Regular Functions

Rajeev Alur
University of Pennsylvania

Joint work with
Pavol Cerny, Loris D'Antoni, Mukund Raghothman and others

DCFS, U. Waterloo, June 2015

Penn Engineering

Regular Languages

- Natural
  - Intuitive operational model of finite-state automata
- Robust
  - Alternative characterizations and closure properties
- Analyzable
  - Algorithms for emptiness, equivalence, minimization, learning …
- Applications
  - Algorithmic verification, text processing …

What is the analog of regularity for defining functions?

Do we really need such a concept ?

FlashFill: Programming by Examples

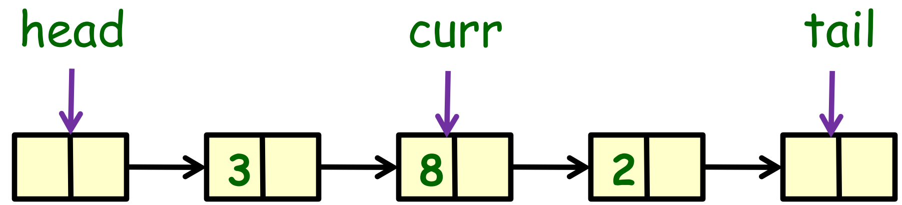
Ref: Gulwani (POPL 2011)

Input	Output
Shallit, Jeffrey	J. Shallit
Alexander Okhotin	A. Okhotin
Colcombet T.	T. Colcombet

- Infers desired Excel macro program
- Iterative: user gives examples and corrections
- Already incorporated in Microsoft Excel

Learning regular languages	:	L* (Angluin'92)
Learning string transformation	:	??

Verification of List-processing Programs




```
function delete
  input ref curr;
  input data v;
  output ref result;
  output bool flag := 0;
  local ref prev;

  while (curr != nil) & (curr.data = v) {
      curr := curr.next;
      flag := 1;
      }
  result := curr;
  prev:= curr;
  if (curr != nil) then {
     curr := curr.next;
     prev.next := nil;
     while (curr != nil) {
         if (curr.data = v) then {
             curr := curr.next;
             flag := 1;
             }
         else {
             prev.next := curr;
             prev := curr;
             curr := curr.next;
             prev.next := nil;
             }
  }
```

Typically a simple function D* → D*

Insert

Delete

Reverse …

But finite-state verification algorithms not applicable, only lots of undecidability results !

Document Transformation

Task: Shift titles one entry up

```
@inproceedings{AC11,
   author = {Alur and Cerny},
   conference = {POPL 2011}
}

@inproceedings{AFR14,
   title = {Streaming transducers},
   conference = {LICS 2014},
   author = {Alur and Freilich and Raghothaman}
}

@inproceedings{ADR15,
   author = {Alur and D'Antoni and Raghothman},
   title = {Regular combinators},
   conference = {POPL 2015}
}
```

Should we use Perl ? sed ?
But these are Turing-complete languages with no "analysis" tools

Complexity Classification of Languages

--- Recursive

--- NP

--- P

--- Linear-time

--- Regular

What if we consider functions?
From strings to strings

No essential change for
Recursive, NP, P, linear-time…

Natural starting point for regular functions:
Variation of classical finite-state automata

Finite-State Sequential Transducers

- Deterministic finite-state control + transitions labeled by (input symbol / string of output symbols)

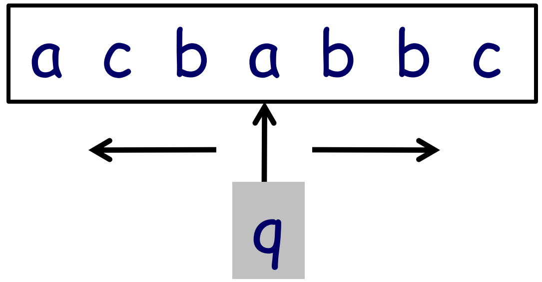
$$q \xrightarrow{a/010} q'$$

- Examples:
  - Delete all a symbols
  - Duplicate each symbol
  - Insert 0 after first b

- Theoretically not that different from classical automata, and have found applications in speech/language processing

Expressive enough ? What about reverse ?

