REPRESENTING PROCESSES AS UPDATE AUTOMATA AND TRANSDUCERS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Bryant Adams

August 2008

REPRESENTING PROCESSES AS UPDATE AUTOMATA AND TRANSDUCERS

Bryant Adams, Ph.D.

Cornell University 2008

In 2007 Nerode introduced the notion of an update automaton and update transducer to model the evolution of interactions of arbitrary functions on arbitrary domains governed by a finite automaton. We show the scope of this methodology by modeling a variety of processes, including database networks, interactive proofs, finite-state transducers, and financial ledgers. We produce a novel solution method and associated transducers for finite systems of arbitrary-order non-homogeneous linear difference equations over a field using Hermite's unimodular polynomial matrix row reduction. These transducers can be viewed as time invariant linear causal maps of the linear space of sequences of vectors over the base field.

BIOGRAPHICAL SKETCH

Bryant was born in Vermont, USA, on September 29, 1979. He graduated from Washington and Lee University in Lexington, Virginia in 2001 with Bachelors of Science and Arts in Physics, Mathematics, and Russian Studies. In 2006 he obtained Masters of Science in Mathematics and Computer Science from Cornell University, and finished his PhD in Mathematics in 2008 with this thesis. Bryant has accepted an Assistant Professorship in the Mathematics and Physical Science department of Wells College in Aurora, NY.

ACKNOWLEDGEMENTS

In reverse chronological order, I begin by thanking my wife, Sarah Rose Adams, whose emotional and logistic support have been indispensable. I thank my thesis advisor, Anil Nerode, for facilitating my ever-expanding interests without letting me get lost. I thank my friend Heather Root, for not accepting "because I started" as a sufficient reason in isolation. I thank my advisor, Hod Lipson, for getting me started on the wandering path that eventually led to this point. And, of course, I thank my parents and friends, who have managed to provide anchors and supports with no strings attached.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Nerode introduced the notion of *update automata* and *update transducers* in 2007 (first announced in 2008 at Tbilisi[18]) for the purpose of modeling the way in which finite automata can govern operations on arbitrary structures. As the first development of this subject, he assigned two dissertation topics, one to Edoardo Carta, the other to Bryant Adams, in order to begin investigation of this modeling process. Carta's dissertation[3] deals with finite systems of linear recurrences over an arbitrary semiring with zero and one. This development led to explicit formulas for solutions of first-order systems and the development of the corresponding automata. This is noteworthy in that there is no use of operation of subtraction or division, and in that respect gave a new way of solving first-order linear recurrences with variable coefficients.

This thesis extends the representation by automata of first order systems of linear recurrences of Carta to higher order systems of linear recurrences over a field using as the main tool the Hermite reduction of unimodular matrices of polynomials to triangular form.

1.1 Range of intended applications

The notions of update transducers and automata are intended to cover a wide variety of applications, including difference equations, interactive proofs, accounting systems, cellular automata, neural networks, database networks, and environmental control systems. In this case we propose a primitive model for

databases, and something of a very similar sort has been proposed in the previous year by Tannen[10] whose representation can be made into update transducers.

All of the applications will require further development in order to be useful. We are giving the application and developing the models, but it is beyond the scope of the thesis to apply them to specific problems in Computer Science.

1.2 Highlights

A core concept behind the notion is the separation of the development of an evolving computation, which will be represented by the update automata or transducer, from the recognition of a point at which that the automaton has computed a particular result. The archetypal example of this is allowing a Turing machine to run indefinitely (corresponding to an update automaton) but declaring that once it enters a steady state (in particular, a designated 'halt' state) the computation has finished and the final output of the machine has been achieved (corresponding to a separate recognition act.) The notion of an update automata does not include the notion of a termination condition. This must be imposed from the outside.

In addition to the separation of 'processing' from 'acceptance', there are other aspects of processes which may be highlighted by modeling them as update automata. Both implicit pipelining and the natural mapping between an update automaton to a network of agents are relevant to parallelization processes. The verification of some programs (such as those for difference equations) can be highly transparent in the update automaton setting. The ability to

compose or decompose given update automata into update automata with simpler or more complex internal structures (and correspondingly more complex or simpler update rules) may be used to extract programs from specifications or, conversely, examine the specifications met by a particular program.

