Coalgebras for Concurrency
– or –
A bridge between automata and concurrency theory.

Alexandra Silva

Radboud University Nijmegen
Centrum Wiskunde & Informatica

September 6, 2014
TRENDS 2014
Rome, Italy
Automata are basic structures in Computer Science.
Language equivalence: well-studied, several algorithms.
Renewed attention (POPL’11, ’13, ’14).
Concurrency: a spectrum of equivalences.
Checking usually done by reducing to bisimilarity.
An alternative road

- Many efficient algorithms for equivalence of automata.
- Applications in concurrency?
Various spectrum equivalences

= Language equivalence of a *transformed* system

= Automaton with outputs and structured state space (Moore automata).

Bonsangue, Bonchi, Caltais, Rutten, S. MFPS 12
From automata to concurrency

- Generalization of existing algorithms to Moore automata.
- Brzozowski’s and Hopcroft/Karp algorithms for van Glabbeek’s spectrum.
- Cleaveland and Hennessy’s acceptance graphs for must/may testing = Moore automata.
- Brzozowski’s and Hopcroft/Karp algorithms algorithm for must/may testing.

Bonchi, Caltais, Pous, Silva. APLAS 2013
From automata to concurrency

- Generalization of existing algorithms to Moore automata.
- Brzozowski’s and Hopcroft/Karp algorithms for van Glabbeek’s spectrum.
- Cleaveland and Hennessy’s acceptance graphs for must/may testing = Moore automata.
- Brzozowski’s and Hopcroft/Karp algorithms algorithm for must/may testing.

Bonchi, Caltais, Pous, Silva. APLAS 2013
The approach
1. Brief introduction to coalgebra.
2. Two algorithms for language equivalence and generalizations.
3. Trends and opportunities.
Specify and reason about systems.
Specify and reason about systems.

state-machines
e.g. DFA, LTS, PA, ...
(Co)algebra

Specify and reason about systems.

Syntax
RE, CCS, ...

\[ b^*a(b^*a)^* \]
\[ a.b.0 + a.c.0 \]
\[ a.(1/2.0 \oplus 1/2.0) + ... \]

state-machines
e.g. DFA, LTS, PA, ...
Specify and reason about systems.

Syntax
RE, CCS, ...

Axiomatization
KA, ...

state-machines
e.g. DFA, LTS, PA, ...

\[b^*a(b^*a)^*\]

\[a \cdot b \cdot 0 + a \cdot c \cdot 0\]

\[a \cdot (\frac{1}{2} \cdot 0 \oplus \frac{1}{2} \cdot 0) + \cdots\]

\[1 + a \cdot a^* = a^*\]

\[p + 0 = p\]

\[p \cdot p \oplus p' \cdot p = (p + p') \cdot p\]
Specify and reason about systems.

Syntax
RE, CCS, ...

Axiomatization
KA,...

Syntax
b^*a(b^*a)^*

b^*a(b^*a)^* = 1 + a a^* = a^*

Axiomatization
p + 0 = p

Axiomatization
p.P \oplus p'.P = (p + p').P

Can we do all of this uniformly in a single framework?
What do this things have in common?

\[(S, t : S \to 2 \times S^A)\]

\[(S, t : S \to \mathcal{P}S^A)\]

\[(S, t : S \to \mathcal{P}\mathcal{D}_\omega(S)^A)\]

\[(S, t : S \to TS) \quad T\text{-coalgebras}\]
What do this things have in common?

\[(S, t : S \to 2 \times S^A)\]

\[(S, t : S \to \mathcal{P}S^A)\]

\[(S, t : S \to \mathcal{P}\mathcal{D}_\omega(S)^A)\]

\[(S, t : S \to TS) \quad T\text{-coalgebras}\]
What do this things have in common?

\[(S, t : S \rightarrow 2 \times S^A)\]

\[(S, t : S \rightarrow \mathcal{P}S^A)\]

\[(S, t : S \rightarrow \mathcal{P}\mathcal{D}_\omega(S)^A)\]

\[(S, t : S \rightarrow TS)\quad T\text{-coalgebras}\]
What do this things have in common?