Deterministic Two-way Transducers



- Unlike acceptors, two-way transducers more expressive than one-way model (Aho, Ullman 1969)
  - Reverse
  - Duplicate entire string (map w to w.w)
  - Delete a symbols if string ends with b (regular look-ahead)

Theory of Two-way Finite-state Transducers

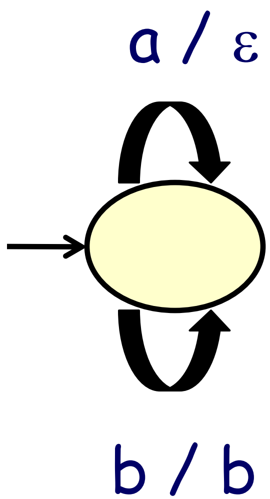
- Closed under sequential composition (Chytil, Jakl, 1977)
- Checking functional equivalence is decidable (Gurari 1980)
- Equivalent to MSO (monadic second-order logic) definable graph transductions (Engelfriet, Hoogeboom, 2001)
- Challenging theoretical results
  - Not like finite automata (e.g. Image of a regular language need not be regular !)
  - Complex constructions
  - No known applications

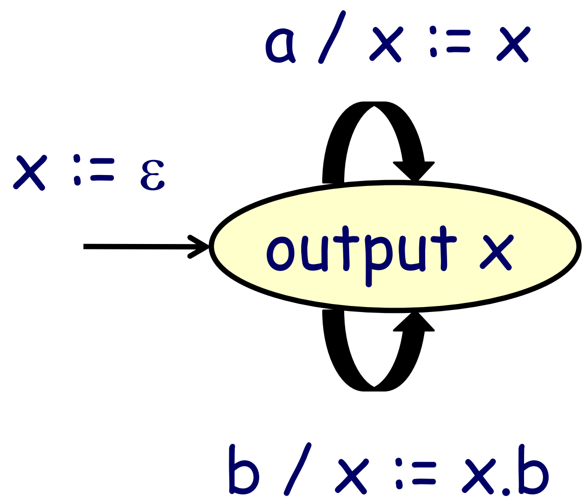
Talk Outline

- ➲ Machine model: Streaming String Transducers
- ❑ DReX: Declarative language for string transformations
- ❑ Regular Functions: Beyond strings to strings

Example Transformation 1: Delete

$Del_a(w)$ = String w with all a symbols removed



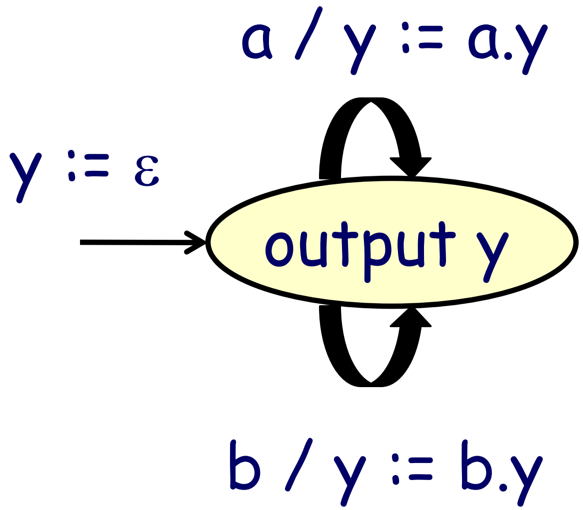


Traditional transducer

Finite-state control + Explicit string variable to compute output

Example Transformation 2: Reverse

Rev(w) = String w in reverse

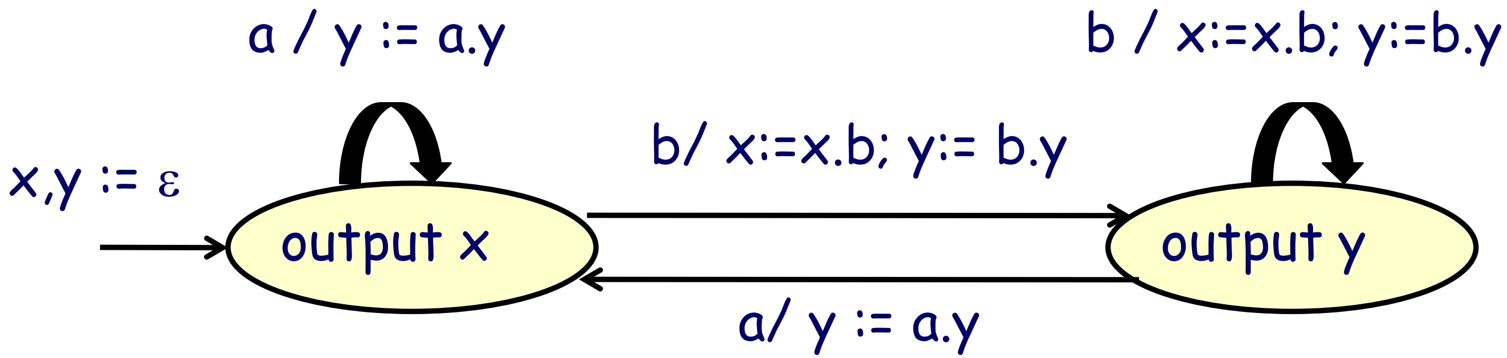


String variables updated at each step as in a program

Key restriction: No tests ! Write-only variables !

