Regular Functions

Rajeev Alur
University of Pennsylvania
Regular Languages

- **Natural**
  Intuitive operational model of finite-state automata

- **Robust**
  Alternative characterizations and closure properties

- **Analyzeable**
  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)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vechev, Martin</td>
<td>Martin Vechev</td>
</tr>
<tr>
<td>Martin Abadi</td>
<td>Martin Abadi</td>
</tr>
<tr>
<td>Rinard, Martin C.</td>
<td>Martin Rinard</td>
</tr>
</tbody>
</table>

- 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 : ??
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* \rightarrow D*
Insert
Delete
Reverse ...

But finite-state verification algorithms not applicable, only lots of undecidability results!
Should we use Perl? sed?
But these are Turing-complete languages with no “analysis” tools
Complexity Classification of Languages

- Recursive
  - What if we consider functions? From strings to strings
- NP
- P
- Linear-time
- Regular
  - 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)

- 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

$\text{Del}_a(w) = \text{String } w \text{ with all } a \text{ symbols removed}$

Traditional transducer

Finite-state control + Explicit string variable to compute output
Example Transformation 2: Reverse

Rev(w) = String w in reverse

- a / y := a.y
- y := ε
- b / y := b.y

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 \# ... \) = v_1 : u_1 \# v_2 : u_2 \# ... \quad u_i \text{ and } v_i : \{a, b\}^* \]

\[ \sigma / y \overset{\varepsilon}{=} \gamma. \sigma \]

\[ \sigma / x \overset{\varepsilon}{=} \chi. \sigma \]

\[ \chi, \gamma \overset{\varepsilon}{=} \varepsilon \]

\[ \# / x \overset{\varepsilon}{=} \chi : \gamma, \#; \gamma \overset{\varepsilon}{=} \varepsilon \]

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

- Output function and variable update function required to be copyless: each variable $x$ can be used at most once
- Configuration = (state $q$, valuation $\alpha$ from $X$ to $\Gamma^*$)
- Semantics: Partial function from $\Sigma^*$ to $\Gamma^*$
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 = $O(\text{length of input})$.

- SST transduction can be computed in linear time.

- Finite-state control: String variables not examined.

- SST cannot implement merge:
  
  $$f(u_1u_2...u_k\#v_1v_2...v_k) = u_1v_1u_2v_2...u_kv_k$$

- 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
Map $f$: $Q \rightarrow Q$ 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).
Heap-manipulating Programs

Sequential program +
Heap of cells containing data and next pointers +
Boolean variables +
Pointer variables that reference heap cells

Program operations can add cells, change next pointers, and traverse the heap by following next pointers

How to restrict operations to capture exactly regular transductions
Representing Heaps in SST

Shape (encoded in state of SST):
\[ x : u_1 u_2 z ; y : u_4 u_2 z ; z : u_3 \]

String variables: \( u_1, u_2, u_3, u_4 \)

Shape + values of string vars enough to encode heap
Simulating Heap Updates

Consider program instruction

\[ y.\text{next} := z \]

How to update shape and string variables in SST?
Simulating Heap Updates

New Shape: $x: u_1 z; y: z; z: u_3$

Variable update: $u_1 := u_1 u_2$

Special cells:
- Cells referenced by pointer vars
- Cells that 2 or more (reachable) next pointers point to

Contents between special cells kept in a single string var

Number of special cells = $2(# \text{ of pointer vars}) - 1$
Regular Heap Manipulating Programs

Update

\[ x\.next := y \]  \hspace{1cm} \text{(changes heap shape destructively)}
\[ x := \text{new} (a) \]  \hspace{1cm} \text{(adds new cell with data a and next nil)}

Traversal

\[ \text{curr} := \text{curr\.next} \]  \hspace{1cm} \text{(traversal of input list)}
\[ x := y\.next \]  \hspace{1cm} \text{(disallowed in general)}

Theorem: Programs of above form can be analyzed by compiling into equivalent SSTs

- Single pass traversal of input list possible
- Pointers cannot be used as multiple read heads
Manipulating Data

- Each string element consists of (tag t, data d)
  - Tags are from finite set
  - Data is from unbounded set D that supports = and < tests
  - Example of D: Names with lexicographic order

- SSTs and list-processing programs generalized to allow
  - Finite set of data variables
  - Tests using = and < between current value and data vars
  - Input and output values

- Checking equivalence remains decidable (in PSPACE)!

