Automata over infinite alphabets: automata with registers and histories

Nikos Tzevelekos, QMUL

jointly with:

Andrzej Murawski, University of Warwick

Steven Ramsay, University of Oxford

Radu Grigore, University of Kent

Oxford Verification Seminar, Nov 2016

infinite alphabets & program behaviour

```
public void foo() {
  // Create new list
  List x = new ArrayList();

  x.add(1); x.add(2);
  Iterator i = x.iterator();
  Iterator j = x.iterator();
  i.next(); i.remove(); j.next();
}
```

infinite alphabets & program behaviour

```
public void foo() {
  // Create new list
  List x = new ArrayList();

  x.add(1); x.add(2);
  Iterator i = x.iterator();
  Iterator j = x.iterator();
  i.next(); i.remove(); j.next();
}
```

Programs with usage of resources/names can go beyond finite alphabets (cf. modelling/analysis of programs)

– but in a parametric way

What this talk is about

Automata models over *infinite alphabets* akin to finite-state automata:

finite-state + registers + history access

We give an overview of their expressiveness & talk about

- emptiness, closures
- bisimilarity
- applications in program verification

Automata for infinite alphabets

Let $\Sigma = \{a_1, a_2, \ldots, a_n, \ldots\}$ be an infinite alphabet of names

→ examine languages over Σ^*

- or languages over $(F \cup \Sigma)^*$
- or languages over $(F \times \Sigma)^*$
  - usually called *data words* (XML)

can only be compared for equality

a finite set of constants

many (finitely many) automata models

History-Dependent Automata

- π-calculus models, "named sets", symmetries, bisimulation

[Montanari & Pistore '98, Pistore '99; Montanari & Pistore '00, Ferrari, Montanari & Pistore '02]

Register Automata (aka FMA)

- FSAs with registers, regularity, data words & XML, extensions

[Kaminski & Francez '94, Neven, Schwentick & Vianu '04]
[Sakamoto & Ikeda '00, Demri & Lazić '09; Libkin, Tan & Vrgoc '15; Jurdzinski & Lazić '11, Figueira '12]
[Cheng & Kaminski '98, Segoufin '06]
[Bojańczyk, Muscholl, Schwentick, Segoufin & David '06 , Bjorklund & Schwentick '10]

Nominal Automata

- Finite → finite orbit, used on nominal sets & other group actions

[Bojańczyk, Klin & Lasota '11, '14]

[Kaminski & Francez '94]

Register Automata (RA)

Let $\Sigma = \{a_1, a_2, \ldots, a_n, \ldots\}$ be an infinite alphabet of names

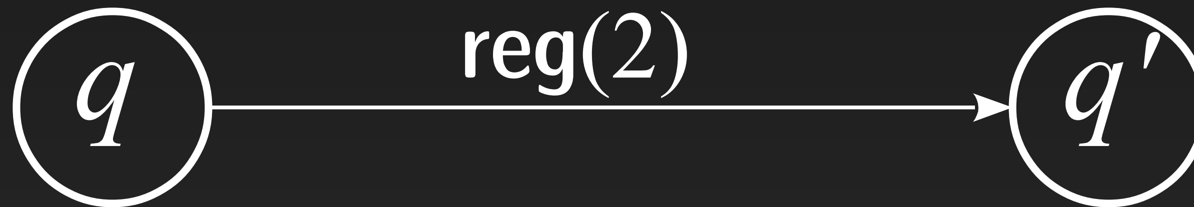
finitely many (say R) registers

registers store names

Label λ of the form:

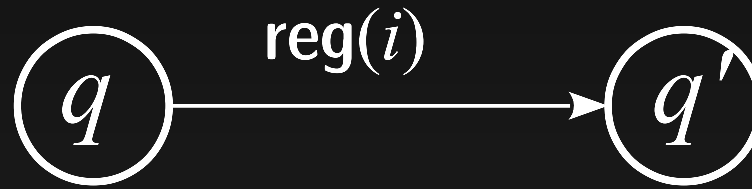
- **reg**(i), $i \in \{1,\ldots,R\}$
- **diff**(i), $i \in \{1,\ldots,R\}$

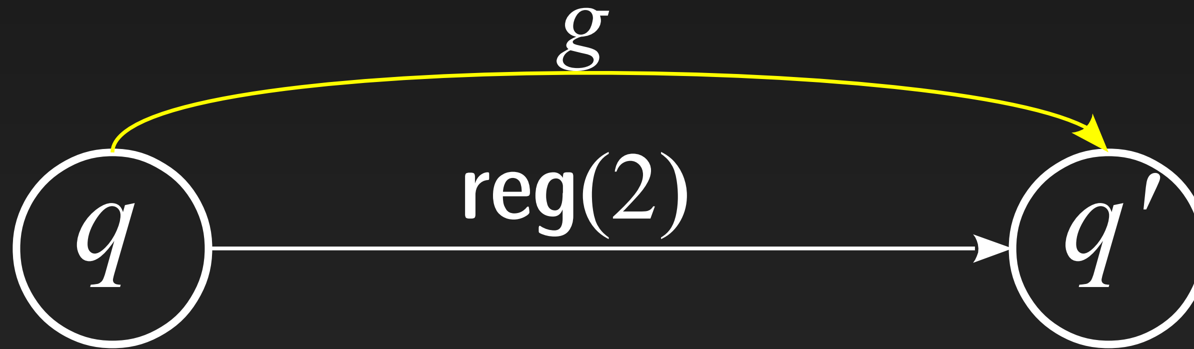
Transitions:



a	g	b

Transitions:





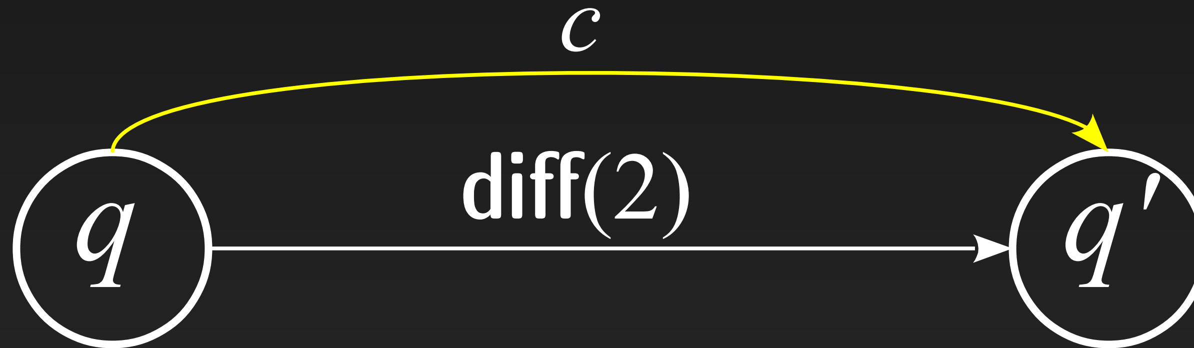
a	g	b

a	g	b

Transitions:

a	g	b

Transitions:

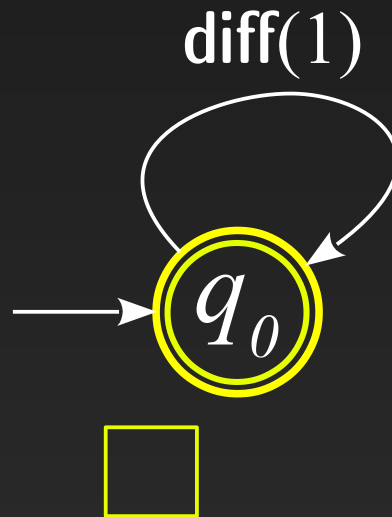


different from
current registers

Example

$$L_1 = \{\ a_1a_2\ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)



Example

$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

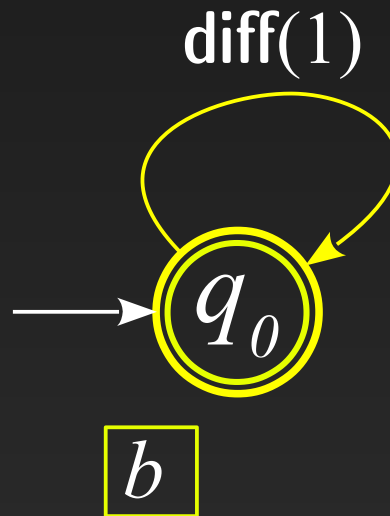
(all strings where each name is distinct from its predecessor)

a

Example

$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

(all strings where each name is distinct from its predecessor)

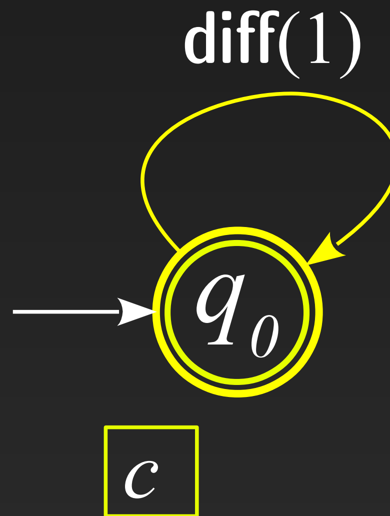


$a\,b$

Example

$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

(all strings where each name is distinct from its predecessor)

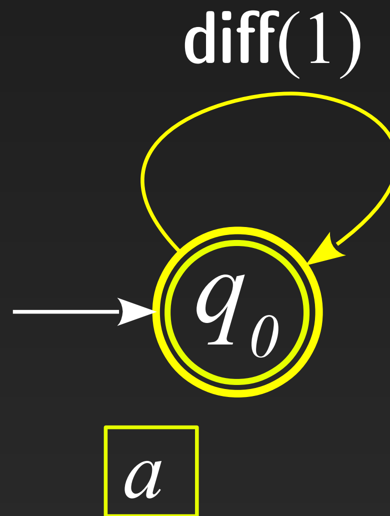


$a\,b\,c$

Example

$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

(all strings where each name is distinct from its predecessor)

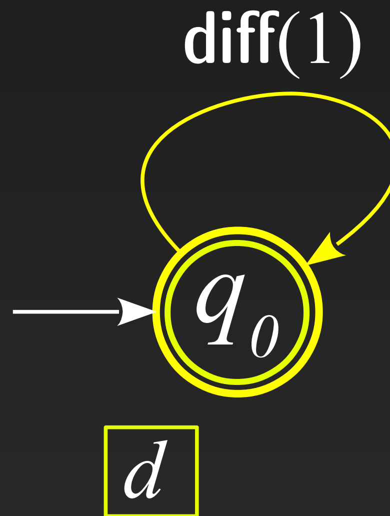


$a\,b\,c\,a$

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)

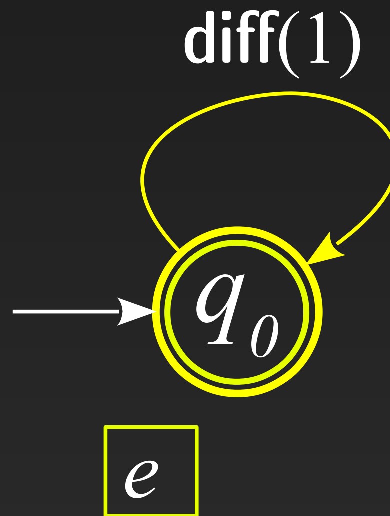


$a\ b\ c\ a\ d$

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)



$a\ b\ c\ a\ d\ e$

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)

$a\ b\ c\ a\ d\ e\ b$

Example

$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

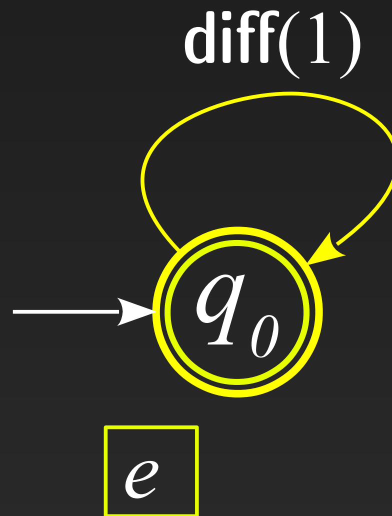
(all strings where each name is distinct from its predecessor)