Example Transformation 3: Regular Choice

f(w)= If input ends with b, then Rev(w) else Del_a(w)

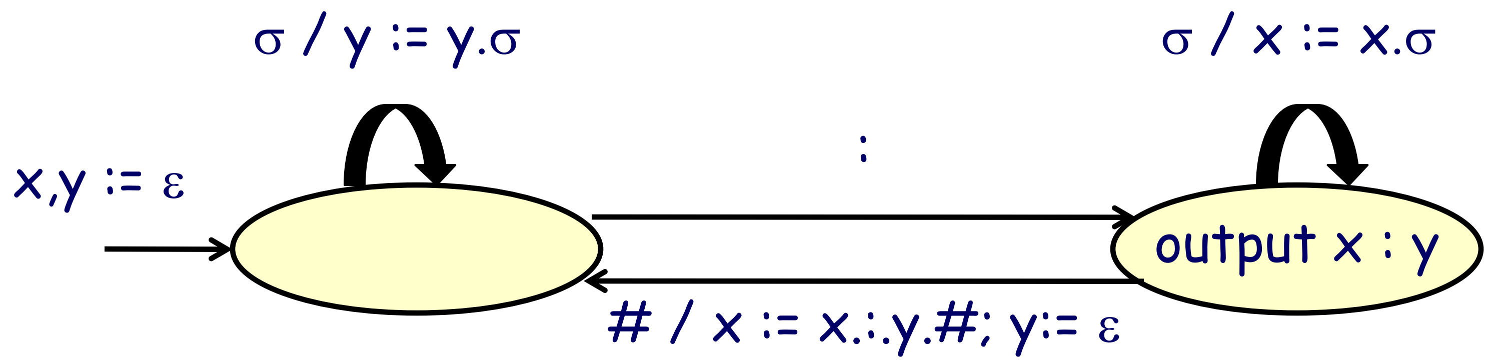


Multiple string variables used to compute alternative outputs

Model closed under “regular look-ahead”

Example Transformation 4: Swap

$f(u_1 : v_1 \# u_2 : v_2 \# \ldots) = v_1 : u_1 \# v_2 : u_2 \# \ldots$ u_i and v_i : $\{a,b\}^*$



Concatenation of string variables allowed (and needed)

Restriction: if x := x.y then y must be assigned a constant

Streaming String Transducer (SST)

1. Finite set Q of states
2. Input alphabet Σ
3. Output alphabet Γ
4. Initial state q_0
5. Finite set X of string variables
6. Partial output function $F : Q \rightarrow (\Gamma \cup X)^*$
7. State transition function $\delta : Q \times \Sigma \rightarrow Q$
8. Variable update function $\rho : Q \times \Sigma \times X \rightarrow (\Gamma \cup X)^*$

- Output function and variable update function required to be copyless: each variable x can be used at most once
- Configuration = (state q, valuation α from X to Γ^*)
- Semantics: Partial function from Σ^* to Γ^*

SST Properties

- At each step, one input symbol is processed, and at most a constant number of output symbols are newly created

- Output is bounded: Length of output = O(length of input)

- SST transduction can be computed in linear time

- Finite-state control: String variables not examined

- SST cannot implement merge

  $f(u_1 u_2 \ldots u_k \# v_1 v_2 \ldots v_k) = u_1 v_1 u_2 v_2 \ldots u_k v_k$

- Multiple variables are essential

  For $f(w)=w^k$, k variables are necessary and sufficient

Decision Problem: Type Checking

Pre/Post condition assertion: { L } S { L' }

Given a regular language L of input strings (pre-condition), an SST S, and a regular language L' of output strings (post-condition), verify that for every w in L, S(w) is in L'

Thm: Type checking is solvable in polynomial-time

Key construction: Summarization

Decision Problem: Equivalence

Functional Equivalence;

Given SSTs S and S' over same input/output alphabets, check whether they define the same transductions.

Thm: Equivalence is solvable in PSPACE

(polynomial in states, but exponential in no. of string variables)

Open problem: Lower bound / Improved algorithm

Expressiveness

Thm: A string transduction is definable by an SST iff it is regular

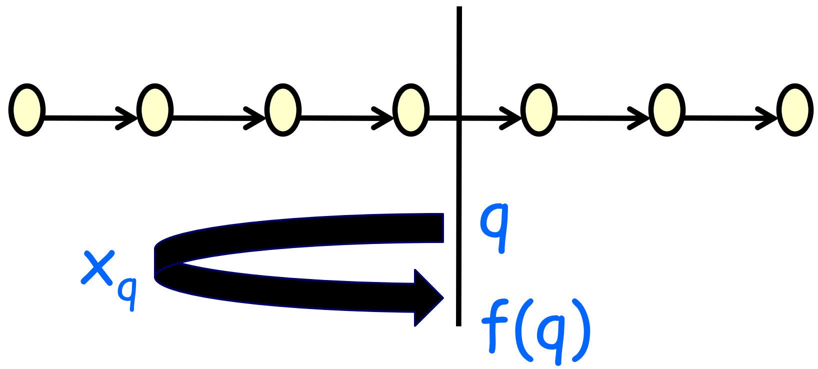
1. SST definable transduction is MSO definable
2. MSO definable transduction can be captured by a two-way transducer (Engelfriet/Hoogeboom 2001)
3. SST can simulate a two-way transducer