When we show that an automaton which is developed in this thesis carries out the mathematical computations that are represented by, for example, a particular difference equation, this is actually a verification that the program represented by the input transducer does indeed satisfy its program specification. There, the program specification is the system of difference equations, and the argument given is the proof that they satisfy the program specification. For other automata, informal remarks will give a way of demonstrating that the automata develops the desired solution. While we do not look into more practical phenomena, there are similar problems and similar solutions: give a mathematical specification as to how the system develops, and then prove that it develops that way.

1.3 Contributions

In section 4.5.5, we use unimodular row reduction to produce a solved form for a finite system of arbitrary-order non-homogeneous linear difference equations with constant coefficients over a field. These give rise to transducers which produce solutions to the original system when supplied with initial conditions and the values of constraining functions. These transducers can be viewed as time invariant linear causal maps of the linear space of sequences of vectors over the base field. Such a solution to general systems of recurrence relations does not

appear to exist in the current literature, and these transducer-based solutions for recurrences over arbitrary fields do not look like the usual exponential formulas for solutions over $\mathbb{R}$ or $\mathbb{C}$.

CHAPTER 2

UPDATE TRANSDUCERS AND AUTOMATA

2.1 Definitions

An *update transducer* (UT) is composed of a set $\mathbb{C}$ of cells, a designated subset $O \subseteq C$ of output cells, and a set $\mathcal{N}$ of input nodes (generally just 'nodes'.) Associated with every cell or node x is an associated domain $\mathcal{D}_x$. Associated with every cell c is an *update function* (or *combining function*) $\tau_c : (\prod_{d \in C} \mathcal{D}_d \times \prod_{n \in \mathcal{N}} \mathcal{D}_n) \to \mathcal{D}_c$. In the case of output cells c, the domains are constrained to cartesian products$D_c = D_i \times D_o$, where D_i is referred to as the *internal domain* and D_o is the *output domain* (no constraints are placed on D_i or D_o, however.)

The state of an update transducer is a function S assigning to each cell c an element of $\mathcal{D}_c$. The image of any node or cell under S is the *value* of the cell in the current state. If every node n is associated with a function $\sigma_n : \omega \to \mathcal{D}_n$ mapping times $\omega = \{0, 1, 2, ...\}$ to the node's domain $\mathcal{D}_n$, and a initial state S_0 is given for the transducer, then (letting cells be numbered $c_1, c_2, ...$ and similarly for nodes) we recursively define the *state S_i at time i* to assign the value $S_i(c) = \tau_c(S_{i-1}(c_1), S_{i-1}(c_2), ..., \sigma_{n_1}(i-1), \sigma_{n_2}(i-1), ...)$. Intuitively, we update the values in each of the cells by applying all update functions (each to their own cell) simultaneously to the previous values of all the cells.

We will also refer to the vector $(S_t(c_1), S_t(c_2), ...)$ as S_t or the state at time t, when the usage is clear in context. We similarly refer to the vector $I_t = (\sigma_{n_1}(t), \sigma_{n_2}(t), ...)$ as the input (to the transducer) at time t, and O_t is the vector of output values (the projection of values of output cells onto their output do-

main.)

Also, by convention, if c is an output cell, update functions of other cells only refer to the internal domain of c. The values of the output domain component of each output cell are considered the outputs of the transducer (or automaton.) The functions σ_n that assign values to the input nodes are the inputs to the transducer.

An *update automaton* (UA) is an update transducer with no inputs: $\mathcal{N} = \emptyset$.

2.1.1 Associated graph

We may associate a directed graph with any UT as follows: Each node and cell of the UA is a node in the graph, (the input nodes and output cells may be specially marked,) and there is an edge from node or cell x_1 to cell (but not node) c_2 exactly when τ_{c_2} depends nontrivially on the value of x_1.