- Many common routines fall in this class
  - Check if list is sorted
  - Insert an element in a sorted list
  - Delete all elements that equal input value
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
Potential Application: String Sanitizers

- BEK: A domain specific language for writing string manipulating sanitizers on untrusted user data

- Analysis tool translates BEK program into (symbolic) transducer and checks properties such as
  - Is transduction idempotent: \( f(f(w)) = f(w) \)
  - Do two transductions commute: \( f_1(f_2(w)) = f_2(f_1(w)) \)

- Recent success in analyzing IE XSS filters and other web apps

- Example sanitizer that BEK cannot capture (but SST can):
  Rewrite input \( w \) to suffix following the last occurrence of “dot”

---

Fast and precise sanitizer analysis with BEK.
Hooimeijer et al. USENIX Security 2011
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: $\varepsilon$, $a$, 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 $\Sigma^*$ to $\Gamma^*$**
  - $\text{Dom}(f)$: Set of strings $w$ for which $f(w)$ is defined
  - In our calculus, $\text{Dom}(f)$ will always be a regular language
Base Functions

- For $a$ in $\Sigma$ and $\gamma$ in $\Gamma^*$, $a / \gamma$
  - If input $w$ equals $a$ then output $\gamma$, else undefined

- For $\gamma$ in $\Gamma^*$, $\varepsilon / \gamma$
  - If input $w$ equals $\varepsilon$ then output $\gamma$ else undefined
Choice

- $f \text{ else } g$
  - Given input $w$, if $w$ in $\text{Dom}(f)$, then return $f(w)$ else return $g(w)$

- Analog of union in regular expressions
  - Asymmetric (non-commutative) nature ensures that the result $(f \text{ else } g)(w)$ is uniquely defined

- Examples:
  - $\text{Id}_1 = (a / a) \text{ else } (b / b)$
  - $\text{Del}_a 1 = (a / \varepsilon) \text{ else } \text{Id}_1$
Concatenation and Iteration

- **split (f, g)**
  - Given input string \( w \), if there exist unique \( u \) and \( v \) such that \( w = u \cdot v \) and \( u \) in \( \text{Dom}(f) \) and \( v \) in \( \text{Dom}(g) \) then return \( f(u) \cdot g(v) \)
  - 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 \( w = u_1 \cdot u_2 \cdots u_k \) and each \( u_i \) in \( \text{Dom}(f) \) then return \( f(u_1) \cdots f(u_k) \)

- **left-split (f, g)**
  - Similar to split, but return \( g(v) \cdot f(u) \)

- **left-iterate (f)**
  - Similar to iterate, but return \( f(u_k) \cdots f(u_1) \)
Examples

- $\text{Id1} = (a / a) \text{ else } (b / b)$
- $\text{Del}_a1 = (a / \varepsilon) \text{ else } \text{Id1}$

- $\text{Id} = \text{iterate (Id1)} : \text{maps w to itself}$

- $\text{Del}_a = \text{iterate (Del}_a1) : \text{Delete all a symbols}$

- $\text{Rev} = \text{left-iterate (Id1)} : \text{reverses the input}$

- If $w$ ends with b then delete a’s else reverse
  $\text{split (Del}_a, b / b) \text{ else Rev}$

- Map $u\#v$ to $v.u$
  $\text{left-split ( split ( Id, \# / \varepsilon), Id )}$
Function Combination