Evidence of robustness of class of regular transductions

Closure properties with effective constructions

1. Sequential composition: $f_1(f_2(w))$
2. Regular conditional choice: if w in L then $f_1(w)$ else $f_2(w)$

From Two-Way Transducers to SSTs



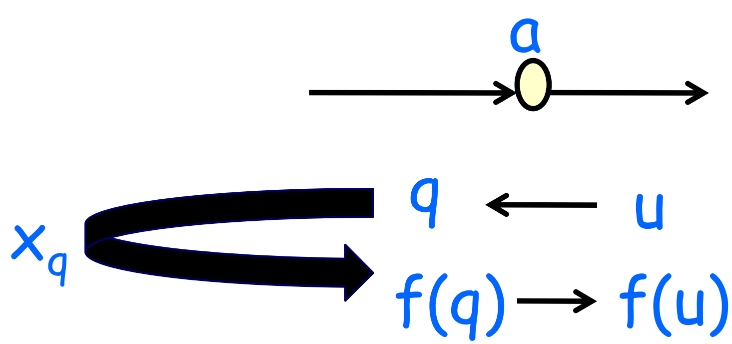
Two-way transducer *A* visits each position multiple times

What information should SST *S* store after reading a prefix?

For each state q of *A*, *S* maintains summary of computation of *A* started in state q moving left till return to same position

1. The state f(q) upon return
2. Variable x_q storing output emitted during this run

Challenge for Consistent Update



Map f: Q -> Q and variables x_q need to be consistently updated at each step

If transducer A moving left in state u on symbol a transitions to q, then updated f(u) and x_u depend on current f(q) and x_q

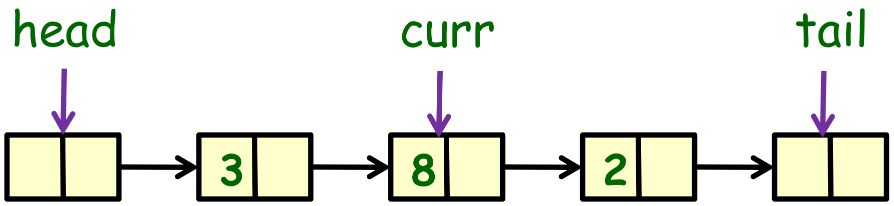
Problem: Two distinct states u and v may map to q

Then x_u and x_v use x_q, but assignments must be copyless !

Solution requires careful analysis of sharing (required value of each x_q maintained as a concatenation of multiple chunks)

Decidable Class of List-processing Programs

```
function delete
  input ref curr;
  input data v;
  output ref result;
  output bool flag := 0;
  local ref prev;

  while (curr != nil) & (curr.data = v) {
      curr := curr.next;
      flag := 1;
      }
  result := curr;
  prev:= curr;
  if (curr != nil) then {
     curr := curr.next;
     prev.next := nil;
     while (curr != nil) {
         if (curr.data = v) then {
             curr := curr.next;
             flag := 1;
             }
         else {
             prev.next := curr;
             prev := curr;
             curr := curr.next;
             prev.next := nil;
             }
  }
```



Decidable Analysis:

1. Assertion checks
2. Pre/post condition
3. Full functional correctness

Talk Outline

- ✓ Machine model: Streaming String Transducers
- ➲ DReX: Declarative language for string transformations
- ❑ Regular Functions: Beyond strings to strings

Search for Regular Combinators

- Regular Expressions
  - Basic operations: ε, a, Union, Concatenation, Kleene-*
  - Additional constructs (e.g. Intersection) : Trade-off between ease of writing constraints and complexity of evaluation

- What are the basic ways of combining functions?
  - Goal: Calculus of regular functions

- Partial function from Σ^* to Γ^*
  - Dom(f): Set of strings w for which f(w) is defined
  - In our calculus, Dom(f) will always be a regular language

Base Functions

- For a in Σ and γ in Γ^*, **a / γ**
  - If input w equals a then output γ, else undefined
- For γ in Γ^*, **ε / γ**
  - If input w equals ε then output γ else undefined

Choice

- **f else g**
  - Given input w, if w in Dom(f), then return f(w) else return g(w)
- Analog of union in regular expressions
  - Asymmetric (non-commutative) nature ensures that the result (f else g)(w) is uniquely defined
- Examples:
  - Id1 = (a / a) else (b / b)
  - Del_a1 = (a / ε) else Id1