2.1.2 On the form of update transducers

In Carta's thesis[3], transducers were similarly treated as a collection of cells with update functions, but with the constraints that there was exactly one input node and one output node for each cell, and each cell's update function could access only the value of its own input node. Also, output nodes did not update in the same fashion as cells (i.e. based on previous values of cells) but were determined by the *current* value of their associated cell. The restriction on referencing input nodes has some undesirable consequences on composition prop-

erties. Specifically, it may induce a reindexing on the input-output mappings as additional delays are added to the system. In this work, we have folded output nodes into their associated cells (now *output cells*) to retain the same instantaneous relationship, and relax the restriction on referencing input nodes to allow any cell's update function to call upon the values of any number of nodes, as well as any number of cells.

2.2 Matrix representations

In the case where the update functions of an Update Automaton are all linear, note that if S_t is an n-vector, then there is an $n \times n$ matrix M_I associated with each possible input I such that $M_I S_t = S_{t+1}$.

Focusing instead on the Transducer aspect, each possible state S corresponds to a matrix M_S, such that given an input I, the output is given as $O_t = M_S I$. Note that this probably leaves the transducer in a new state as well!

A view more connected to the UT graph is that, for each UT, there is a $(n + m) \times n$ matrix T such that if I is an m-vector and S is an n-vector, then letting IS be the $n + m$ vector formed by concatenation, the next state S' is $S' = T \times IS$. In this view, the matrix obtained from T by replacing all non-zero entries with 1 is exactly the adjacency matrix of the UT graph.

2.3 Cellular Automata

Recall that a Cellular Automaton (CA)[9] is an n-dimensional grid with identical finite state machines at each node, each of whose inputs come from a fixed neighborhood of the grid about that node. Treating CA as a type of automata network is not new[20], and in the case of update automata both the basic embedding and embeddings of various CA extensions are very natural.

Let C be $\mathbb{Z}^m$ for some dimension m, and let $Nbr(c)$ denote the neighborhood of cell c. Such neighborhoods are symmetrically preserved under pointwise translation. If every cell c shares the same finite domain $\mathcal{D}_c$, and the update function for each cell c only makes use of values of cells in $Nbr(c)$ then the update automaton specified is a cellular automaton. Essentially, a cellular automaton is an update automaton with a restriction on domains and update functions.

2.3.1 Broadened CA

There are various broadenings of the notion of cellular automata to include features such as inhomogeneous transition rules[4] or aperiodic tilings, and each of these extensions are easily realized in UA-embedded CA by relaxing some of the restrictions on update functions and what cells they may refer to.

Various boundary conditions may also be achieved. Assigning inhomogeneous transition rules to certain cells allows for an edge effect of truncating a cells' neighborhood. Altering the topology of the grid allows for periodic boundary conditions (domain tiling.) Replacing some cells with nodes, and thus embedding into an UT instead of an UA, allows for 'probe cells' whose val-

ues are determined by external conditions. Instead of an orthogonal grid, other tilings of space by cells are easily achieved by imposing different topologies on the set of nodes. Continuous state automata simply have the finite-domain restriction lifted. Continuous spatial automata similarly have the restriction of a countable domain lifted.

Probabilistic CA rules could, among other means, be implemented by augmenting the neighborhood of each node with a single input cell, again shifting the embedding substrate from an UA to an UT, which supplies a random input, used by the node's transition rule to make probabilistic choices.

2.4 Spreadsheet view

Recall that a spreadsheet, in its modern incarnation, is a computer program which presents a tabular view of data and allows each datum to be a manually input value or the result of a computation taking the data from other cells as inputs. Generally, circular references among data cells are not allowed, and (in modern implementations) results of changes are computed immediately. Compare to an UA, in which cells may be mutually referencing (arrows going both ways in the UA graph,) and the data in a cell depends on immediately previous values of other cells, not any present values. Within constraints imposed by hardware, software implementations, and design choices, the cells of a spreadsheet may have arbitrary domains, and function cells may take any number of other cells as inputs. The *Dependency Graph* of a spreadsheet is defined by taking the set of cells as nodes, and a directed edge from a to b whenever cell b contains a function which refers to cell a.