\[(S, t : S \to 2 \times S^A)\]

\[(S, t : S \to \mathcal{P}S^A)\]

\[(S, t : S \to \mathcal{P}\mathcal{D}_\omega(S)^A)\]

\[(S, t : S \to TS) \quad T\text{-coalgebras}\]
What do this things have in common?

\[(S, t : S \to 2 \times S^A)\]

\[(S, t : S \to \mathcal{P}S^A)\]

\[(S, t : S \to \mathcal{P}\mathcal{D}_\omega(S)^A)\]

\[(S, t : S \to TS) \quad T\text{-coalgebras}\]
The power of $T$

$(S, t : S \rightarrow TS)$

The functor $T$ determines:

1. notion of observational equivalence (coalg. bisimulation)
   E.g. $T = 2 \times (-)^A$: language equivalence
2. behaviour (final coalgebra)
   E.g. $T = 2 \times (-)^A$: languages over $A - 2^A$
3. set of expressions describing finite systems
4. axioms to prove bisimulation equivalence of expressions
The power of $T$

$(S, t : S \rightarrow TS)$

The functor $T$ determines:

1. notion of observational equivalence (coalg. bisimulation)
   E.g. $T = 2 \times (-)^A$: language equivalence

2. behaviour (final coalgebra)
   E.g. $T = 2 \times (-)^A$: languages over $A – 2^A^*$

3. set of expressions describing finite systems

4. axioms to prove bisimulation equivalence of expressions
The power of $T$

$$(S, t : S \rightarrow TS)$$

The functor $T$ determines:

1. notion of observational equivalence (coalg. bisimulation)
   E.g. $T = 2 \times (-)^A$: language equivalence

2. behaviour (final coalgebra)
   E.g. $T = 2 \times (-)^A$: languages over $A - 2^A$

3. set of expressions describing finite systems

4. axioms to prove bisimulation equivalence of expressions
The power of $T$

$$(S, t : S \rightarrow TS)$$

The functor $T$ determines:

1. notion of observational equivalence (coalg. bisimulation)
   E.g. $T = 2 \times (-)^A$: language equivalence

2. behaviour (final coalgebra)
   E.g. $T = 2 \times (-)^A$: languages over $A - 2^A^*$

3. set of expressions describing finite systems

4. axioms to prove bisimulation equivalence of expressions

1 + 2 are classic coalgebra; 3 + 4 are recent work.
Current state of affairs

- Coalgebra/coinduction – semantic side of the world: operational/denotational semantics, logics, . . .
- Key role in current development of functional languages, type theory, . . .
- This talk: uniform derivation of algorithms and applications to concurrency.
Brzozowski’s algorithm

Brzozowski’s algorithm, (co)algebraically – Kozen’s festschrift 2012
Brzozowski’s algorithm (by example)

• initial state: \( x \)  
• final states: \( y \) and \( z \)

• \( L(x) = \{a, b\}^* a \)

• \( X \) is reachable but not minimal: \( L(y) = \varepsilon + \{a, b\}^* a = L(z) \)
Brzozowski’s algorithm (by example)

- initial state: $x$
- final states: $y$ and $z$
- $L(x) = \{a, b\}^* a$
- $X$ is reachable but not minimal: $L(y) = \varepsilon + \{a, b\}^* a = L(z)$
Reversing the automaton: $\text{rev}(X)$

- Transitions are reversed
- Initial states $\leftrightarrow$ final states
- $\text{rev}(X)$ is non-deterministic
Reversing the automaton: \( \text{rev}(X) \)

- Transitions are reversed.
- Initial states \( \Leftrightarrow \) final states.
- \( \text{rev}(X) \) is non-deterministic.
Reversing the automaton: $\text{rev}(X)$

- transitions are reversed
- initial states $\Leftrightarrow$ final states
- $\text{rev}(X)$ is non-deterministic
Reversing the automaton: \( \text{rev}(X) \)

- transitions are reversed
- initial states ⇔ final states
- \( \text{rev}(X) \) is non-deterministic
Making it deterministic again: $\text{det}(\text{rev}(X))$

- new state space: $2^X = \{ V \mid V \subseteq \{x, y, z\} \}$
- initial state: $\{y, z\}$
- final states: all $V$ with $x \in V$
- $V \xrightarrow{a} W$ $W = \{ w \mid v \xrightarrow{a} w, v \in V \}$
Making it deterministic again: \( \text{det}(\text{rev}(X)) \)