- combine \((f, g)\)
  - If \(w\) in both \(\text{Dom}(f)\) and \(\text{Dom}(g)\), then return \(f(w) \cdot g(w)\)

- combine\((\text{Id}, \text{Id})\) maps an input string \(w\) to \(w \cdot w\)

- Needed for expressive completeness

- Reminiscent of Intersection for languages
@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}
}

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 u₁,...uₖ such that w = u₁.u₂...uₖ and each uᵢ in Dom(r) then return f(u₁u₂).f(u₂u₃)...f(uₖ₋₁uₖ)

**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: $\varphi(x) / \gamma$
  - 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 $f \text{ else } g$, $\text{Dom}(f)$ and $\text{Dom}(g)$ should be disjoint.

- In $\text{combine}(f, g)$, $\text{Dom}(f)$ and $\text{Dom}(g)$ should be identical.

- In $\text{split}(f, g)$, for every string $w$, there exists at most one way to split $w = u.v$ such that $u$ in $\text{Dom}(f)$ and $v$ in $\text{Dom}(g)$.

- 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 $\text{split}(f,g)(w)$, whenever a prefix of $w$ matches $\text{Dom}(f)$, 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)
  - Integration in Python/Java?
Talk Outline

- Machine model: Streaming String Transducers
- DReX: Declarative language for string transformations
- Regular Functions: Beyond strings to strings
  - Parameterized Definition of Regularity
  - Additive Cost Register Automata
  - Regular functions over a semi-ring
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 analysis
Cost Register Automata:
Finite control + Finite number of registers
Registers updated explicitly on transitions
Registers are write-only (no tests allowed)
Each (final) state associated with output register
CRA Example

At any time, $x =$ cost of coffees during the current month

Cost register $x$ reset to 0 at each end-of-month
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:** $t := c \mid (t+c)$
- **Plus:** $t := c \mid (t+t)$
- **Min-Inc:** $t := c \mid (t+c) \mid \min(t,t)$
- **Inc-Scale:** $t := c \mid (t+c) \mid (t\ast d)$

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 $\Gamma^*$ of strings with concatenation
Regular Function

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

A (partial) function $f: \Sigma^* \rightarrow D$ 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
MSO-definable String-to-tree Transformations

- **MSO over strings**
  \[ \Phi := a(x) \mid X(x) \mid x = y + 1 \mid \sim \Phi \mid \Phi \& \Phi \mid \text{Exists } x. \Phi \mid \text{Exists } X. \Phi \]

- **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: \( t := c \mid (t+c) \mid \text{min}(t,t) \)
Interpretation: Natural numbers with usual meaning of + and min
\( \Sigma = \{C,M\} \)
\( f(w) = \) Minimum number of \( C \) symbols between successive \( M \)'s

Input \( w = \) C C M C C C M

Tree:

Value = 2
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)
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 $\text{Rev}(f)(w) = f(\text{reverse}(w))$. If $f$ is regular w.r.t. a cost model $C$, then so is $\text{Rev}(f)$.

- Costs grow linearly with the size of the input string: Term corresponding to a string $w$ is $O(|w|)$. 
Cost model: $\Gamma^*$ with binary function concatenation

Interpretation for $\cdot$ is non-commutative, associative, identity $\varepsilon$

Cost grammar $G(.)$: $t := \sigma | (t \cdot t)$  \quad $\sigma$ is a string

Cost grammar $G(\sigma)$: $t := \sigma | (t \cdot \sigma) | (\sigma \cdot t)$

Thm: Regular functions w.r.t $G(.)$ is a strict superset of regular functions w.r.t. $G(\sigma)$

Classical model of Sequential Transducers captures only a subset of regular functions w.r.t. $G(\sigma)$

SSTs capture exactly regular functions w.r.t. $G(.)$
Regular Functions over Commutative Monoid

Cost model: D with binary function +
Interpretation for + is commutative, associative, with identity 0