While a spreadsheet is not identical to an UA, there are marked similarities. In fact, by imposing two straightforward constraints on the design of a spreadsheet, we can directly implement many different UA, and at the same time display the UA in a form where its entire trace (history of computation) may be viewed synoptically. The first constraint is that every function in a given column must be identical, up to a relative relabeling of row references. That is, if the cell at $Row_nColumn_m$ refers to the cell $Row_kColumn_l$, then for all δ, the cell at $Row_{n+\delta}Column_m$ must use the same reference to the cell $Row_{k+\delta}Column_l$. The second constraint is that every function in a given row may only refer to cells of the previous row. That is, cells with addresses $Row_nColumn_m$ may only refer to cells with addresses $Row_{n-1}Column_l$. Since any cell in row $n-1$ may be referenced, l is unconstrained.

With these constraints in place, we can view every row of the spreadsheet as one and the same Update Transducer, just in different states. Specifically, each row displays the values of the cells of the transducer at consecutive time steps. Cells with manually entered values are input nodes. Cells with formulas are cells. The formulas in the cells are the update functions of that cell. The data in the cells are the values from the domain. The domain of a cell is implicitly defined by taking the range of the update function.

Implementing an UA or UT in this way, most spreadsheet programs would likely throw an error, because the formulas in row 0 would try to refer to cells of a nonexistent row -1. As such, the cells of row 0 should also have manually entered data, which corresponds to the initial state of the automaton or transducer.

The UT Graph may be directly recovered from the Dependency Graph G of

the spreadsheet: note that the subgraph R of G given by only considering cells in rows n and $n+1$, for $n \geq 0$, is independent of the choice of n. Also note there are no edges between cells whose row number differs by any number other than one. By taking R and identifying nodes $Row_n Column_m$ with nodes $Row_{n+1} Column_m$, the UA Graph is recovered.

2.5 Intrinsic properties

Intrinsic Properties are those which can be derived solely from the transducer mapping without needing reference to specific implementations of the transducer. Our principal concern is with *causal* transducers, those for which the output at time t is uniquely determined by inputs at times strictly less than t. We treat transducers as functions from sequences of inputs to sequences of outputs.

2.5.1 Intrinsic states

We define the relation $\sim$ (as in the Myhill-Nerode theorem[16], also denoted $\sim_0$) on the finite sequences of inputs to a transducer T. Let $\overline{a} = [a_0, a_1, a_2, ..., a_n], \overline{b} = [b_0, b_1, b_2, ..., b_m]$ be finite input sequences, and $\overline{x} = [x_0, x_1, x_2, ...]$ an arbitrary infinite input. Let $T([c_0, c_1, c_2, ...])$ denote the sequence $T(c_0), T(c_1), T(c_2), ...$ We say $\overline{a} \sim \overline{b}$ iff $\forall \overline{x}, T(\overline{ax}) - T(\overline{a}) = T(\overline{b}\overline{x}) - T(\overline{b})$. Here, $-$ is cancellation of sequences. For example, $[c_0, c_1, c_2, c_3, c_4] - [c_0, c_1] = [c_2, c_3, c_4]$. Note that as long as T is causal, it is deterministic, so $T(\overline{a})$ is an initial segment of $T(\overline{ax})$, and thus the cancelation is well-defined.

Note that $\sim_0$ is an equivalence relation. We say that the equivalence classes of $\sim_0$ are the *intrinsic states* of the transducer T.

2.5.2 k-intrinsic states

Using the notation $\overline{a}[m..k] = [a_m, a_{m+1}, ..., a_k]$ and $|\overline{a}| = n$ (for infinite sequences, we also say $\overline{x}[k..\infty] = [x_k, x_{k+1}, ...]$ and $|\overline{x}| = \infty$) we could rephrase the condition on $\sim_0$ as $T(\overline{ax})[1 + |\overline{a}|..\infty] = T(\overline{b}\overline{x}[1 + |\overline{b}|..\infty]$. More generally, under the same conditions we could require $T(\overline{ax})[k + 1 + |\overline{a}|..\infty] = T(\overline{b}\overline{x}[k + 1 + |\overline{b}..\infty]$ for an integer k, in which case we say $\overline{a} \sim_k \overline{b}$. Again, for each k it is the case that $\sim_k$ is an equivalence relation, and further $m \leq n \Rightarrow (\overline{a} \sim_m \overline{b} \Rightarrow \overline{a} \sim_n \overline{b})$. We say the equivalence classes of $\sim_k$ are the *k-intrinsic states*. Intuitively, k-intrinsic states may not look identical immediately, but given identical inputs, after at most k time steps of "clearing residual values out", their subsequent outputs are identical.