Concatenation and Iteration

- **split (f, g)**
  - Given input string w, if there exist unique u and v such that w=u.v and u in Dom(f) and v in Dom(g) then return f(u).g(v)
  - Similar to "unambiguous" concatenation
- **iterate (f)**
  - Given input string w, if there is unique k and unique strings $u_1,\ldots u_k$ such that w = $u_1.u_2\ldots u_k$ and each u_i in Dom(f) then return $f(u_1)\ldots f(u_k)$
- **left-split (f, g)**
  - Similar to split, but return g(v).f(u)
- **left-iterate (f)**
  - Similar to iterate, but return $f(u_k)\ldots f(u_1)$

Examples

- Id1 = (a / a) else (b / b)
- Del_a1 = (a / ε) else Id1
- Id = iterate (Id1) : maps w to itself
- Del_a = iterate (Del_a1) : Delete all a symbols
- Rev = left-iterate (Id1) : reverses the input
- If w ends with b then delete a's else reverse

  split (Del_a, b / b) else Rev
- Map u#v to v.u

  left-split (split (Id, # / ε), Id)

Function Combination

- **combine (f, g)**
  - If w in both Dom(f) and Dom(g), then return f(w).g(w)
- combine(Id, Id) maps an input string w to w.w
- Needed for expressive completeness
- Reminiscent of Intersection for languages

Document Transformation Example

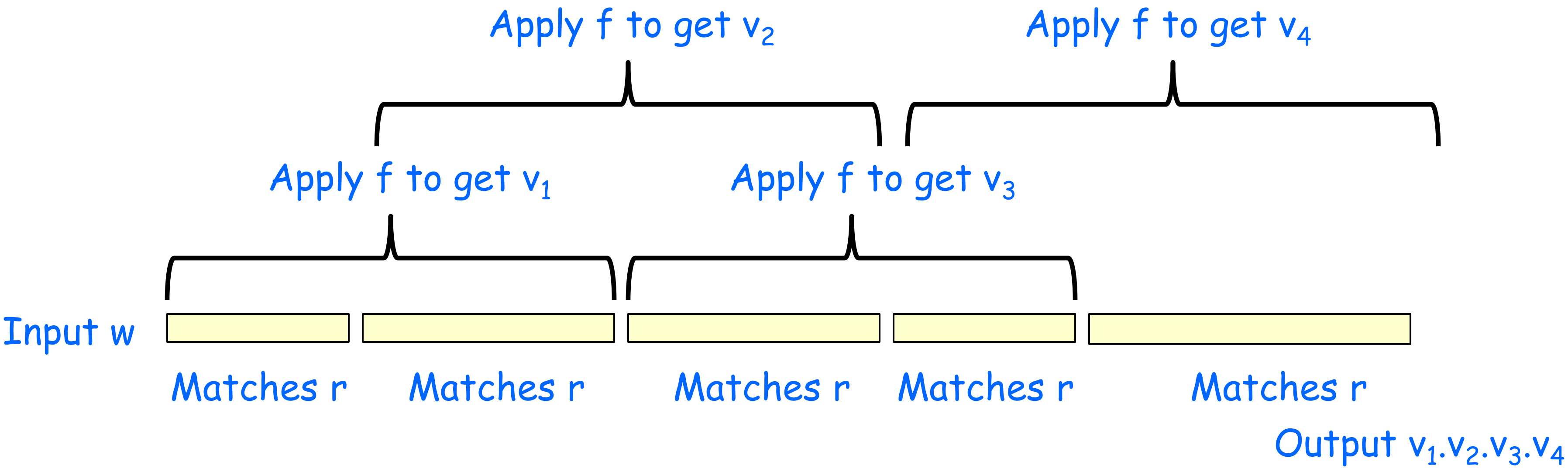
```
@inproceedings{AC11,
   author = {Alur and Cerny},
   conference = {POPL 2011}
}

@inproceedings{AFR14,
   title = {Streaming transducers},
   conference = {LICS 2014},
   author = {Alur and Freilich and Raghothaman}
}

@inproceedings{ADR15,
   author = {Alur and D'Antoni and Raghothman},
   title = {Regular combinators},
   conference = {POPL 2015}
}
```

Task: Shift titles one entry up

Does not seem expressible with combinators discussed so far…
Cannot compute this by splitting document in chunks, transforming them separately, and combining the results

Chained Iteration

chain (f, r) : Given input string w, if there is unique k and unique strings $u_1,\ldots u_k$ such that $w = u_1.u_2\ldots u_k$ and each u_i in Dom(r) then return $f(u_1u_2).f(u_2u_3)\ldots f(u_{k-1}u_k)$



Thm: A partial function $f : \Sigma^* \rightarrow \Gamma^*$ is regular iff it can be constructed using base functions, choice, split, left-split, combine, chain, and left-chain.