- new state space: \( 2^x = \{ V \mid V \subseteq \{ x, y, z \} \} \)
- initial state: \( \{ y, z \} \)  
  final states: all \( V \) with \( x \in V \)
- \( V \xrightarrow{a} W \)  
  \( W = \{ w \mid v \xrightarrow{a} w, v \in V \} \)
Making it deterministic again: $\text{det}(\text{rev}(X))$

- new state space: $2^X = \{ V \mid V \subseteq \{x, y, z\} \}$
- initial state: $\{y, z\}$ final states: all $V$ with $x \in V$
- $V \xrightarrow{a} W$ $W = \{ w \mid w \xrightarrow{a} w, v \in V \}$
Making it deterministic again: \( \text{det}(\text{rev}(X)) \)

- new state space: \( 2^X = \{ V \mid V \subseteq \{ x, y, z \} \} \)
- initial state: \( \{ y, z \} \)
- final states: all \( V \) with \( x \in V \)
- \( V \xrightarrow{a} W \quad W = \{ w \mid v \xrightarrow{a} w, v \in V \} \)
The automaton $\text{det}(\text{rev}(X))$ . . .

. . . accepts the reverse of the language accepted by $X$:

$$L(\text{det}(\text{rev}(X))) = a\{a,b\}^* = \text{reverse}(L(X))$$

. . . and is observable!
The automaton $\text{det}(\text{rev}(X))$ . . .

- . . . accepts the reverse of the language accepted by $X$:

$$L(\text{det}(\text{rev}(X))) = a\{a, b\}^* = \text{reverse}(L(X))$$

- . . . and is observable!
The automaton $\text{det}(\text{rev}(X))$ . . .

- . . . accepts the reverse of the language accepted by $X$:

$$L(\text{det}(\text{rev}(X))) = a\{a, b\}^* = \text{reverse}(L(X))$$

- . . . and is observable!
If: a deterministic automaton $X$ is *reachable* and accepts $L(X)$

then: $\text{det}(\text{rev}(X))$ is *minimal* and

$$L(\text{det}(\text{rev}(X))) = \text{reverse}(L(X))$$
Brzozowski’s Theorem

If: a deterministic automaton \( X \) is reachable and accepts \( L(X) \) then: \( \text{det}(\text{rev}(X)) \) is minimal and

\[
L(\text{det}(\text{rev}(X))) = \text{reverse}(L(X))
\]
Taking the reachable part of $\text{det}(\text{rev}(X))$

- $\text{reach}(\text{det}(\text{rev}(X)))$
Taking the reachable part of $\text{det}(\text{rev}(X))$

- $\text{reach}(\text{det}(\text{rev}(X)))$
Taking the reachable part of \( \det(\text{rev}(X)) \)

- \( \text{reach}(\det(\text{rev}(X))) \) is reachable (by construction)
Repeating everything, now for $reach(\text{det}(\text{rev}(X)))$

... gives us $reach(\text{det}(\text{rev}(reach(\text{det}(\text{rev}(X))))))$

... which is (reachable and) minimal and accepts $\{a, b\}^* a$. 
Repeating everything, now for $reach(det(rev(X)))$

- ... gives us $reach(det(rev(reach(det(rev(X)))))$)
- which is (reachable and) minimal and accepts $\{a, b\}^* a$. 
Repeating everything, now for \( \text{reach}(\text{det}(\text{rev}(X))) \)

\[ \xymatrix{ & a, b \\ x, y, z \\ y, z \\ \emptyset \\ & a, b } \quad \xymatrix{ & b \\ s \\ & a \\ t \\ a } \]

- . . . gives us \( \text{reach}(\text{det}(\text{rev}(\text{reach}(\text{det}(\text{rev}(X)))))\))
- which is (reachable and) minimal and accepts \( \{a, b\}^* a \).
Repeating everything, now for $\text{reach}(\text{det}(\text{rev}(X)))$

- . . . gives us $\text{reach}(\text{det}(\text{rev}(\text{reach}(\text{det}(\text{rev}(X)))))$)
- which is (reachable and) minimal and accepts $\{a, b\}^* a$. 
All in all: Brzozowski’s algorithm

