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)

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

Model Checking Problem
(Simple Case, for safety properties)

Given

G: higher-order recursion scheme

A: trivial automaton [Aehlig CSL06]

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

does A accept Tree(G)?

Model Checking Problem: General Case

Given

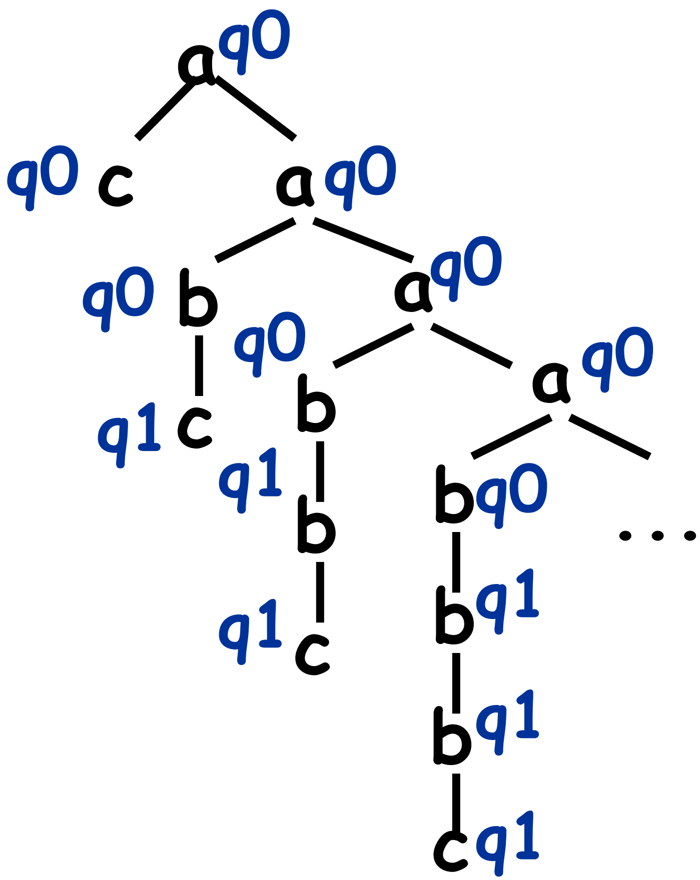
G: higher-order recursion scheme

A: alternating parity tree automaton

(or modal μ-calculus formula)

Does A accept Tree(G)?

Alternating parity tree automata for infinite trees



Positive boolean formulas

$\delta(q0, a) = ((1, q0) \wedge (2, q0)) \vee (1, q1)$

$\delta(q0, b) = (1, q1)$

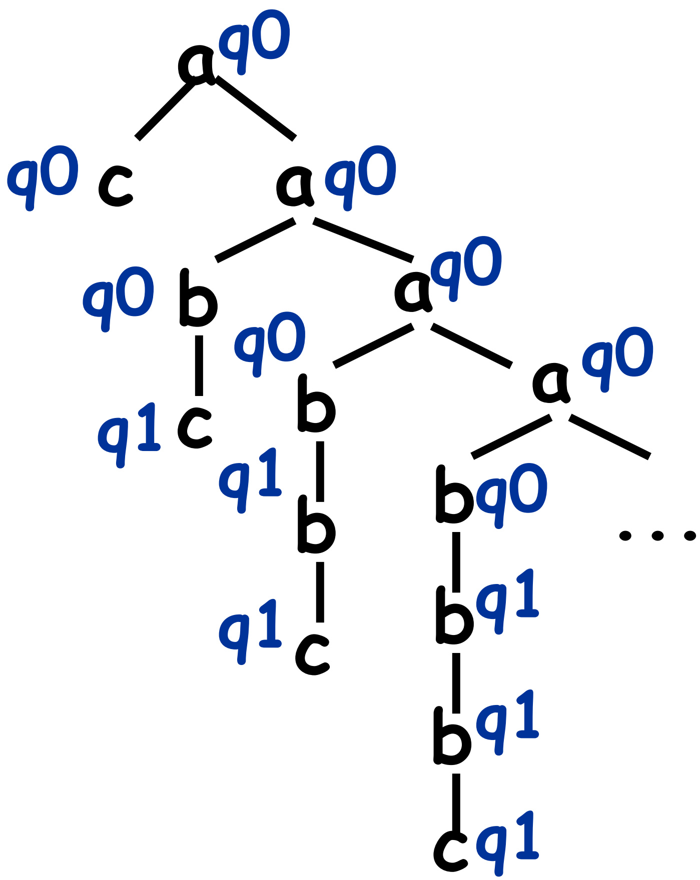
$\delta(q1, b) = (1, q1)$

$\delta(q0, c) = \text{true}$

$\delta(q1, c) = \text{true}$

Alternating parity tree automata for infinite trees

Positive boolean formulas



$\delta(q0, a) = ((1,q0) \wedge (2,q0)) \vee (1, q1)$

$\delta(q0, b) = (1, q1)$

$\delta(q1, b) = (1, q1)$

$\delta(q0, c) = \text{true}$

$\delta(q1, c) = \text{true}$

Priority function:

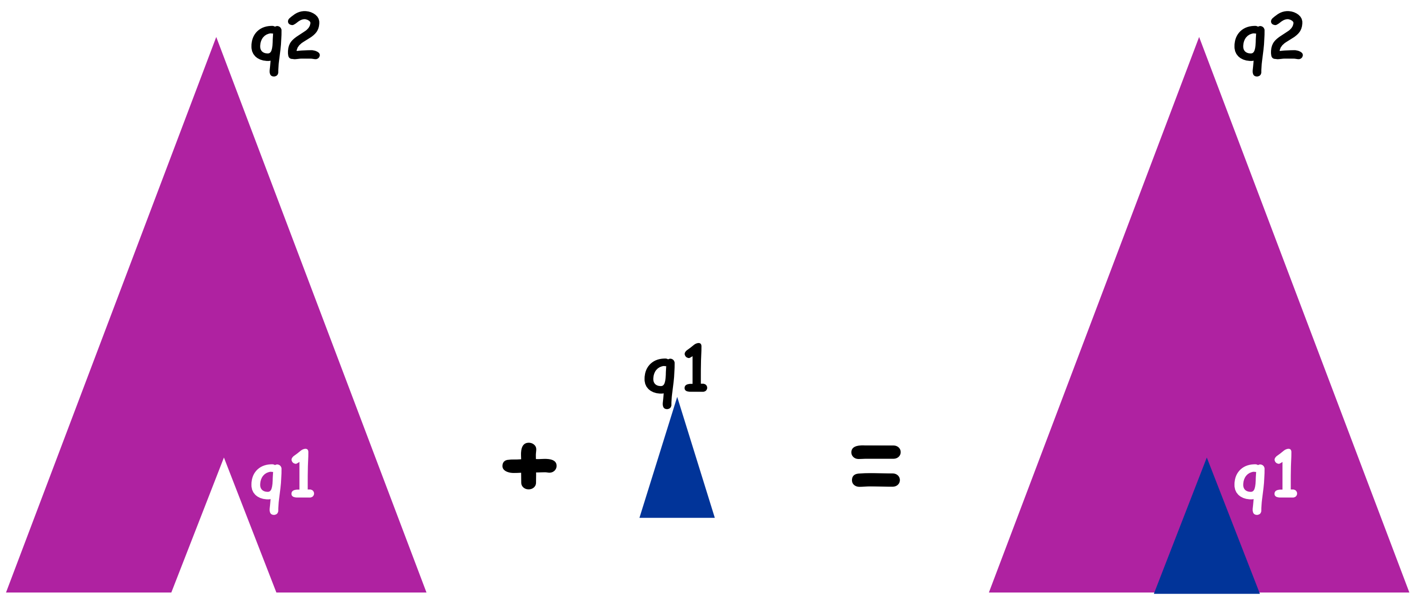
$\Omega(q0) = 1$

$\Omega(q1) = 2$

Acceptance condition: For any infinite path of the run tree, the largest priority visited infinitely often must be even.