Cost grammar $G(+)$: $t ::= c \mid (t+t)$

Cost grammar $G(+c)$: $t ::= c \mid (t+c)$

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
- Each register is initially 0
- Registers updated using assignments \( x := y + c \)
- Each final state labeled with output term \( x + c \)

Given commutative monoid \((D,+,0)\), an ACRA defines a partial function from \( \Sigma^* \) to \( D \)
Regular Functions and ACRAs

- Thm: Given a commutative monoid \((D,+\,0)\), a function \(f:\Sigma^*\rightarrow D\) is definable using an ACRA iff it is regular w.r.t. grammar \(G(+).\)

- Establishes ACRA as an intuitive, deterministic operational model to define this class of regular functions

- Proof relies on the model of SSTT (Streaming string-to-tree transducers) that can define all regular string-to-tree transformations
Single-Valued Weighted Automata

- **Weighted Automata:**
  Nondeterministic automata with edges labeled with costs

- **Single-valued:**
  Each string has at most one accepting path

- **Cost of a string:**
  Sum of costs of transitions along the accepting path

- **Example:** When you fill out a survey, each coffee during that month gets the discounted cost.
  Locally nondeterministic, but globally single-valued

- **Thm:** ACRAs and single-valued weighted automata define the same class of functions
Decision Problems for ACRAs

- **Min-Cost:** Given an ACRA $M$, find $\min \{ M(w) \mid w \in \Sigma^* \}$
  - 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 $k$
Towards a Theory of Additive Regular Functions

Goal: Machine-independent characterization of regularity
Similar to Myhill-Nerode theorem for regular languages
Registers should compute necessary auxiliary functions

Example: \( \Sigma = \{C, S\} \)

\[ f(w) = \begin{cases} |w| & \text{if } w \text{ contains } S \\ 2|w| & \text{else} \end{cases} \]

\( f_1(C^i) = i \) and \( f_2(C^i) = 2i \) are necessary and sufficient

Thm: Register complexity of a function is at least \( k \) iff there exist strings \( \sigma_0, \ldots \sigma_m \), loop-strings \( \tau_1, \ldots \tau_m \), and suffixes \( w_1, \ldots w_m \), and \( k \) distinct vectors \( c_1, \ldots c_k \) such that for all numbers \( x_1, \ldots x_m \),

\[ f(\sigma_0 \tau_1^{x_1} \sigma_1 \tau_2^{x_2} \ldots \sigma_m w_i) = \sum_j c_{ij} x_j + d_i \]
Regular Functions over Semiring

- **Cost Domain:** Natural numbers + Infty

- **Operation Min:** Commutative monoid with identity Infty

- **Operation +:** Monoid with identity 0

- **Rules:**
  
  \[ a + \text{Infty} = \text{Infty} + a = \text{Infty} \]
  
  \[ a + \min(b,c) = \min(a+b, a+c); \min(b,c)+a = \min(b+a, c+a) \]

- **Cost grammar MinInc:** \( t := c \mid \min(t,t) \mid (t+c) \)

- **Goal:** Understand class of regular functions w.r.t. MinInc
Weighted Automata

- **Weighted Automata:**
  Nondeterministic automata with edges labeled with costs

- **Interpreted over the semiring cost model:**
  cost of string $w = \min$ of costs of all accepting paths over $w$
  cost of a path = sum of costs of all edges in a path

- **Widely studied (Handbook of Weighted Automata, Droste et al)**
  Minimum cost problem solvable
  Equivalence undecidable over $(N, \min, +)$
  Not determinizable
  Natural model in many applications
  Recent interest in CAV/LICS community for quantitative analysis
CRA over Min-Inc Semiring

Output = Minimum number of coffees consumed during a month
CRA(\text{min},+c) = \text{Weighted Automata}

