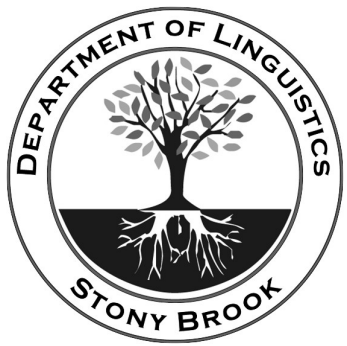
Learning reduplication with 2-way finite-state transducers

Hossep Dolatian & Jeffrey Heinz



ICGI
Wrocław University of Science and Technology
Sept 7, 2018

Acknowledgments

Hossep Dolatian

- This research is supported by NIH grant #R01-HD087133 to JH.

Copying sequential information

Copying (=duplication, doubling, mimicry)

- biological sciences
- planning and control (robotics)
- natural language...

→ word-formation or morphology (=reduplication)

Copying in Natural Language

Many languages (~83%) use reduplication to mark meaning

Indonesian plural

- buku → buku~buku, 'book'→'books'
- wanita→ wanita~wanita, 'woman'→'women'

Tohono O'odham plural

- kotwa → kok~twa, 'shoulder' → 'shoulders'
- sikul → sis~kul, 'younger sibling' → 'younger siblings'

(Rubino, 2013; Cohn, 1989) and (Anderson and Smith 2017)

In this talk, we...

- Present (the old) deterministic 2-way finite-state transducer (FST) as a new way to represent reduplicative processes;
- Identify a subclass of those transducers which covers most reduplication patterns we studied;
- Show how this subclass is learnable from examples.
  The trick is to decompose the 2-way FSTs into the concatenation of 1-way FSTs and learn the 1-way FSTs with known methods.

Studying Linguistic Variation/Typology

Requires two books:

- "encyclopedia of categories"
- "encyclopedia of types"

Wilhelm Von Humboldt

Basic typology of reduplication

- Typology: Wide variation in how natural languages copy:

(1) Total reduplication = unbounded copy (~83%)
wanita→wanita~wanita 'woman'→'women' (Indo.)

(2) Partial reduplication = bounded copy (~75%)

a.	C:	gen→g~gen 'to sleep'→'to be sleeping'	(Shilh)
b.	CV:	guyon→gu~guyon 'to jest'→'to jest repeatedly'	(Sundanese)
c.	CVC:	takki→ tak~takki 'leg'→'legs'	(Agta)
d.	CVCV:	banagaɲu→bana~banagaɲu 'return'	(Dyirbal)

Basic typology of reduplication

And it gets wider

(3) Triplication:
roar→ roar~roar-roar
'give a shudder' →'continue to shudder' (Mokilese)

(4) Final reduplication:
erasi→erasi~rasi
'he is sick'→'he continues being sick' (Siriono)

(5) Subconstituent copying:
ku-haata→ku-haata~haata
'to ferment'→'to start fermenting' (KiHehe)

(6) Left-right copying:
lú:t'ux^{w}→lúx^{w}~lút'ux^{w}
'to value'→'... (plural)' (Nisgha)

Basic Typology of Reduplication

(7) Syllable-counting:

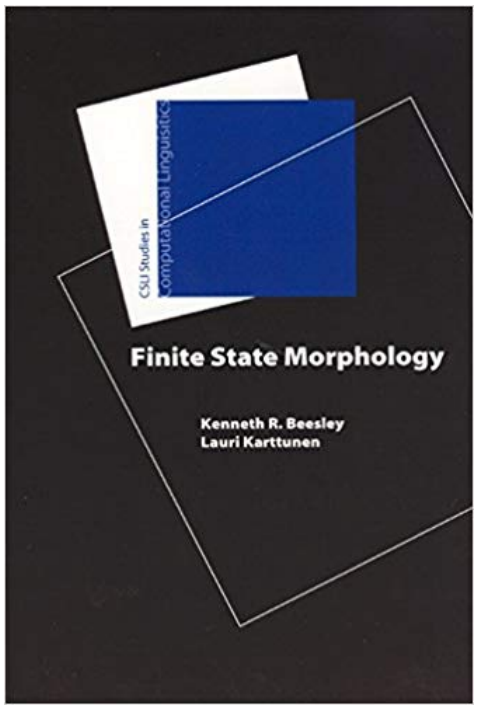
a. jang→jang~jang
'sheet'→'every sheet' (Mandarin)

b. jialuen→meei jialuen
'gallon'→'every gallon'

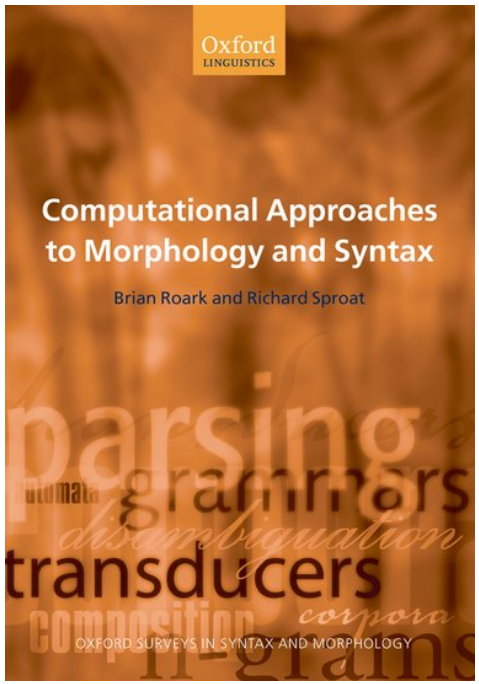
(8) Echo reduplication:
tras→tras~vras
'grief'→'grief schmief' (Hindi)

Computational Nature of Word Formation

Word formation processes are rational relations, analyzable with (1-way) finite-state methods



Beesley and Karttunen 2003



Roark and Sproat 2007

1-way FSTs and reduplication

- 1-way FSTs memorize a large but finite list of strings and their copies
- For partial reduplication = bounded # of segments copied:
  - **Extension**: productively modeled ☺
  - **Size**: burdensome because of state explosion ☹
  - **Intension**: treated as 'remembering' and not 'copying' ☹
- For total reduplication = unbounded # of segments copied:
  - **Extension**: If we assume a finite lexicon, can be modeled ☺...
  - but can't be extended productively to new words ☹
  - output language is non-regular L_{ww}=$\{ ww \mid w \in \Sigma^* \}$
  - **Size**: larger state explosion ☹!
  - **Intension**: can't capture productivity + 'remembering' again ☹
- Appendix: more contrasts + difference in 'remembering' vs. 'copying' using origin semantics (Bojańczyk, 2014)

Responses to the 1-way problem

- Approximate:
  - Stick to 1-way FST approximations (Walther, 2000; Cohen-Sygal and Wintner, 2006; Beesley and Karttunen, 2003; Hulden, 2009)
  - **But**: impose un-linguistic restrictions (e.g. a finite bound on word size,...) and don't directly capture reduplication
- Non-finite-state mechanisms:
  - MCFGs (Albro, 2005), HPSG (Crysmann, 2017), pushdown accepters with queues (Savitch, 1989)
  - **But:** those are recognizers not transducers

2-way FSTs

- Mainstream FSTs are 1-way FSTs because they read the input once from left to right.
- 2-way FSTs are an enriched class of FSTs that can go back and forth on the input (Engelfriet and Hoogeboom, 2001; Savitch, 1982).
- A 2-way FST can do everything a 1-way FST can do, and more.
- Equivalances to logical transduction, other kinds of machines:

$$\begin{aligned} &[\![\text{2-way FSTs}]\!] \\ =\ &[\![\text{MSO-definable transductions}]\!] \\ =\ &[\![\text{Streaming String Transducers}]\!] \end{aligned}$$

(Courcelle, 1997; Engelfriet and Hoogeboom, 2001; Alur, 2010)

Definition

2-way deterministic FST

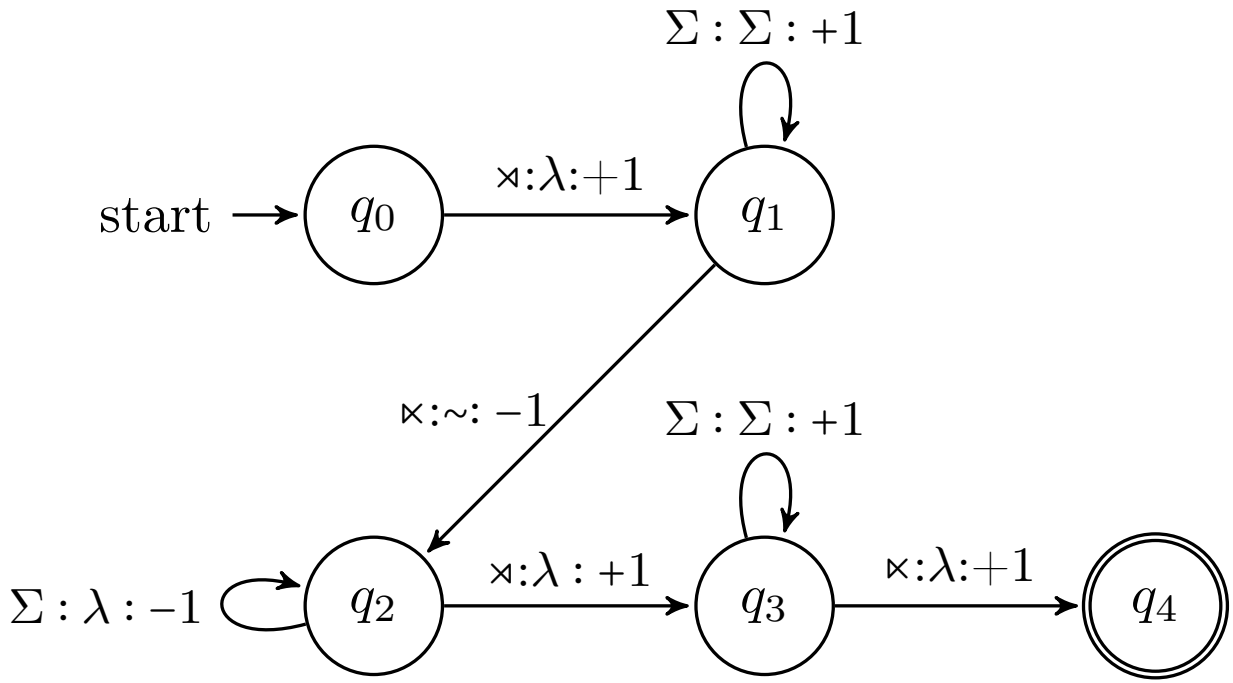
A 2-way, deterministic FST is a six-tuple $(Q, \Sigma_{\ltimes}, \Gamma, q_0, F, \delta)$ such that:

- Q is a finite set of states,
- $\Sigma_{\ltimes} = \Sigma \cup \{\rtimes, \ltimes\}$ is the input alphabet,
- Γ is the output alphabet,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- $\delta : Q \times \Sigma \to Q \times \Gamma^* \times D$ is the transition function where the direction $D = \{-1, 0, +1\}$.

2-way FSTs - Total reduplication

- Total reduplication copies an unbounded size

  wanita→wanita~wanita ‘woman’→‘women’ (Indo.)
- 2-way FST reads the input left-to-right (+1), goes back (-1), and reads it again (+1)



2-way FSTs - Total Reduplication

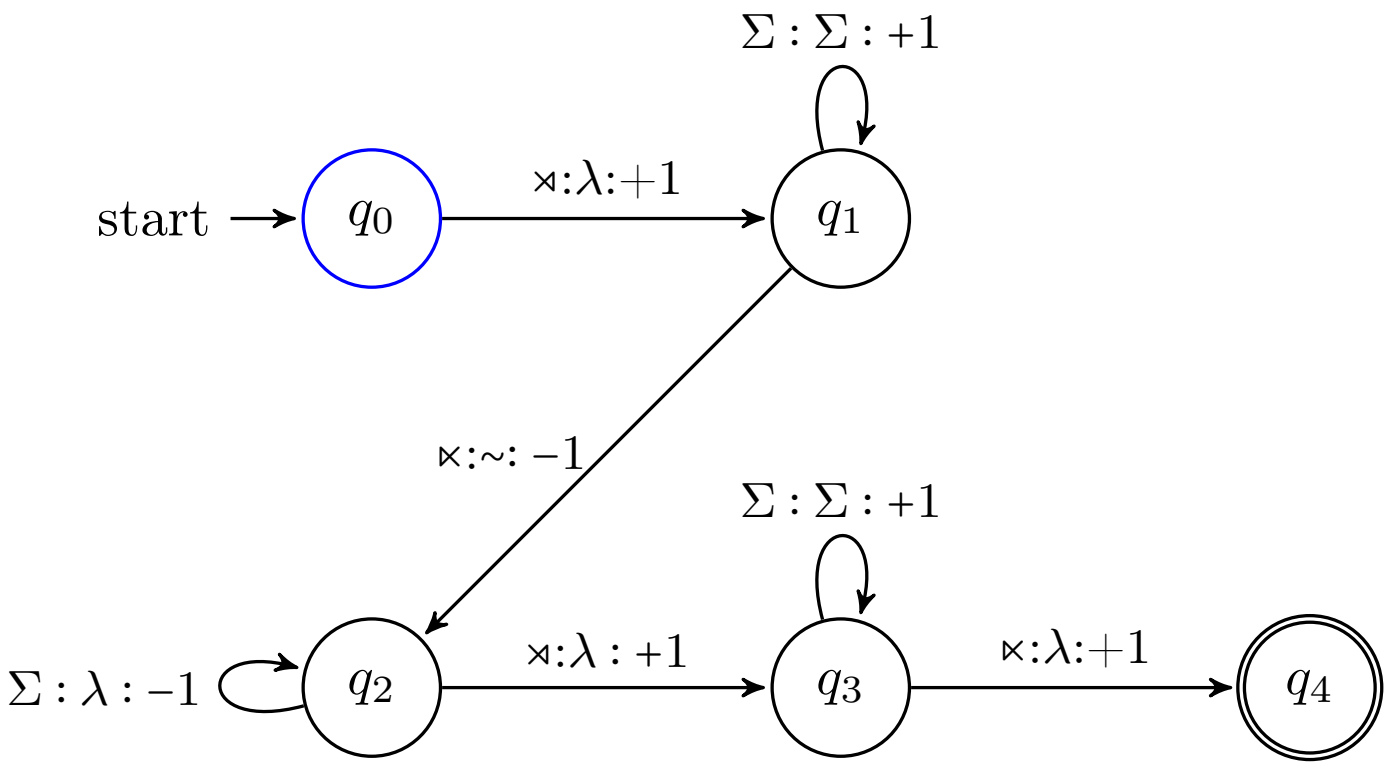
- Indonesian example: wanita→wanita~wanita
- Working example: bye→?

2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

  Input: ⋊ b y e ⋉

  Output:

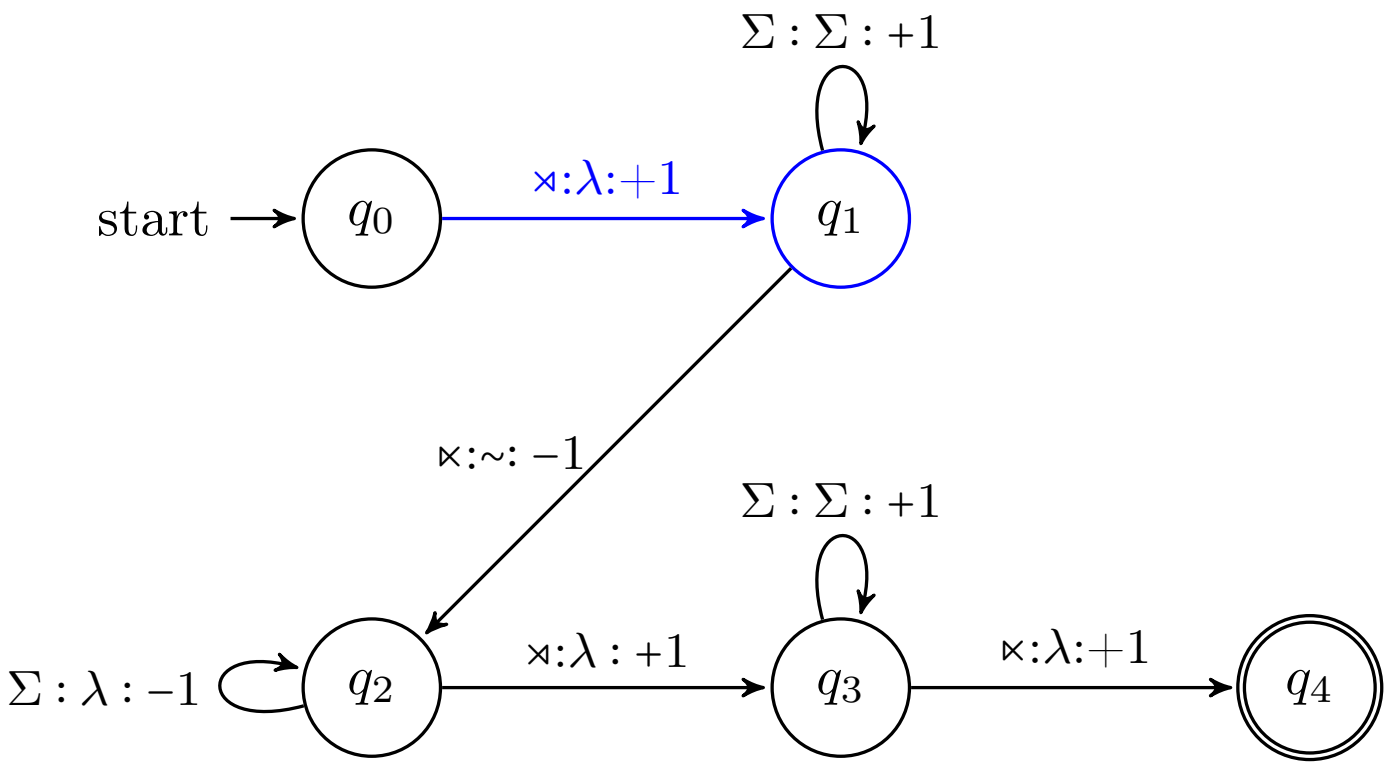


2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

  Input: ⋊ b y e ⋉

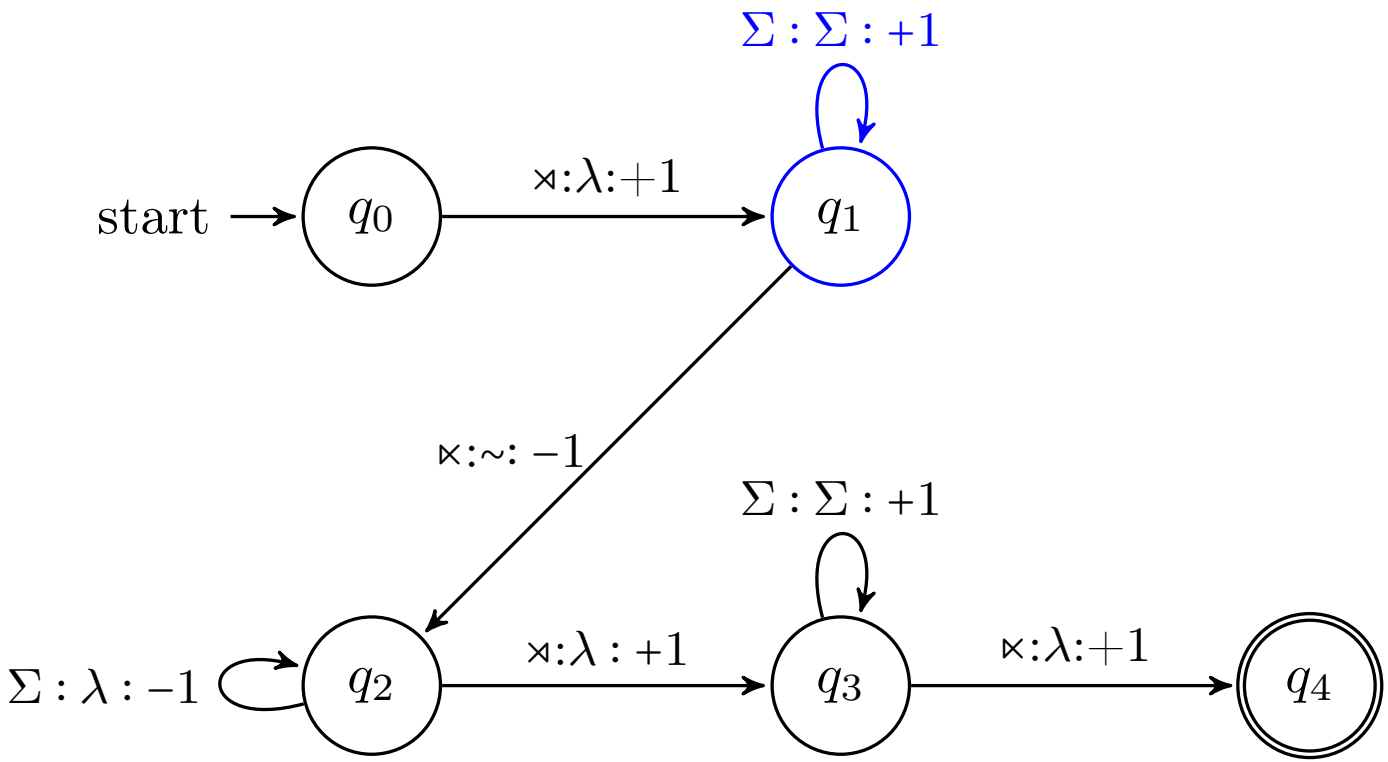
  Output:



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

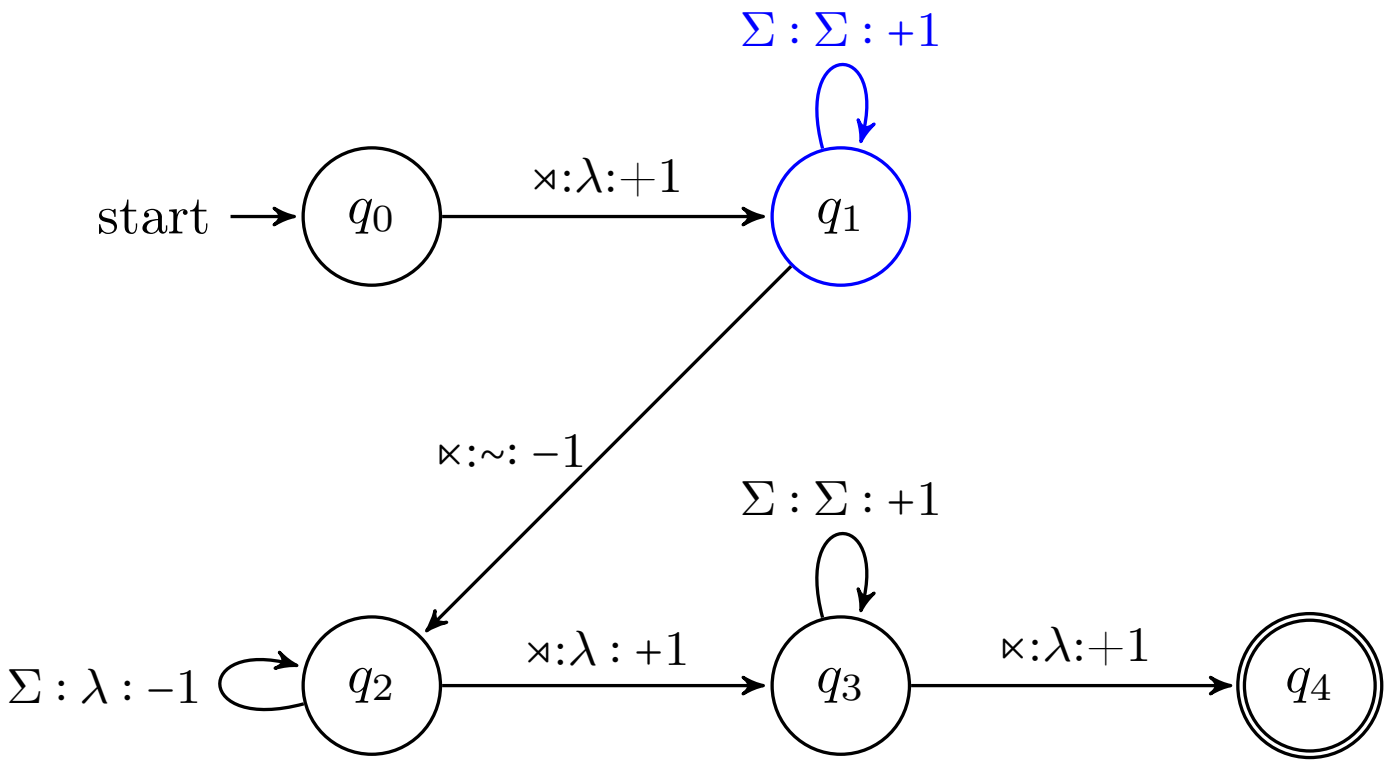
Input:	⋊	b	y	e	⋉
Output:		b			



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

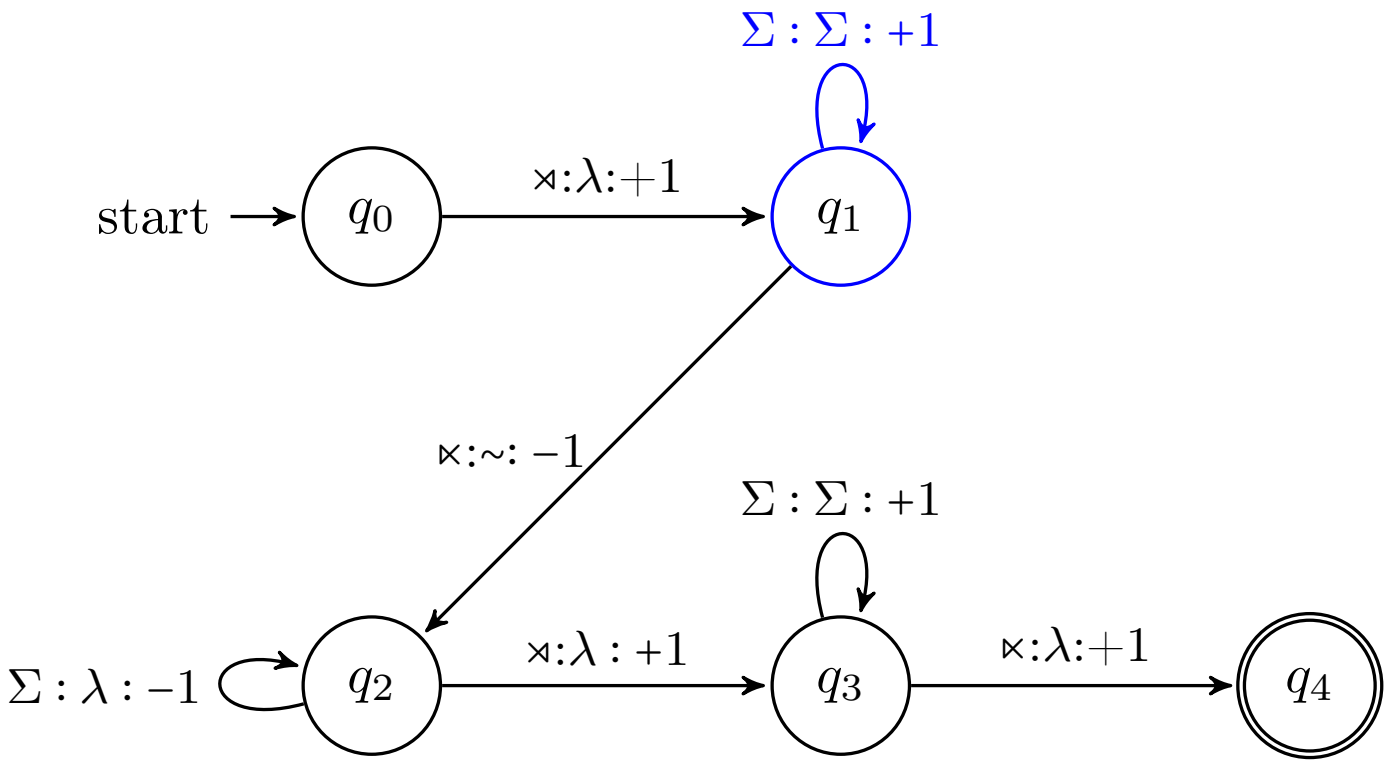
Input:	⋊	b	y	e	⋉
Output:		b	y		



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

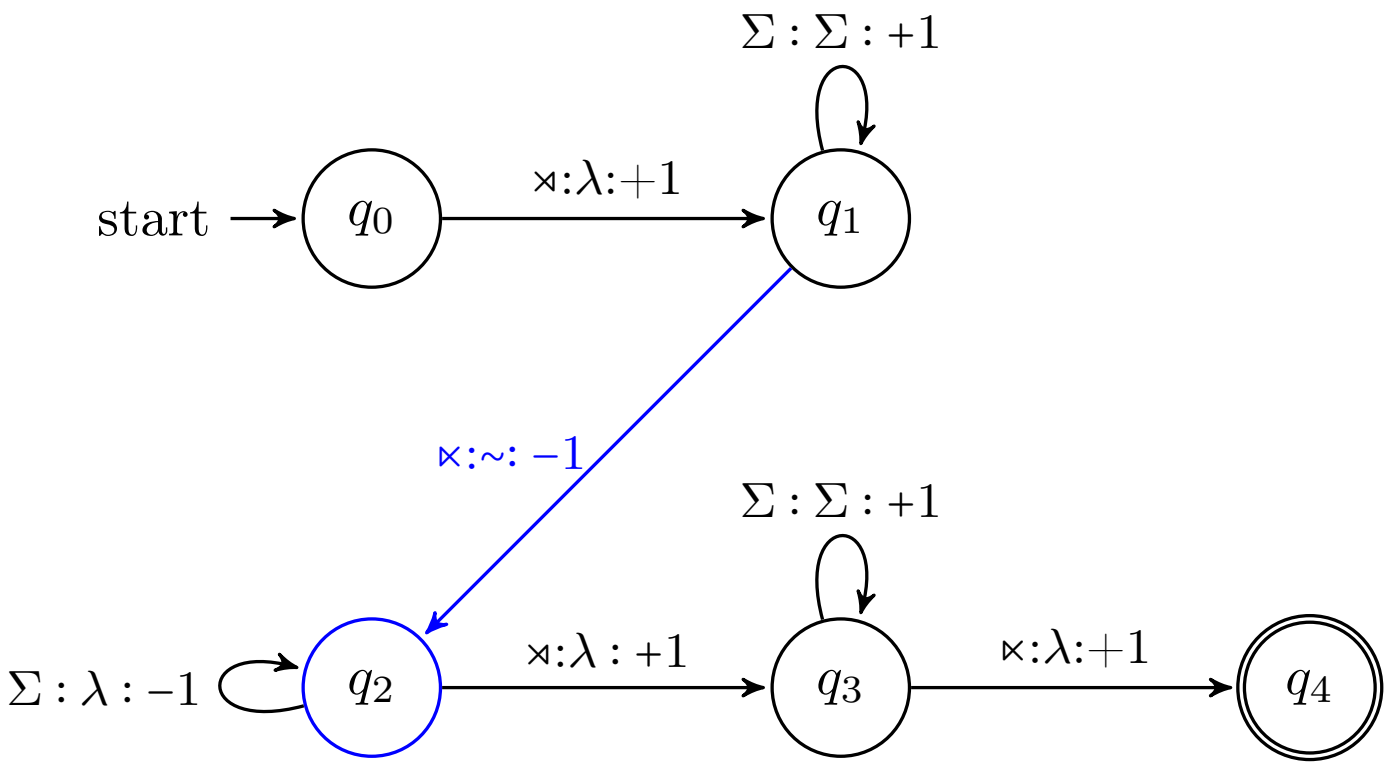
Input:	⋊	b	y	e	⋉
Output:		b	y	e	



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

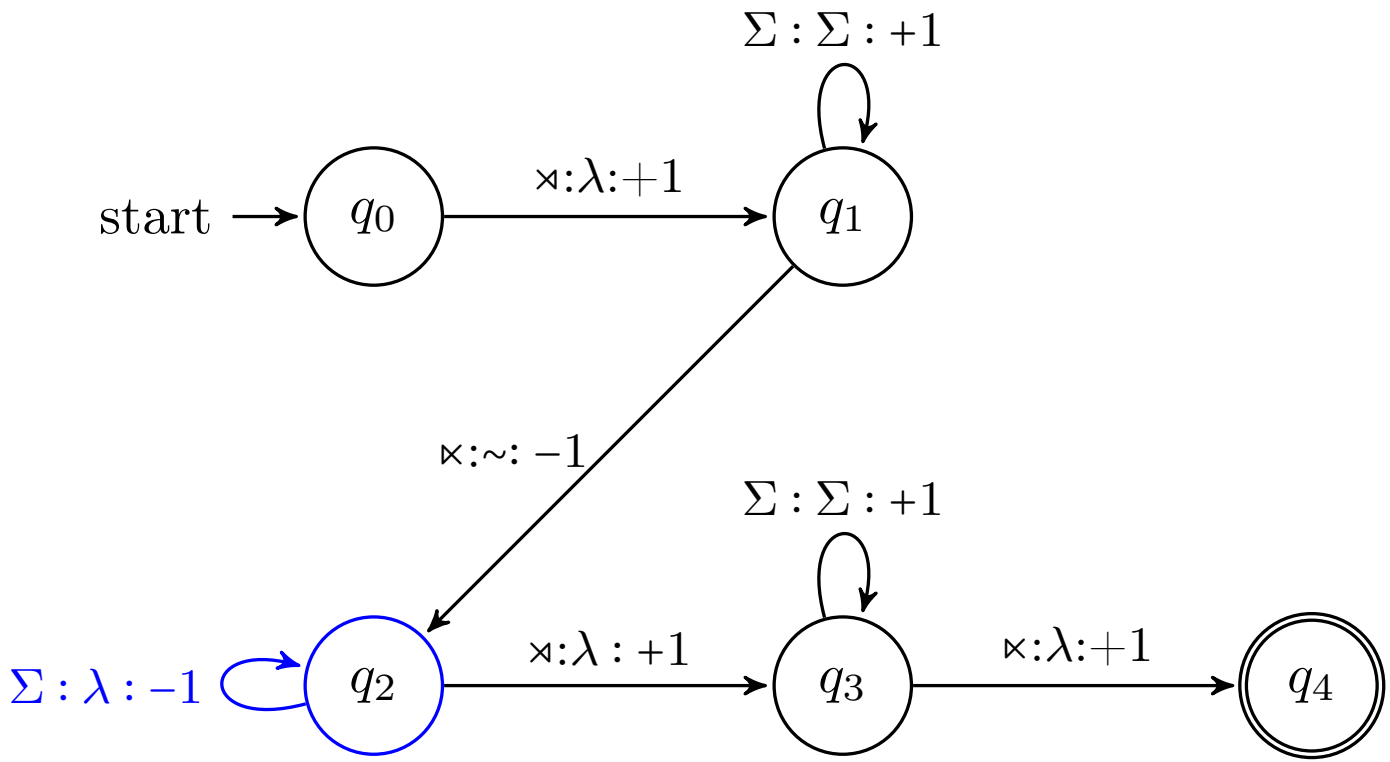
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

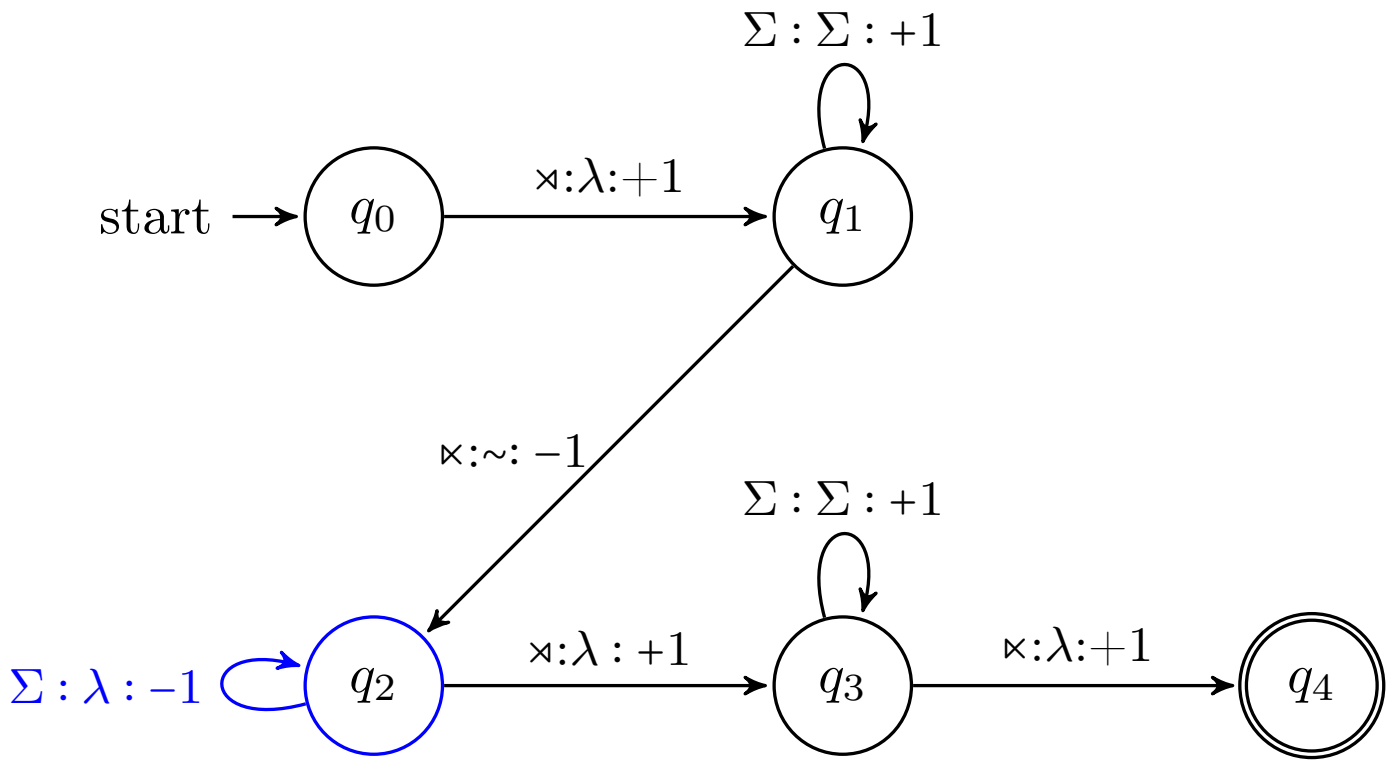
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

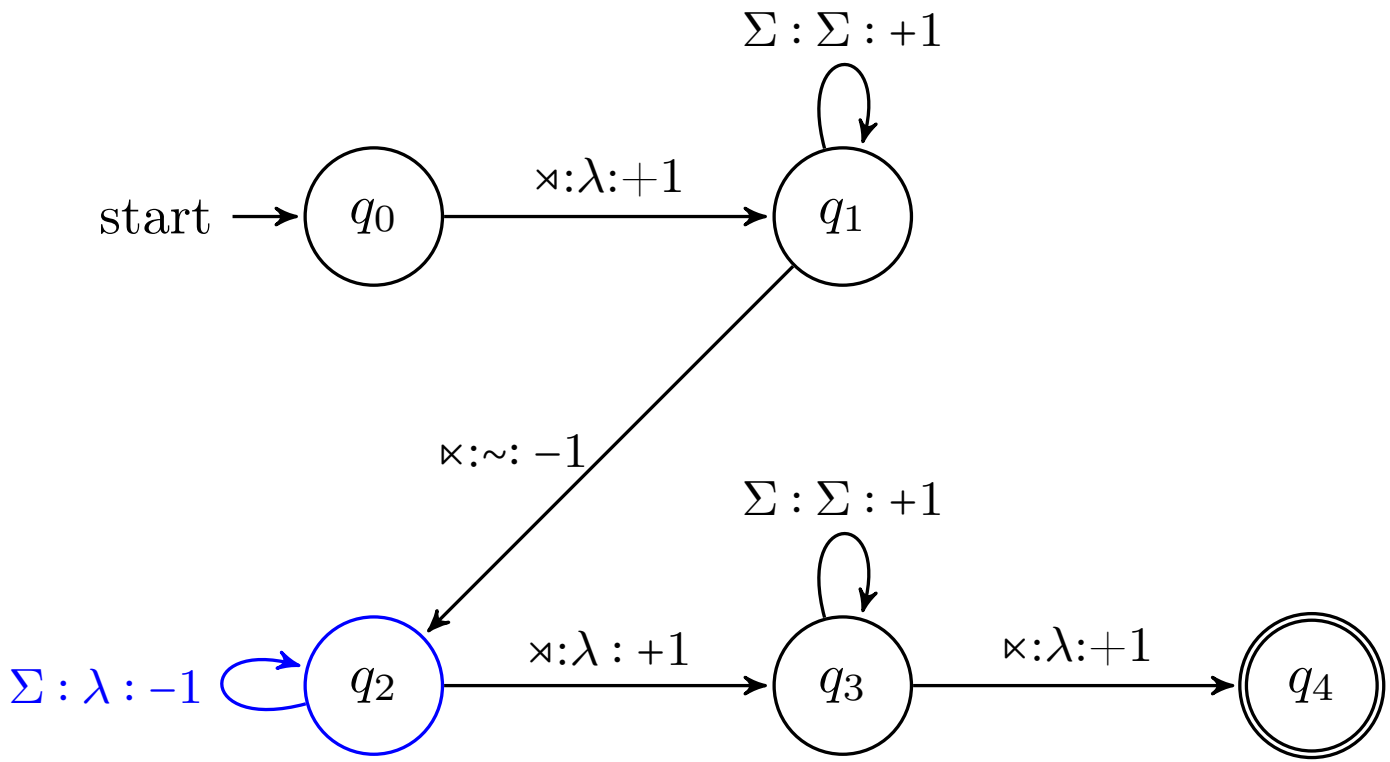
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