Types extended with priorities

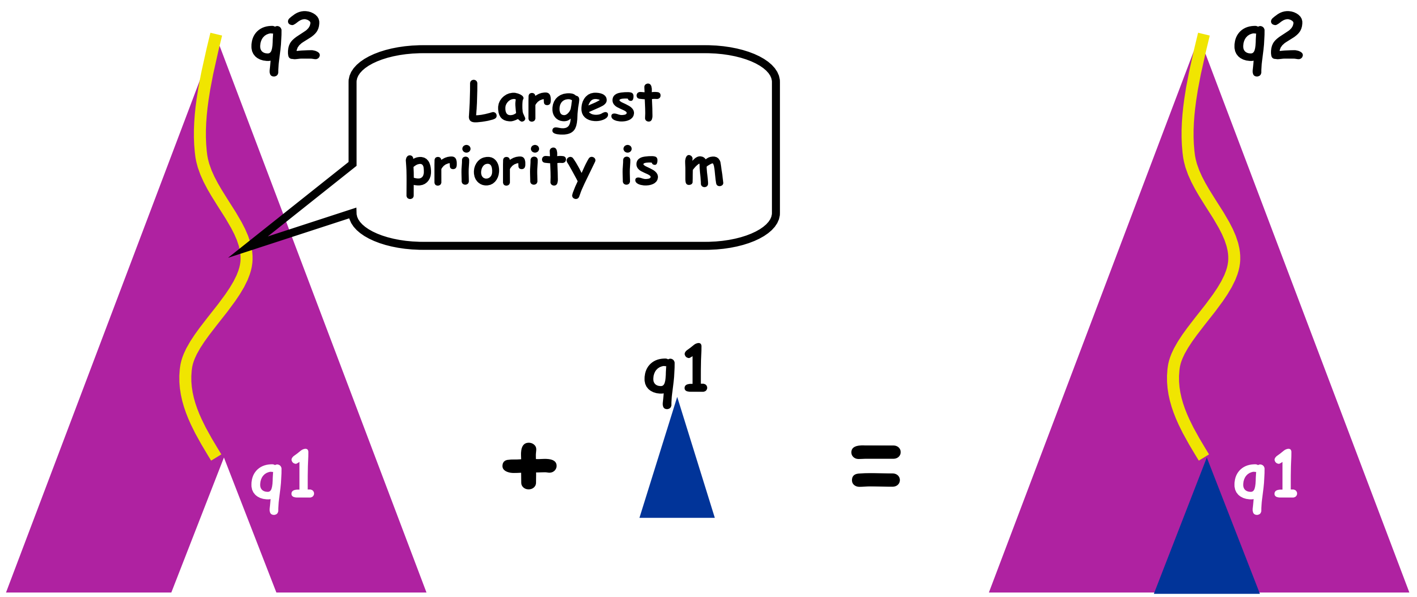
q1→ q2: functions that take a tree of type q1
and return a tree of q2



Types extended with priorities

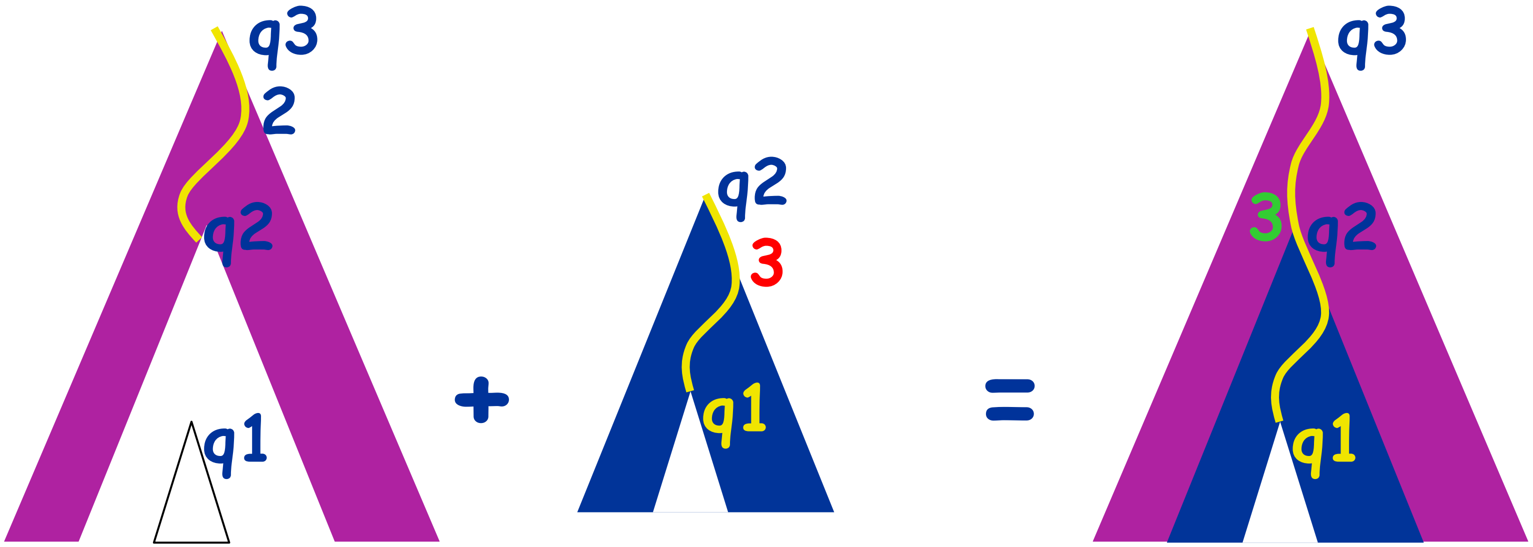
(q1, m) → q2: functions that take a tree of type q1 and return a tree of q2

priority



Types extended with priorities

((q1, 3) → q2, 2) → (q1, 3) → q3 :

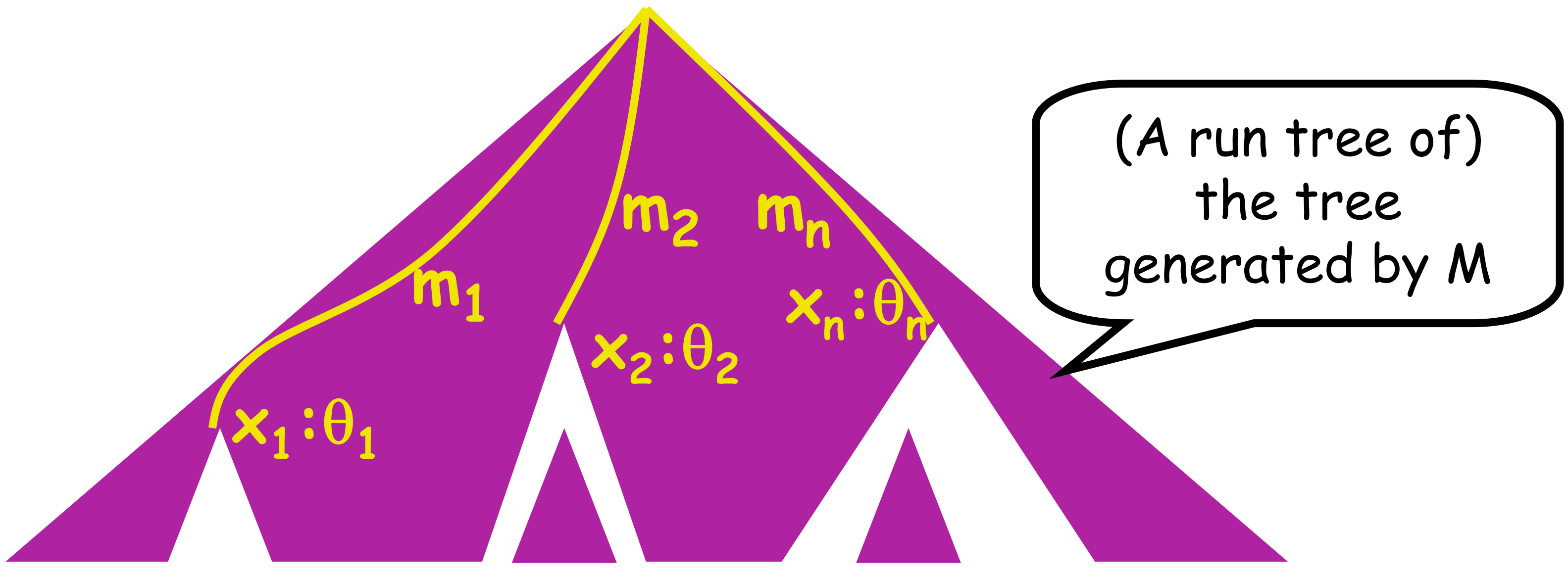


Type judgment

$x_1\colon (\theta_1, m_1), \ldots, x_n\colon (\theta_n, m_n) \vdash M\colon \theta$