- $X$ is reachable and accepts $\{a, b\}^* a$
- $\text{reach}(\text{det}(\text{rev}(\text{reach}(\text{det}(\text{rev}(X)))))$) also accepts $\{a, b\}^* a$
- ... and is minimal!!
All in all: Brzozowski’s algorithm

- $X$ is reachable and accepts $\{a, b\}^* a$
- $\text{reach}(\text{det}(\text{rev}(\text{reach}(\text{det}(\text{rev}(X)))))),\text{rev})$ also accepts $\{a, b\}^* a$
- \ldots and is minimal!!
All in all: Brzozowski’s algorithm

- $X$ is reachable and accepts $\{a, b\}^* a$
- $\text{reach} (\text{det}(\text{rev}(\text{reach} (\text{det}(\text{rev}(X))))) )$ also accepts $\{a, b\}^* a$
- . . . and is minimal!!
All in all: Brzozowski’s algorithm

- $X$ is reachable and accepts $\{a, b\}^* a$
- $\text{reach}(\text{det}(\text{rev}(\text{reach}(\text{det}(\text{rev}(X)))))))$ also accepts $\{a, b\}^* a$

... and is minimal!!
All in all: Brzozowski’s algorithm

- $X$ is reachable and accepts $\{a, b\}^* a$
- $\text{reach}(\text{det}(\text{rev}(\text{reach}(\text{det}(\text{rev}(X)))))))$ also accepts $\{a, b\}^* a$
- . . . and is minimal!!
Brzozowski’s algorithm

(1) reverse and determinize;
(2) take the reachable part;
(3) reverse and determinize;
(4) take the reachable part.

Checking language equivalence
Minimize both automata and check for isomorphism.

Crucial observation for generalizations
Reverse and determinize is more general than at first sight!
Beyond deterministic automata

**Brzozowski (X)**

1. reverse and determinize;
2. take the reachable part;
3. reverse and determinize;
4. take the reachable part.

**Checking language equivalence**

Minimize both automata and check for isomorphism.

**Crucial observation for generalizations**

Reverse and determinize is more general than at first sight!
Brzozowski’s algorithm

(1) reverse and determinize;
(2) take the reachable part;
(3) reverse and determinize;
(4) take the reachable part.

Checking language equivalence
Minimize both automata and check for isomorphism.

Crucial observation for generalizations
Reverse and determinize is more general than at first sight!
Reverse and determinize

\[ 2^{(-)} : g \quad \mapsto \quad 2^g \]

where \( 2^V = \{ S \mid S \subseteq V \} \) and, for all \( S \subseteq W \),

\[ 2^g(S) = g^{-1}(S) \quad (= \quad \{ v \in V \mid g(v) \in S \} ) \]

- This construction is \textit{contravariant} and . . . .
- Works for general \( B^{(-)} \) and . . . .
- For structured sets (change in category).
Reverse and determinize

\[ 2^{(-)} : g \quad \rightarrow \quad 2^g \]

where \( 2^V = \{ S \mid S \subseteq V \} \) and, for all \( S \subseteq W \),

\[ 2^g(S) = g^{-1}(S) \quad ( = \{ v \in V \mid g(v) \in S \} ) \]

- This construction is *contravariant* and . . . . .
- Works for general \( B^{(-)} \) and . . . . .
- For structured sets (change in category).
Reverse and determinize

\[
2^{(\_\_\_)} : V \xrightarrow{g} W \quad \iff \quad 2^W \quad \xleftarrow{2^g}
\]

where \(2^V = \{ S \mid S \subseteq V \}\) and, for all \(S \subseteq W\),

\[
2^g(S) = g^{-1}(S) \quad (= \quad \{ v \in V \mid g(v) \in S \})
\]

- This construction is *contravariant* and . . . .
- Works for general \(B^{(\_\_\_)}\) and . . . .
- For structured sets (change in category).
Brzozowski’s algorithm generalized

Deterministic automata  
\[ X \rightarrow B \times X^A \quad \mathcal{P}(A^*) \]
Brzozowski’s algorithm generalized