- From WA to CRA(\text{min},+c):
  Generalizes subset construction for determinization
  For every state \( q \) of WA, CRA maintains a register \( x_q \)
  \( x_q = \text{min of costs of all paths to } q \text{ on input read so far} \)
  Update on \( a \): \( x_q := \text{min} \{ x_p + c \mid \text{p }-(a,c)-\rightarrow q \text{ is edge in WA} \} \)

- From CRA(\text{min},+c) to WA:
  State of WA = (state \( q \) of CRA, register \( x \))
  min simulated by nondeterminism
  To simulate \( p - (a, x:=\text{min}(y,z)) -\rightarrow q \) in CRA,
    add \( a \)-labeled edges from \((p,y)\) and \((p,z)\) to \((q,x)\)
  Distributivity of + over \text{min} critical
CRA(min,+c) > Min-Plus Regular Functions

Input $w$: $w_1 \ M \ w_2 \ M \ ... \ M \ w_n$
Each $w_i$ in $\{C,S\}^*$
$c_i$ = Number of C’s in $w_i$
$s_i$ = Number of S’s in $w_i$

$\text{Cost}(w) = \min_j \{ c_1+...+c_j+s_{j+1}+...+s_n \}$

Thm: The class of regular functions w.r.t. Min-Inc semiring is a strict subset of weighted automata

Above function is not regular: cost term is quadratic in input
Machine Model for Semiring Regular Functions

- Updates to registers must be copyless
  - Each register appears at most once in a right-hand-side
  - Update \([x, y] := [\min(x, y), y]\) not allowed
  - Necessary to maintain “linear” growth

- Need ability to simulate substitution
  - Register \(x\) carries two values \(c\) and \(d\)
  - Stands for the parameterized expression \(\min(c, ?) + d\)
  - Besides min and inc, can substitute \(?\) with a value

- Resulting model coincides with regular functions over semiring

- Open: Decidability of equivalence over \((\mathbb{N}, \min, +c)\)
Discounted Cost Regular Functions

- Basic element: \((\text{cost } c, \text{ discount } d)\)
- Discounted sum: \((c_1, d_1) \times (c_2, d_2) = (c_1 + d_1 c_2, d_1 d_2)\)
- Example of non-commutative monoid
- Classical Model: Future discounting

  Cost of a path: \((c_1, d_1) \times (c_2, d_2) \times \ldots \times (c_n, d_n)\)

  Polynomial-time algorithm for “generalized” shortest path

- Past discounting

  Cost of a path: \((c_n, d_n) \times (c_{n-1}, d_{n-1}) \times \ldots \times (c_1, d_1)\)

  Same PTIME algorithm works for shortest paths

- Prioritized double discounting

  Cost = \((c_1, d_1) \times \ldots \times (c_n, d_n) \times (c'_1, d'_1) \times \ldots \times (c'_n, d'_n)\)

  Shortest path: NExpTime algorithm

- Open: Shortest path for Discounted Cost Register Automata
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
Acknowledgements and References

- Streaming String Transducers (with P. Cerny; POPL’11, FSTTCS’10)
- Transducers over Infinite Strings (with E. Filiot, A. Trivedi; LICS’12)
- Streaming Tree Transducers (with L. D’Antoni; ICALP’12)
- Regular Functions and Cost Register Automata
  (with L. D’Antoni, J. Deshmukh, M. Raghothaman, Y. Yuan; LICS’13)
- Decision problems for Additive Cost Regular Functions
  (with M. Raghothaman; ICALP’13)
- Infinite-String to Infinite-Term Regular Transformations
  (with A. Durand, A. Trivedi; LICS’13: Next session)
- Min-cost problems for Discounted Sum Regular Functions
  (with S. Kannan, K. Tian, Y. Yuan; LATA’13)
- Regular combinators for string transformations
  (with A. Freilich and M. Raghothaman; LICS’13)
- DReX (with L. D’Antoni and M. Raghothaman; POPL’15)