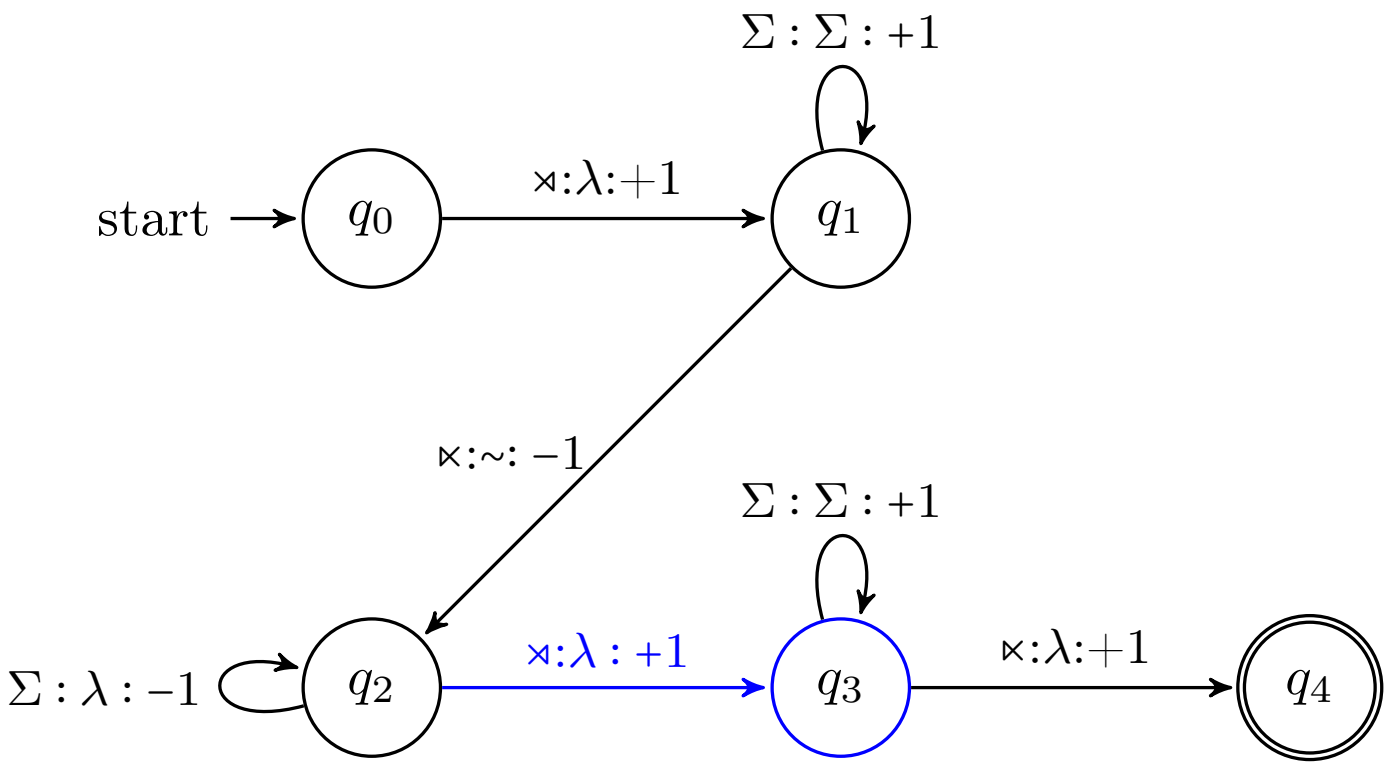
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

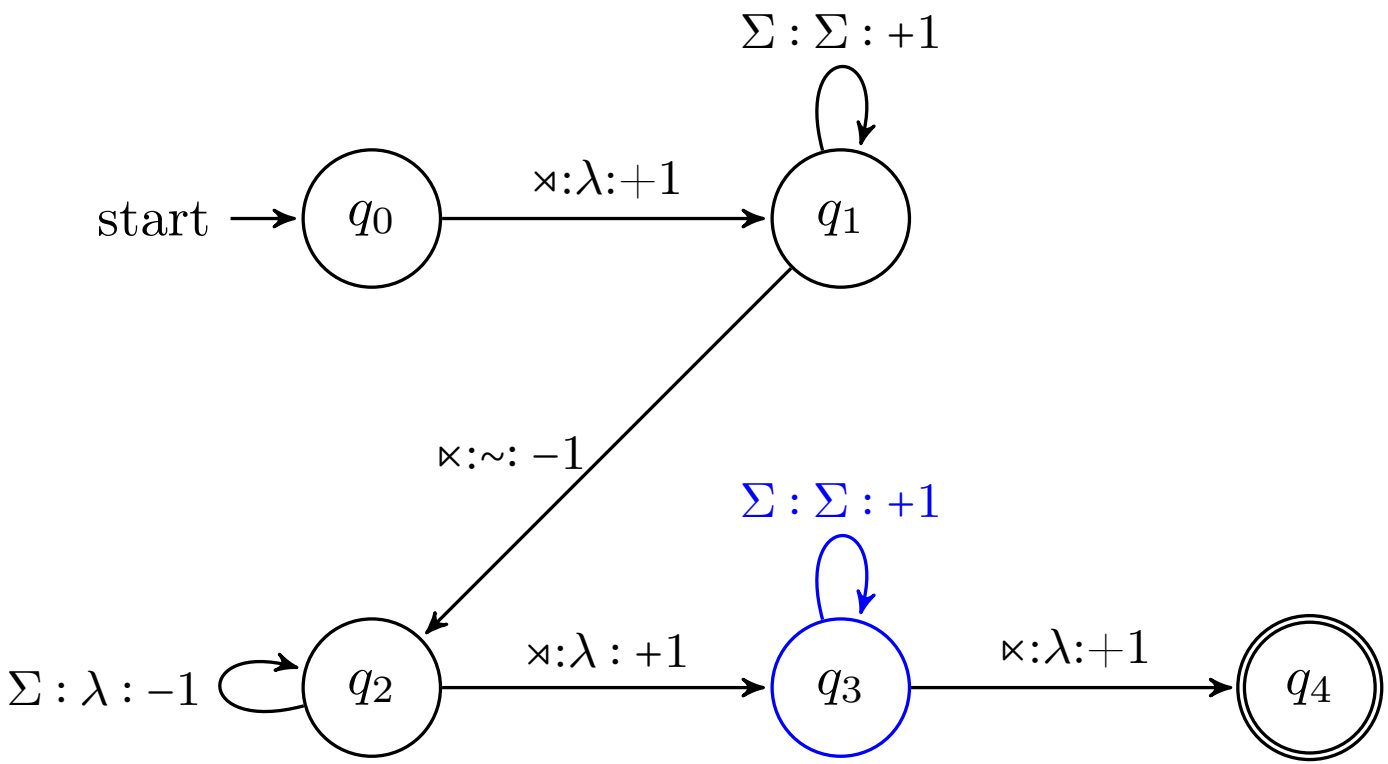
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

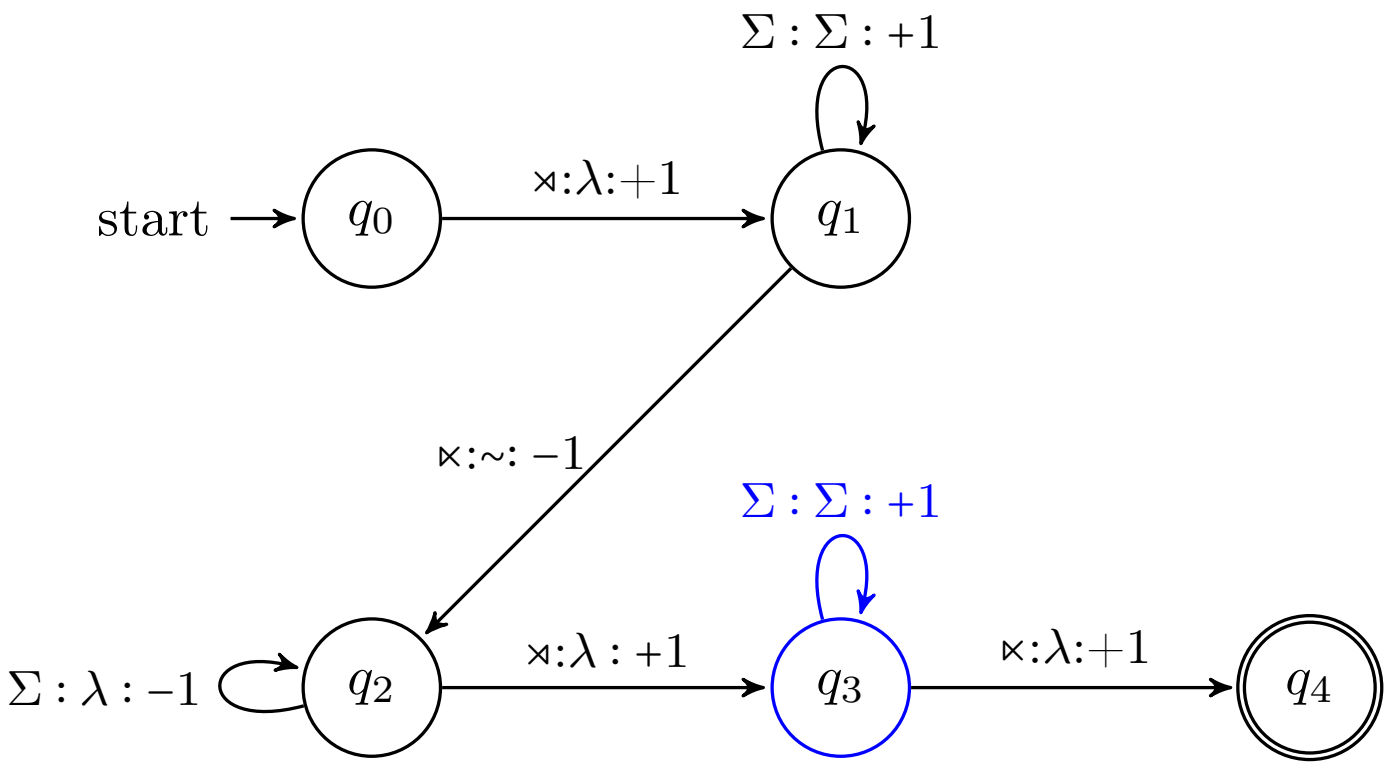
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~
		b			



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

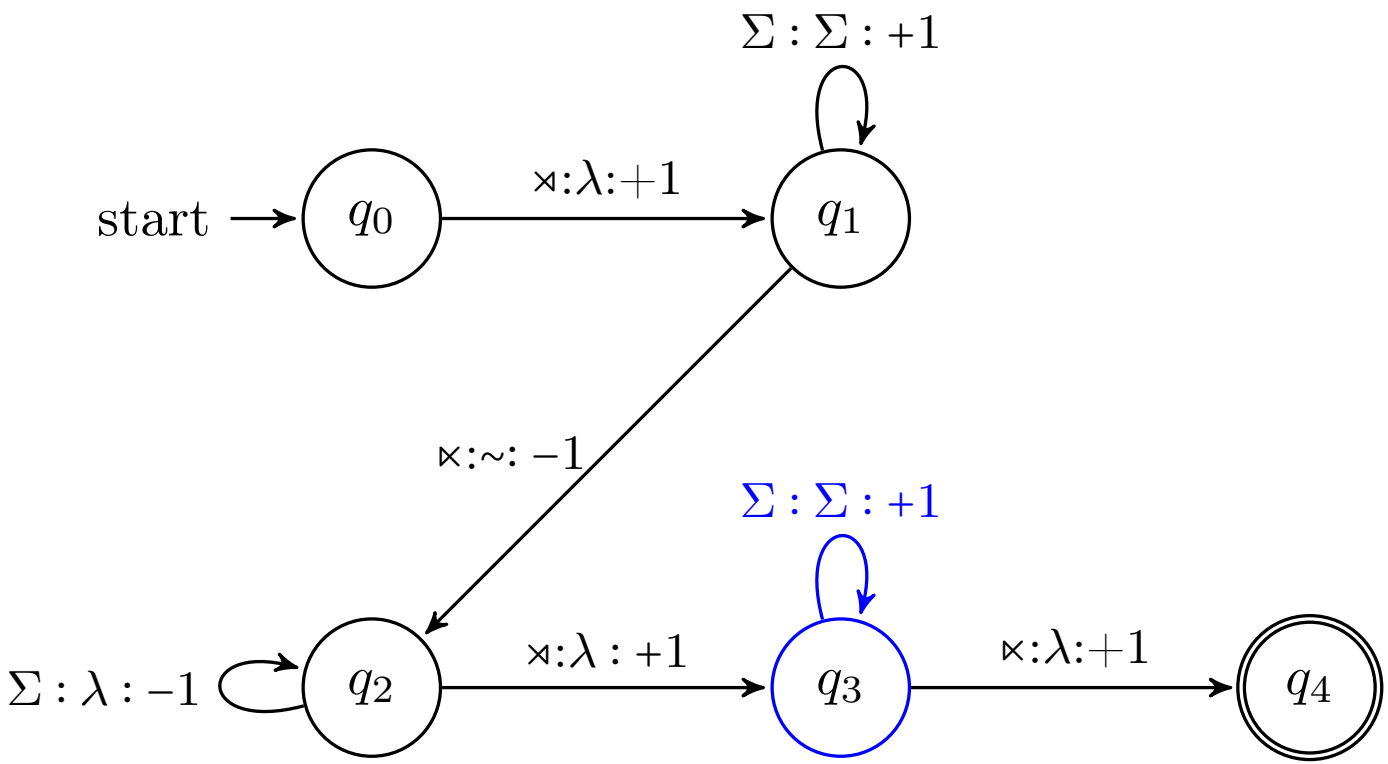
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~
		b	y		



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

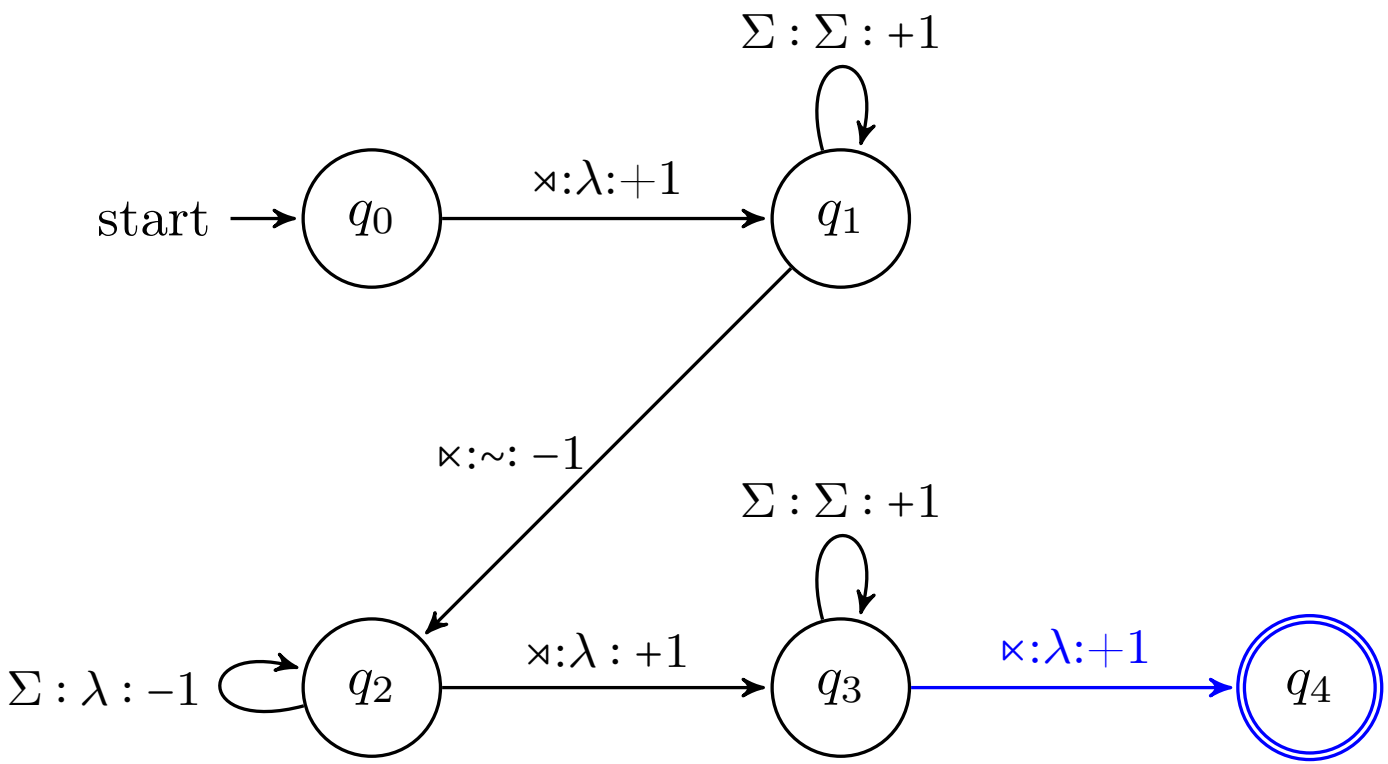
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~
		b	y	e	



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

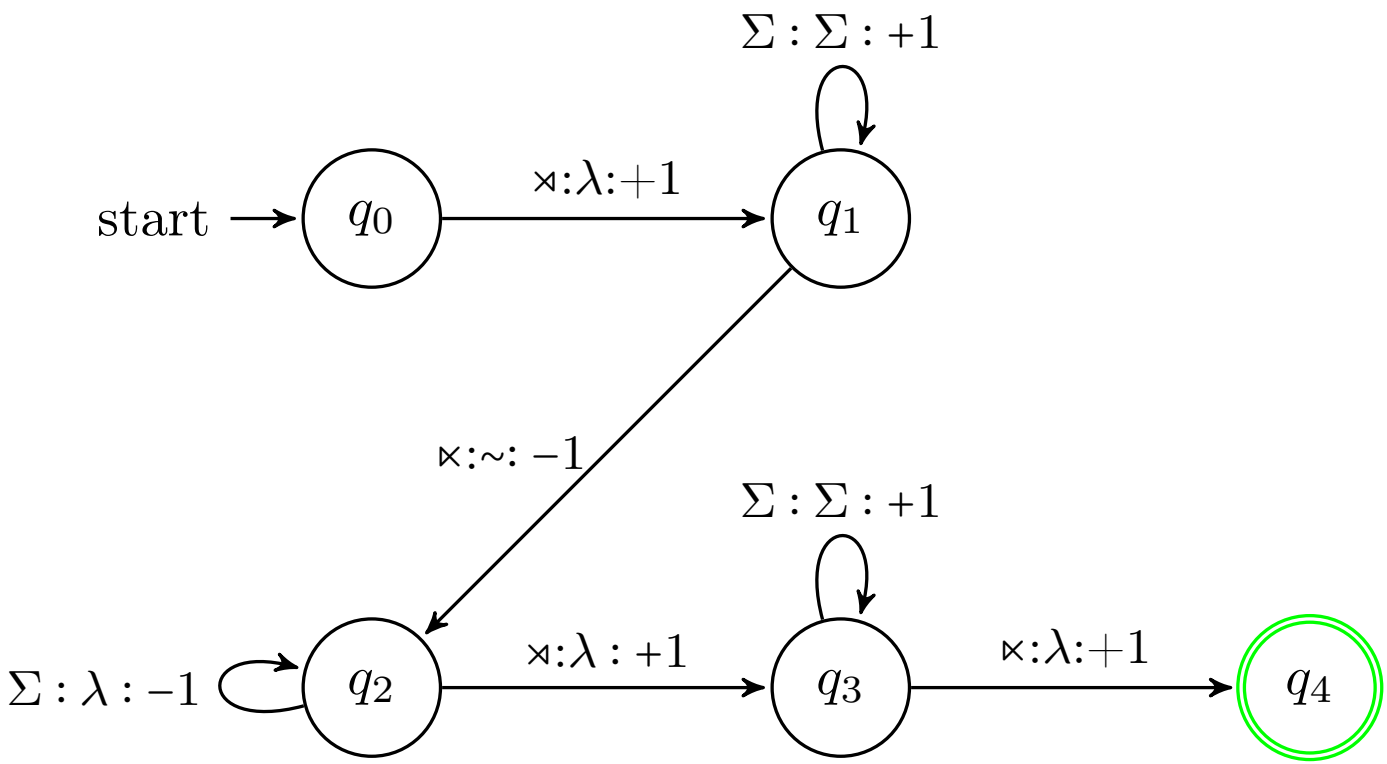
Input:	⋊	b	y	e	⋉
Output:		b	y	e	~
		b	y	e	



2-way FSTs - Total Reduplication

- Indonesian example: wanita→wanita~wanita
- Working example: bye→bye~bye

Input:	⋊	b	y	e	⋉
Output:		b	y	e	~
		b	y	e	



Observe

Total reduplication

$R(x) = f(x) \cdot g(x)$ where $f = g = id$.