$a\ b\ c\ a\ d\ e\ b\ a\ g\ c\ a\ b$

Example

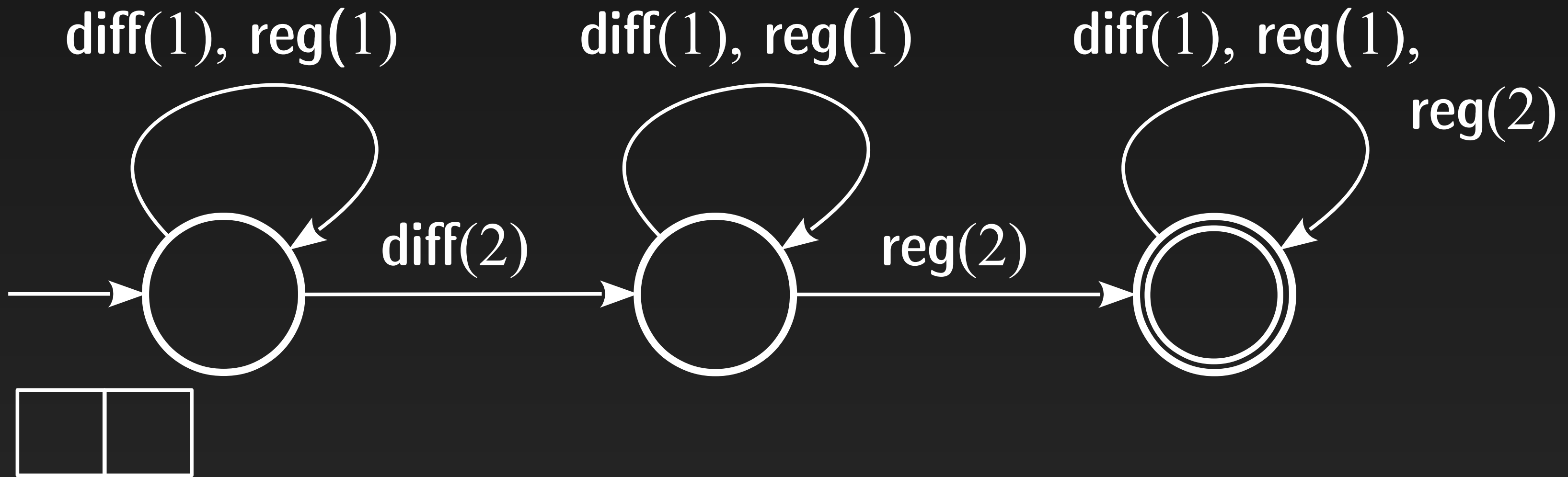
$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

(all strings where each name is distinct from its predecessor)

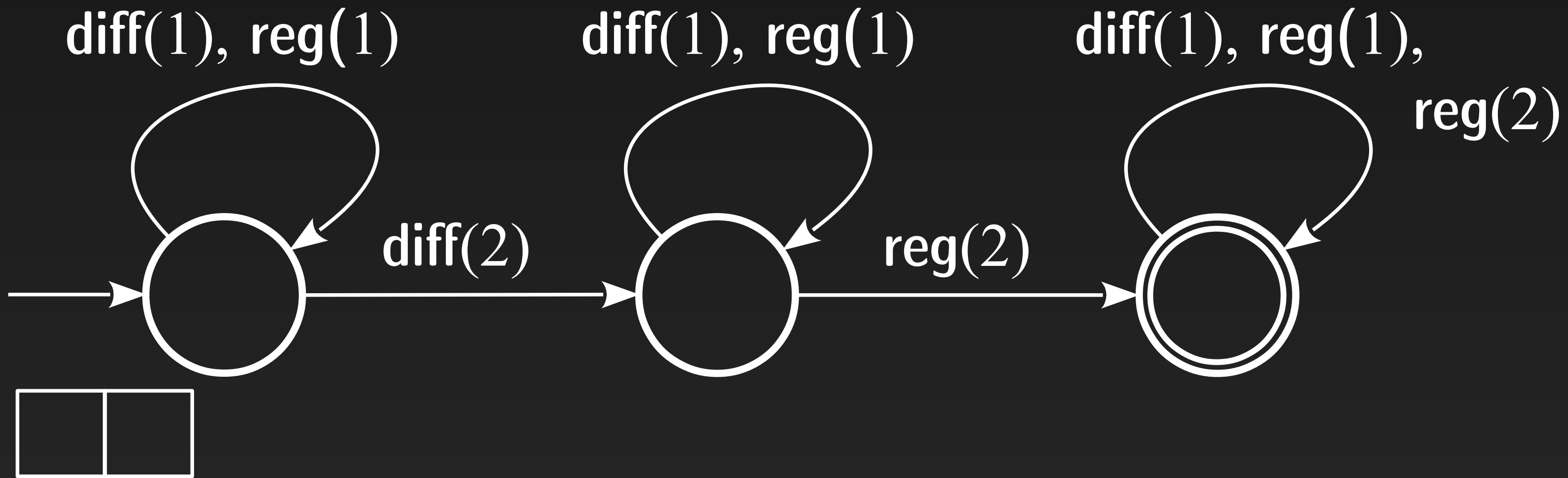


$a\,b\,c\,a\,d\,e\,b\,a\,g\,c\,a\,b\,a\,n\,d\,w\,e\,l\,o\,v\,e\,c\,a\,k\,e$

Quiz



Quiz



$$L_2 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \exists i \neq j.\ a_i = a_j \}$$

(all strings where some name appears twice)

$$L_{\mathrm{fr}} = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i \neq j.\ a_i \neq a_j \}$$

(all strings of pairwise distinct names)

– what about the complement of L_{fr} *? And that of* $L_{\mathrm{fr}} \bullet L_{\mathrm{fr}}$ *?*

Example revisited

```
public void foo() {
  // Create new list
  List x = new ArrayList();

  x.add(1); x.add(2);
  Iterator i = x.iterator();
  Iterator j = x.iterator();
  i.next(); i.remove(); j.next();
}
```

here is a safety property φ:

if an iterator modifies its collection x then other iterators of x become invalid

e.g. the code on the left is bad.

We can express such “chaining” properties using RAs

- and dynamically verify them

[Grigore, Distefano, Petersen & T. '13]

Example revisited

```
public void foo() {
  // Create new list
  List x = new ArrayList();

  x.add(1); x.add(2);
  Iterator i = x.iterator();
  Iterator j = x.iterator();
  i.next(); i.remove(); j.next();
}
```

but we cannot capture new!

here is a safety property φ:

if an iterator modifies its collection x then other iterators of x become invalid

e.g. the code on the left is bad.

We can express such "chaining" properties using RAs

- and dynamically verify them

[Grigore, Distefano, Petersen & T. '13]

Fresh-Register Automata (FRA)

Let $\Sigma = \{a_1, a_2, \ldots, a_n, \ldots\}$ be an infinite alphabet of names

finitely many (say R) registers

registers store names

Label λ of the form:

- **reg**(i), $i \in \{1,\ldots,R\}$
- **diff**(i), $i \in \{1,\ldots,R\}$
- **fresh**(i), $i \in \{1,\ldots,R\}$

global freshness oracle

Transitions:

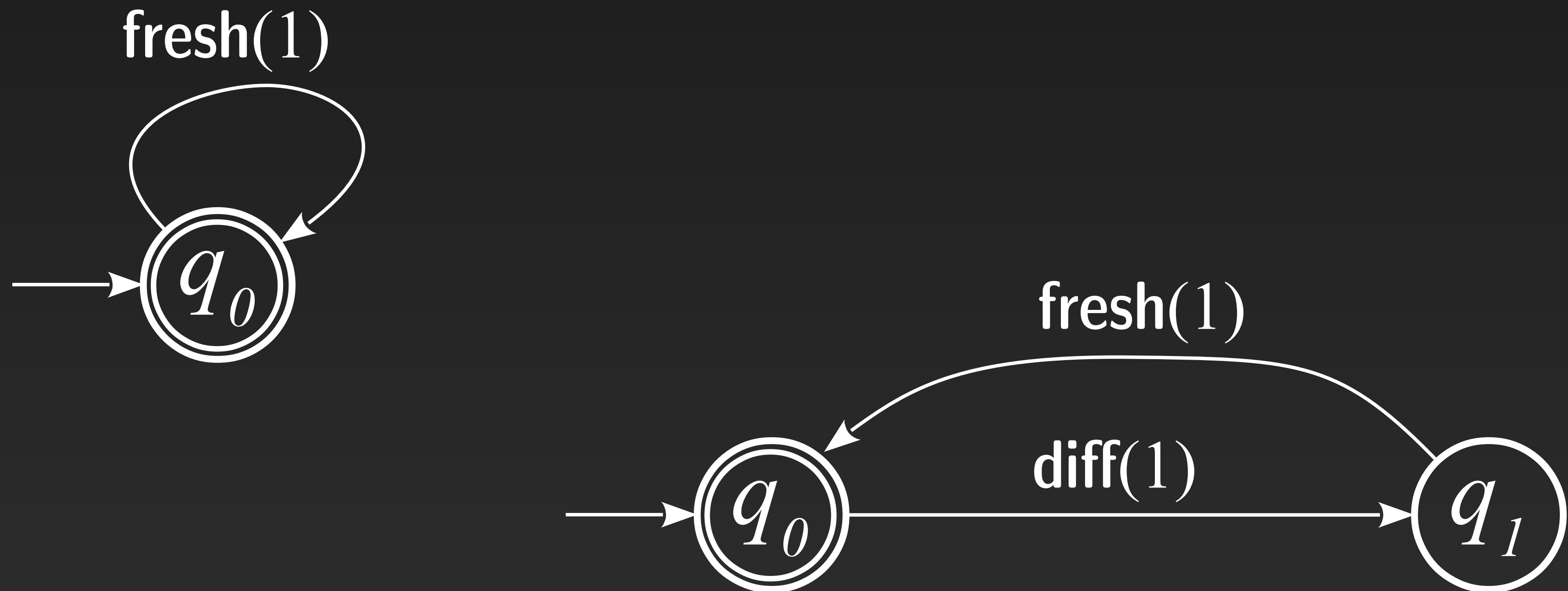
Transitions:

globally fresh

Examples

$$L_{\mathrm{fr}} = \{\, a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i \neq j.\ a_i \neq a_j \,\}$$

(all strings of pairwise distinct names)



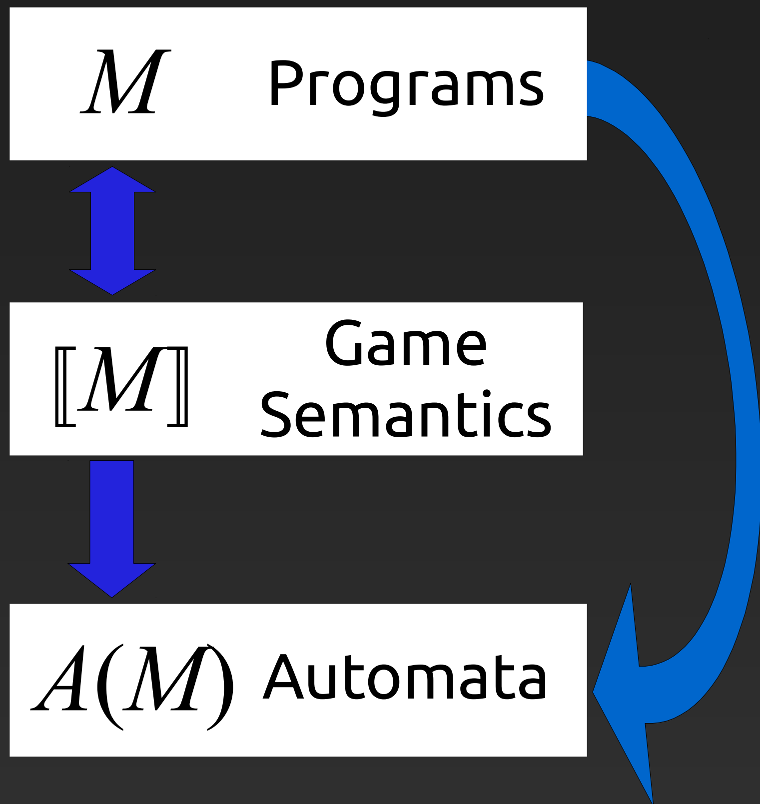
$$L_3 = \{\, a_1 a_2 \ldots a_{2n} \in \Sigma^* \mid n \geq 0, \forall i < 2n.\ a_i \neq a_{i+1}$$
$$\forall i \leq n, j < 2i.\ a_j \neq a_{2i} \,\}$$

