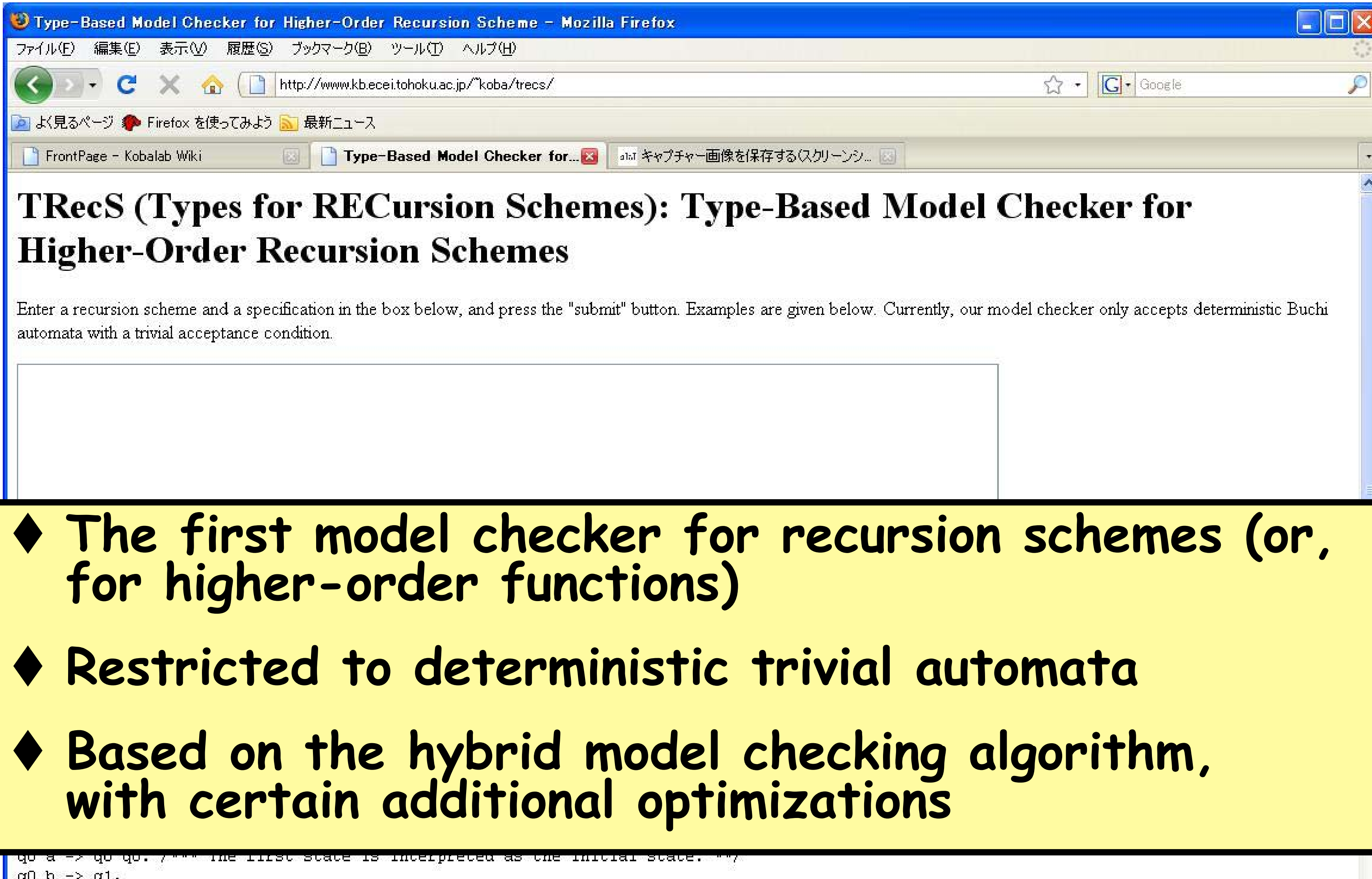
Step 1:
Run the recursion scheme a finite number of steps
Property violated?
yes
Error path
no
Step 2: Extract type environment Γ_0
S: q_0
F: $q_0 \wedge q_1 \to q_0$
F: $q_0 \to q_0$
F: $T \to q_0$
Step 3: Compute $\Gamma = \bigcap_k H^k(\Gamma_0)$
S: q_0
F: $q_0 \wedge q_1 \to q_0$
S:$q_0 \in \Gamma$?
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)
- Restricted to deterministic trivial automata
- Based on the hybrid model checking algorithm, with certain additional optimizations

Experiments

	order	rules	states	result	Time (msec)
Twofiles	4	11	4	Yes	2
FileWrong	4				
TwofilesE	4				
FileOcamlC	4	23	4	Yes	5
Lock	4	11	3	Yes	5
Order5	5	9	4	Yes	2

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
```

Demonstration

Conclusion (for Part I)

- Recursion schemes are very relevant to program verification, hence of practical interest
- Type-based approach gives a simple, efficient model checking algorithm
- Despite the disappointing worst case complexity, the model checking of recursion schemes may be tractable for realistic inputs