2-way FSTs - Partial Reduplication

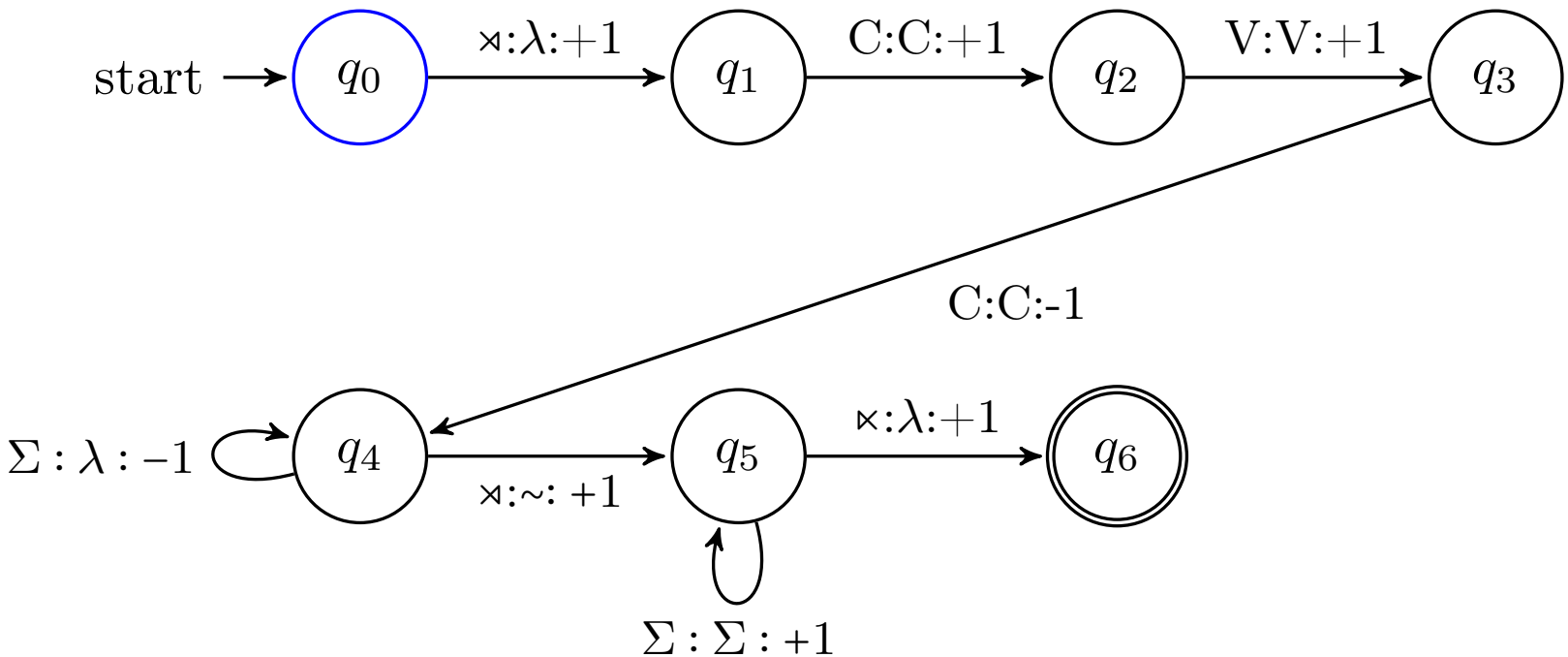
- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→?

2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

  Input: ⋊ c o p i e s ⋉

  Output:

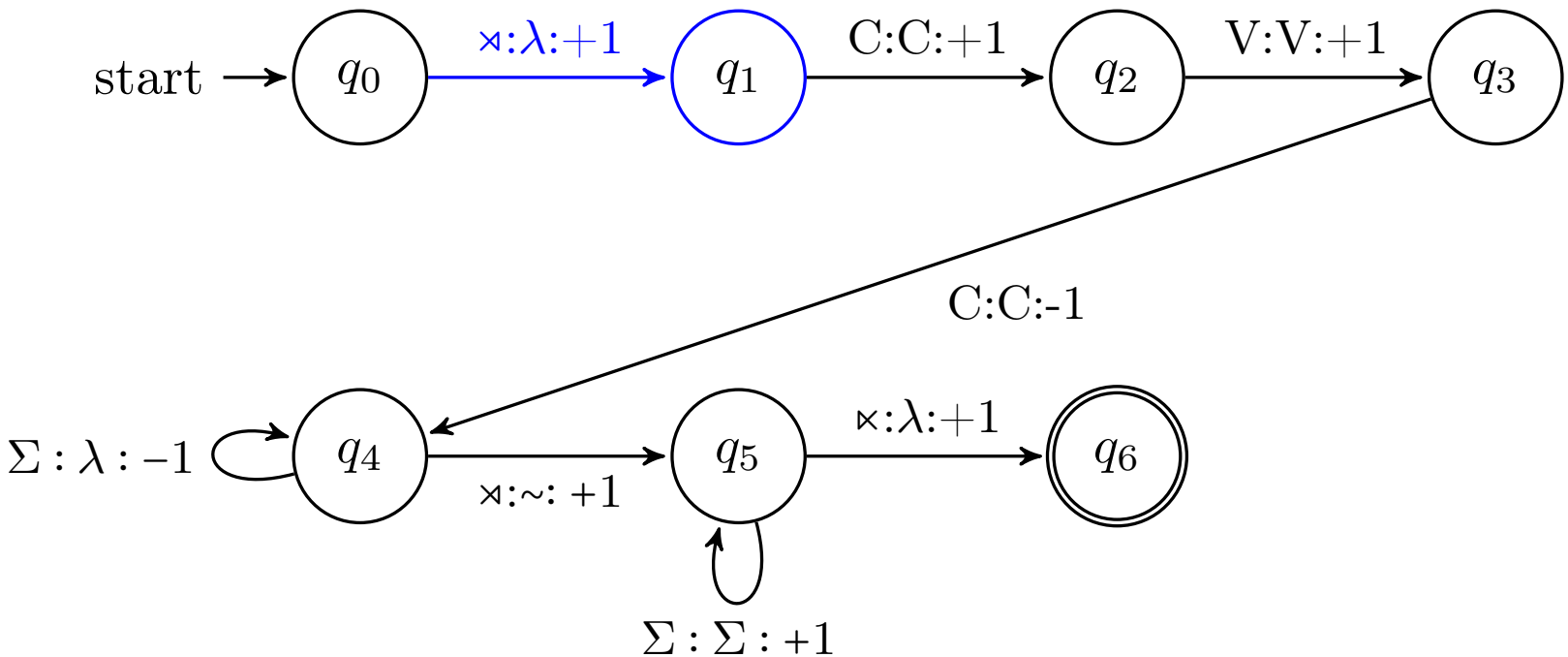


2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

  Input: ⋊ c o p i e s ⋉

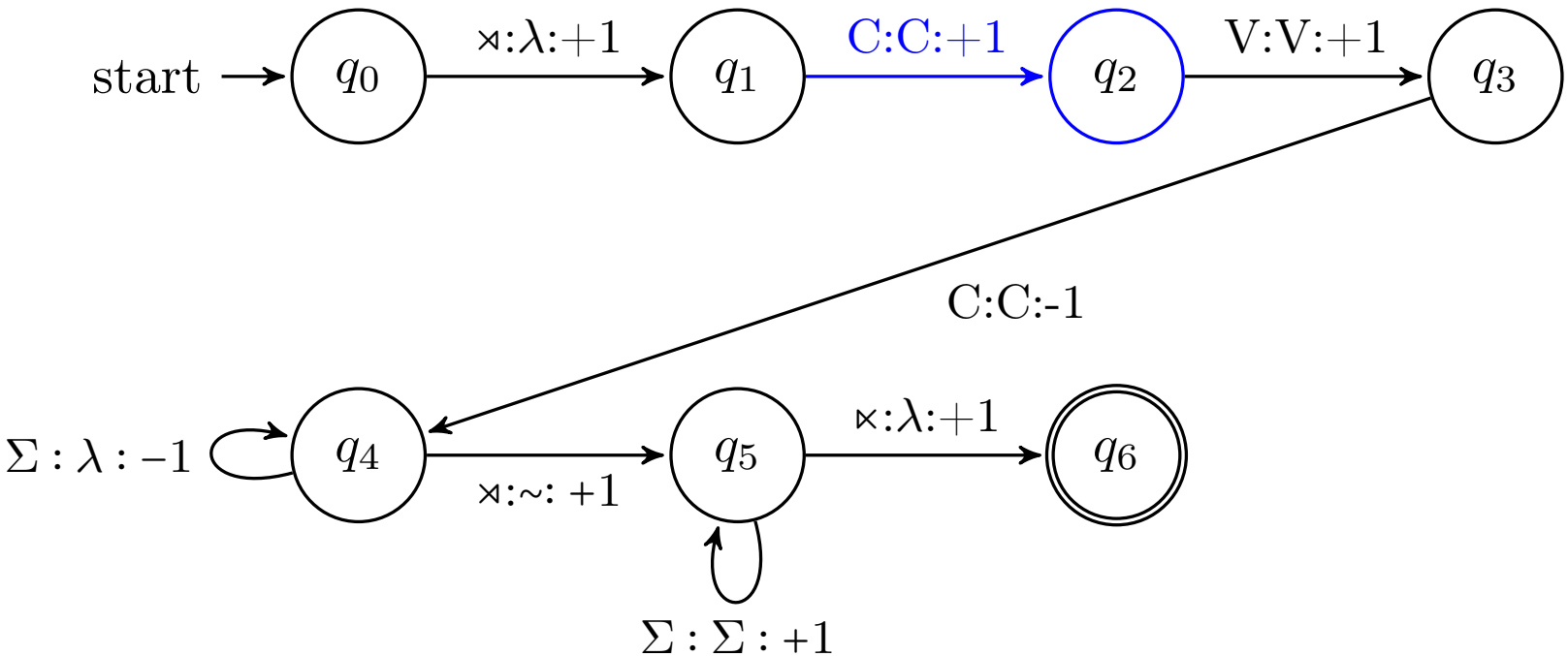
  Output:



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

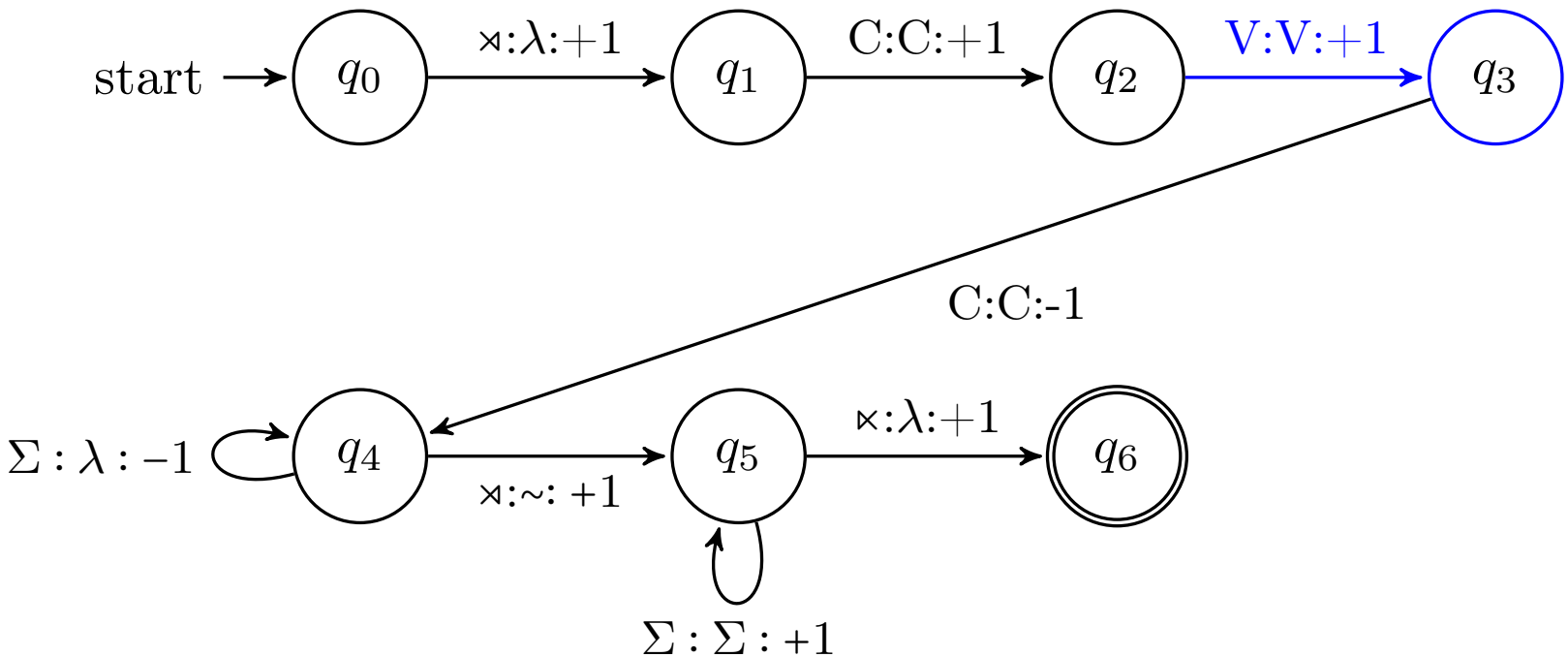
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c						



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

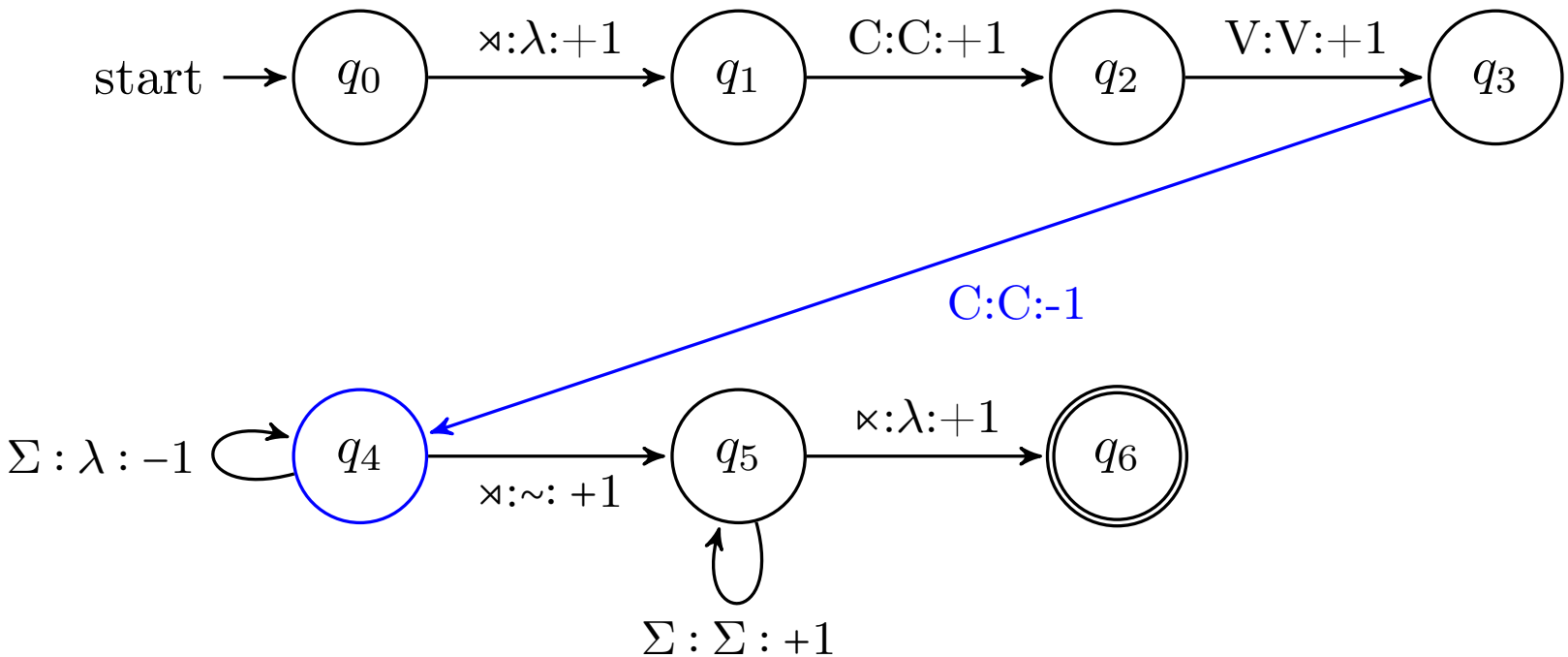
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o					



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

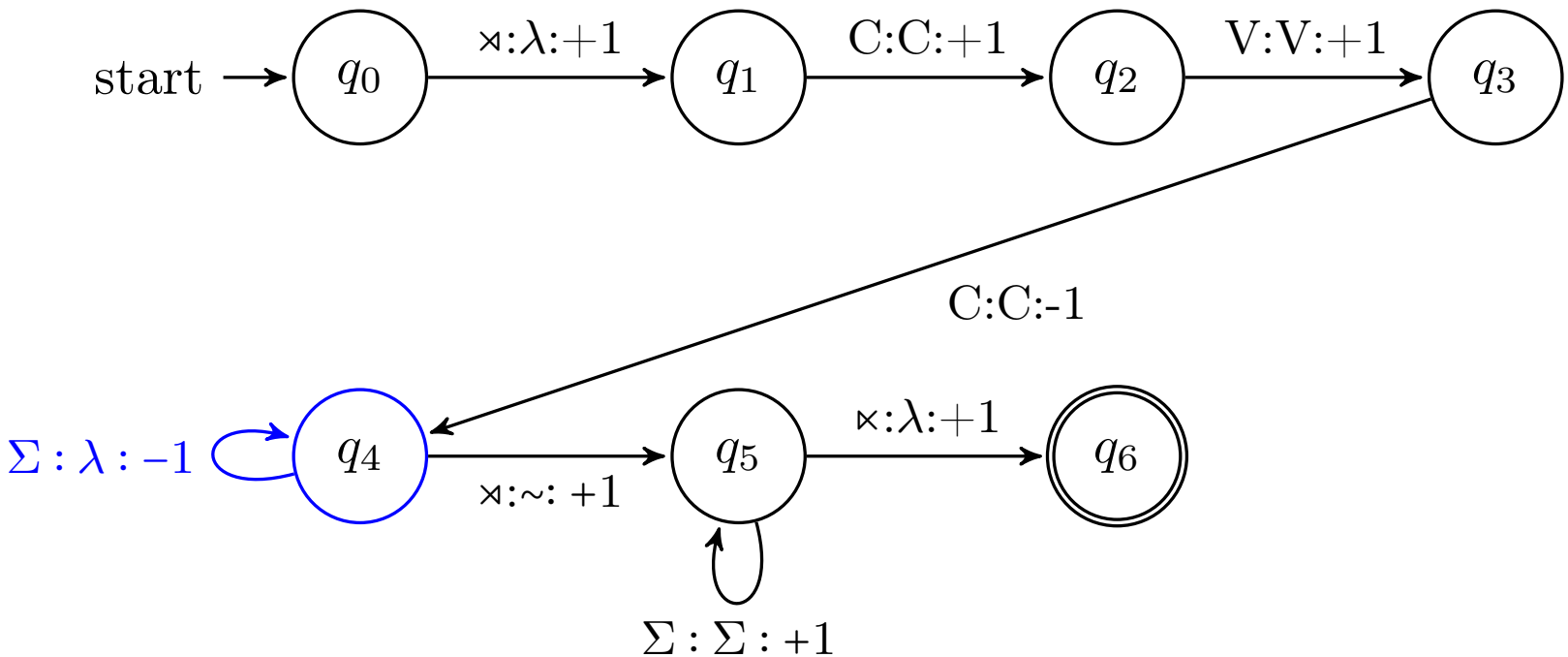
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