FRA properties

- Closed under $\cup$, $\cap$, but not under $\bullet$, $*$
- Not closed under complement & not determinisable
- Decidable emptiness (same as RAs):
  - complexity depends on register "mode" (NL → NP → PSPACE)
- Universality / equivalence undecidable (from RAs)
- Bisimilarity: decidable [T.11]

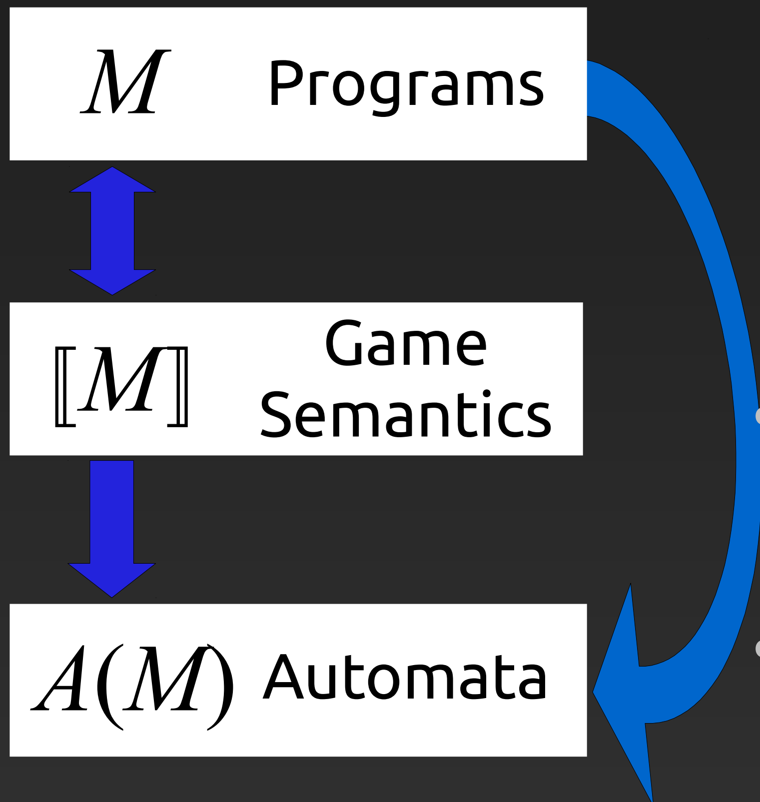
FRAs for program equivalence

The modelling power of FRAs can be used to model name-using programs e.g. via game semantics



FRAs for program equivalence

The modelling power of FRAs can be used to model name-using programs e.g. via game semantics



- Programs modelled fully abstractly:

$$M \cong M' \iff [\![M]\!] = [\![M']\!]$$

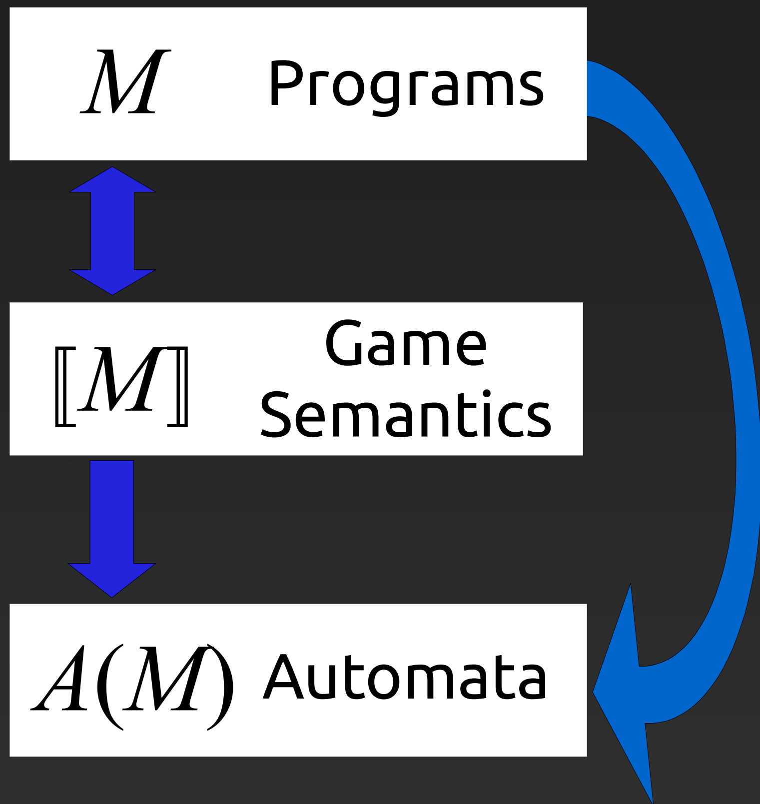
[$\cong$: same behaviour in every context i.e. contextual equivalence]

- Each program M translated into a set $[\![M]\!]$ of traces of specific kind
- These traces can be captured by automata (FRAs + …)

$$[\![M]\!] = [\![M']\!] \iff A \sim A' \iff A \otimes A' = 0$$

FRAs for program equivalence

The modelling power of FRAs can be used to model name-using programs e.g. via game semantics



effectively:

two programs are equivalent $\iff$ their FRAs are language equivalent

what we get:

- decision procedures for ML fragments
  [Murawski & T. '11, '12]
- same for Interface Middleweight Java
  http://bitbucket.org/sjr/coneqct/wiki/Home
  [Murawski, Ramsay & T. '15]

Coneqct

Atlassian, Inc. (US) https://bitbucket.org/sjr/coneqct/wiki/Home

Atlassian Bitbucket Features Pricing Find a repository... English Sign up Log in

sjr
coneqct

ACTIONS
- Clone
- Compare
- Fork

NAVIGATION
- Overview
- Source
- Commits
- Branches
- Pull requests
- Wiki
- Downloads 1

Wiki

Clone wiki

coneqct / Home

View History

Coneqct: a contextual equivalence checking tool for Interface Middleweight Java

Home | **Downloads** | **Syntax** | **Examples**

Requirements

The checker runs on the .NET platform (>=4.5), and hence requires a recent implementation of the .NET Common Language Infrastructure (CLI) to be installed on your system.

- On Windows we recommend Microsoft's ".NET Framework", the latest stable version is 4.5.2: https://www.microsoft.com/en-us/download/details.aspx?id=42643.
- On Linux or Mac we recommend Xamarin's "Mono": http://www.mono-project.com/download/

Installation

1. Download the latest assemblies from downloads. All the required assemblies are packaged together in a zip file named "coneqct-XXX.zip" where "XXX" denotes the revision number.
2. Unzip to any convenient location, this creates a new directory "coneqct-XXX" in which resides the executable "coneqct.exe".
3. To verify that all is well, on the command line navigate to the directory "coneqct-XXX" and run the command:
   - ".\coneqct.exe" on Windows or,
   - "mono ./coneqct.exe" on Linux or Mac. If the installation is working correctly, the usage message will be printed out to the terminal.

Usage

On Windows, to check the equivalence of two IMJ terms defined in the file "terms.inp", run:

```
> .\coneqct.exe \path\to\terms.inp
```

On Linux or Mac you should prefix this command by "mono" (and use appropriate slashes):

```
> mono ./coneqct.exe /path/to/terms.inp
```

A number of example inputs are bundled with the installation. For example, after navigating to the root of the directory "coneqct-XXX", to verify the "extended types" equivalence adapted from Benton and Leperchey's "Relational Reasoning in a Nominal Semantics for Storage" on Mac, run:

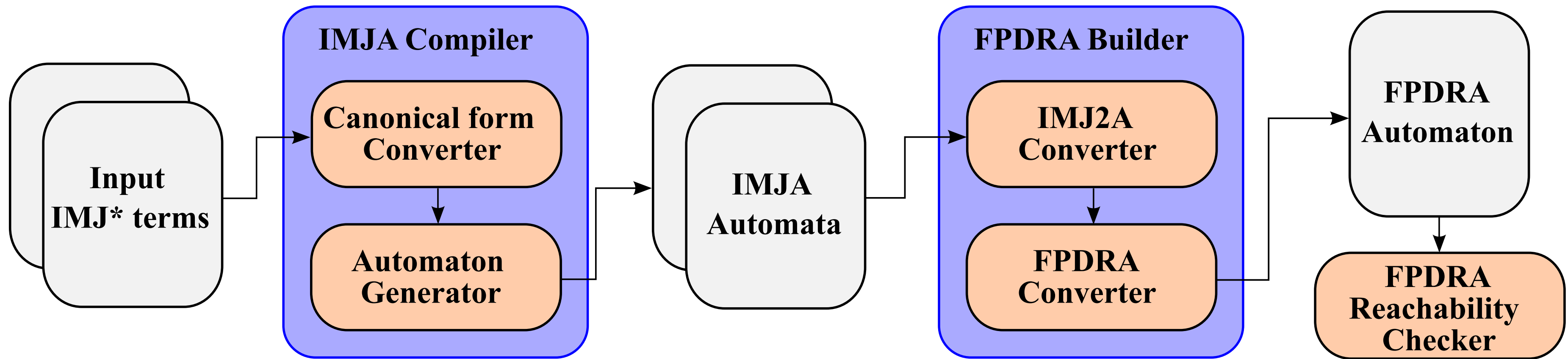
```
> mono ./coneqct.exe inputs/inp2.imj
```

See Syntax for a detailed description of the syntax of the input file format.

The tool can be configured using some command-line options:

- -maxint <int>: set the value of maxint to <int>.