Towards a Prototype Language

- Goal: Design a DSL for regular string transformations

- Allow "symbolic" alphabet
  - Symbols range over a "sort"
  - Base function: $\varphi(x)$ / γ
  - Set of allowed predicates form a Boolean algebra
  - Inspired by Symbolic Automata of Veanes et al

- Given a program P and input w, evaluation of P(w) should be fast!
  - Natural algorithm is based on dynamic programming: $O(|w|^3)$

Consistency Rules

- In **f else g**, Dom(f) and Dom(g) should be disjoint
- In **combine(f,g)**, Dom(f) and Dom(g) should be identical
- In **split(f,g)**, for every string w, there exists at most one way to split w = u.v such that u in Dom(f) and v in Dom(g)
- Similar rules for left-split, iterate, chain, and so on

DReX: Declarative Regular Transformations

- Syntax based on regular combinators + Type system to enforce consistency rules

- Thm: Restriction to consistent programs does not limit the expressiveness (DReX captures exactly regular functions)

- Consistency can be checked in poly-time in size of program

- For a consistent DReX program P, output P(w) can be computed in single-pass in time O(|w|) (and poly-time in |P|)
  - Intuition: To compute split(f,g)(w), whenever a prefix of w matches Dom(f), a new thread is started to evaluate g. Consistency is used to kill threads eagerly to limit the number of active threads

DReX Prototype Status

- Prototype implementation
  - Type checking
  - Linear-time evaluation
- Evaluation
  - How natural is it to write consistent DReX programs?
  - How does type checker / evaluator scale ?
- Ongoing work
  - Syntactic sugar with lots of pre-defined operations
  - Support for analysis (e.g. equivalence checking)

Try it out at www.drexonline.com

Talk Outline

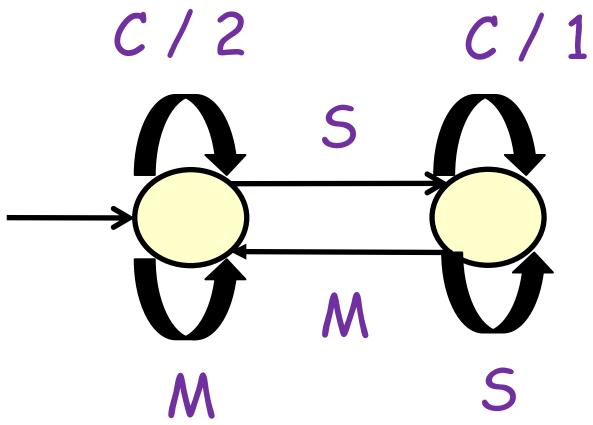
- ✓ Machine model: Streaming String Transducers
- ✓ DReX: Declarative language for string transformations
- ➲ Regular Functions: Beyond strings to strings

Mapping Strings to Numerical Costs

C: Buy Coffee

S: Fill out a survey

M: End-of-month

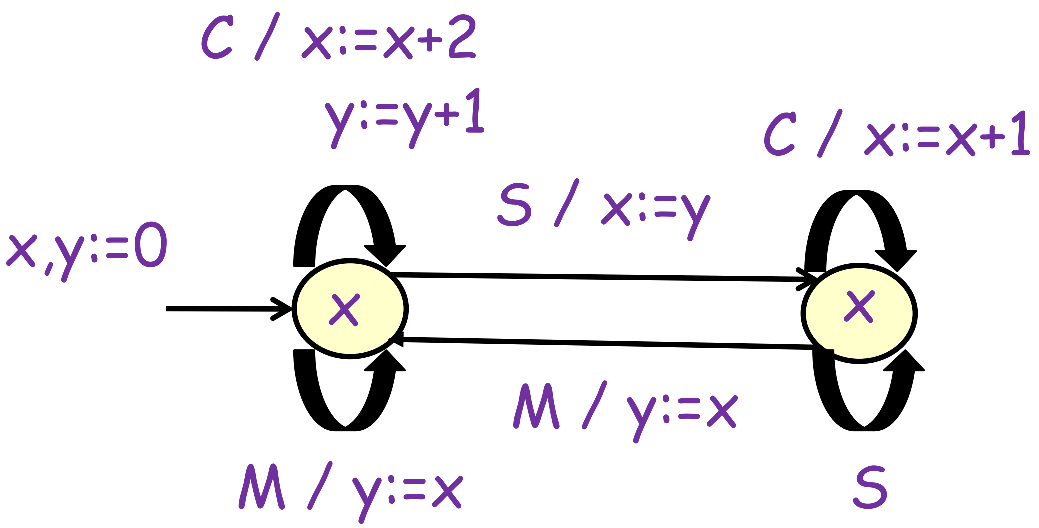


Maps a string over {C,S,M} to a cost value:

Cost of a coffee is 2, but reduces to 1 after filling out a survey until the end of the month