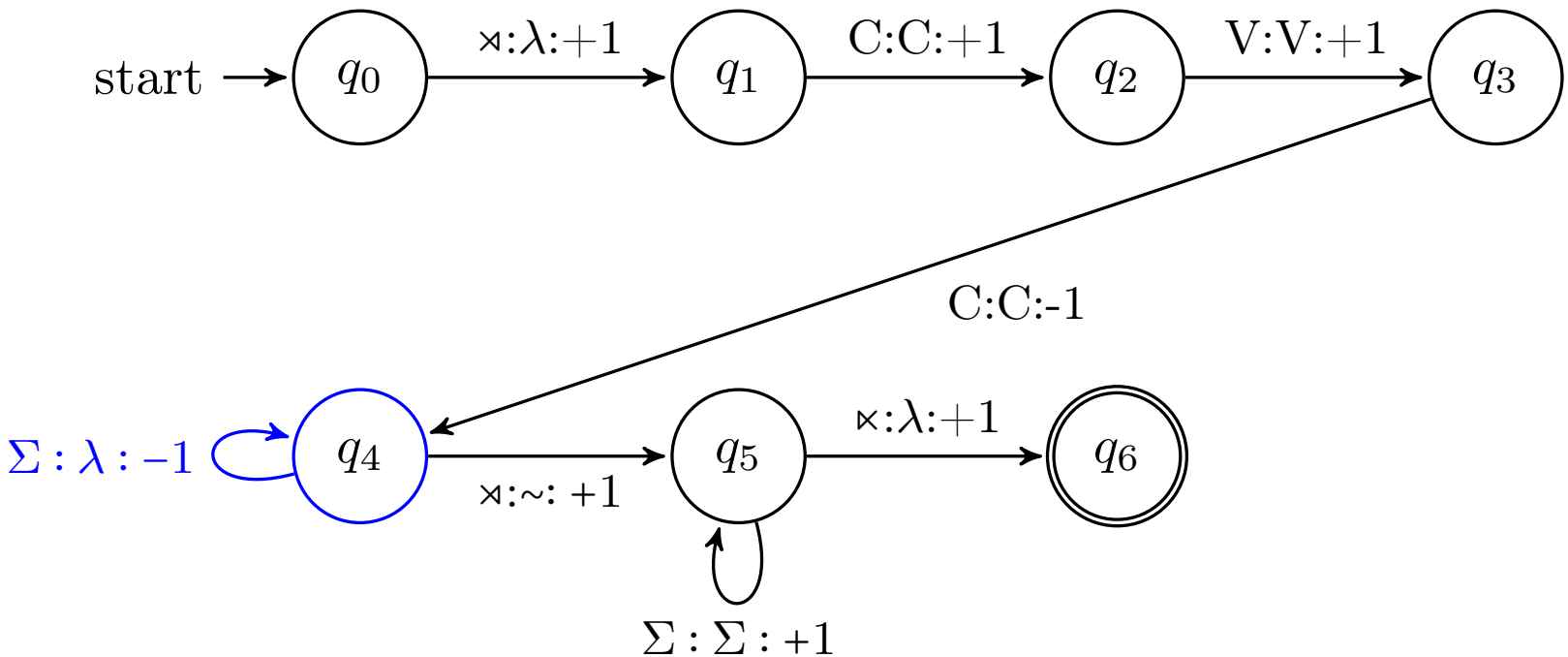
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

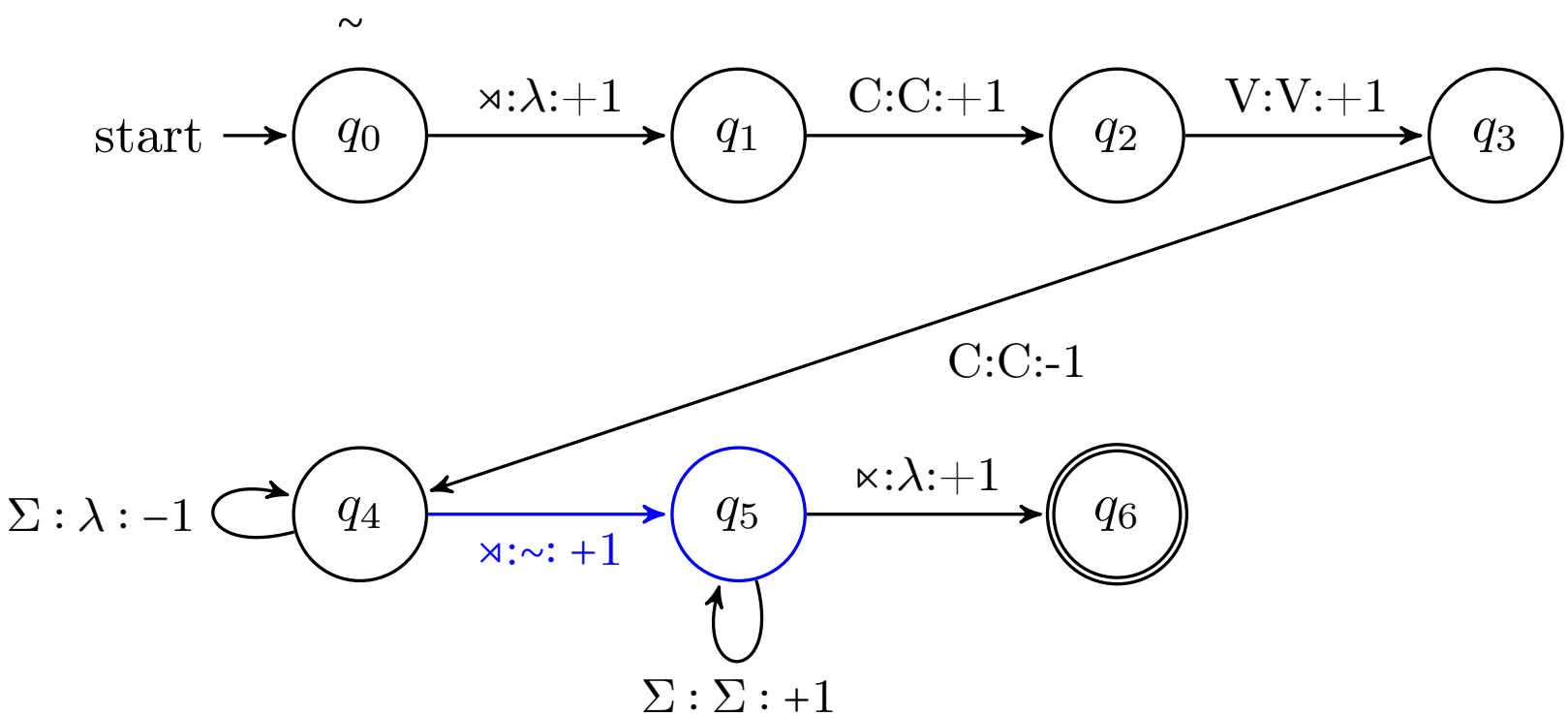
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

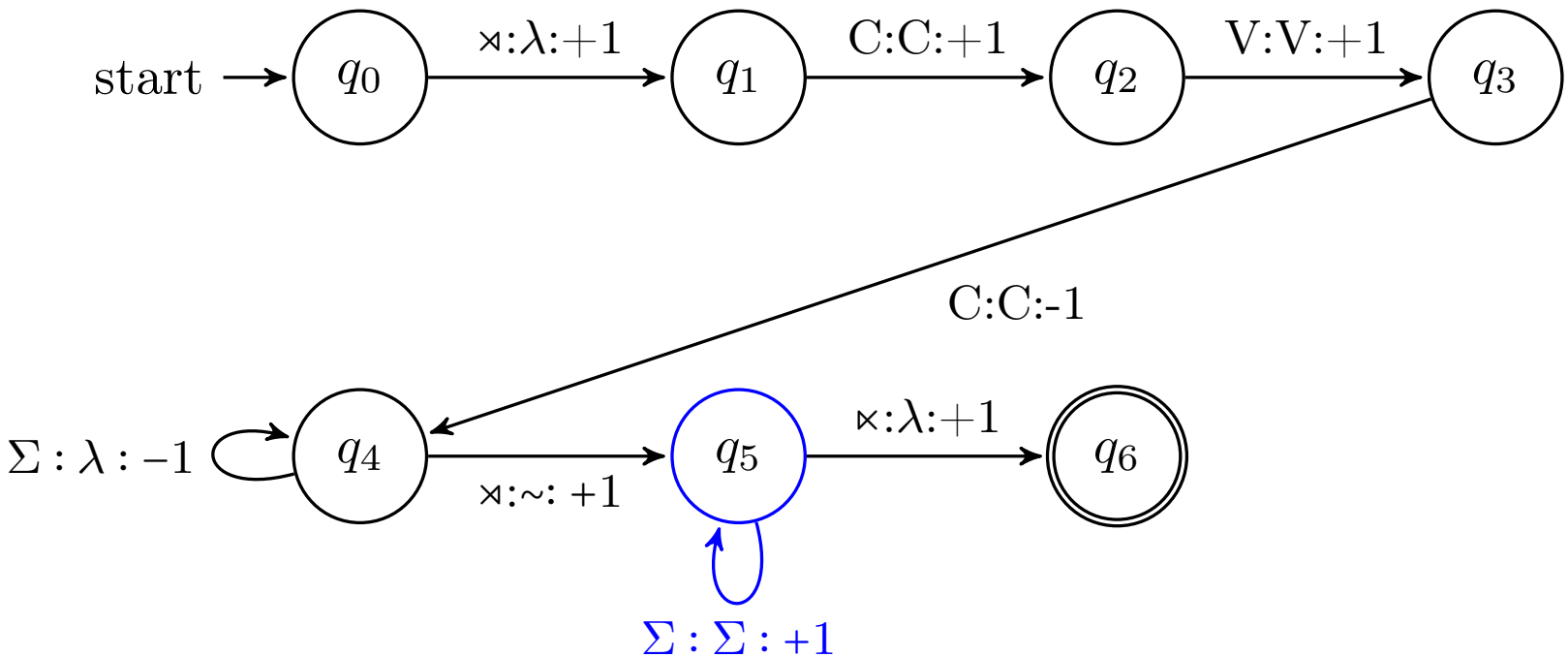
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

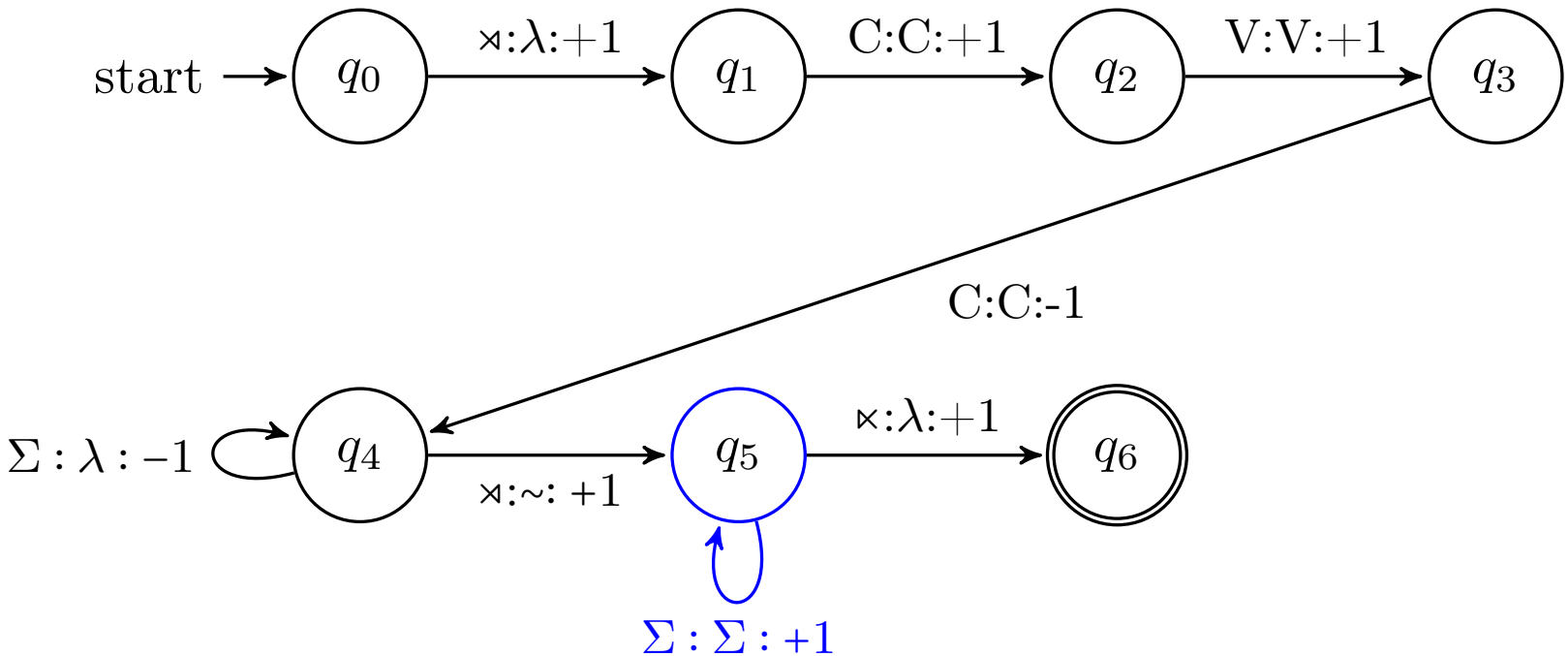
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c						



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

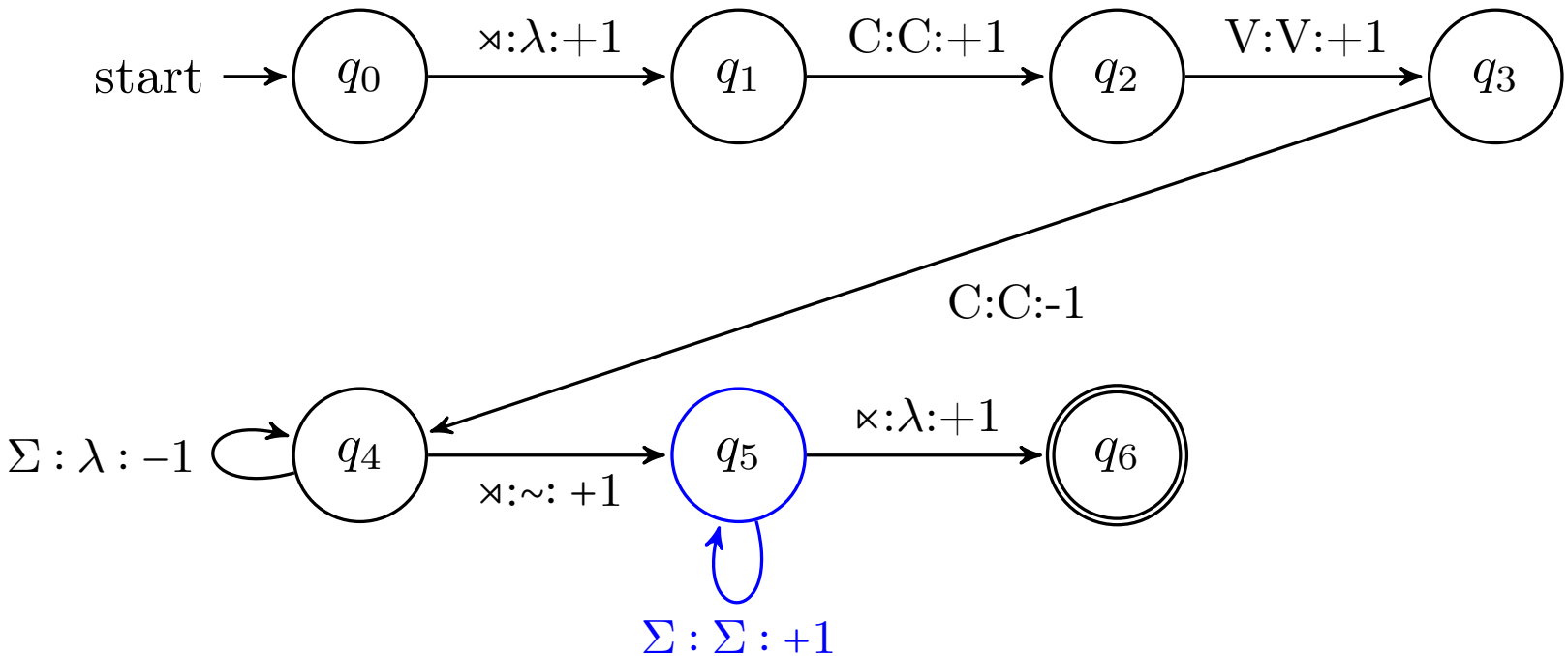
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c	o					



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

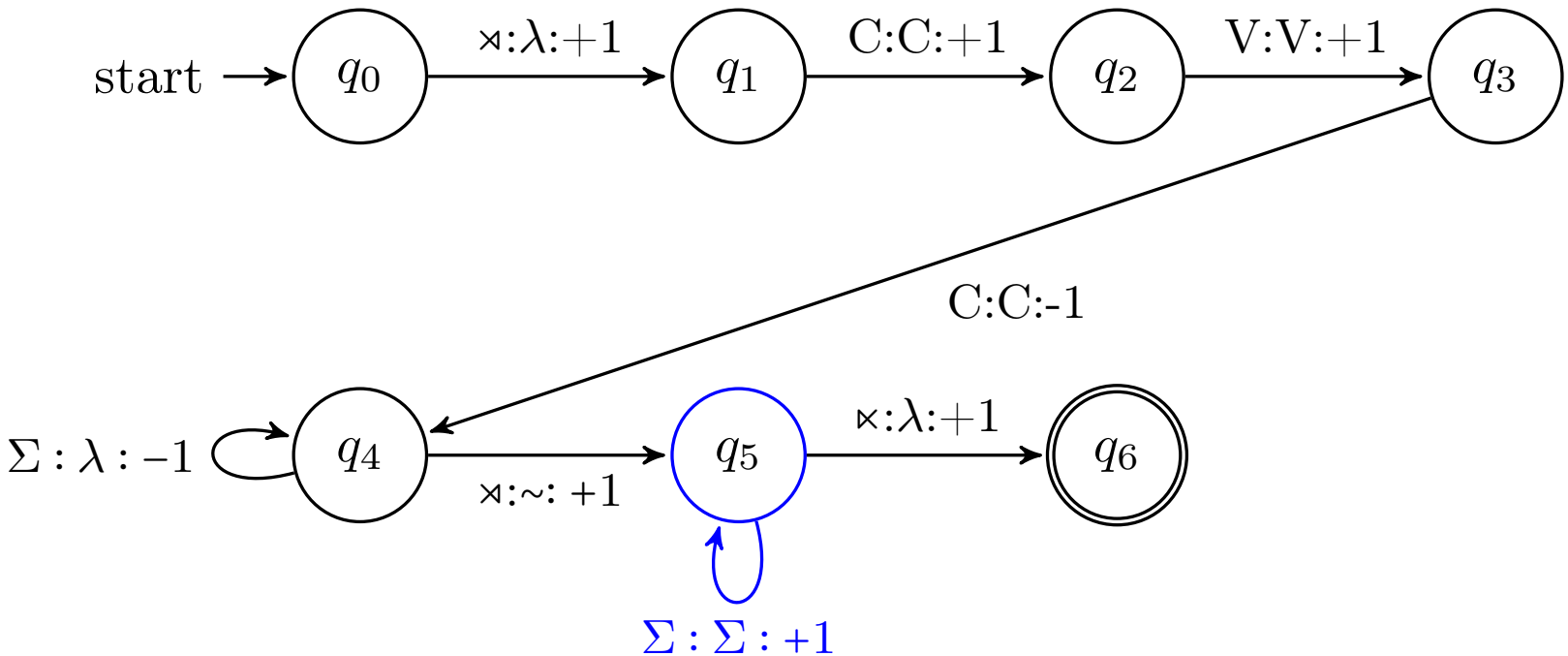
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c	o	p				