Coneqct



Input: two IMJ* programs

Output: equivalent / inequivalent

[Murawski, Ramsay & T. '15]

More applications and variants

History-Dependent Automata

- freshness via "black holes" (histories)
- verification of LTL + allocation

[Pistore '99, Distefano, Rensink & Katoen '02, '04]

Session automata and learning

- freshness, but no **diff**
- canonical forms, decide equivalence

[Bollig, Habermehl, Leucker & Monmege '14]

Kleene algebras for languages with binders

- NKA: KA with ν-binder → match with automata

[Gabbay & Ciancia '11; Kozen, Mamouras, Petrisan & Silva '15]

Investigations in FRAs

Bisimilarity for FRAs (complexity)

- Depends on register mode (NP → PSPACE → EXPTIME)
  - approach uses permutation group theory

[Murawski, Ramsay & T. '15]

Context-freeness: Pushdown FRA

[Cheng & Kaminski '98; Segoufin '06]
[Murawski & T. '12]

- Reachability EXPTIME-complete
- Global reachability via “saturation”

[Murawski, Ramsay & T. '14]

Freshness oracle: from one to many histories

- History Register Automata (cf. DA/CMA)

[Grigore & T. '16]

Semantics formally: configurations

Semantics of FRAs given by configuration graphs:

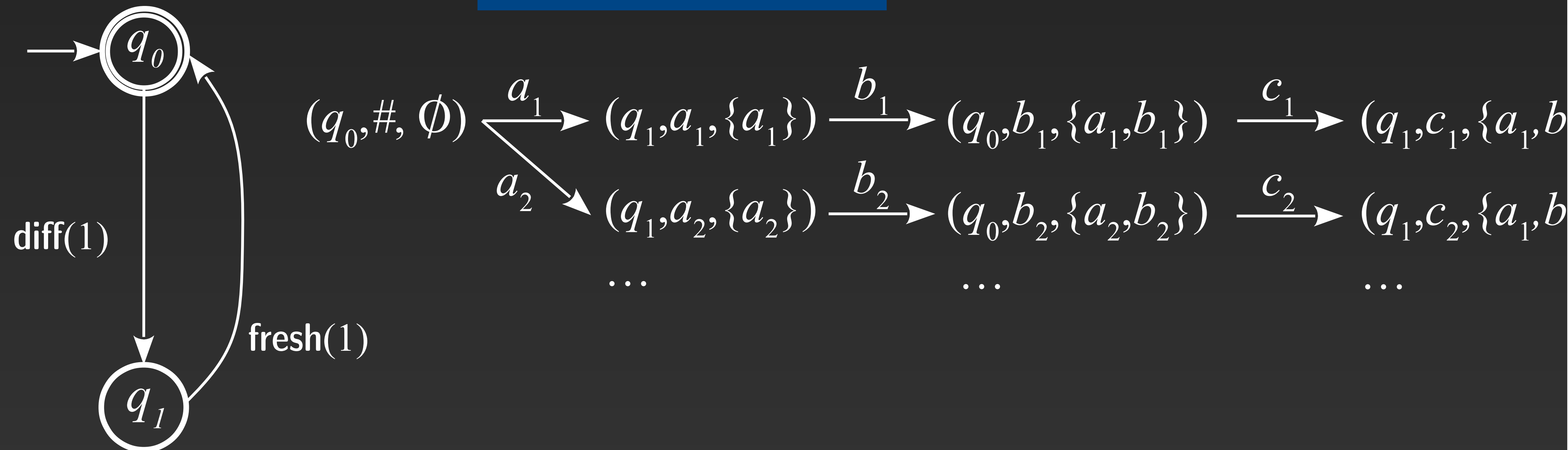
configuration

$$(q,\rho,H) \xrightarrow{a} (q',\rho',H')$$

state

register assignment: $\rho : \{1,\ldots,R\} \to \Sigma \cup \{\#\}$

history: $H \subseteq_{\text{fin}} \Sigma$



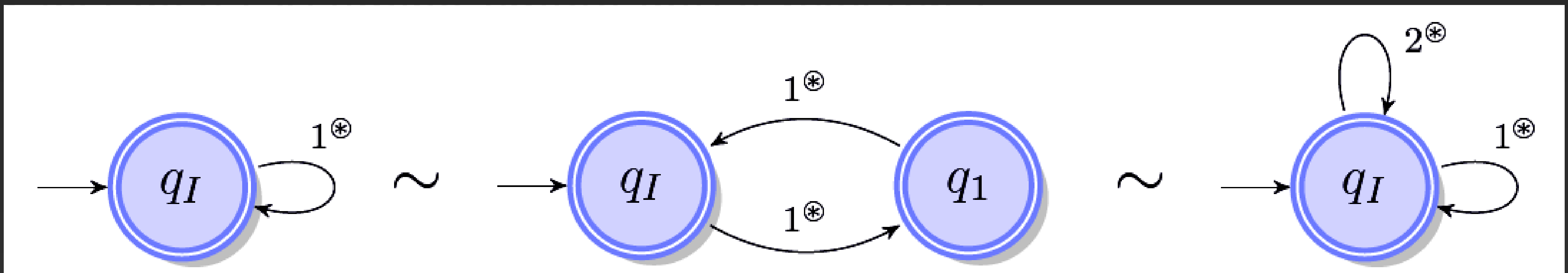
Bisimilarity

A behavioural notion of equivalence:

two configurations κ_1, κ_2 are bisimilar ($\kappa_1 \sim \kappa_2$) if they can simulate one another name-by-name

We say that two FRAs are bisimilar if their initial configurations are (in the combined conf. graph).

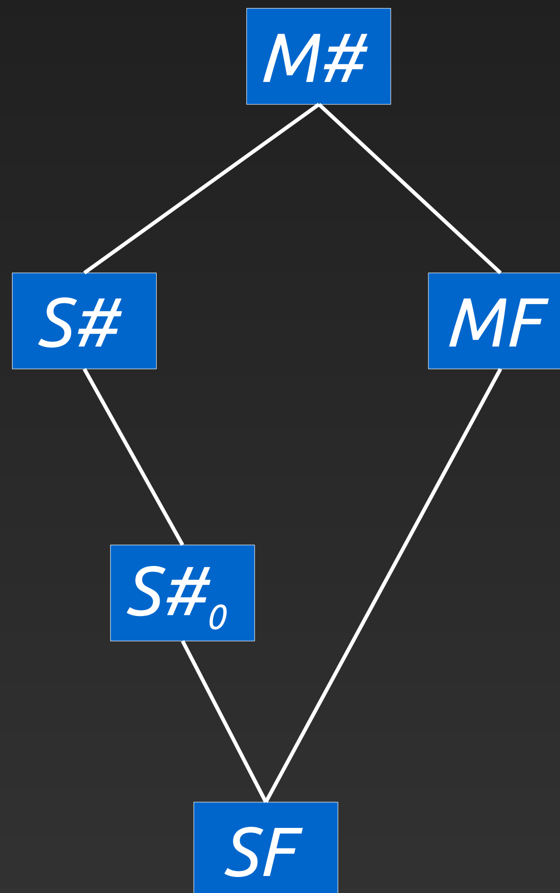
e.g. (writing $1^{\circledast}$ for fresh(1)):



A small detail: register modes

So far we assumed: registers initially empty, not possible to erase them or have name duplicates. We can generalise:

Name multiplicity
- *(S) single*
- *(M) multiple*



Register fullness
- *(F) full*
- *($\#_0$) initially empty*
- *(#) eraseable*

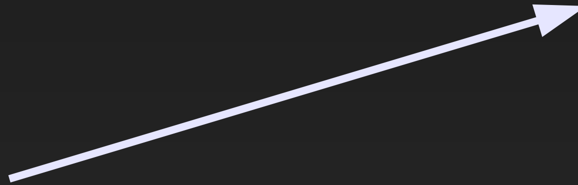
is for empty register content

$(S\#) \rightarrow (MF)$

no multiplicities,
but erasing allowed

multiplicities, but
no empty registers

z	a	g	z	b	z



a	g	$\#$	b	$\#$

neat, but erasing
gives exponentially
large labels

$(S\#) \rightarrow (MF)$

no multiplicities,
but erasing allowed

multiplicities, but
no empty registers

z	a	g	z	b	z

neat, but erasing
gives exponentially
large labels

a	g	$\#$	b	$\#$

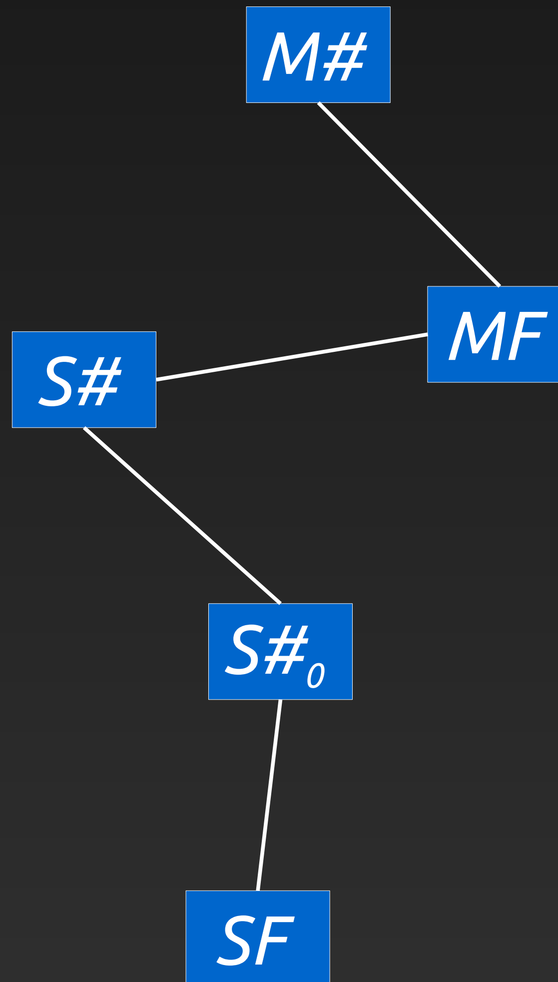
a'	g'	c	b'	d	a	g	c	b	d

concise, as each
name appears
at most twice

Complexity Picture (bisimilarity)

Multiplicity:
- (*S*) single
- (*M*) multiple

Fullness:
- (*F*) full
- ($\#_0$) initially empty
- (#) eraseable



Complexity Picture (bisimilarity)

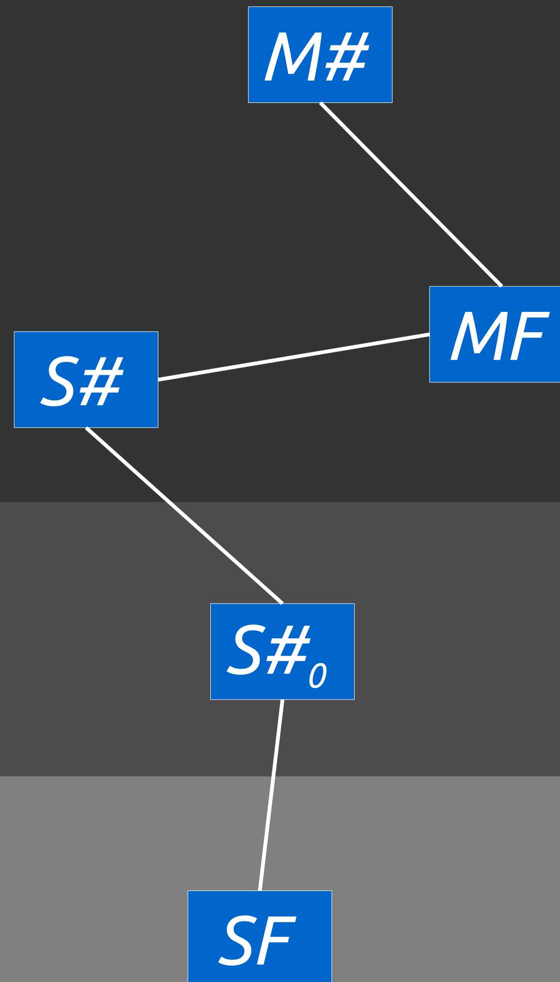
Multiplicity:
- (S) single
- (M) multiple

Fullness:
- (F) full
- ($\#_0$) initially empty
- (#) eraseable

EXPTIME-c

PSPACE-c

NP



EXPTIME solvability

To decide bisimilarity of two configurations of size R:

- we need $2R$ names to represent all possible name matchings between them
- plus one name that stands for "different"
- and another one for "fresh"