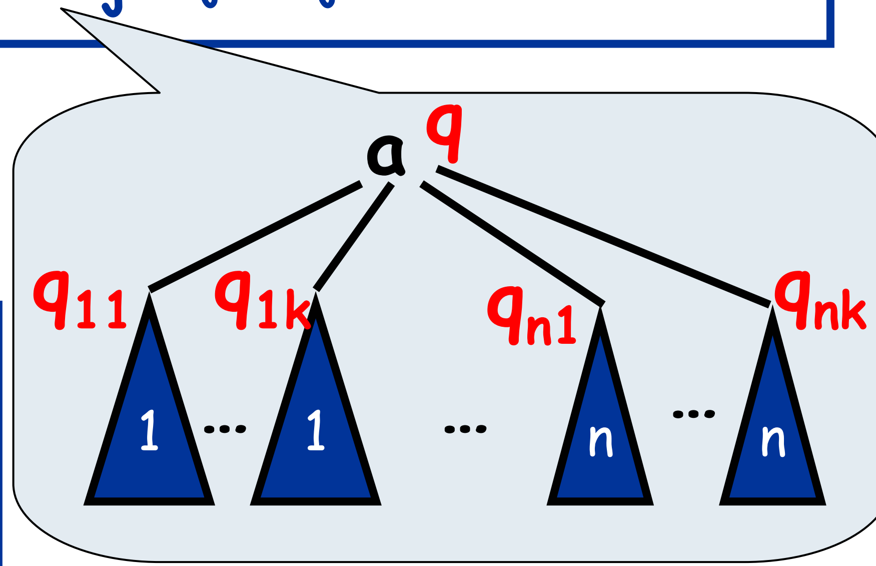
where

$$\theta ::= q \mid (\theta_1, m_1) \wedge \ldots \wedge (\theta_n, m_n) \to \theta$$



Typing

$$\frac{\{(i,q_{ij}) \mid i\in 1,\ldots,n,\ j\in 1,\ldots,k_i\} \text{ satisfies } \delta(q,a) \qquad m_{ij} = \max(\Omega(q_{ij}),\Omega(q))}{\vdash a : \wedge_j(q_{1j},m_{1j}) \to \ldots \to \wedge_j(q_{nj},m_{nj}) \to q}$$



$$x:(\theta,\Omega(\theta)) \vdash x:\theta$$

$$\frac{\Gamma, x:\tau_1,\ldots, x:\tau_k \vdash t:\theta \qquad k\le n}{\Gamma \vdash \lambda x.t: \tau_1\wedge\ldots\wedge\tau_n \to \theta}$$

$$\frac{\Gamma_0 \vdash t_1: (\theta_1,m_1)\wedge\ldots\wedge(\theta_n,m_n) \to \theta \qquad \Gamma_i \vdash t_2:\theta_i\ (i=1,\ldots n)}{\Gamma_0 \cup \Gamma_1\uparrow m_1 \cup \ldots\cup \Gamma_n\uparrow m_n \vdash t_1\, t_2:\theta}$$

Typing

$\{(i,q_{ij}) \mid i \in 1,..,n,\ j \in 1,..,k_i\}$ satisfies $\delta(q,a)$

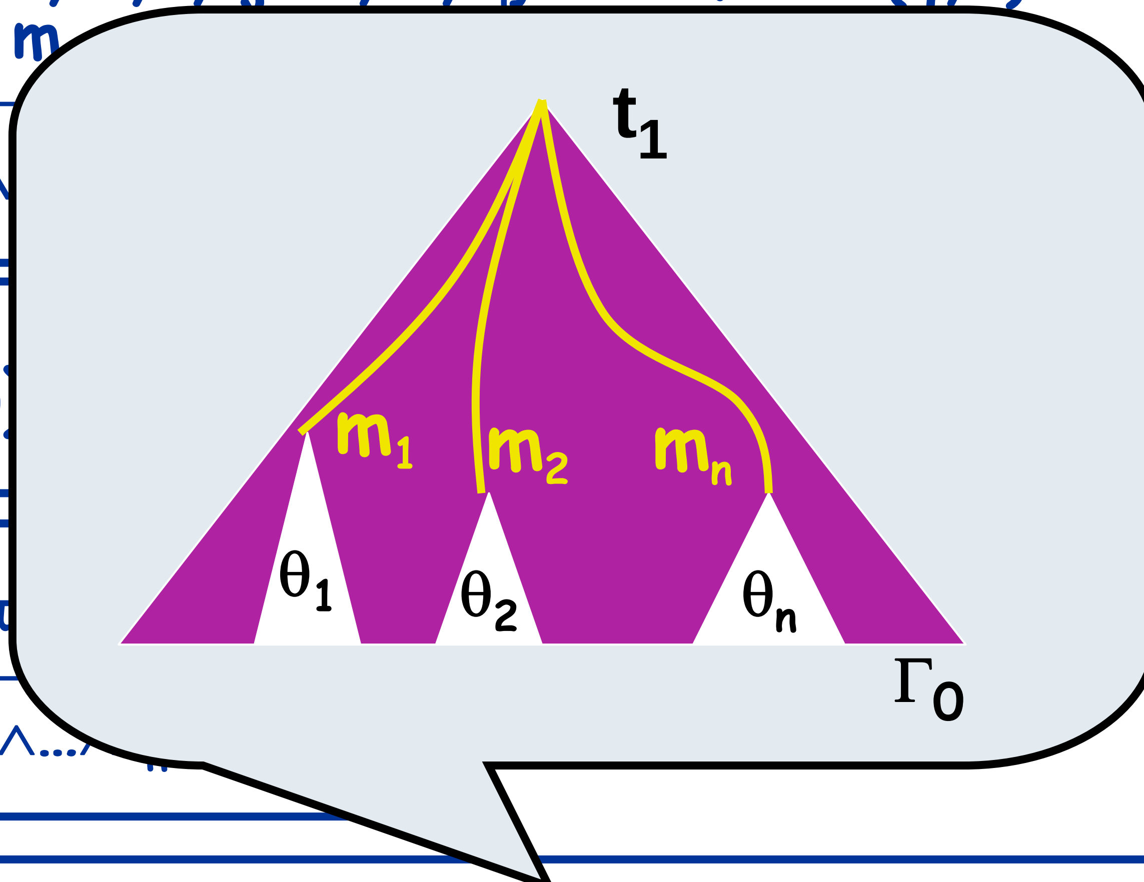
m

$\vdash a : \wedge$

$\Gamma, x{:}(\theta,\Omega(\theta))$

$\Gamma, x{:}\tau_1, \ldots, x{:}\tau$

$\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge$



$$\frac{\Gamma_0 \vdash t_1{:}\ (\theta_1,m_1)\wedge\ldots\wedge(\theta_n,m_n) \rightarrow \theta \qquad \Gamma_i \vdash t_2{:}\theta_i\ (i=1,..n)}{\Gamma_0 \cup \Gamma_1\uparrow m_1 \cup \ldots \cup \Gamma_n\uparrow m_n \vdash t_1\ t_2{:}\theta}$$