Deterministic automata
\[ X \rightarrow B \times X^A \]

Moore automata
\[ X \rightarrow B \times X^A \]

Linear weighted automata
\[ V \rightarrow \mathbb{R} \times V^A \]

Guarded strings automata
\[ \mathcal{B} \rightarrow B \times \mathcal{B}^{B \times A} \]

\[ \mathcal{P}(A^*) \]
\[ B^A^* \]
\[ \mathbb{R}^{A^*} \]
\[ \mathcal{P}((\mathcal{A} \cdot A)^* \cdot \mathcal{A}) \]

Correctness and generalizations in [BBRS’12, BBHPRS’13].
Brzozowski’s algorithm in concurrency

Cleaveland and Hennessy’s acceptance graphs for must/may testing = Moore automata.

Several equivalences of the spectrum (failure, ready-trace, ... ) = regular behaviors Moore automata.

See APLAS paper for details.
Brzozowski’s algorithm in concurrency

Cleaveland and Hennessy’s acceptance graphs for must/may testing = Moore automata.

Several equivalences of the spectrum (failure, ready-trace, . . .) = regular behaviors Moore automata.

See APLAS paper for details.
Intermezzo

- Brzozowski’s algorithm can be uniformly generalized based on the type functor.
Brzozowski’s algorithm can be uniformly generalized based on the type functor.

Second example: up-to algorithm (HKC).
Up-to techniques

Tools and proof techniques for systems equivalence

Methodology:
1. characterise coinductively a given notion of equivalence
2. improve the associated proof method
Deterministic finite automata

The states $x$ and $u$ are language equivalent
Deterministic finite automata

The states $x$ and $u$ are language equivalent
The states $x$ and $u$ are language equivalent
Deterministic finite automata

The states $x$ and $u$ are language equivalent
Deterministic finite automata

The states $x$ and $u$ are language equivalent
Deterministic finite automata

The states $x$ and $u$ are language equivalent.
Complexity

The previous algorithm is *quadratic*
Complexity

The previous algorithm is *quadratic*.

3 pairs
The previous algorithm is *quadratic*
The previous algorithm is *quadratic*.
The previous algorithm is *quadratic*.
The previous algorithm is *quadratic*.
Complexity

The previous algorithm is *quadratic*
The previous algorithm is \textit{quadratic}

9 pairs
The previous algorithm is *quadratic*
Complexity

The previous algorithm is *quadratic*

11 pairs
The previous algorithm is \textit{quadratic}
The previous algorithm is *quadratic*.
Complexity

The previous algorithm is *quadratic*.
The previous algorithm is *quadratic*.
Complexity

The previous algorithm is \textit{quadratic}.

16 pairs
The previous algorithm is *quadratic*
The previous algorithm is *quadratic*
The previous algorithm is *quadratic*.
The previous algorithm is *quadratic*.
Complexity

The previous algorithm is *quadratic*
The previous algorithm is \textit{quadratic}

22 pairs
The previous algorithm is *quadratic*
Complexity

The previous algorithm is *quadratic*.

21 pairs
The previous algorithm is *quadratic*
First improvement

One can stop much earlier

Complexity: almost linear

21 pairs

[Tarjan ’75]
First improvement

One can stop much earlier

Complexity: almost linear

\[ \leq 20 \text{ pairs} \]

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

21 19 pairs

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

21 18 pairs [Tarjan '75]
First improvement

One can stop much earlier

\[ \begin{array}{cccccc}
\cdot & \rightarrow & \cdot & \rightarrow & \cdot & \rightarrow \\
\cdot & \rightarrow & \cdot & \rightarrow & \cdot & \rightarrow \\
\cdot & \rightarrow & \cdot & \rightarrow & \cdot & \rightarrow \\
\cdot & \rightarrow & \cdot & \rightarrow & \cdot & \rightarrow \\
\cdot & \rightarrow & \cdot & \rightarrow & \cdot & \rightarrow \\
\cdot & \rightarrow & \cdot & \rightarrow & \cdot & \rightarrow \\
\end{array} \]

\( \geq 17 \text{ pairs} \)