→ $2R+2$ names, that we can encode inside states:

$$Q \longrightarrow Q \times (2R+2)^R$$

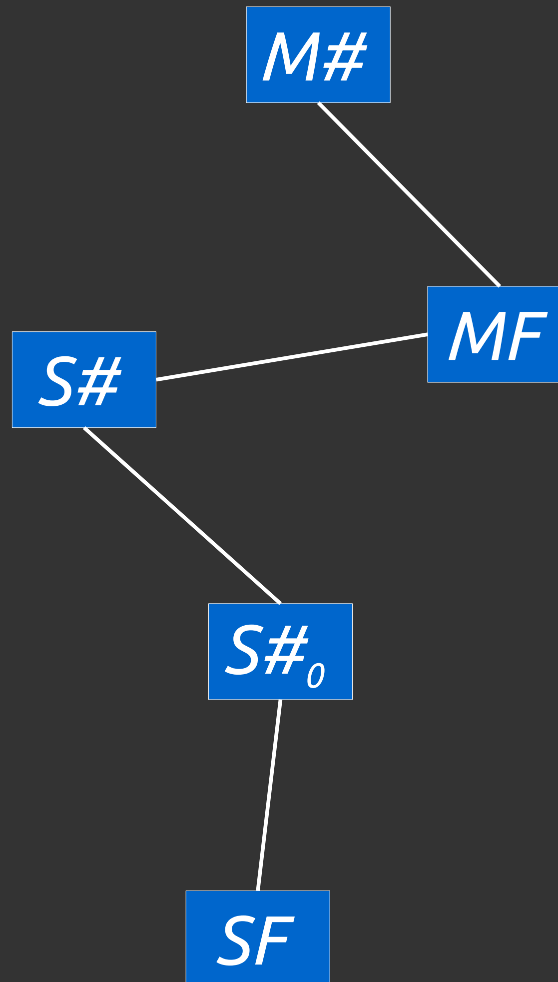
(bisimilarity for finite-state automata is in PTIME)

Complexity Picture

Multiplicity:
- (S) single
- (M) multiple

Fullness:
- (F) full
- ($\#_0$) initially empty
- (#) eraseable

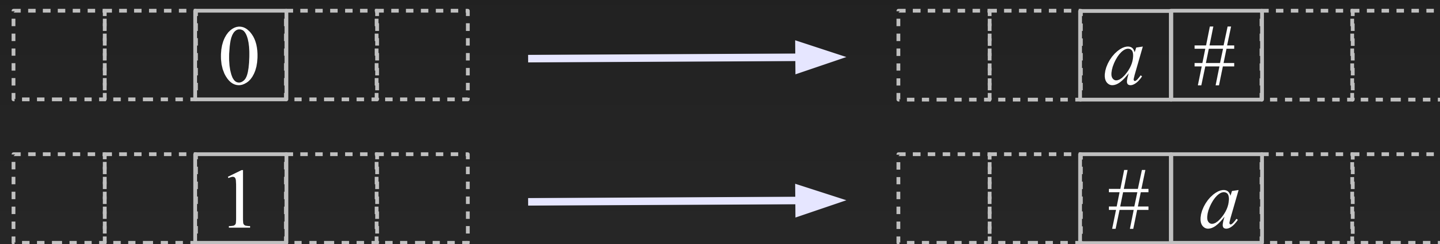
EXPTIME



EXPTIME hardness

The (*S*#) case is EXPTIME-hard:

- reduce from alternating TMs with linear-size tape (ALBA)
- represent each cell by two registers:



- Prepare two (RA) representations of the ALBA:
  - identical apart at rejecting final states (→ non-bisim)
- Bisimulation game (Attacker (A) vs Defender (D)):
  - A controls universal states, D controls existential ones
  - use *Defender forcing* [Jancar & Srba '08]

Bisimulation defined as a game

Two players, an Attacker (A) and a Defender (D), play a game on a configuration graph, starting from given configurations κ_1 and κ_2

- **A** wants to show them not equivalent, D the opposite
- **A** picks a configuration, say κ_1, and an edge out of it, say with label a
- **D** must then pick an a-edge out of κ_2 – or he loses!
- now we are at κ_1' **and** κ_2' – and the game continues

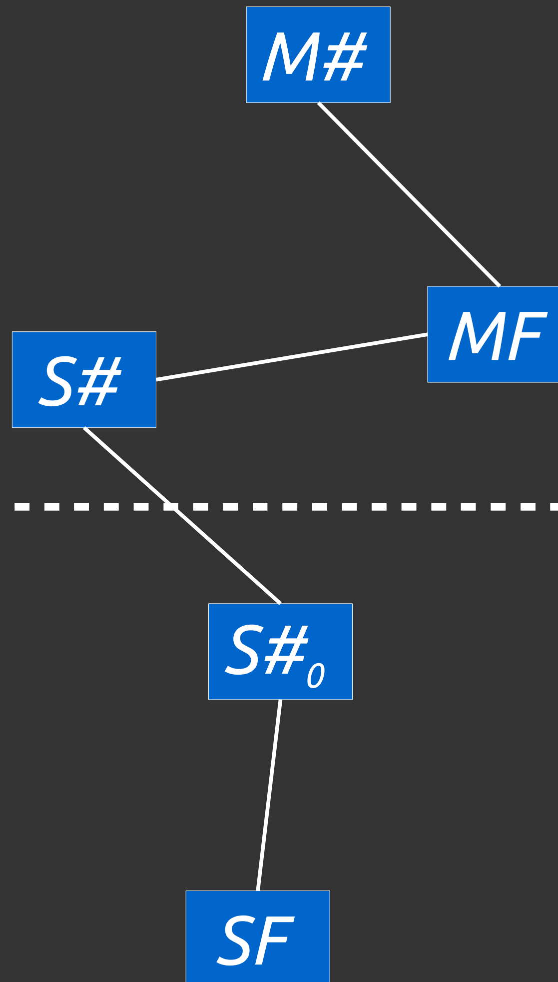
κ_1 and κ_2 are called *bisimilar*, write $\kappa_1 \sim \kappa_2$, if D has a strategy for not losing the game

Complexity Picture

Multiplicity:
- (S) single
- (M) multiple

Fullness:
- (F) full
- ($\#_0$) initially empty
- (#) eraseable

EXPTIME



EXPTIME

The original case ($S\#_0$)

Disallowing erasures makes impossible our modelling of a linear-size tape...

In fact, the problem is PSPACE complete

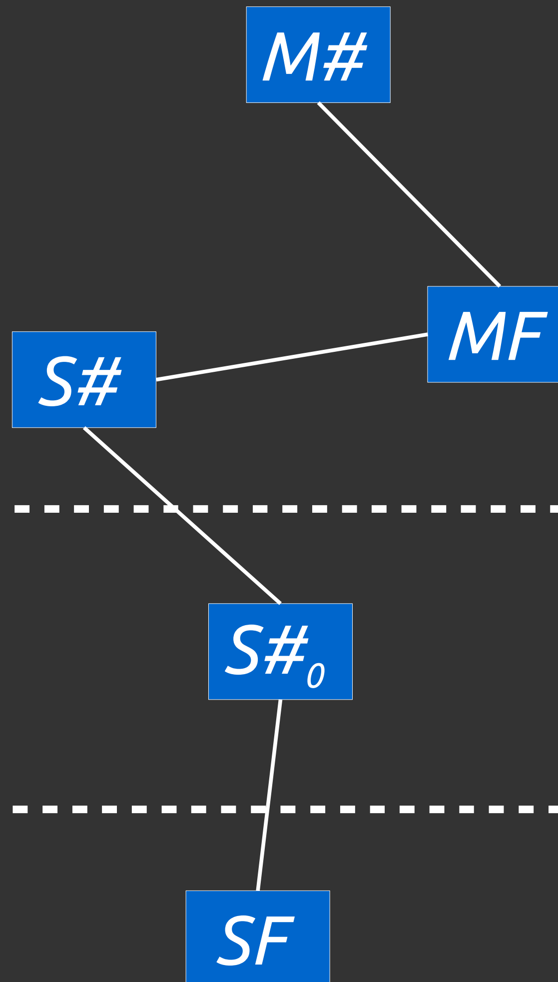
First, we can model boolean assignments (cf. write-once tape), which are enough for PSPACE-hardness:

- we reduce from QBF
- Attacker chooses universal variables
- Defender chooses existential ones (via forcing)

Complexity Picture

Multiplicity :
- (*S*) single
- (*M*) multiple

Fullness :
- (*F*) full
- ($\#_0$) initially empty
- (#) eraseable



EXPTIME

EXPTIME

PSPACE

PSPACE solvability: difficult

We use the fact: APTIME = PSPACE

- problem: while we cannot simulate a linear tape, we still have a lot of configurations!

We look into internal symmetries of FRAs:

- symbolic reasoning: we only look at how configurations are related, not their actual content
- group representations: we express these interrelations compactly via permutation groups
- bounded history: it suffices to consider histories of size up to $2R$

PSPACE solvability

$$\overset{0}{\sim} \supseteq \overset{1}{\sim} \supseteq \overset{2}{\sim} \supseteq \ldots \supseteq \overset{i}{\sim} \supseteq \ldots \quad \text{and} \quad \sim_{\mathrm{s}} = \bigcap_{i \in \omega} \overset{i}{\sim}$$

Reasoning symbolically:

- each decrease in the indexed chain can be traced back to one of polynomially many factors!

use fact that strict subgroup chains have bounded length

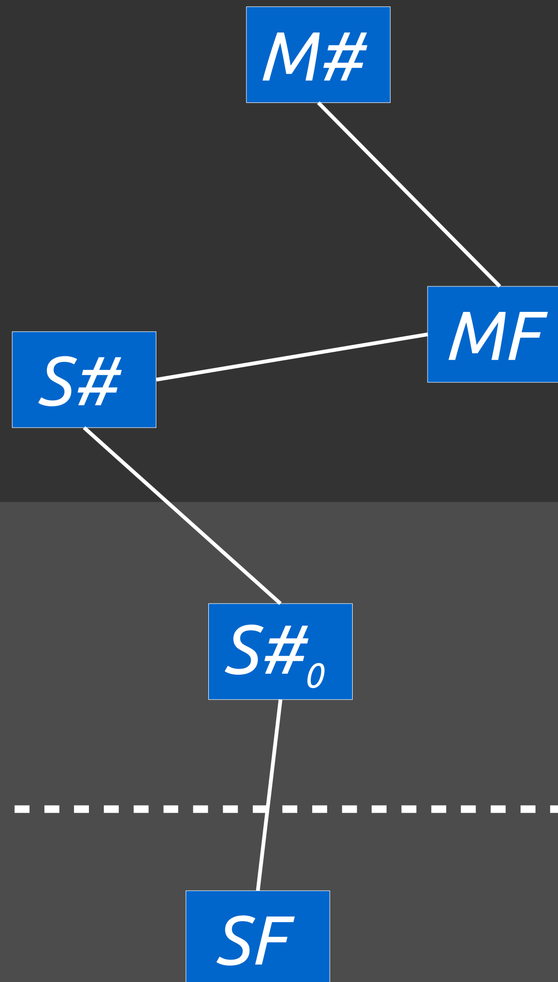
This means there is a final polynomial-size i

- polynomial bound for bisimulation game → APTIME

Complexity Picture

Multiplicity:
- (S) single
- (M) multiple

Fullness:
- (F) full
- ($\#_0$) initially empty
- (#) eraseable



EXPTIME-c

PSPACE

PSPACE

Bisimilarity for (F)RAs

Multiplicity:

- (S) single
- (M) multiple

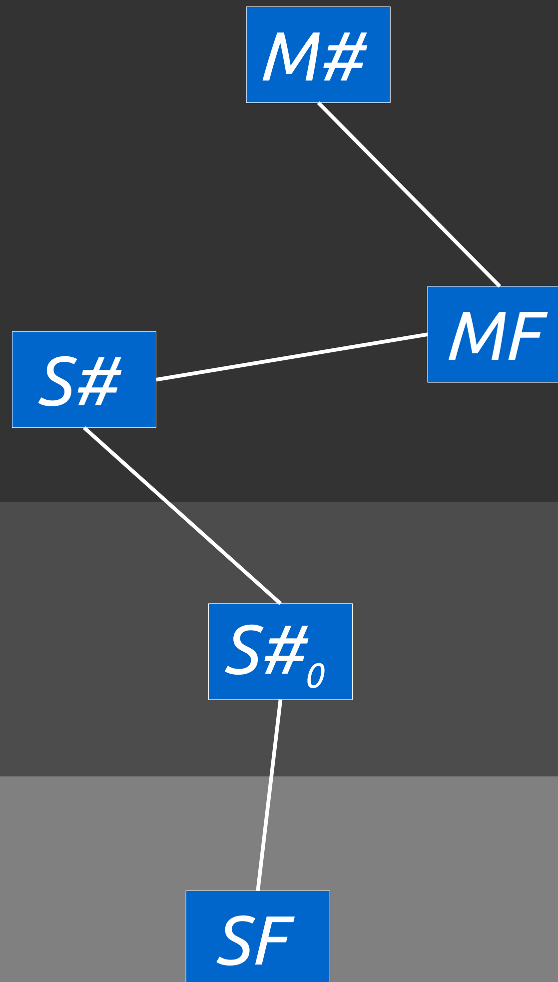
Fullness:

- (F) full
- ($\#_0$) initially empty
- ($\#$) eraseable

EXPTIME-c

PSPACE-c

NP



Investigations in FRAs

Bisimilarity for FRAs (complexity)

- Depends on register mode (NP → PSPACE → EXPTIME)
  - approach uses permutation group theory [Murawski, Ramsay & T. '15]

Context-freeness: Pushdown FRA

[Cheng & Kaminski '98; Segoufin '06]
[Murawski & T. '12]

- Reachability EXPTIME-complete
- Global reachability via "saturation" [Murawski, Ramsay & T. '14]

Freshness oracle: from one to many histories

- History Register Automata (cf. DA/CMA) [Grigore & T. '16]

Limitations of FRAs

- Not closed under concatenation, repetition, interleaving and complementation

  → *due to history being "monolithic"*

- Expressivity:
  - Cannot express non-freshness of names
  - What about consuming names
  - or classifying between fresh names

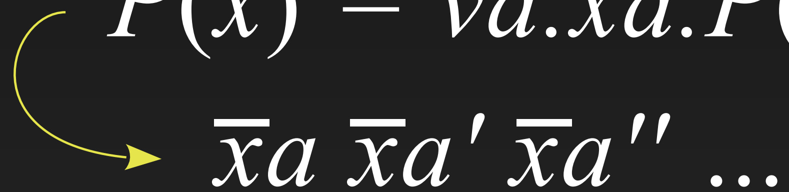
Interleaving

$$P(x) = \nu a.\overline{x}a.P(x)$$

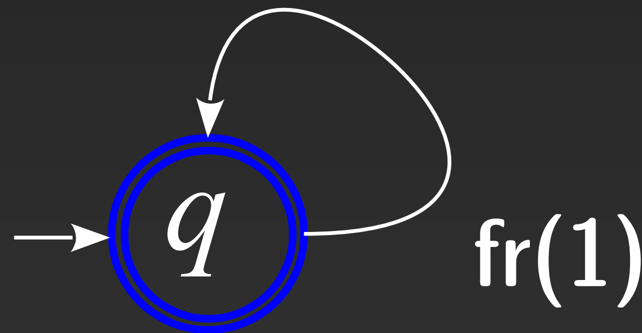
(name generator)

Interleaving

$$P(x) = \nu a.\overline{x}a.P(x)$$ (name generator)



$$L = \{ a\ a'\ a''\ \ldots \mid \text{all names distinct} \}$$



Interleaving

$$P(x) = \nu a.\overline{x}a.P(x)$$

(name generator)

$$P(x) \;|{*}|\; P(y)$$

Name producer
– produces and sends names on x

Name producer
– produces and sends (possibly the same) names on y

Consumption

Name producer
– produces and sends names on x

$$P(x) \parallel C(x,y)$$

Name consumer
– Receives names on x and consumes them on y

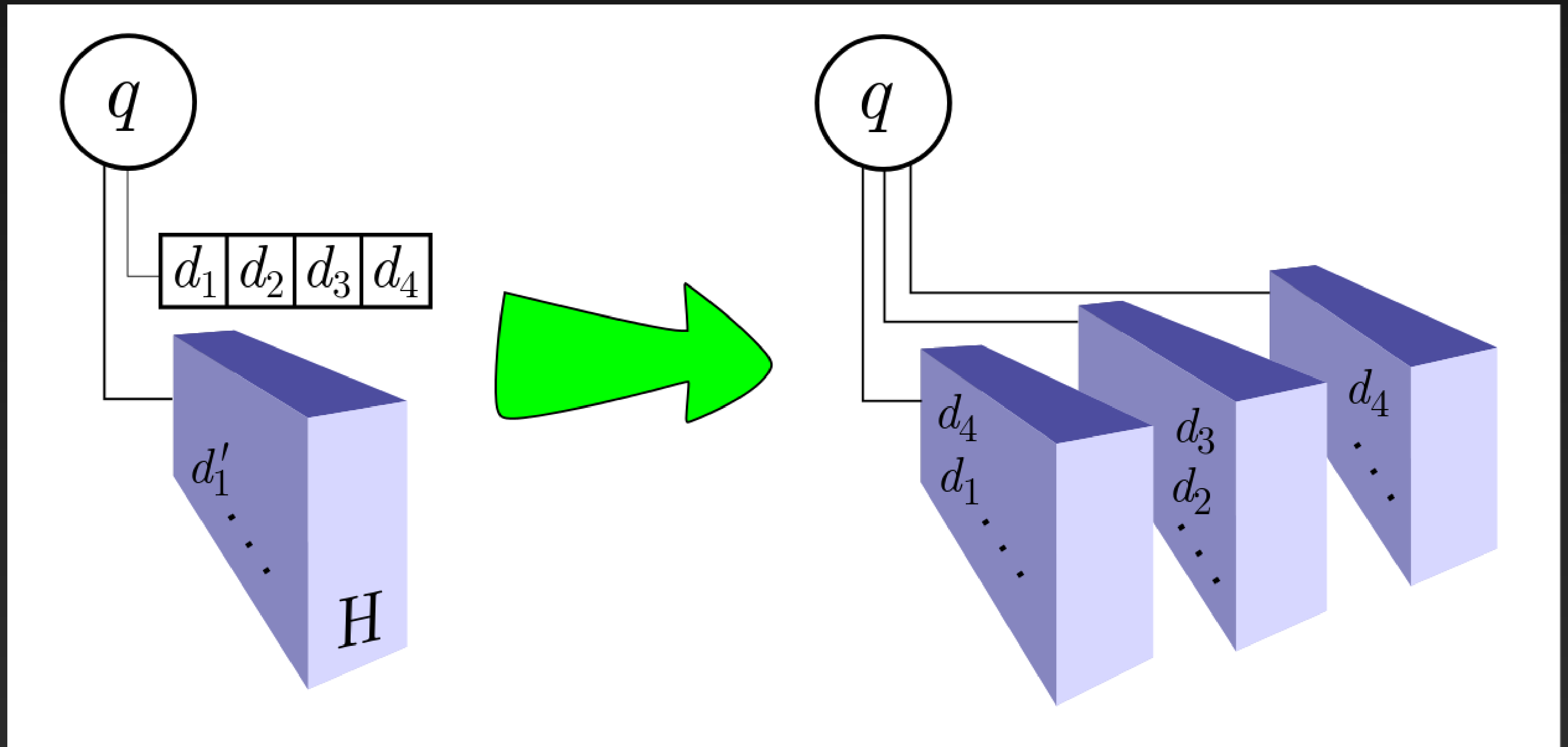
Classification

Name producers – produce and send names on x and y

$$P(x) \mid\mid P(y) \mid\mid Q(x,y)$$

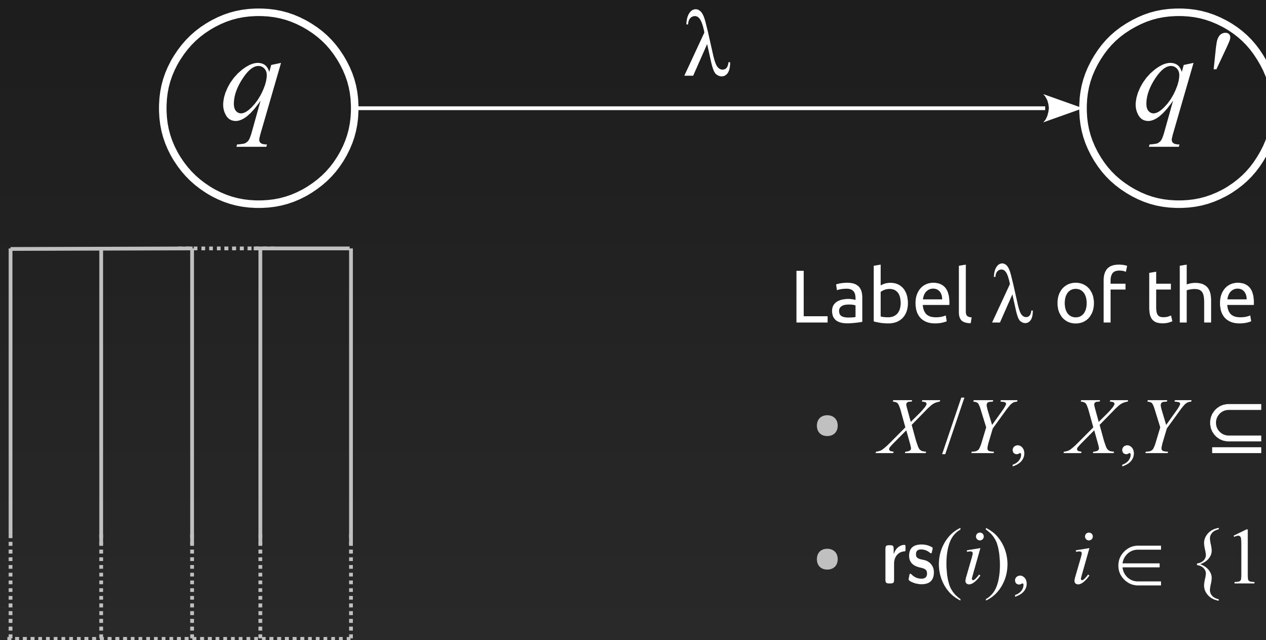
Receives names on x and y and uses them for different purposes

HRAs: from registers to histories



histories = registers with unboundedly many equivalent elements

History-Register Automata (HRAs)



Label λ of the form:

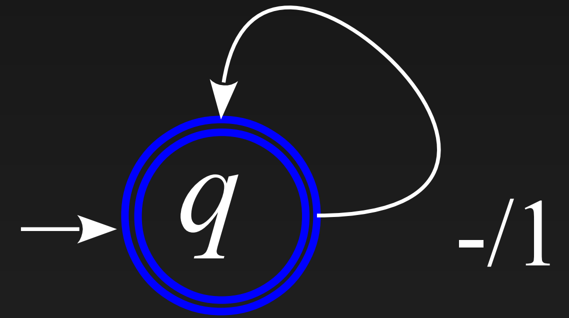
- $X/Y, \;\; X,Y \subseteq \{1,\ldots,R\}$
- $\mathbf{rs}(i), \;\; i \in \{1,\ldots,R\}$

finitely many (say R) histories

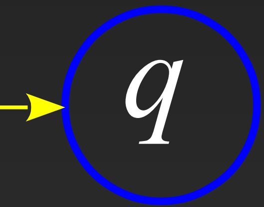
histories store names

Name generator

$$P(x) = \nu a.\overline{x}a.P(x)$$

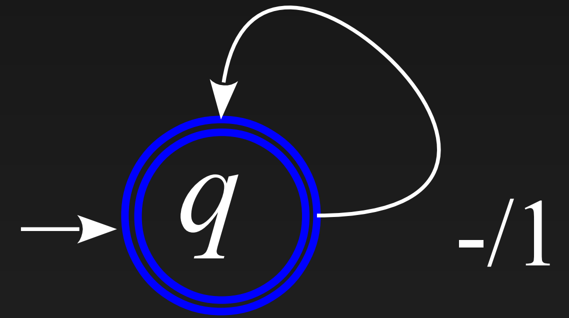


$$L = \{a\ a'\ a''\ \ldots \mid \text{all names distinct}\}$$



Name generator

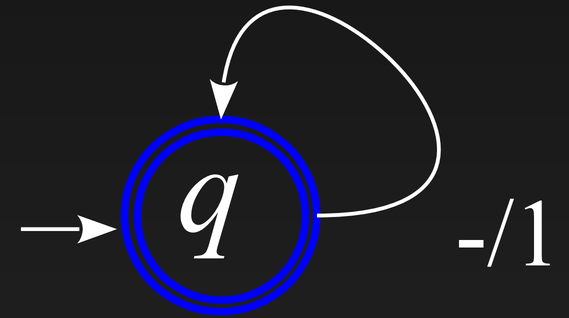
$$P(x) = \nu a.\overline{x}a.P(x)$$



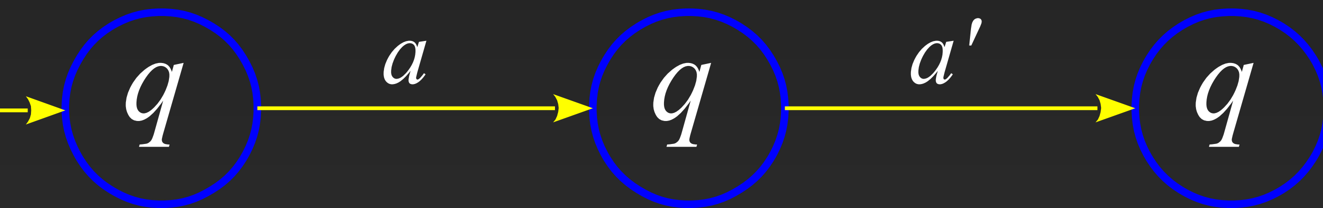
$$L = \{a\ a'\ a''\ \ldots \mid \text{all names distinct}\ \}$$