Typing

$\{(i,q_{ij}) \mid i \in 1,..,n,\ j \in 1,..,k_i\}$ satisfies $\delta(q,a)$

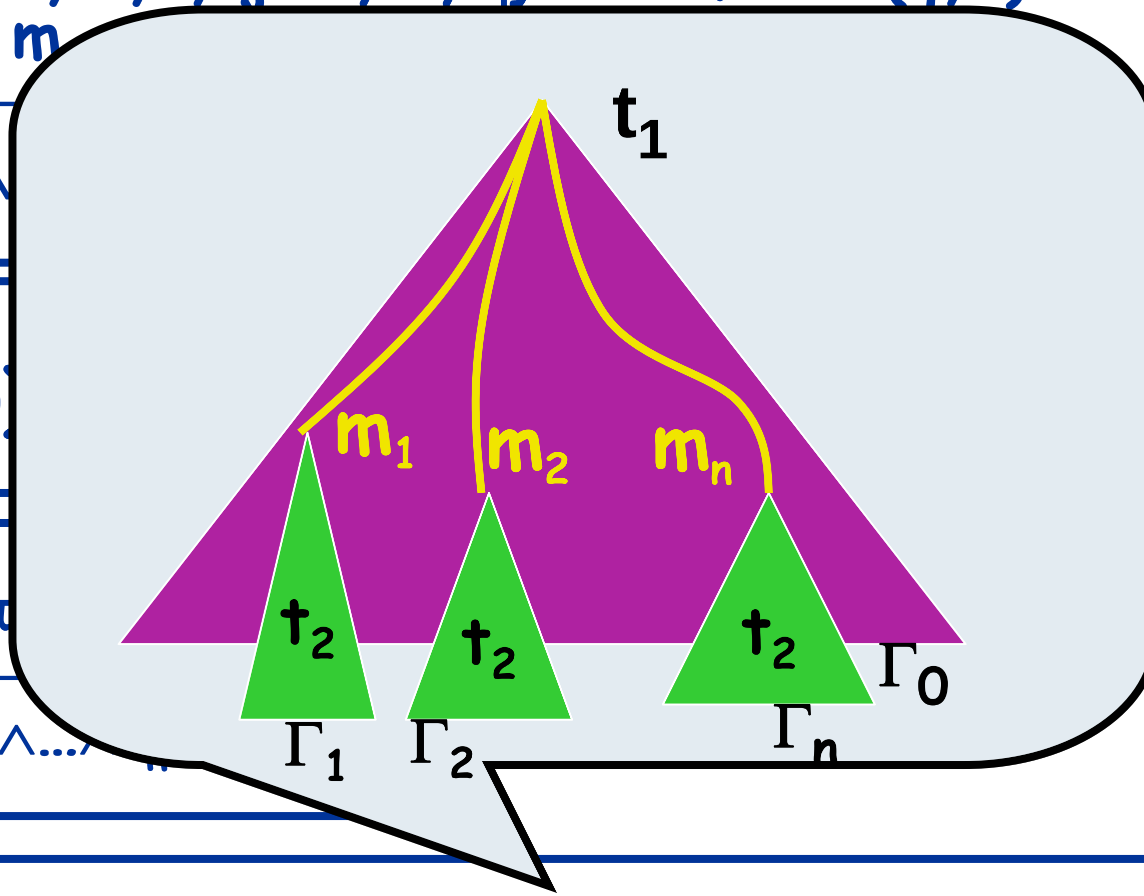
m

$\vdash a : \wedge$

$\Gamma, x{:}(\theta,\Omega(\theta))$

$\Gamma, x{:}\tau_1, \ldots, x{:}\tau$

$\Gamma \vdash \lambda x.t{:}\ \tau_1 \wedge \ldots \wedge$



$\Gamma_0 \vdash t_1{:}\ (\theta_1,m_1) \wedge \ldots \wedge (\theta_n,m_n) \rightarrow \theta \qquad \Gamma_i \vdash t_2{:}\theta_i\ (i=1,..n)$

$\Gamma_0 \cup \Gamma_1 \uparrow m_1 \cup \ldots \cup \Gamma_n \uparrow m_n \vdash t_1\ t_2{:}\theta$

Typing for Recursion?

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

Parity conditions are not respected!

Recursion and parity conditions

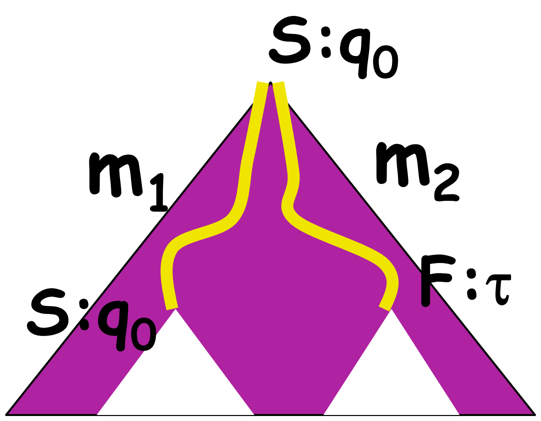
Recursion scheme:
$S \rightarrow t$
$F \rightarrow u$

Typing:
S: (q_0, m_1), F: (τ, m_2) |- t: q_0
S: (q_0, m_3), F: (τ, m_4) |- u: τ

Recursion and parity conditions

Recursion scheme:
$S \rightarrow t$
$F \rightarrow u$

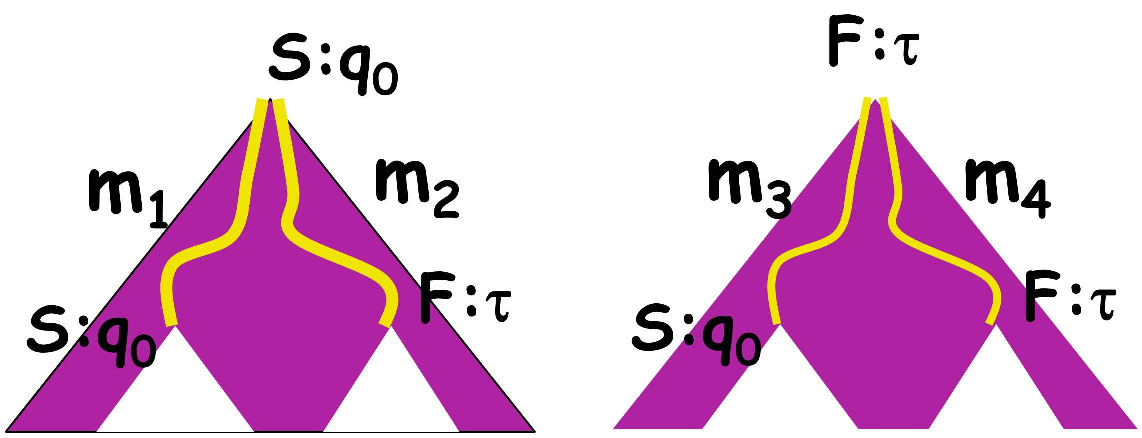
Typing:
$S: (q_0, m_1), F: (\tau, m_2) \vdash t: q_0$
$S: (q_0, m_3), F: (\tau, m_4) \vdash u: \tau$



Recursion and parity conditions

Recursion scheme:
$S \rightarrow t$
$F \rightarrow u$

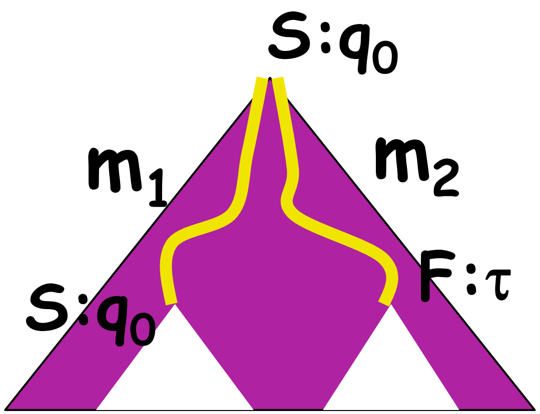
Typing:
$S: (q_0, m_1), F: (\tau, m_2) \vdash t: q_0$
$S: (q_0, m_3), F: (\tau, m_4) \vdash u: \tau$