**Complexity**: almost linear

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

21 16 pairs

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

[21 15 pairs] [Tarjan ‘75]
First improvement

One can stop much earlier

Complexity: almost linear

21 14 pairs

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

$21 \downarrow 13$ pairs

[Tarjan ‘75]
First improvement

One can stop much earlier

Complexity: almost linear

$\geq 12$ pairs

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

21 11 pairs

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

\[\text{\#\# 10 pairs}\]

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

$21 \leftrightarrow 9$ pairs

[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

[Hopcroft and Karp '71]
[Tarjan '75]
First improvement

One can stop much earlier

Complexity: almost linear

[Hopcroft and Karp ’71]
[Tarjan ’75]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[ o^\#(S) = \bigvee_{s \in S} o(s) \]

\[ t^\#(S)(a) = \bigcup_{s \in S} t(s)(a) \]

plete

\[ x \quad y \quad z \quad x+y \quad y+z \quad x+y+z \]

\[ u \quad v+w \quad u+w \quad u+v+w \]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[ o^#(S) = \bigvee_{s \in S} o(s) \]
\[ t^#(S)(a) = \bigcup_{s \in S} t(s)(a) \]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[ o^\#(S) = \bigvee_{s \in S} o(s) \]

\[ t^\#(S)(a) = \bigcup_{s \in S} t(s)(a) \]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[
o^\#(S) = \bigvee_{s \in S} o(s)
\]

\[
t^\#(S)(a) = \bigcup_{s \in S} t(s)(a)
\]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[
o^\#(S) = \bigvee_{s \in S} o(s)
\]

\[
t^\#(S)(a) = \bigcup_{s \in S} t(s)(a)
\]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[ o^#(S) = \bigvee_{s \in S} o(s) \]

\[ t^#(S)(a) = \bigcup_{s \in S} t(s)(a) \]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[
o^\#(S) = \bigvee_{s \in S} o(s)
\]

\[
t^\#(S)(a) = \bigcup_{s \in S} t(s)(a)
\]
Non-Deterministic Automata

Hopcroft and Karp *on the fly*, with powerset construction:

\[
o^\#(S) = \bigvee_{s \in S} o(s)
\]

\[
t^\#(S)(a) = \bigcup_{s \in S} t(s)(a)
\]
Non-Deterministic Automata

Hopcroft and Karp \textit{on the fly}, with powerset construction:

\[
o^\#(S) = \bigvee_{s \in S} o(s)
\]

\[
t^\#(S)(a) = \bigcup_{s \in S} t(s)(a)
\]
Non-Deterministic Automata

One can do better:

\[(x, u) + (y, v+w) = (x+y, u+v+w)\]

using bisimulations \textit{up to union}
Non-Deterministic Automata

One can do better:

\[(x, u) + (y, v+w) = (x+y, u+v+w)\]

using bisimulations up to union
One can do better:

\[(x, u) + (y, v+w) = (x+y, u+v+w)\]

using bisimulations up to union
Non-Deterministic Automata

One can do even better:

\[ x + y = u + y \quad (1) \]
\[ = y + z + y \quad (2) \]
\[ = y + z \]
\[ = u \quad (2) \]

using bisimulations up to congruence

this lead to the HKC algorithm [Bonchi, Pous, POPL’13]
Non-Deterministic Automata

One can do even better:

\[ x + y = u + y \quad \text{(1)} \]
\[ = y + z + y \quad \text{(2)} \]
\[ = y + z \]
\[ = u \quad \text{(2)} \]

\[ x \rightarrow y + z \rightarrow x + y \rightarrow x + y + z \]

using bisimulations \textit{up to congruence}

this lead to the HKC algorithm [Bonchi, Pous, POPL’13]
Non-Deterministic Automata

One can do even better:

\[ x + y = u + y \quad (1) \]

\[ = y + z + y \quad (2) \]

\[ = y + z \]

\[ = u \quad (2) \]

\[ x \rightarrow y + z \rightarrow x + y \rightarrow x + y + z \]

using bisimulations *up to congruence*

this lead to the HKC algorithm [Bonchi, Pous, POPL’13]
Non-Deterministic Automata