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

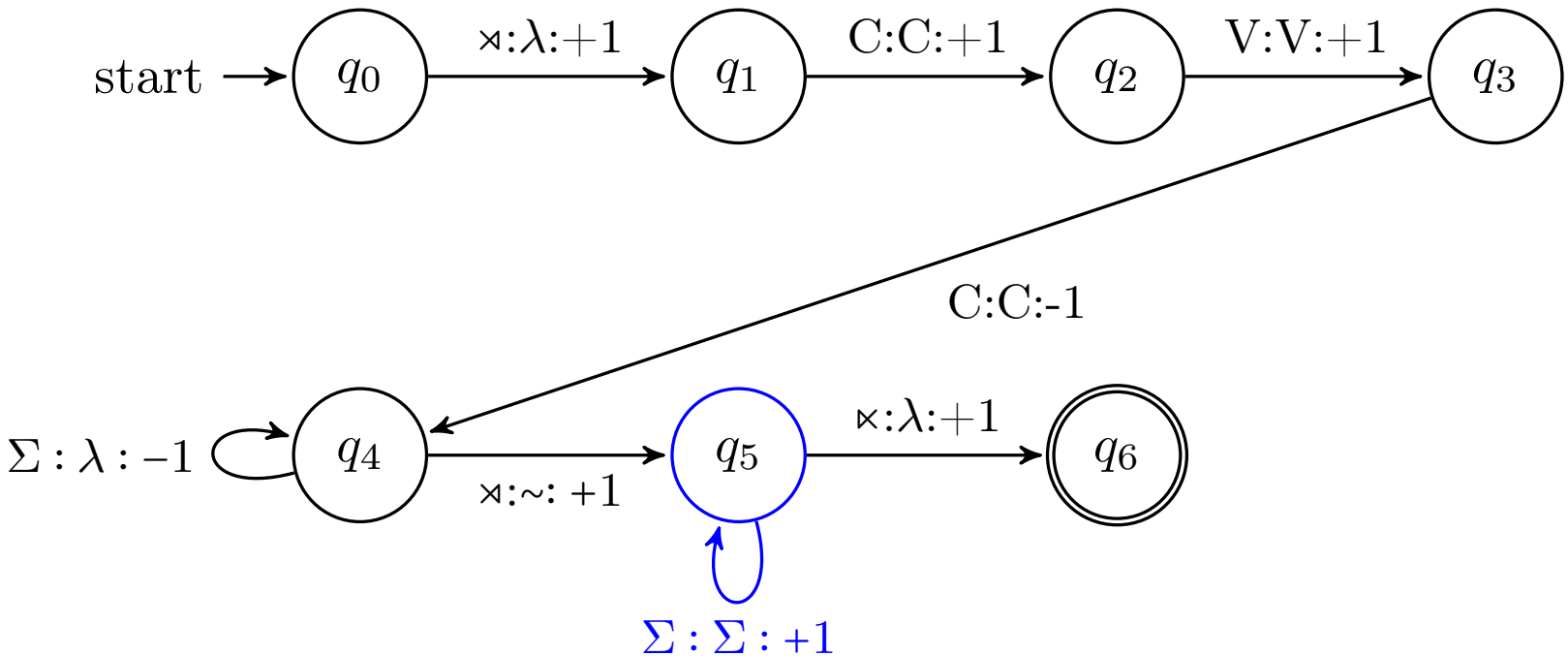
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c	o	p	i			



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

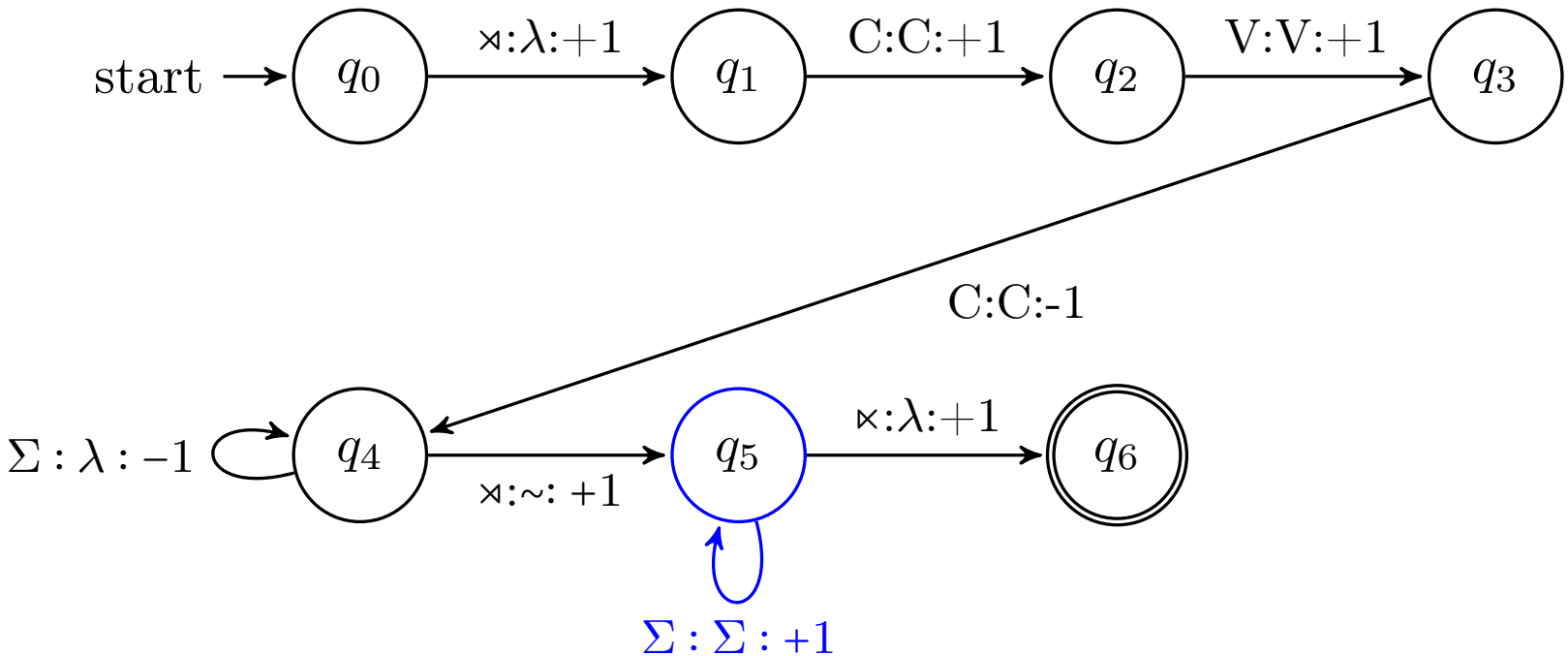
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c	o	p	i	e		



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

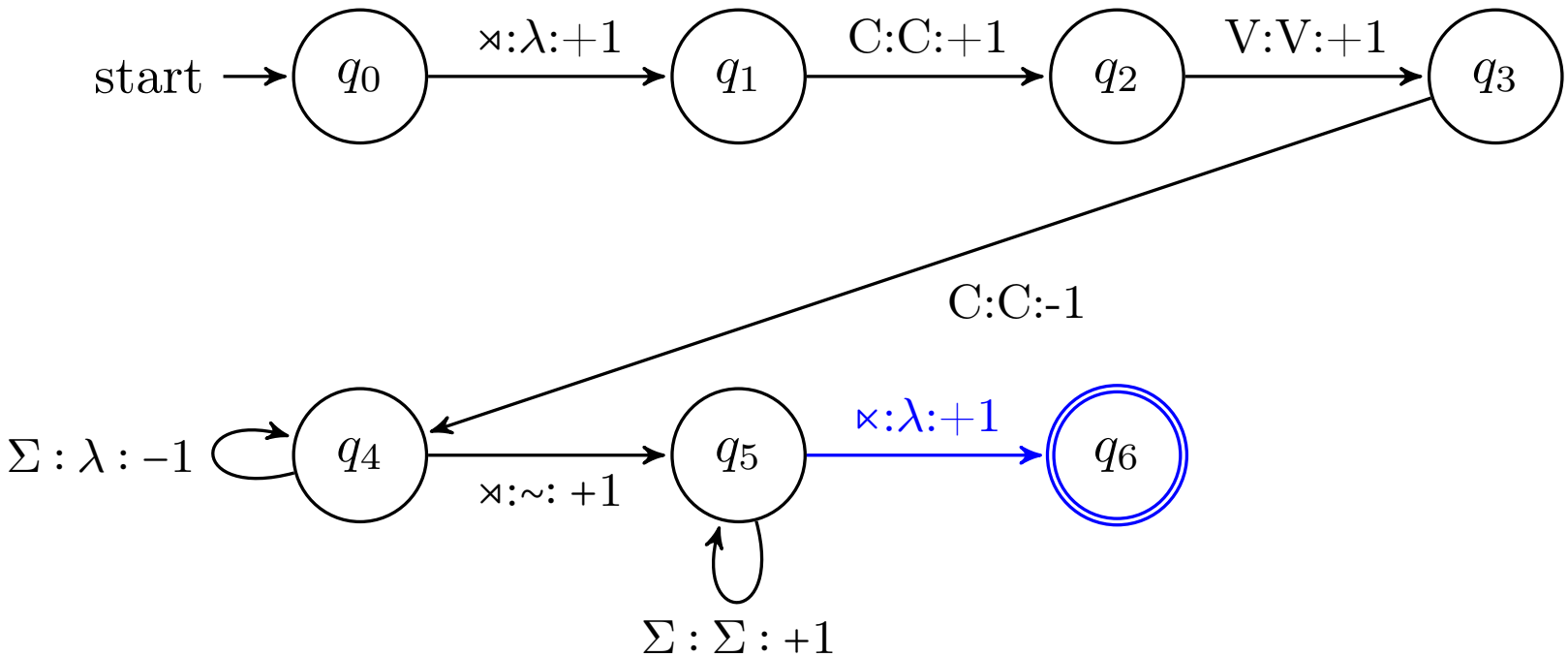
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p			
	~	c	o	p	i	e	s



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

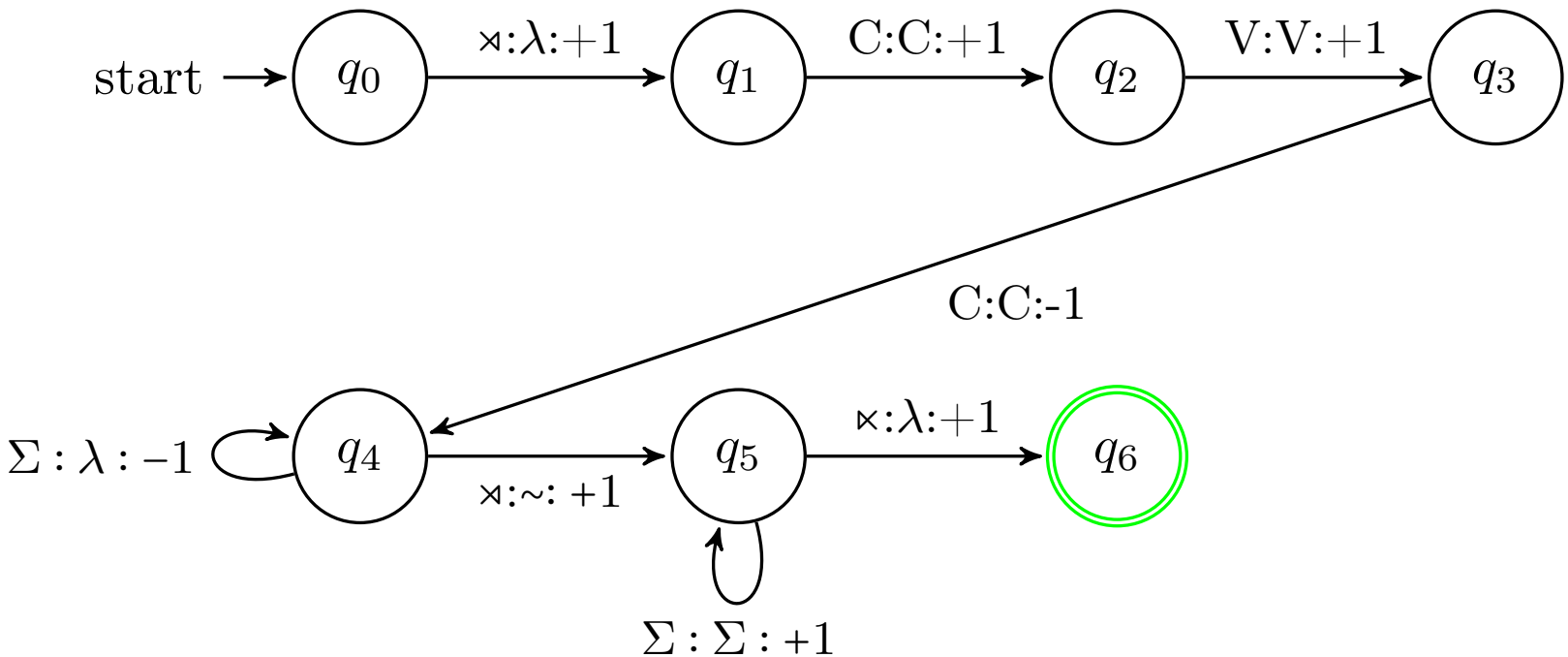
Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c	o	p	i	e	s	



2-way FSTs - Partial Reduplication

- Agta initial-CVC copying: takki→tak~takki
- Working example: copies→cop~copies

Input:	⋊	c	o	p	i	e	s	⋉
Output:		c	o	p				
	~	c	o	p	i	e	s	



Observe

Partial reduplication

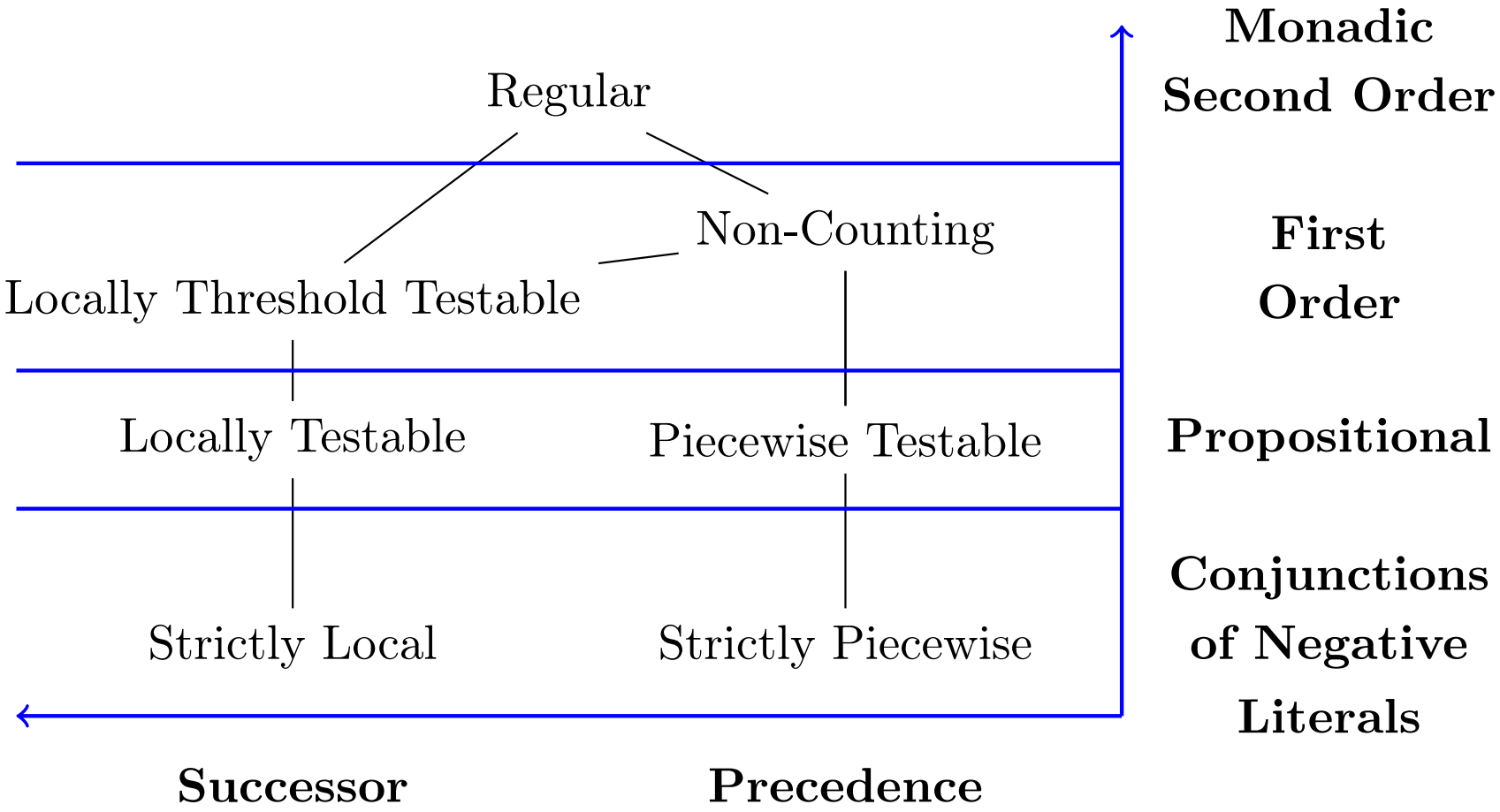
$R(x) = f(x) \cdot g(x)$ where f truncates x and $g = id$.

Building a computationally explicit typology of patterns

RedTyp

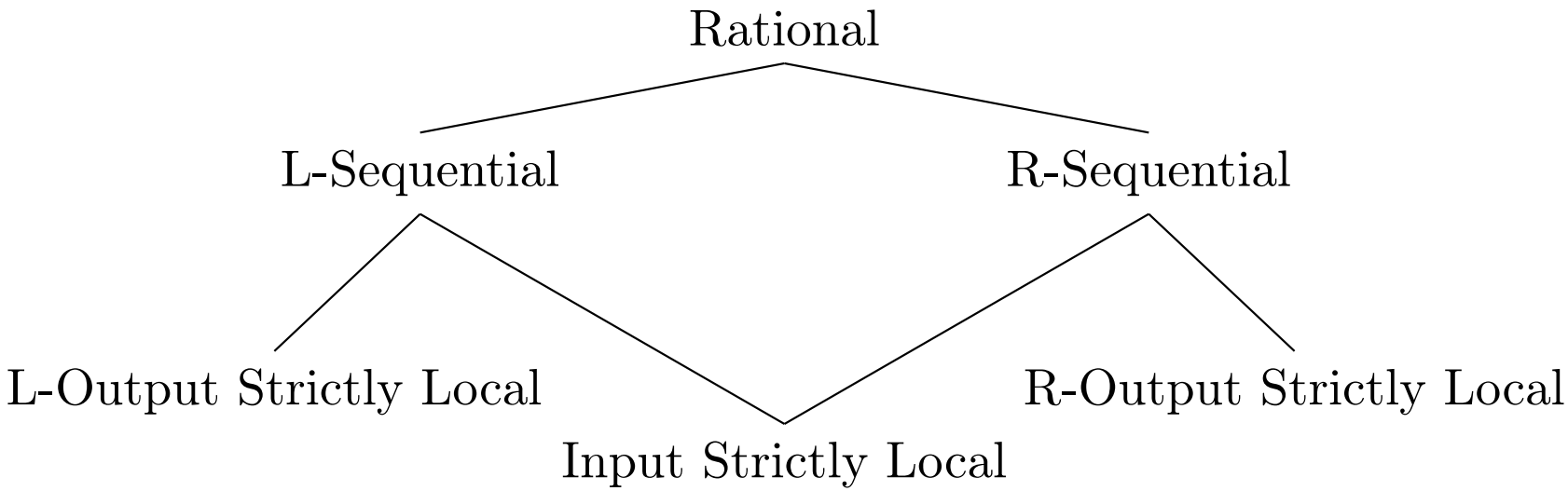
- SQL database of reduplicative processes + 2-way FST
- Modeled 138 reduplicative processes across 90 languages using 57 2-way FSTs
- Average # of states = 8.8
- Largest 2-way FST has 30 states (would be 1000s for a 1-way FST)
- `https://github.com/jhdeov/RedTyp`

Encyclopedia of Categories
Regular languages



(McNaughton and Papert, 1971; Rogers and Pullum, 2011; Rogers et al., 2013)

Encyclopedia of Categories
Rational functions



(Chandlee et al., 2014, 2015)

Left OSL subclass

- The state of any FST computing a Left Output Strictly k-Local (k-LOSL) function depends only on the last *k-1* segments written to the output tape and the last symbol read on the input tape.
- k-LOSL functions are identifiable in quadratic time and data with k-OSLFIA.

(Chandlee et al., 2015)

Truncation belongs to LOSL

- English nicknames: truncate name to first (C)VC

(9)

a.	/dʒɛfɹi/	→ [dʒɛf]	'Jeffrey'→'Jeff'
b.	/deɪvɪd/	→ [deɪv]	'David'→'Dave'
c.	/ælən/	→ [æl]	'Alan→Al'

- 3-OSL because keep track of last 2 segment outputted + the current segment (and skip anything after the first VC)

Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: ‘Samuel’ /sæmjəl/→?