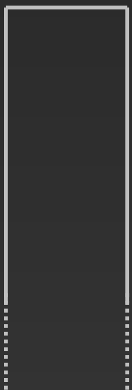
Name generator

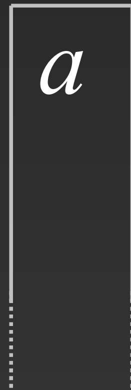
$$P(x) = \nu a.\overline{x}a.P(x)$$



$$L = \{a\ a'\ a''\ \ldots \mid \text{all names distinct}\}$$

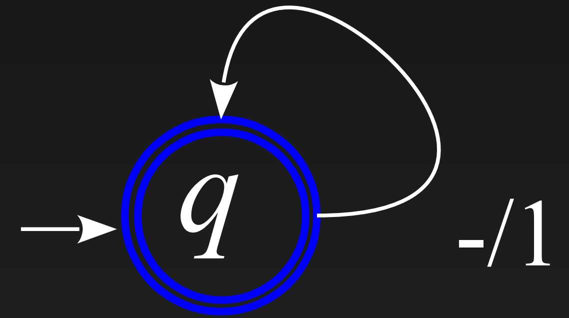




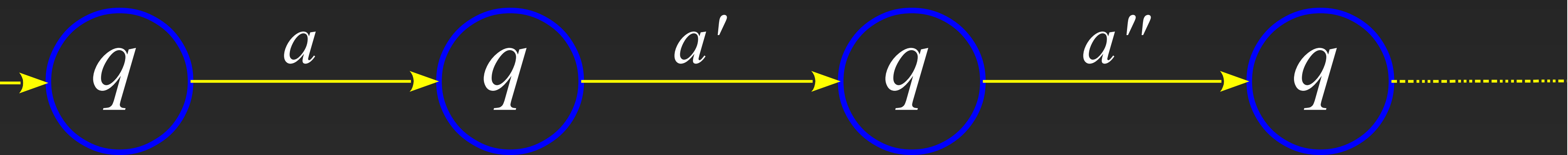


Name generator

$$P(x) = \nu a.\overline{x}a.P(x)$$

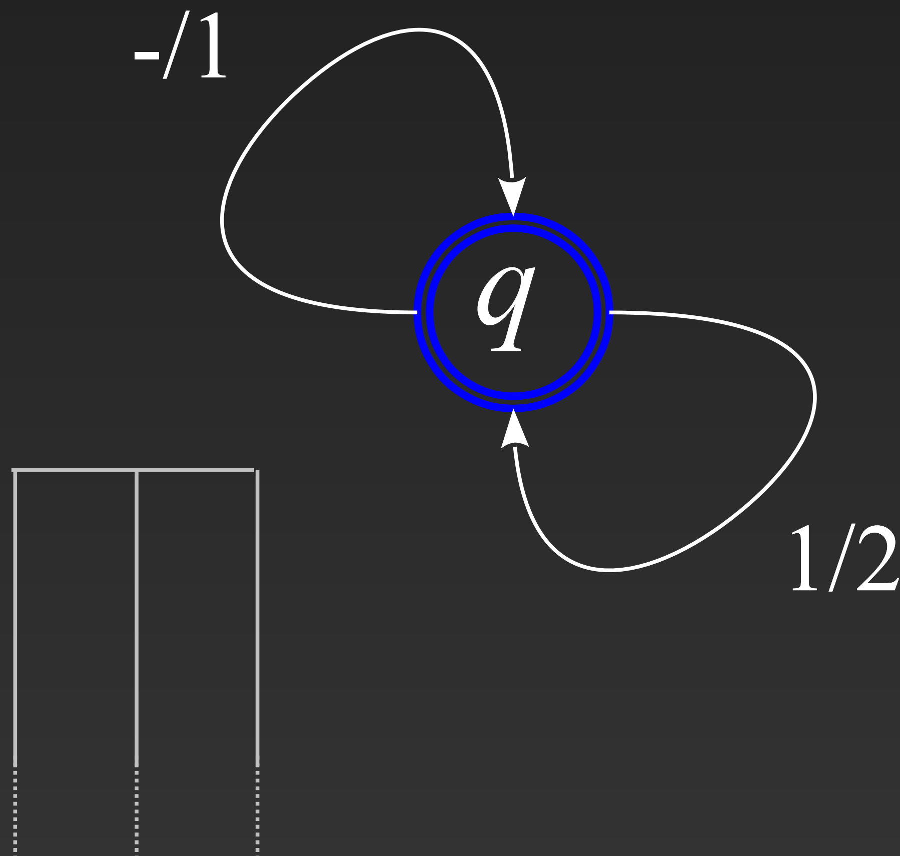


$L = \{a\ a'\ a''\ \ldots \mid$ all names distinct $\}$



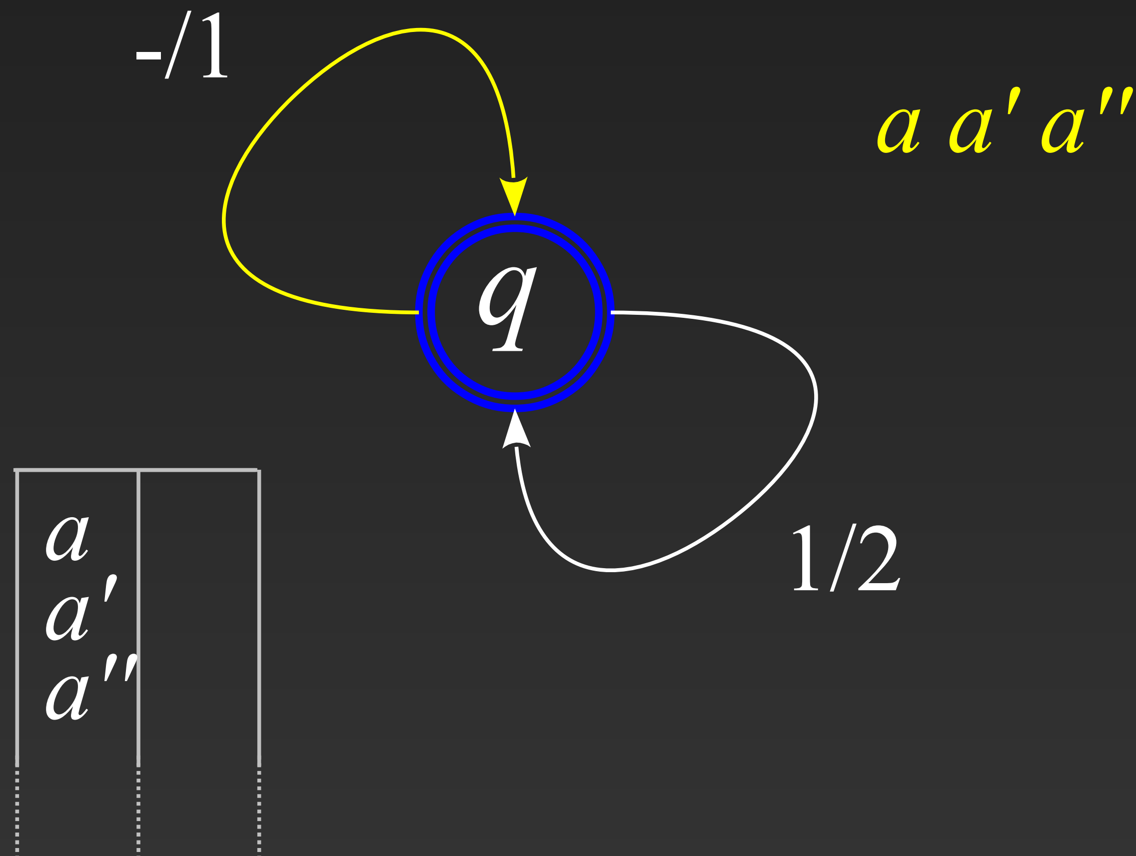
Consumption

$$P(x) \mid\mid C(x,y)$$



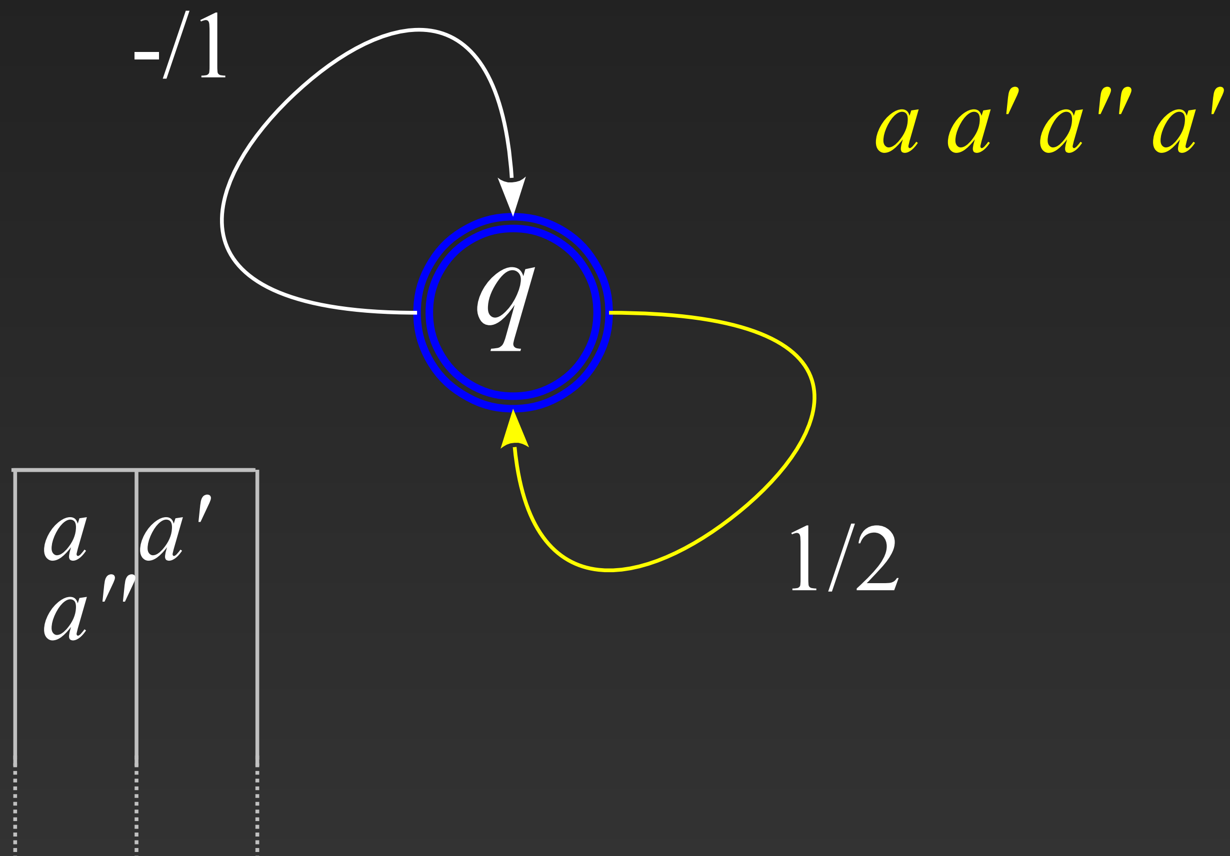
Consumption

$P(x) \parallel C(x,y)$



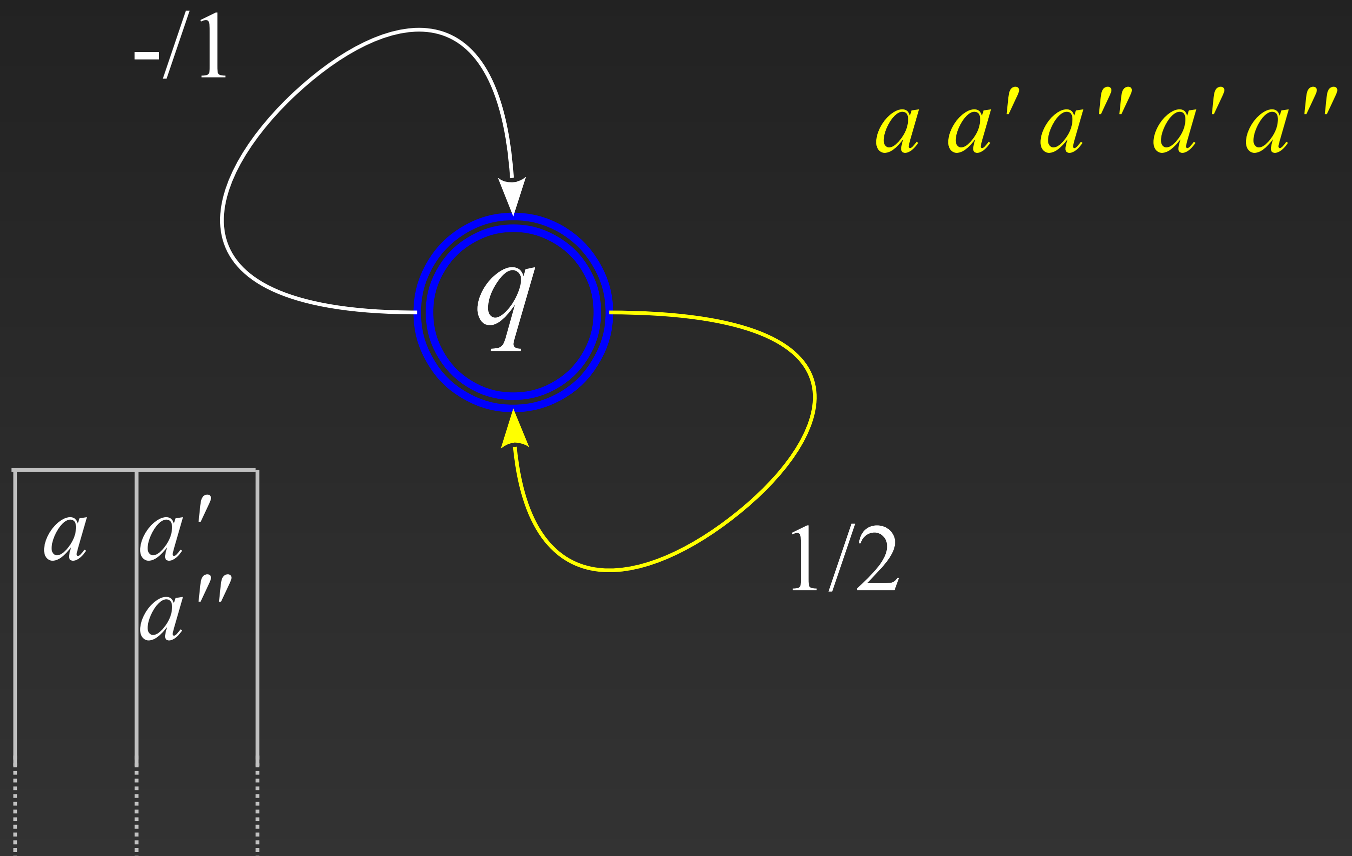
Consumption

$$P(x) \parallel C(x,y)$$



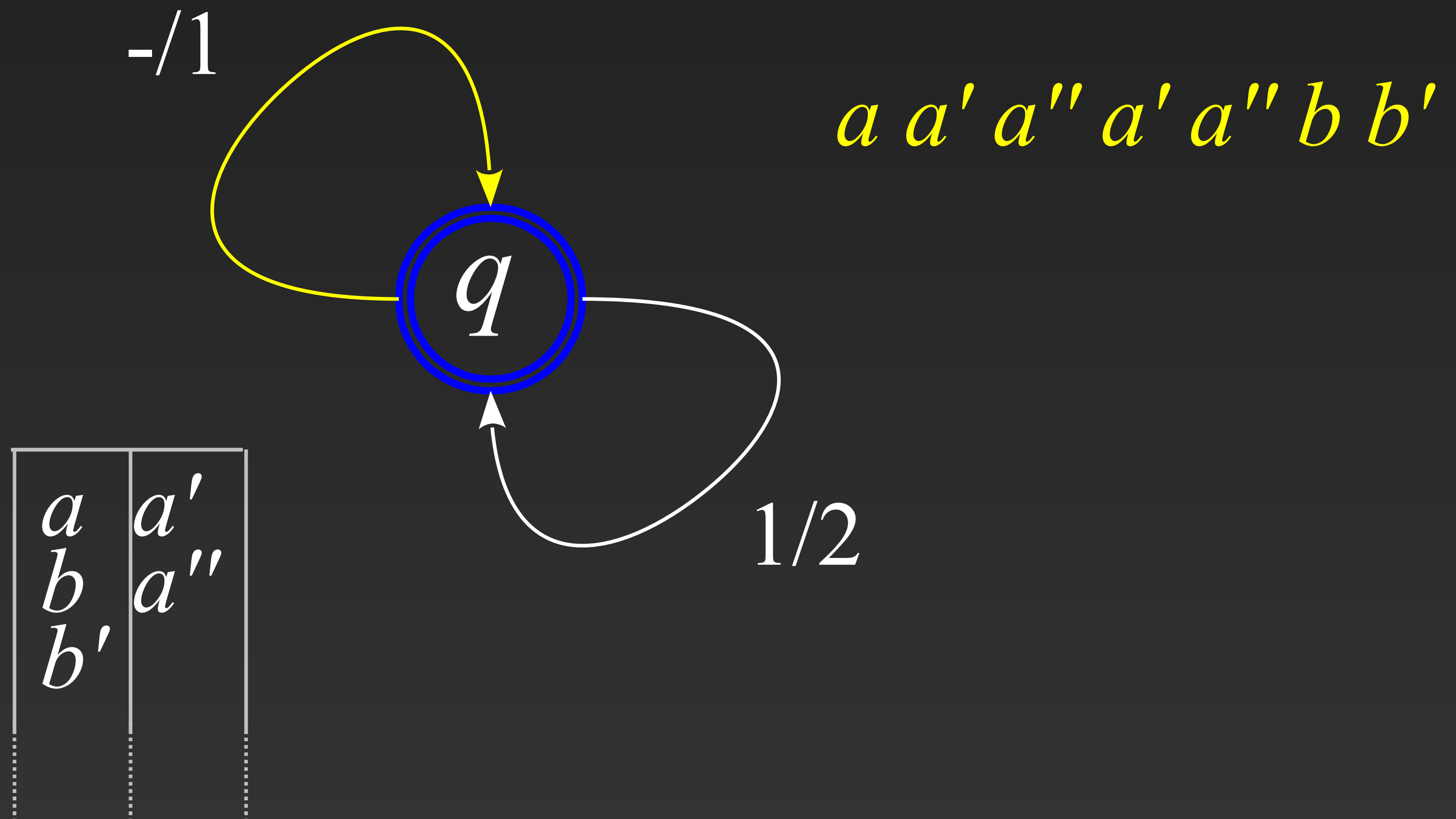
Consumption

$$P(x) \parallel C(x,y)$$



Consumption

$P(x) \parallel C(x,y)$

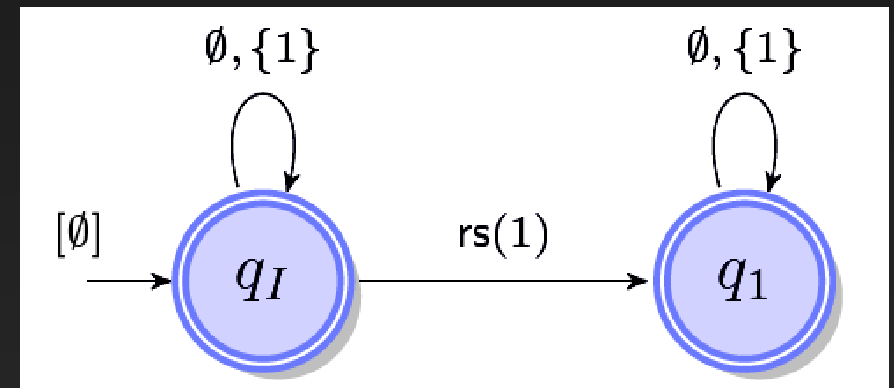


Expressivity