One can do **even** better:

\[ x + y = u + y \]  
\[ = y + z + y \]  
\[ = y + z \]  
\[ = u \] \( (2) \)

using bisimulations **up to congruence**

this lead to the HKC algorithm [Bonchi, Pous, POPL’13]
HKC is also parametric

\[
\text{HKC}(X, Y):
\begin{align*}
(1) & \ R \text{ is empty;} \ todo \text{ is } \{(X', Y')\}; \\
(2) & \text{ while } todo \text{ is not empty, do} \\
(2.1) & \text{ extract } (X', Y') \text{ from } todo; \\
(2.2) & \text{ if } (X', Y') \in c(R \cup todo) \text{ then continue;} \\
(2.3) & \text{ if } o^\sharp(X') \neq o^\sharp(Y') \text{ then return } false; \\
(2.4) & \text{ for all } a \in A, \\
& \quad \text{ insert } (t^\sharp(X')(a), t^\sharp(Y')(a)) \text{ in } todo; \\
(2.5) & \text{ insert } (X', Y') \text{ in } R; \\
(3) & \text{ return } true;
\end{align*}
\]

Powerset construction $o^\sharp$, $t^\sharp$

Generalized to other algebraic structures / functors (weighted, Moore, probabilistic automata, . . .)

Applicable for must/may testing, failure, . . .
HKC is also parametric

\[
\text{HKC}(X, Y): \\
(1) \text{ } R \text{ is empty; } todo \text{ is } \{(X', Y')\}; \\
(2) \text{ while } todo \text{ is not empty, do} \\
(2.1) \text{ extract } (X', Y') \text{ from } todo; \\
(2.2) \text{ if } (X', Y') \in c(R \cup todo) \text{ then continue}; \\
(2.3) \text{ if } o^\#(X') \neq o^\#(Y') \text{ then return } false; \\
(2.4) \text{ for all } a \in A, \\
\quad \text{insert } (t^\#(X')(a), t^\#(Y')(a)) \text{ in } todo; \\
(2.5) \text{ insert } (X', Y') \text{ in } R; \\
(3) \text{ return } true;
\]

Powerset construction $o^\#$, $t^\#$
Generalized to other algebraic structures / functors (weighted, Moore, probabilistic automata, . . .)

Applicable for must/may testing, failure, . . .
Trends / opportunities

Trend I: New language constructs

Trend II: NetKat – applications in networks

Trend III: Automata learning
Trend I: New language constructs

- Extensions of programming languages with coinductive constructs (Agda, CoCaml, ...).
- Algorithms like general HKC enable efficient representation and equivalence check.

Opportunity for concurrency

- New methods to check equivalence of behaviors.
- Automatic derivation of programming constructs for new models.
Trend II: NetKAT – semantic foundations for networks
Anderson, Foster, Guha, Jeannin, Kozen, Schlesinger, Walker, POPL’14

- Specifying and reasoning about networks.
- Based on Kleene algebra with tests (KAT).

Recent work (submitted)
- Coinductive model of KAT extended to NetKAT.
- Brzozowski and HKC for NetKAT.

Opportunity for concurrency
- Foundations of networks: transference of results, new challenges.
Trend II: NetKAT – semantic foundations for networks
Anderson, Foster, Guha, Jeannin, Kozen, Schlesinger, Walker, POPL’14

- Specifying and reasoning about networks.
- Based on Kleene algebra with tests (KAT).

Recent work (submitted)
- Coinductive model of KAT extended to NetKAT.
- Brzozowski and HKC for NetKAT.

Opportunity for concurrency
- Foundations of networks: transference of results, new challenges.
Trend III: automata learning

- Angluin’s algorithm: inference of regular languages.
- Coalgebra enables generalizations to e.g. weighted automata.

Opportunity for concurrency

- Inference of behaviors in distributed systems.
- Applications in security.
Conclusions

- Coalgebra has applications in automata and concurrency.
- Bridge to transfer results and tools.
- (Co)algebra is not only semantics but also algorithms!

Thanks! Questions?
Conclusions

- Coalgebra has applications in automata and concurrency.
- Bridge to transfer results and tools.
- (Co)algebra is not only semantics but also algorithms!

Thanks! Questions?
Conclusions

- Coalgebra has applications in automata and concurrency.
- Bridge to transfer results and tools.
- (Co)algebra is not only semantics but also algorithms!

Thanks! Questions?