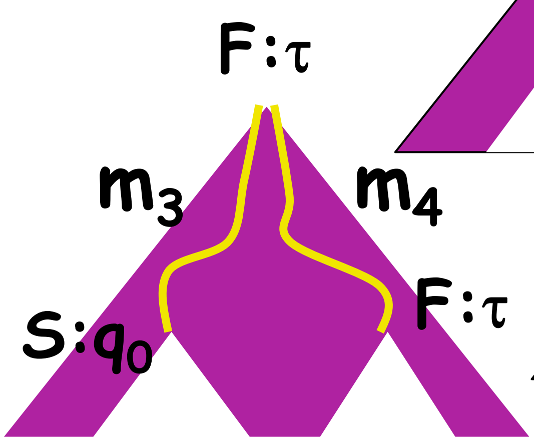


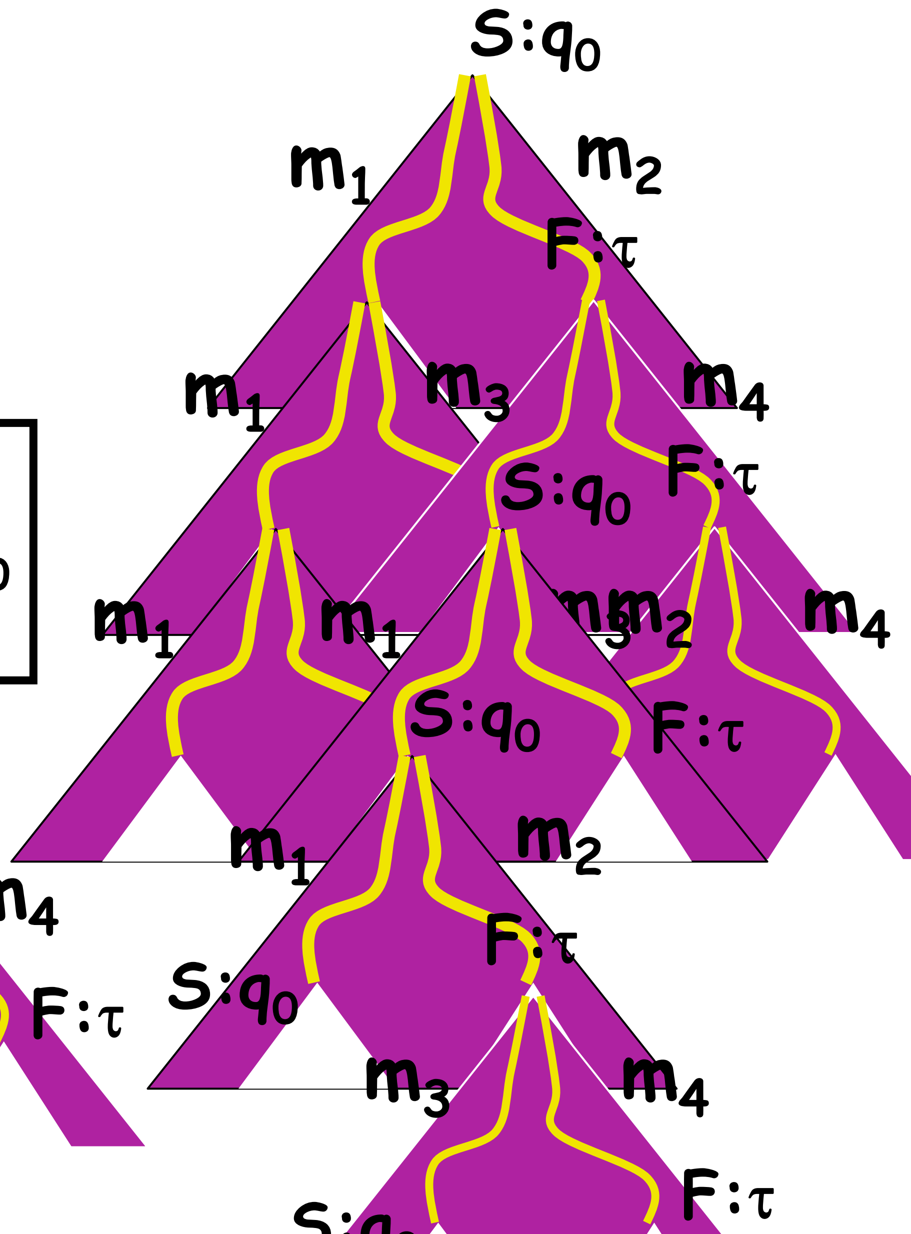
Recursion and parity conditions

Recursion scheme:
$S \rightarrow t$
$F \rightarrow u$

Typing:
$S: (q_0, m_1), F: (\tau, m_2) \mid\!- \ t: q_0$
$S: (q_0, m_3), F: (\tau, m_4) \mid\!- \ u: \tau$







Recursion and parity conditions

Recursion scheme:

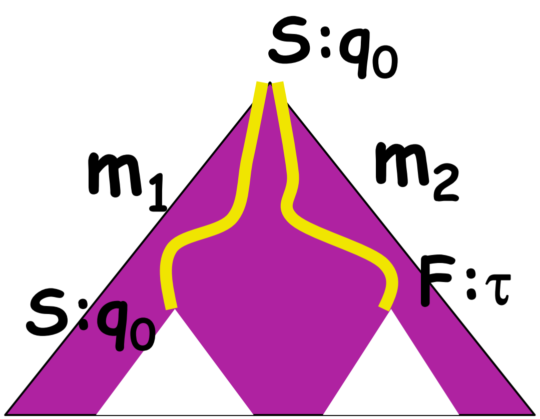
$S \to t$

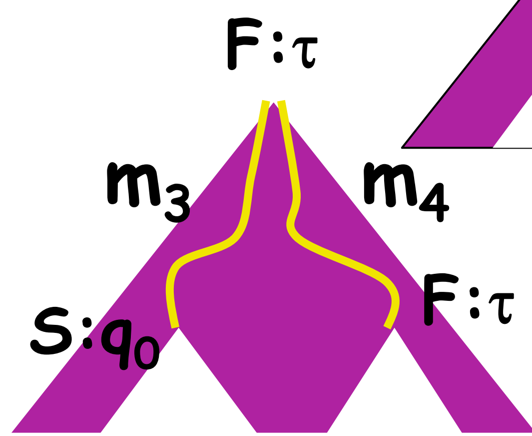
$F \to u$

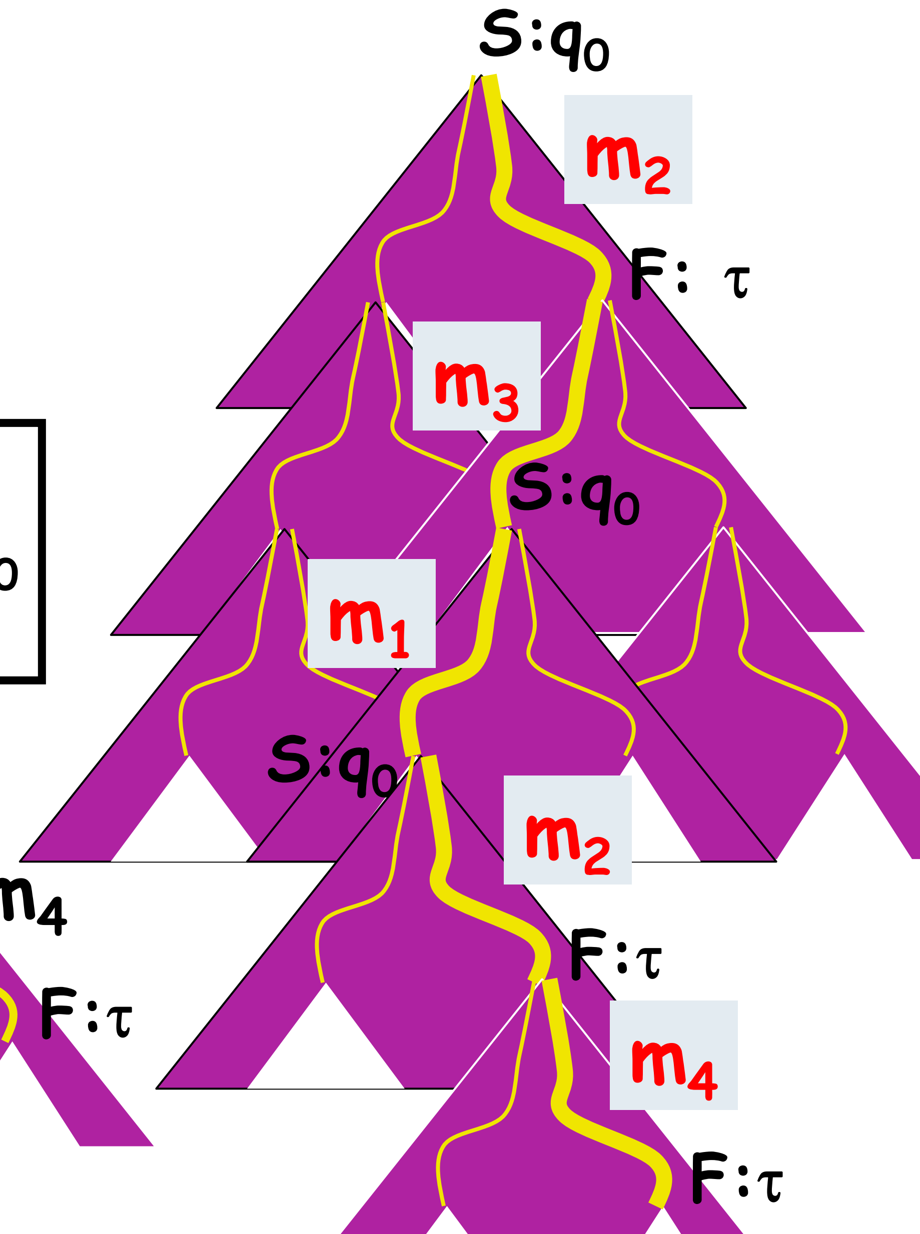
Typing:

S: (q_0, m_1), F: (τ, m_2) |- t: q_0

S: (q_0, m_3), F: (τ, m_4) |- u: τ







Typability as Parity Game

Initial state: S:(q_0, 0)

priority

Player (P): Given F:(τ, m),

pick Γ such that $\Gamma \vdash t_F : \tau$

r.h.s of F's rule

Opponent (O): Given Γ,

pick F:(τ, m) $\in \Gamma$

(and ask P to show why F has type τ)

Definition: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S:(q_0, 0)$

priority

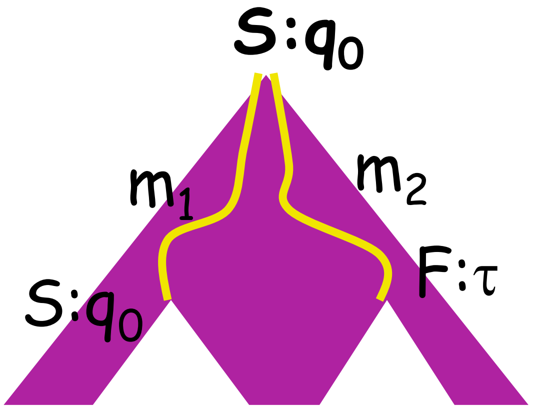
Player (P): Given $F:(\tau, m)$,

pick Γ such that $\Gamma \vdash t_F : \tau$

Opponent (O): Given Γ,

pick $F:(\tau, m) \in \Gamma$

(and ask P to show why F has type τ)