Truncation belongs to LOSL

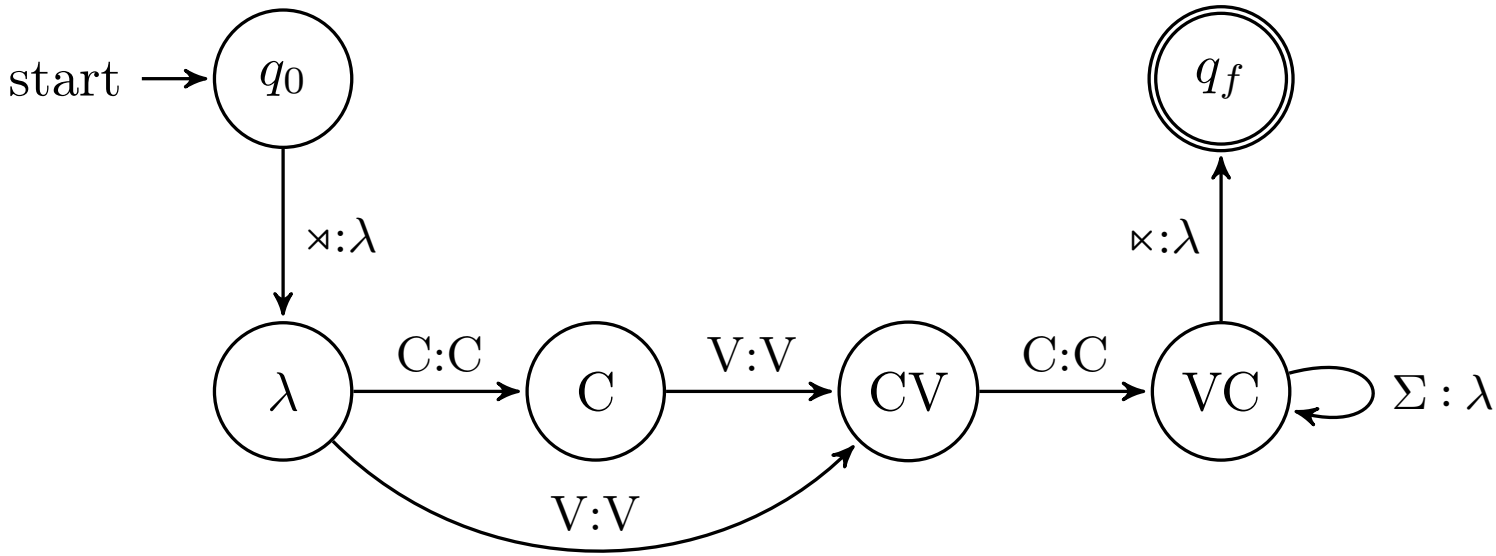
- English: /dʒɛfri/→[dʒɛf]
- Working example: ‘Samuel’ /sæmjəl/→[sæm]

Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

Input: ⋊ s æ m j ə l ⋉

Output:

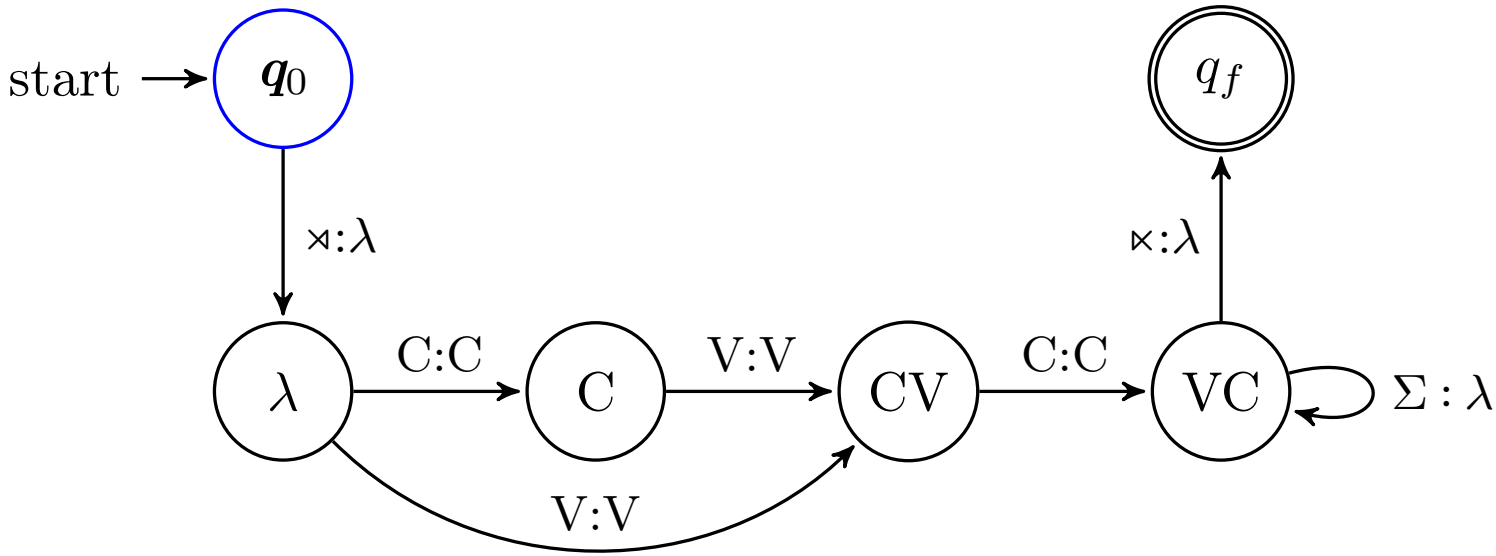


Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

  Input: ⋊ s æ m j ə l ⋉

  Output:

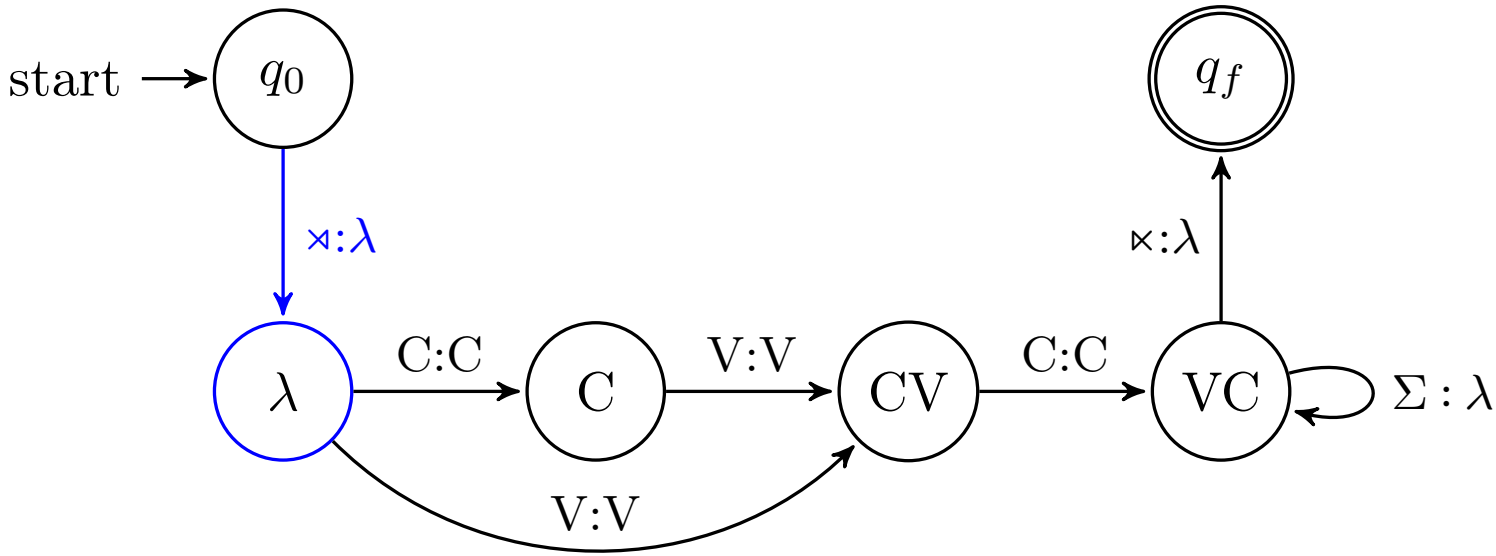


Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

Input: ⋊ s æ m j ə l ⋉

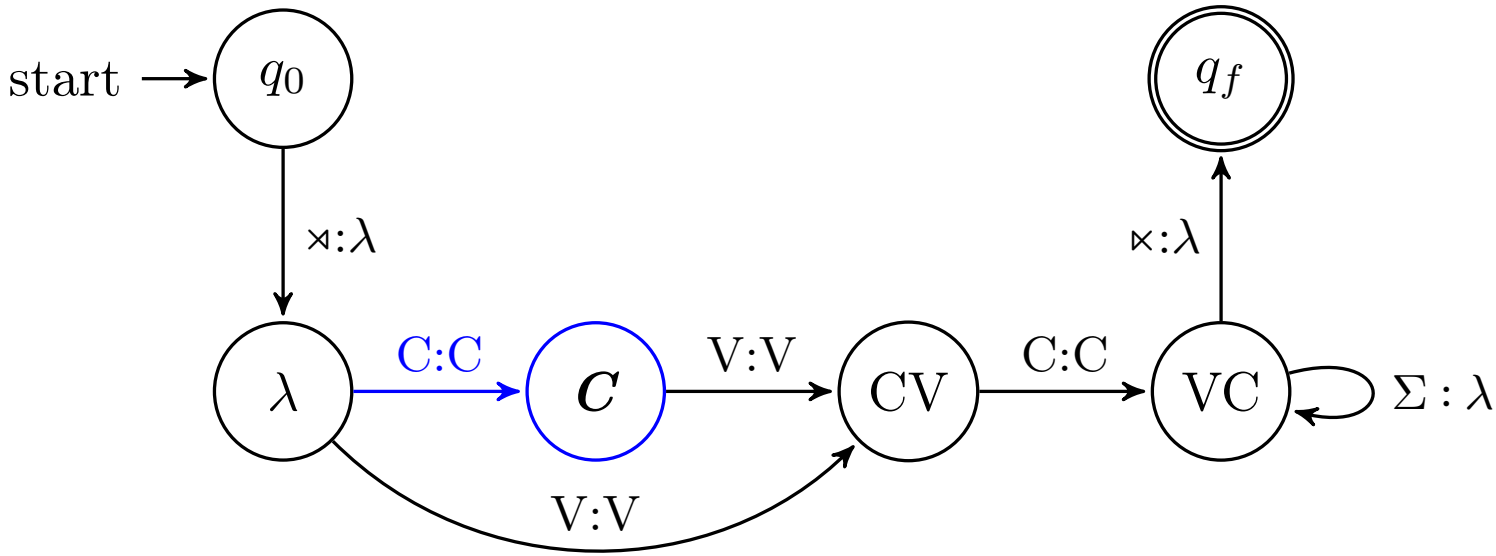
Output:



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

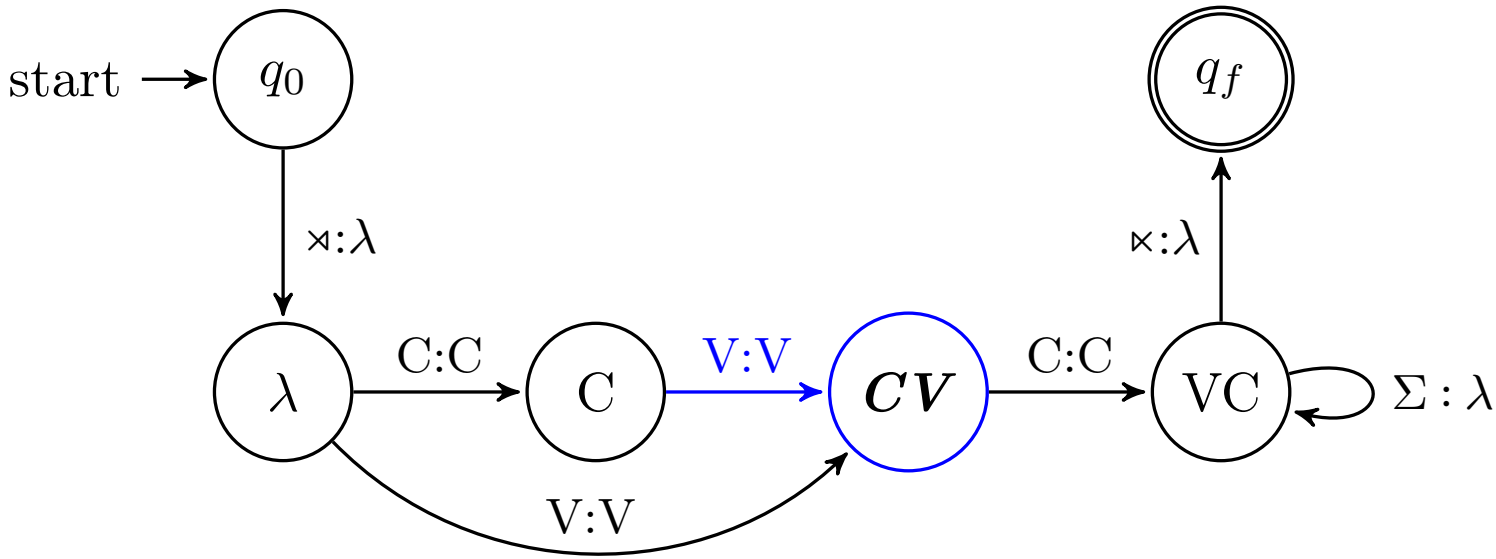
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s						



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

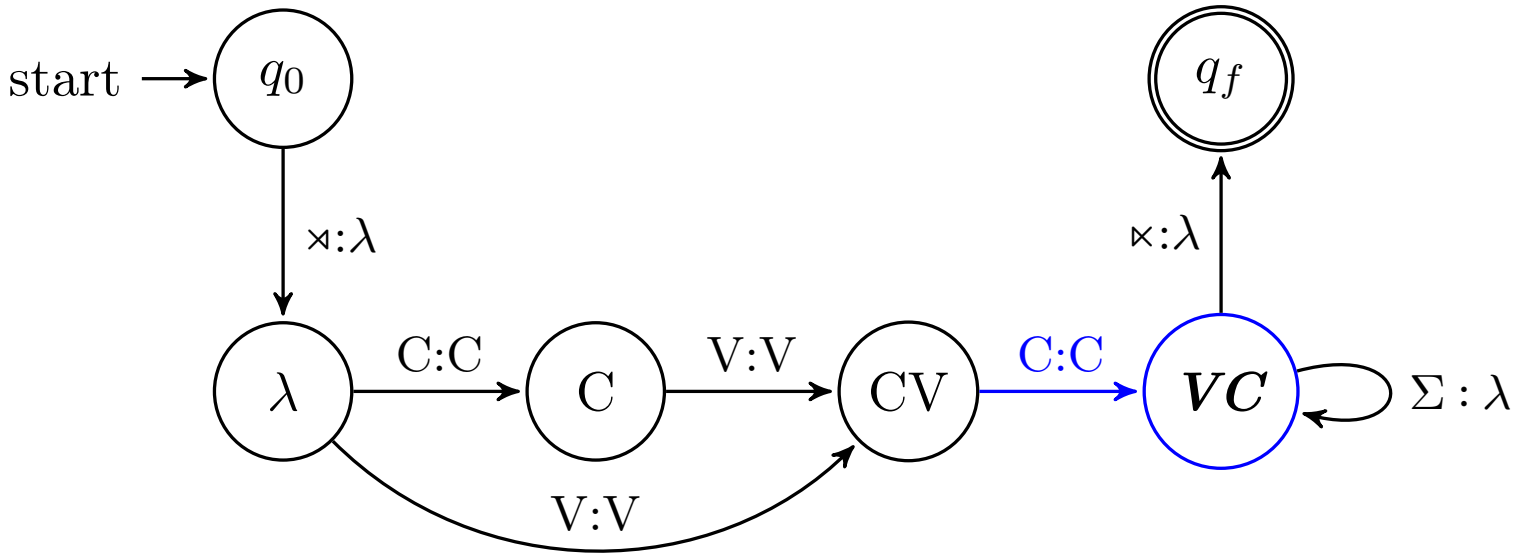
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ					



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

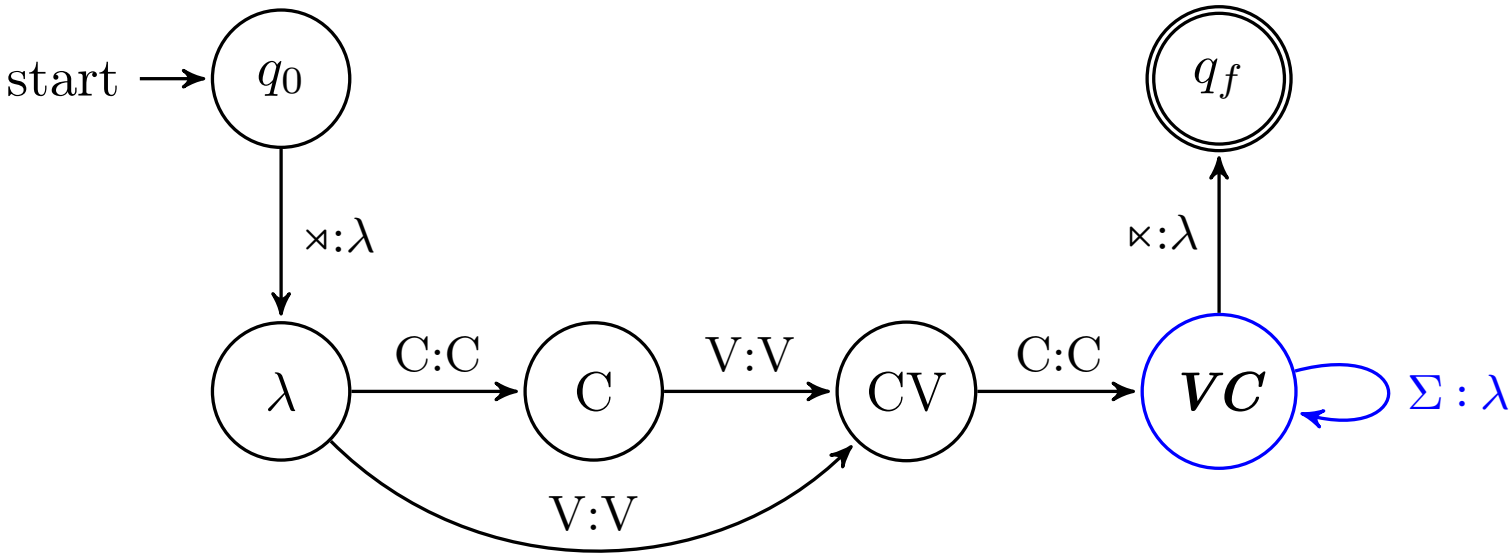
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ	m				



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

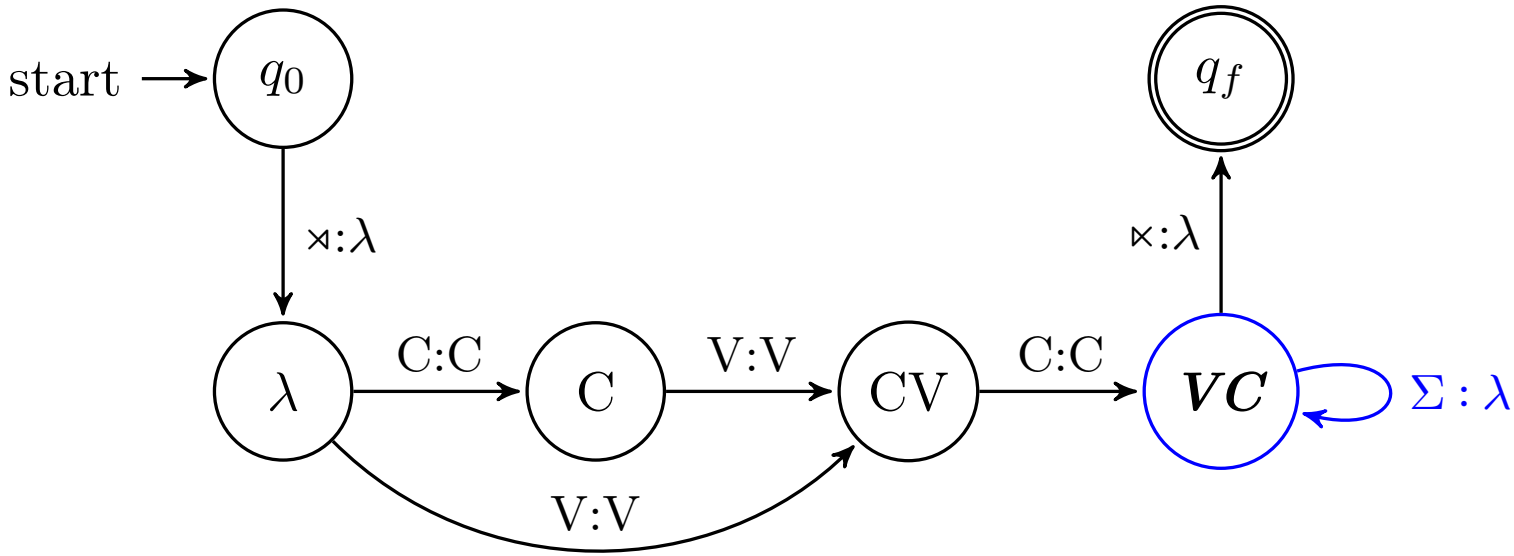
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ	m				