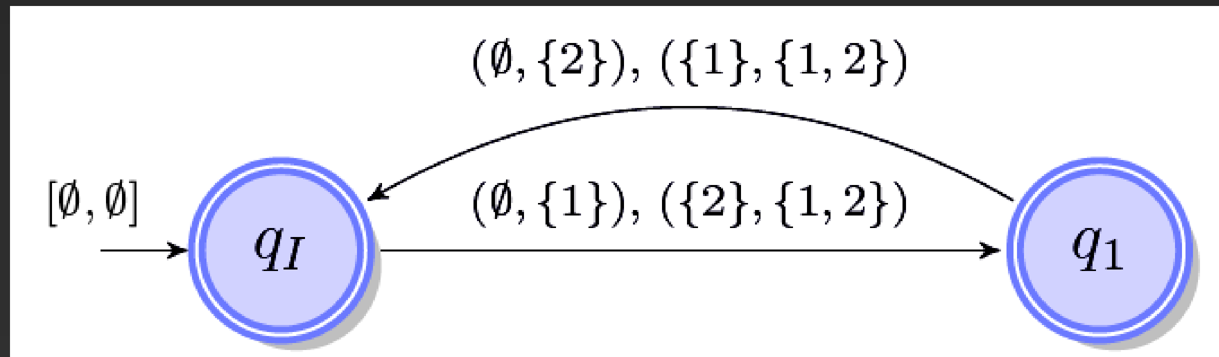
Histories simulate registers → HRAs extend FRAs

Histories can be reset:

→ Closure under Kleene* and concatenation



Several histories → Closure wrt interleavings



HRA properties

- Cleanly extend RAs and FRAs
- Closed under all regular operations apart from complementation
- Closed under interleaving
- Universality undecidable (from RAs)
- Emptiness decidable, non-primitive recursive complexity (~transfer/reset Petri nets)
- Closely related to Data / Class Memory Automata

[Bojańczyk, Muscholl, Schwentick, Segoufin & David '06 , Bjorklund & Schwentick '10]

Concluding

Classes of automata over infinite alphabets

- expressing computation with names/resources
- new landscape of algorithms and results
- applications in verification

Further on:

- open/working problems: automata learning, regular expressions, infinite words, verification logics