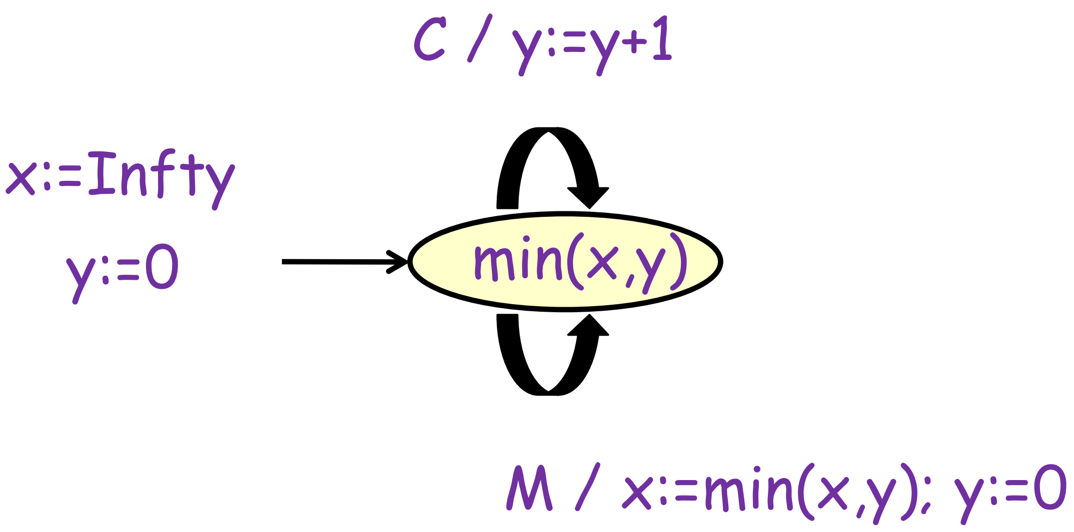
Can we generalize expressiveness using SST-style model?
Potential application: Quantitative queries for data streams

Cost Register Automata (CRA) Example



Filling out a survey gives discount for all coffees during that month

CRA Example



Output = minimum number of coffees consumed during a month

Updates use two operations: increment and min

Can we define a general notion of regularity parameterized by operations on the set of costs ?

Cost Model

Cost Grammar G to define set of terms:

Inc: t := c | (t+c)

Plus: t := c | (t+t)

Min-Inc: t := c | (t+c) | min(t,t)

Inc-Scale: t := c | (t+c) | (t*d)

Interpretation [] for operations:

Set D of cost values

Mapping operators to functions over D

Example interpretations for the Plus grammar:

Set N of natural numbers with addition

Set Γ^* of strings with concatenation

Regular Function

Definition parameterized by the cost model C=(D,G,[])

A (partial) function f:Σ*->D is regular w.r.t. the cost model C if there exists a string-to-tree transformation g such that

(1) for all strings w, f(w)=[g(w)]

(2) g is a regular string-to-tree transformation

Regular String-to-tree Transformations

- Definition based on MSO (Monadic Second Order Logic) – definable graph-to-graph transformations (Courcelle)
- Studied in context of syntax-directed program transformations, attribute grammars, and XML transformations
- Operational model: Macro Tree Transducers (Engelfriet et al)
- Recent proposal: Streaming Tree Transducers (ICALP 2012)

MSO-definable String-to-tree Transformations

- MSO over strings

  Φ := a(x) | X(x) | x=y+1 | ~ Φ | Φ & Φ | Exists x. Φ | Exists X. Φ

- MSO-transduction from strings to trees:

  1. Number k of copies

     For each position x in input, output-tree has nodes $x_1, \ldots x_k$

  2. For each symbol a and copy c, MSO-formula $\Phi_{a,c}(x)$

     Output-node x_c is labeled with a if $\Phi_{a,c}(x)$ holds for unique a

  3. For copies c and d, MSO-formula $\Phi_{c,d}(x,y)$

     Output-tree has edge from node x_c to node x_d if $\Phi_{c,d}(x,y)$ holds

Example Regular Function

Cost grammar Min-Inc: t := c | (t+c) | min(t,t)