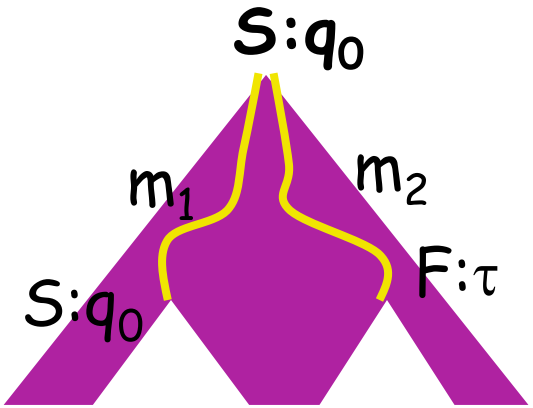


Definition: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S{:}(q_0, 0)$ (priority)

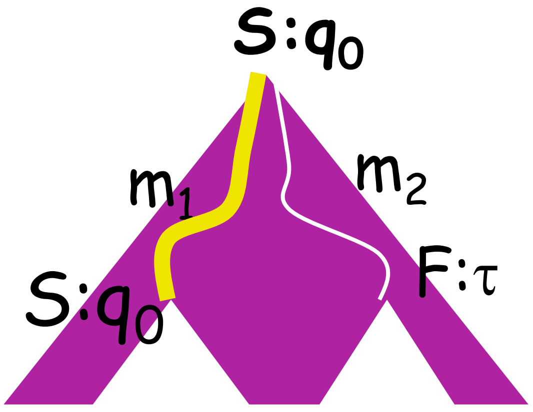
Player (P): Given $F{:}(\tau, m)$,
pick Γ such that $\Gamma \vdash t_F : \tau$



Opponent (O): Given Γ,
pick $F{:}(\tau, m) \in \Gamma$
(and ask P to show why F has type τ)

<u>Definition</u>: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game



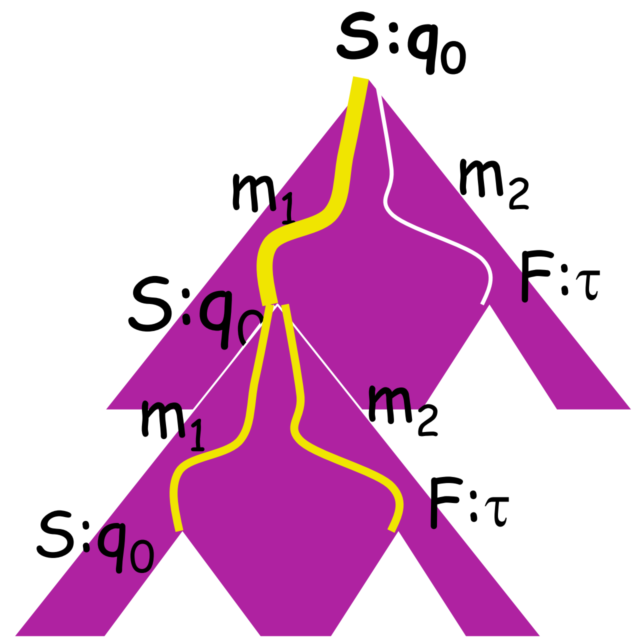
Initial state: S:(q_0, 0)

Player (P): Given F:(τ, m),
pick Γ such that $\Gamma \vdash t_F : \tau$

Opponent (O): Given Γ,
pick F:(τ, m) $\in \Gamma$
(and ask P to show
why F has type τ)

Definition: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game



Initial state: S:(q_0, 0)

Player (P): Given F:(τ, m),
pick Γ such that $\Gamma \vdash t_F : \tau$

Opponent (O): Given Γ,
pick F:(τ, m) $\in \Gamma$
(and ask P to show why F has type τ)

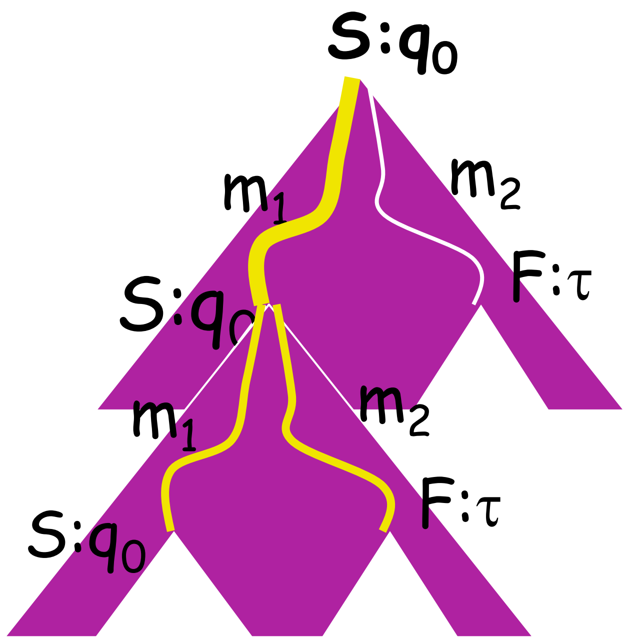
<u>Definition</u>: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S:(q_0, 0)$

Player (P): Given $F:(\tau, m)$,
pick Γ such that $\Gamma \vdash t_F: \tau$

Opponent (O): Given Γ,
pick $F:(\tau, m) \in \Gamma$
(and ask P to show why F has type τ)



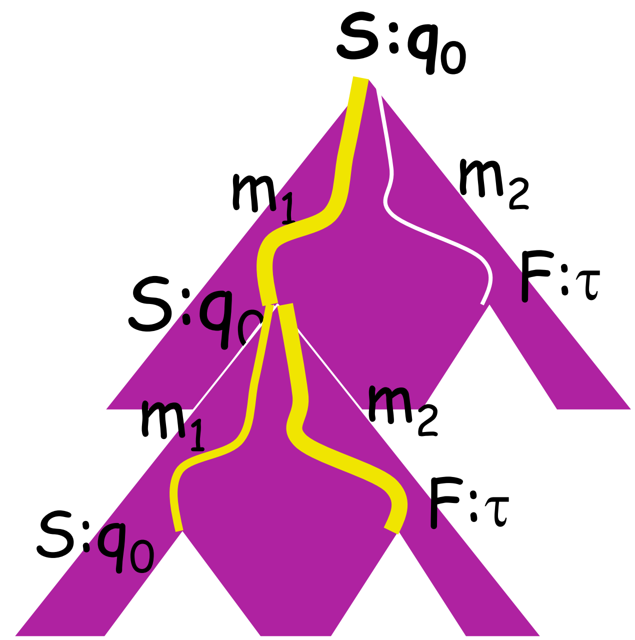
<u>Definition</u>: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S:(q_0, 0)$

Player (P): Given $F:(\tau, m)$,
pick Γ such that $\Gamma \vdash t_F: \tau$

Opponent (O): Given Γ,
pick $F:(\tau, m) \in \Gamma$
(and ask P to show
why F has type τ)



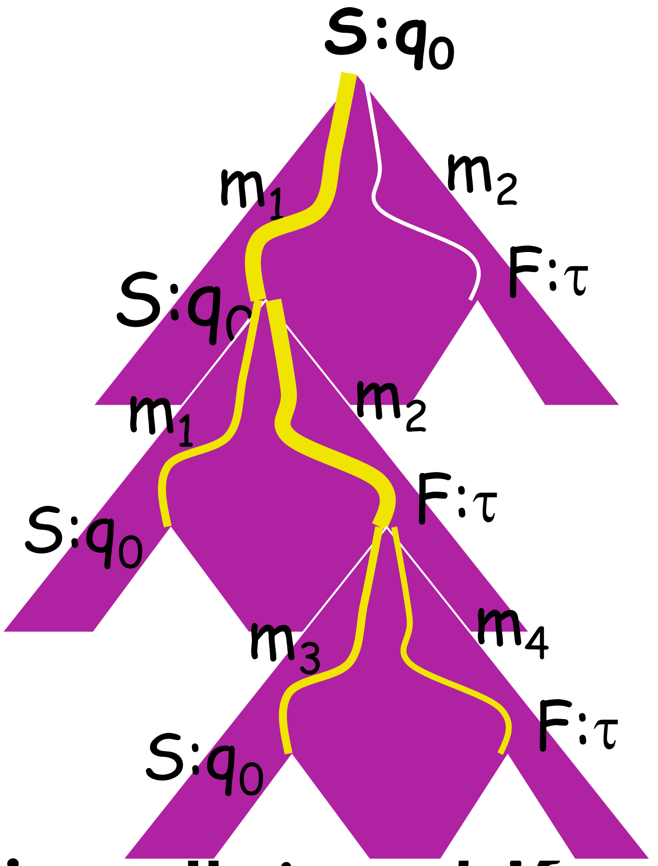
Definition: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S:(q_0, 0)$

Player (P): Given $F:(\tau, m)$,
pick Γ such that $\Gamma \vdash t_F: \tau$

Opponent (O): Given Γ,
pick $F:(\tau, m) \in \Gamma$
(and ask P to show why F has type τ)