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

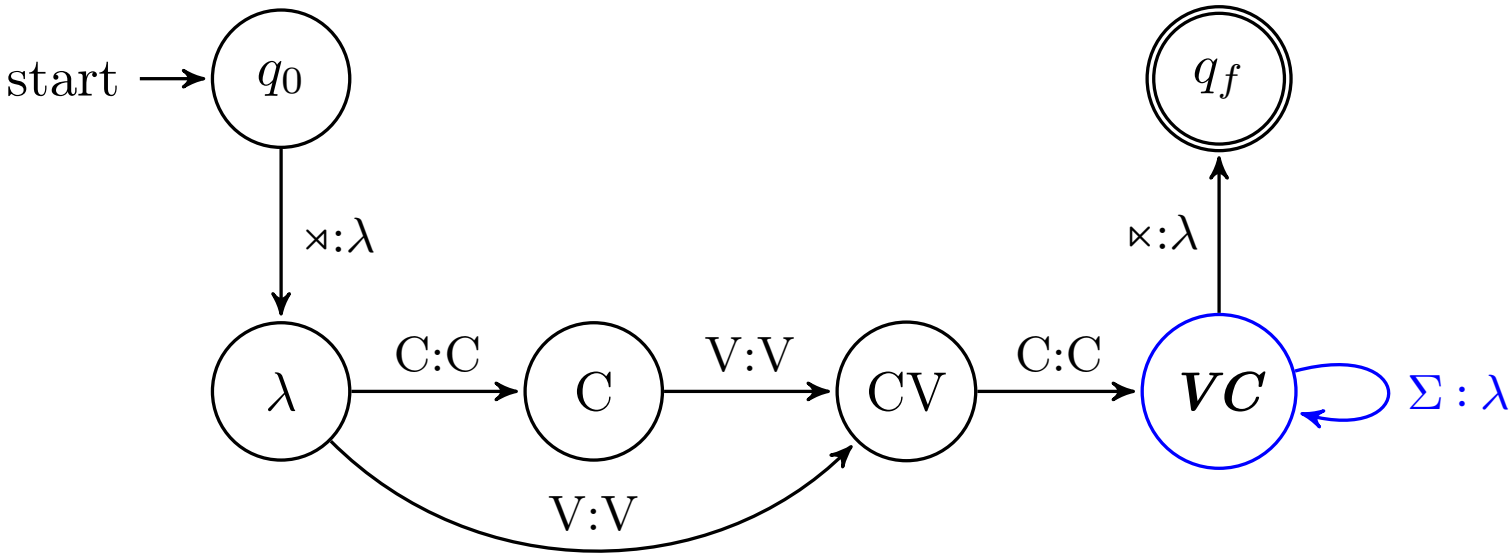
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ	m				



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

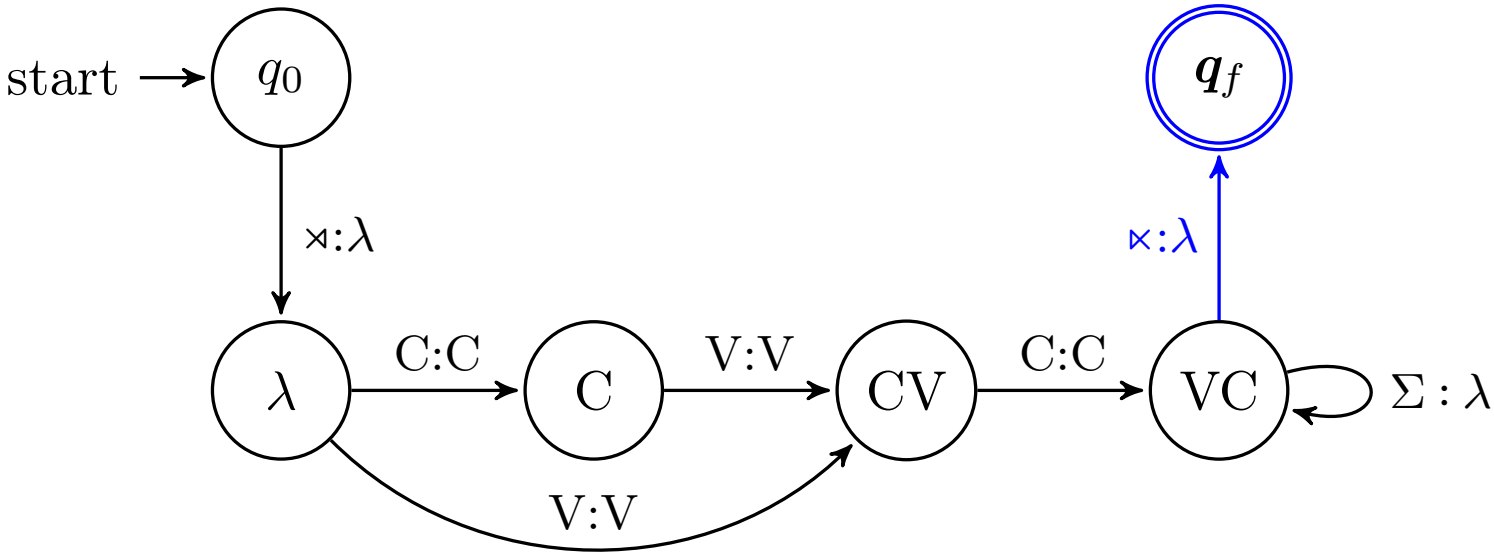
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ	m				



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

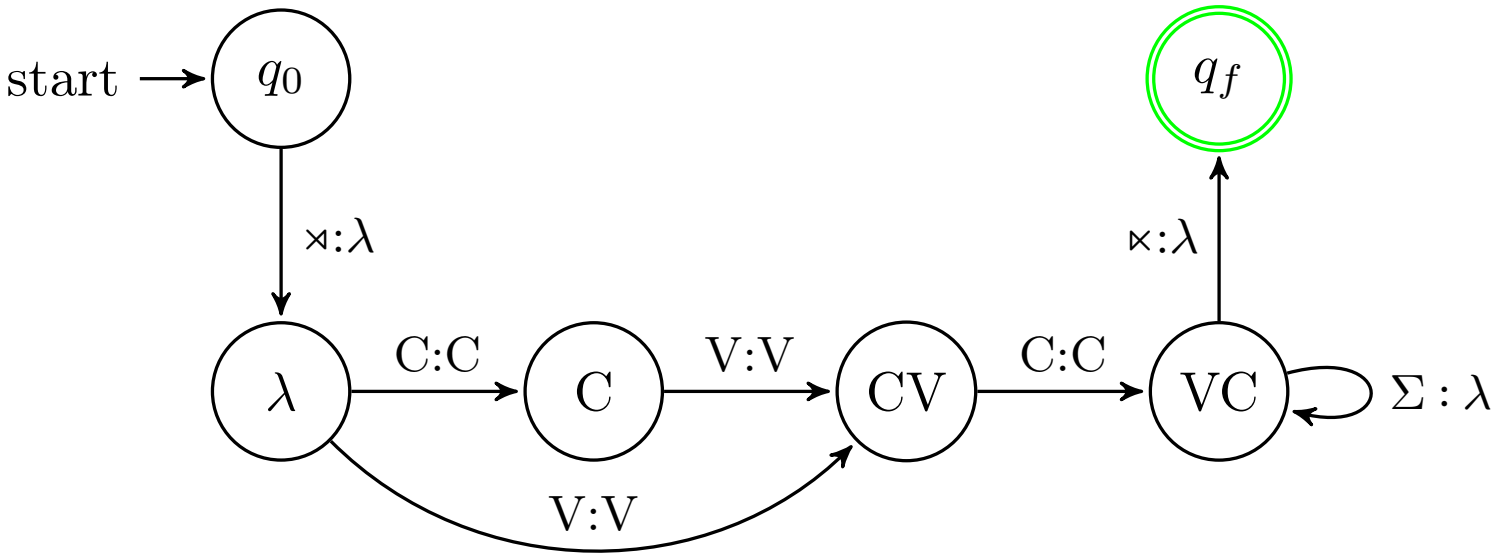
Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ	m				



Truncation belongs to LOSL

- English: /dʒɛfri/→[dʒɛf]
- Working example: 'Samuel' /sæmjəl/→[sæm]

Input:	⋊	s	æ	m	j	ə	l	⋉
Output:		s	æ	m				



Observe

Partial reduplication

$R(x) = f(x) \cdot g(x)$ where f truncates x and $g = id$.

Both f and g are 1-way LOSL functions!

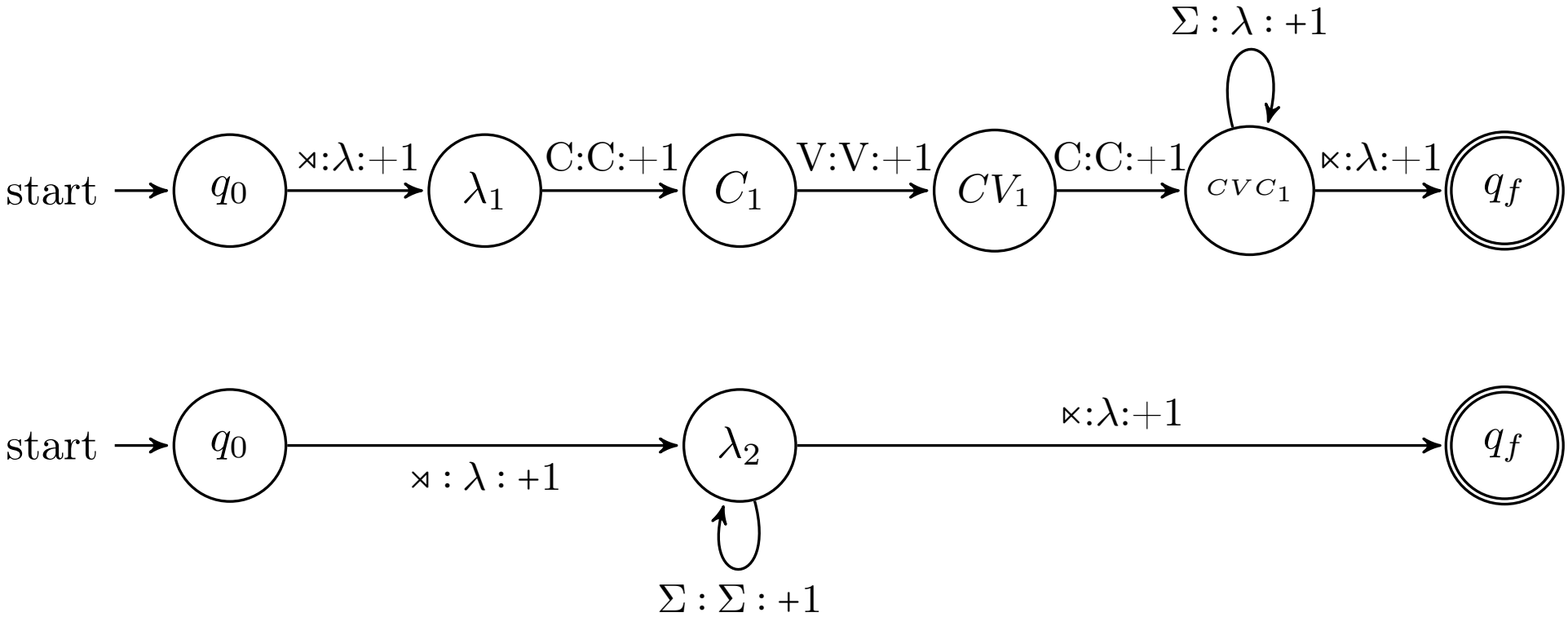
Defining C-LOSL

A function R is C-k-LOSL iff $R(x) = f(x) \cdot g(x)$ & both f, g are k-LOSL.

- Right C-OSL can be similarly defined
- Can be generalized to the concatenation of n 1-way functions.
- Can define a class of C-Sequential functions in this way and so on.

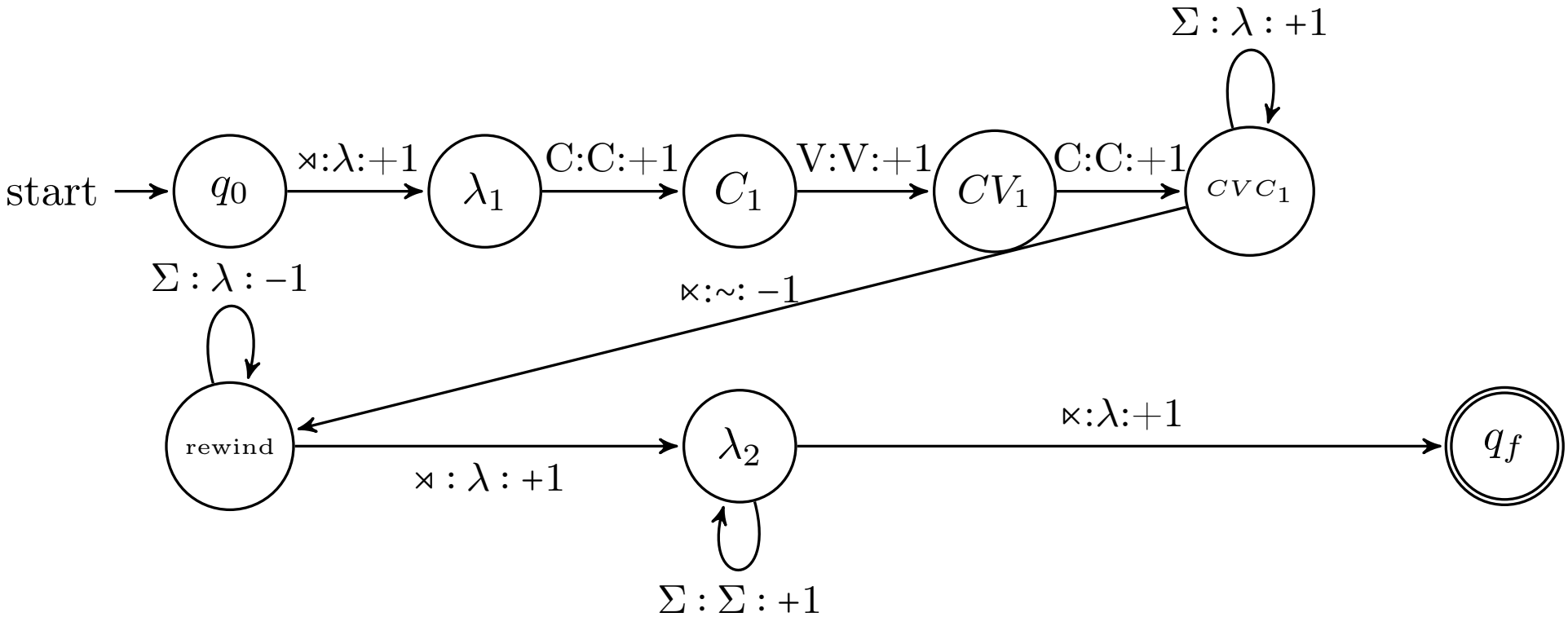
Concatenating 1-way FSTs

Left OSL 1-way FSTs for Trunc(x) and ID(x)



Make C-OSL by concatenation

Concatenating 1-way FSTs



What's C-OSL

- 87% of RedTyp is C-OSL.

(10) Total reduplication
wanita→wanita~wanita (Indonesian)

(11) Partial reduplication

a. CV: guyon→gu~guyon (Sundanese)
'to jest'→'to jest repeatedly'

b. CVC: takki→ tak~takki (Agta)
'leg'→'legs'

c. CVCV: banaganu→bana~banaganu (Dyirbal)
'return'

- The other ~13% appear to require:
  1. Concatenation of *Sequential* functions.
  2. *Compositions* of OSL or Sequential functions.
- More in appendix

Learning C-OSL

- Learning k-C-OSL can be reduced to learning the 1-way k-OSL functions
  1. If the boundary is overtly marked, then this is straightforward.
  2. if the boundary is not overtly marked then the learning reduces to finding this boundary (still an open problem).
- Reducing learning C-OSL to learning OSL modularizes learning and and builds on pre-existing results in GI.

BB-COSLIA - learning with boundaries

BB-COSLIA: Boundary-Based C-OSL learner

- Input: boundary-enriched data sample S and a positive integer k
  - Example $S = \{$(cat, cat~cat), (bird, bir~bird), (music, mus~music) $\ldots\}$
- Algorithm:
  1. Break up S into two data sets
     $H1 = \{(w, u) \mid (w, u \sim v) \in S\}$ and
     $H2 = \{(w, v) \mid (w, u \sim v) \in S\}$
  2. Submit H1 and H2 and k to OSLFIA which outputs 1-way transducers T1 and T2.
  3. Concatenate T1 and T2.

BB-COSLIA Discussion

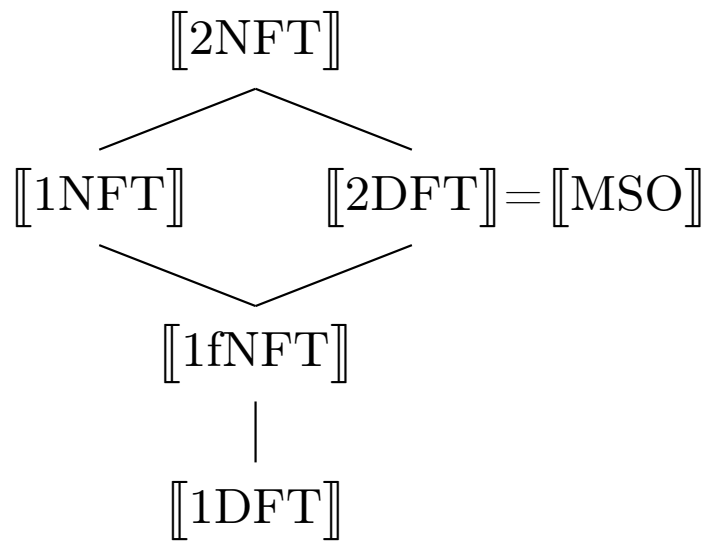
1. BB-COSLL provably learns C-OSL functions iff $H1, H2$ are characteristic samples for $T1, T2$ respectively.
2. What about the boundary?
   - BB-COSLL depends on the boundary ~ in the input.
   - The boundary is an overt manifestation of the concatenation in the "derivation" of the reduplicated form.
   - Potential phonological evidence for the boundary (appendix)
   - If this boundary is unknown, learning reduces to morpheme segmentation, which is an open problem.

(Goldsmith et al., 2017)

2FST Discussion

- Reduplication is widespread in natural languages.
- It has generally been considered beyond finite-state analysis.
- However, it is amenable to an analysis with 2-way FSTs.
- How come 2way FSTs have been off the radar of computational linguists all this time???

  Maybe transducers are harder to study?
  - While $[\![\text{1DFA}]\!] = [\![\text{2DFA}]\!] = [\![\text{1NFA}]\!] = [\![\text{2NFA}]\!] = [\![\text{MSO}]\!]$,



  - Note 50 years passed between Büchi's theorem and Engelfriedt and Hoogeboom's.

(Filiot and Reynier, 2016)

Conclusions

- Contributions
  1. RedTyp database
  2. Show that 2-way FSTs can model virtually the entire typology of reduplication
  3. ~87% of this typology belongs to the C-OSL subclass.
  4. simple learning algorithm for C-OSL which builds off of OSLFIA but uses boundary-enriched sample.
- Future research:
  1. Adding to RedTyp.
  2. Studying the non-C-OSL transformations more carefully.
  3. Learning without boundaries.
  4. Studying the trade-off between 1-way vs. 2-way FSTs for learning partial reduplication.

Thank you
and
Thank you Wrocław [vrɔtswaf] !

References

Albro, D. M. (2005). Studies in Computational Optimality Theory, with Special Reference to the Phonological System of Malagasy. Ph. D. thesis, University of California, Los Angeles, Los Angeles.

Alur, R. (2010). Expressiveness of streaming string transducers. In Proceedings of the 30th Annual Conference on Foundations of Software Technology and Theoretical Computer Science,, Volume 8, pp. 1–12.

Beesley, K. R. and L. Karttunen (2003). Finite-state morphology: Xerox tools and techniques. CSLI Publications.

Bojańczyk, M. (2014). Transducers with origin information. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias (Eds.), Automata, Languages, and Programming, Berlin, Heidelberg, pp. 26–37. Springer.

Chandlee, J., R. Eyraud, and J. Heinz (2014). Learning strictly local subsequential functions. Transactions of the Association for Computational Linguistics 2, 491–503.

Chandlee, J., R. Eyraud, and J. Heinz (2015, July). Output strictly local functions. In Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015), Chicago, USA, pp. 112–125.

Cohen-Sygal, Y. and S. Wintner (2006). Finite-state registered automata for non-concatenative morphology. Computational