Interpretation: Natural numbers with usual meaning of + and min

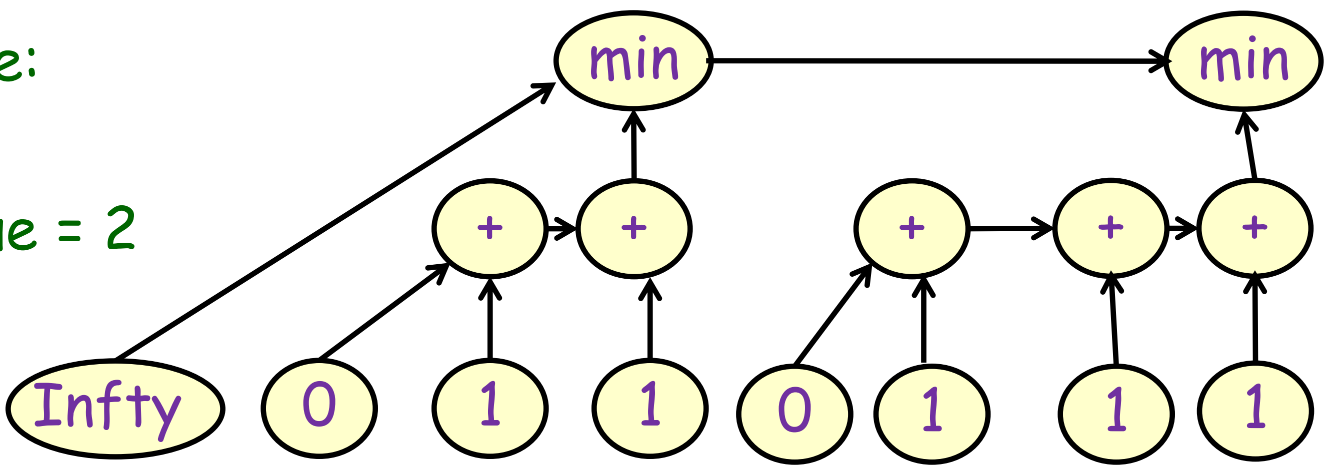
Σ={C,M}

f(w) = Minimum number of C symbols between successive M's

Input w= C C M C C C M

Tree:

Value = 2



Properties of Regular Functions

Known properties of regular string-to-tree transformations imply:

- If f and g are regular w.r.t. a cost model C, and L is a regular language, then "if L then f else g" is regular w.r.t. C
- Reversal: define Rev(f)(w) = f(reverse(w)).
  If f is regular w.r.t. a cost model C, then so is Rev(f)
- Costs grow linearly with the size of the input string:
  Term corresponding to a string w is O(|w|)

Regular Functions over Commutative Monoid

Cost model: D with binary function +

Interpretation for + is commutative, associative, with identity 0

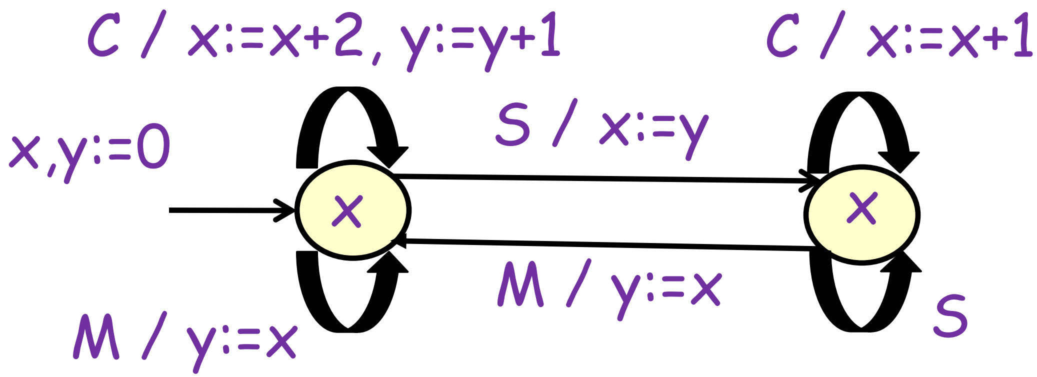
Cost grammar G(+): t := c | (t+t)

Cost grammar G(+c): t := c | (t+c)

Thm: Regularity w.r.t. G(+) coincides with regularity w.r.t. G(+c)

Proof intuition: Show that rewriting terms such as (2+3)+(1+5) to (((2+3)+1)+5) is a regular tree-to-tree transformation, and use closure properties of tree transducers

Additive Cost Register Automata



- DFA + Finite number of registers, initialized to 0
- Registers updated using assignments x := y + c
- Each final state labeled with output term x + c

Thm: For a commutative monoid (D,+,0), a function f:Σ*->D is definable using an ACRA iff it is regular w.r.t. grammar G(+).

Decision Problems for ACRAs

- Min-Cost: Given an ACRA M, find min {M(w) | w in Σ^*}
  - Solvable in Polynomial-time
  - Shortest path in a graph with vertices (state, register)
- Equivalence: Do two ACRAs define the same function
  - Solvable in Polynomial-time
  - Based on propagation of linear equalities in program graphs
- Register Minimization: Given an ACRA M with k registers, is there an equivalent ACRA with < k registers?
  - Algorithm polynomial in states, and exponential in k

Emerging Theory of Regular Functions

- A few classes that have been (partially) studied
  - Finite strings to finite strings
  - Finite strings to commutative monoid
  - Infinite strings to infinite strings
  - Finite strings to semiring (N, +, min)
  - Finite strings to discounted costs
  - Finite trees to finite trees
- Many open problems (and unexplored classes)
  - Decidability of equivalence of functions from Σ^* to (N,+,min)
  - Theory of congruences
  - Learning algorithms…

Conclusions

- Streaming String Transducers and Cost Register Automata
  - Write-only machines with multiple registers to store outputs
- DReX: Declarative language for string transformations
  - Robust expressiveness with decidable analysis problems
  - Prototype implementation with linear-time evaluation
  - Ongoing work: Analysis tools
- Emerging theory of regular functions
  - Some results, new connections
  - Many open problems and unexplored directions