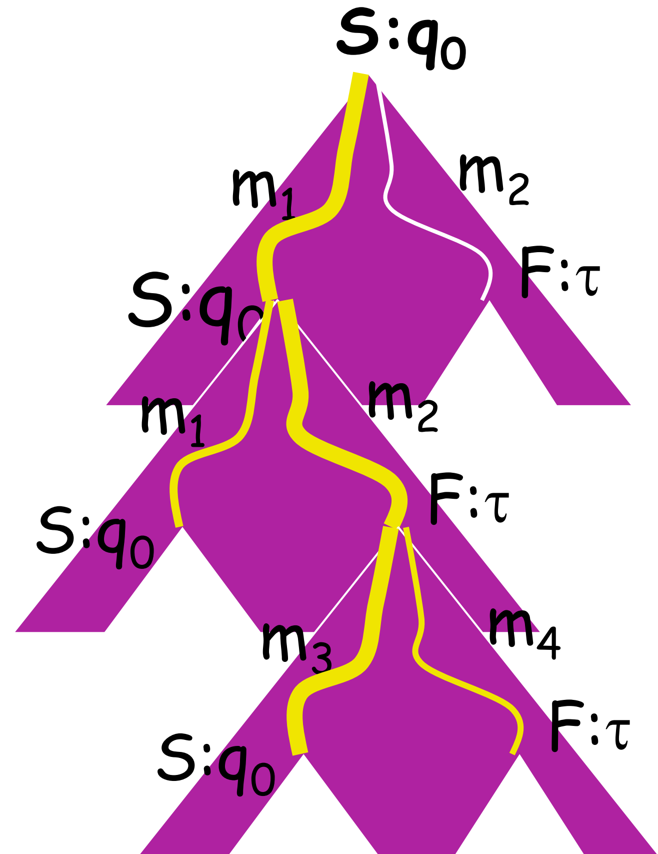
Definition: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S:(q_0, 0)$

Player (P): Given $F:(\tau, m)$,
pick Γ such that $\Gamma \vdash t_F : \tau$

Opponent (O): Given Γ,
pick $F:(\tau, m) \in \Gamma$
(and ask P to show
why F has type τ)



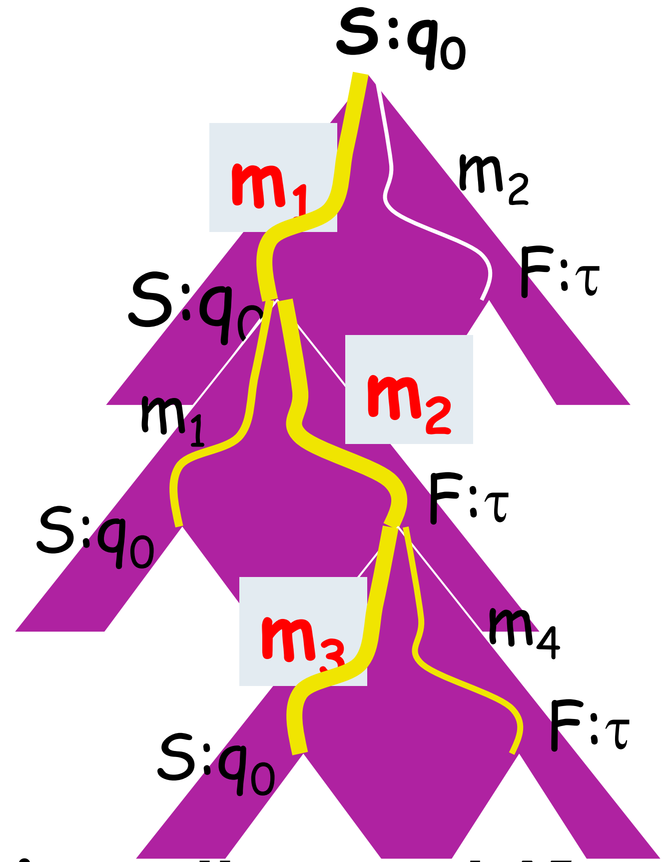
<u>Definition</u>: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Typability as Parity Game

Initial state: $S{:}(q_0, 0)$

Player (P): Given $F{:}(\tau, m)$,
pick Γ such that $\Gamma \vdash t_F : \tau$

Opponent (O): Given Γ,
pick $F{:}(\tau, m) \in \Gamma$
(and ask P to show why F has type τ)



<u>Definition</u>: Recursion scheme G is well-typed if P has a winning strategy for the parity game.

Example

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

Automaton:

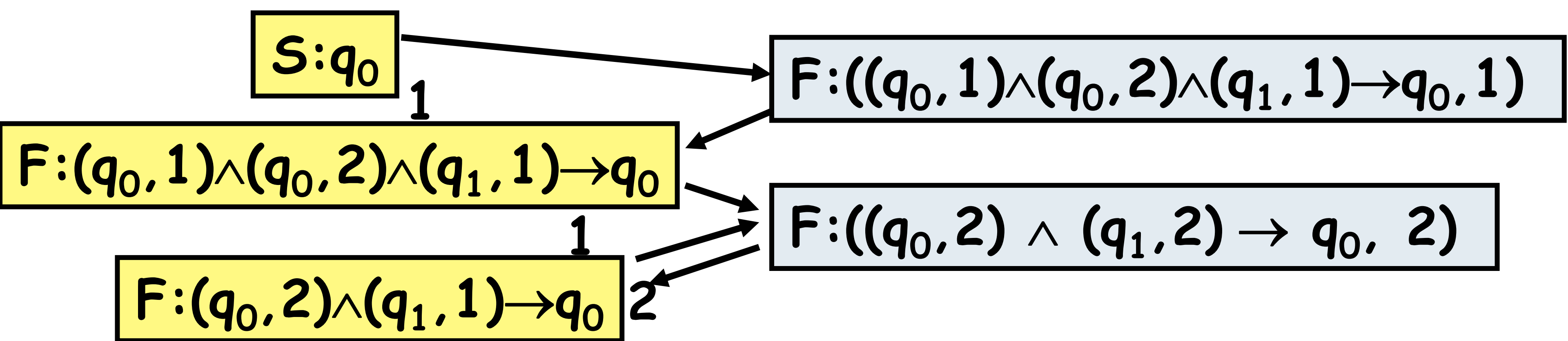
$\delta(q_0,a)=\delta(q_1,a)=(1,q_0)\wedge(2,q_0) \quad \delta(q_0,b)=\delta(q_1,b)=(1,\ q_1)$

$\delta(q_0,\ c) = \delta(q_1,\ c) = \mathrm{true} \quad \Omega(q_0)=1,\ \Omega(q_1)=2$

$F:\ ((q_0,1)\wedge(q_0,2)\wedge(q_1,2) \rightarrow q_0,\ 1)\ |\!-\ F\ c:\ q_0$

$F:\ ((q_0,2)\wedge(q_1,2) \rightarrow q_1,\ 2)$
$\quad |\!-\ \lambda x.a\ x\ (F\ (b\ x)) : (q_0,1)\wedge(q_0,2)\wedge(q_1,2) \rightarrow q_0$

$F:\ ((q_0,2)\wedge(q_1,2) \rightarrow q_1,\ 2)$
$\quad |\!-\ \lambda x.a\ x\ (F\ (b\ x)) : (q_0,2)\wedge(q_1,2) \rightarrow q_1$



Soundness and Completeness

Let

G: Recursion scheme

A: Alternating parity tree automaton

TS(A): Intersection type system (with priorities) derived from A

Then,

Tree(G) is accepted by A if and only if G is well-typed in TS(A)

(Naïve) Model Checking Algorithm (= Type Checking Algorithm)

- Construct an arena for the parity game

  For each $F \rightarrow t \in G$,
  enumerate all valid judgments $\Gamma \vdash t : \tau$

  # of edges and vertices: $O(|G| \exp_n (aQm)^{1+\varepsilon})$

  $|G|$: size of G, n: the largest order of types, a: the largest arity,
  Q: # of states, m: # of priorities

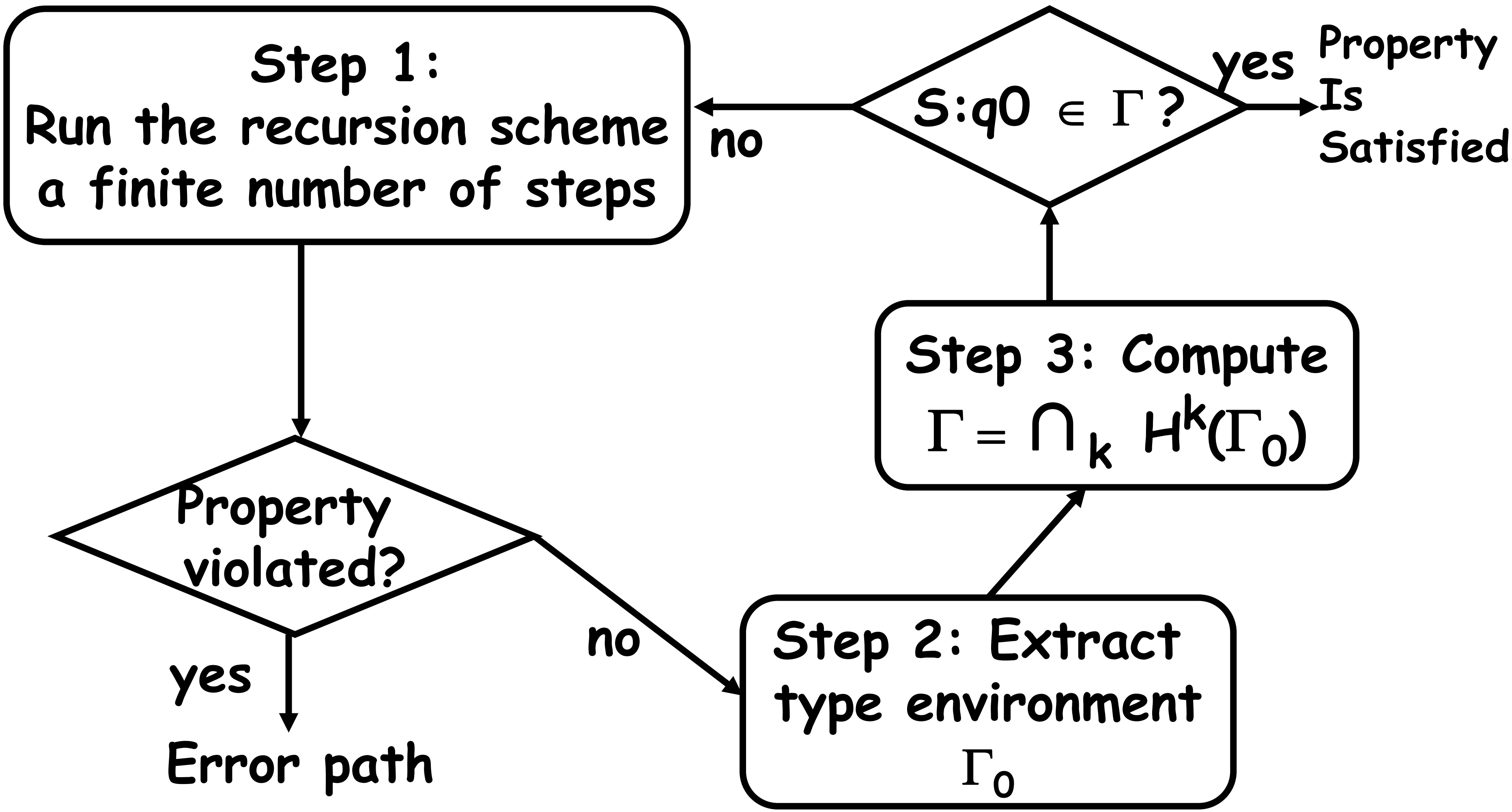
  # of order-n types: $\left.2^{2^{\cdot^{\cdot^{\cdot^{2^{(aQm)^{1+\varepsilon}}}}}}}\right\} n$

- Solve the parity game [Jurdziński 2000]

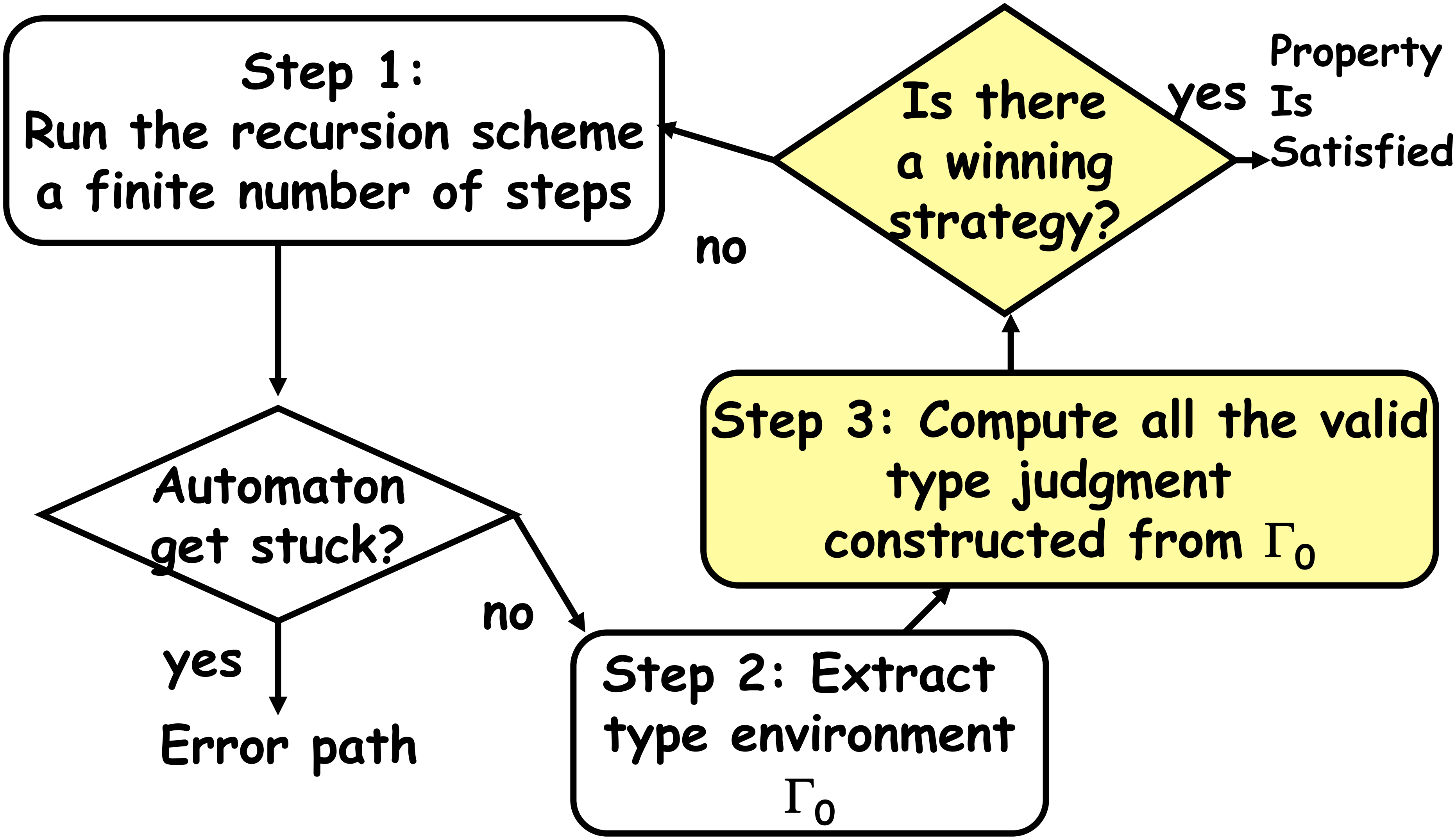
  $O(m\,E\,V^{m/2}) = O(|G|^{1+m/2} \exp_n (aQm)^{1+\varepsilon})$

  Polynomial in $|G|$, if other parameters are fixed

Hybrid Type Checking Algorithm



Hybrid Type Checking Algorithm



Note: One may have to prepare two automaton, one for the property and the other for its negation, and run the algorithm for both automata concurrently.

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
- 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

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 numeric data (c.f. BLAST, SLAM, …)
- From recursion schemes to transducers, to deal with algebraic data types (lists, trees, …)
- Infinite intersection types, to deal with non-simply-typed 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
- 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 (from a program verification point of view)

Challenges (1)

- **More efficient model checker**
  - Limitations of the current implementation
    - Worst-case complexity is not optimal
    - Too heuristic on the choice of expanded nodes
    - Not scalable on the size of tree automata
  - Possible approaches:
    - More language-theoretic properties of recursion schemes (e.g. pumping lemmas), to avoid redundant computation
    - BDD-like representation of intersection types
    - Other approaches to model checking? (e.g. model-theoretic approach?)

Challenges (2)

- Full modal μ-calculus model checker
  - The hybrid algorithm [K. PPDP09] can be extended easily.
  - Getting an efficient implementation remains a challenge.

Challenges (3)

- **Extension of the decidability result**
  - *A larger class of MSO-decidable trees than recursion schemes?*
  - *A larger class of properties that are decidable for the trees generated by recursion schemes?*

Conclusion

- Recursion schemes have important applications in program verification.
- Type-theoretic approach yields a practical model checking algorithm,
  (despite the extremely high worst-case complexity)
- More (both theoretical and practical) studies on recursion schemes are required to get practical software model checkers

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