Example

Consider an update transducer with three cells, c_2, c_3, c_4, input node i_1 (marked with an incoming arrow,) and output cell c_4 (marked with a double ring.) Its graph is shown in figure 2.1. The domains are as follows: n^{in}, c_4 have letters

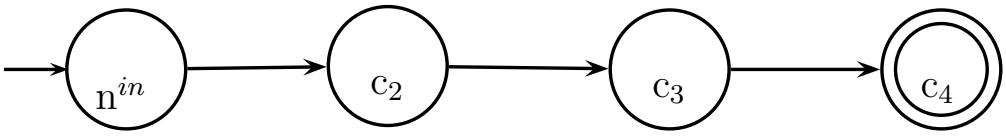

Figure 2.1: Simple substitution cipher transducer

'a' through 'z' as their domain, along with the symbol $\perp$. c_2, c_3 have integers

from 0 to 25 as their domain, as well as the symbol $\perp$. The update function for c_2 translates the letter in n^{in} into a number, $a \to 0, b \to 1, c \to 2, ...z \to 25$ (and $\perp \to \perp$.) The update function for c_3 yields the mod-26 product of 5 and the number in c_2 (if c_2 contains $\perp$, c_3 takes the value $\perp$.) The update function for c_4 performs the reverse translation of the number in c_3 into a letter, $0 \to a, 1 \to b, 2 \to c, ...25 \to z$ (and $\perp \to \perp$.)

Assume the initial state of the transducer has $\perp$ as the value for all cells.

This Update Transducer implements a transducer T with 19683 intrinsic states (corresponding exactly to number of elements of the cartesian product P of the three cells' 27-element domains) and a 27-letter input (and output) alphabet A. The states encode the process of reading a letter, changing it to numeric form, performing an operation on that number, and decoding the number back to a letter.

Even though the mapping $T : A \times P \to A$ is somewhat messy, it should be clear that if the input at time t is some letter ℓ, then regardless of the state of the machine at time t, the output at time $t + 3$ will be the mod-5 encoding of ℓ, $Crypt(\ell)$. As a result, it is the case that for all states $S_a, S_b \in P$, we have $S_a \sim_3 S_b$, and there is only one 3-intrinsic state to the machine.

There are also 27 2-intrinsic states (those which produce the letter 'a' before output $Crypt(\ell)$, those which produce the letter 'b', and so forth,) and 27^2 1-intrinsic states (which correspond to producing 'aa' before output $Crypt(\ell)$, output 'ab' first, etc, or alternately to the actual states wherein (c_1, c_2) values are $(a, 0)$ and $(a, 1)$ and so forth.)

2.5.3 ∞-intrinsic states

If it is the case that, for any infinite input sequence $\overline{x}$, $T(\overline{ax}) = T(\overline{b}\overline{x})$ at all but finitely many locations, we say $\overline{a} \sim_\infty \overline{b}$. Note that there may not be any $k < \infty$ such that $\overline{a} \sim_k \overline{b}$, as the finite delay before the outputs fall in step could depend on the particular $\overline{x}$ used and be unbounded. However, this too is an equivalence relation, and $\overline{a} \sim_k \overline{b} \implies \overline{a} \sim_\infty \overline{b}$ for any finite k. All ∞-intrinsic states are guaranteed to eventually have their outputs converge if they are given the same inputs for long enough, but there is no guarantee on how long it will be before that will happen.

2.5.4 Input-output delays

Given a specific update transducer, a quick look at the associated graph is sufficient to answer the questions "What is the least number of time steps that can pass during which an input has no impact on the transducer's outputs" and "What is the greatest number of steps that must pass after which an input can have no impact on any output". The number of edges in the shortest path from an input node to an output cell answers the first question. The number of edges in the longest path from an input to an output (which may be infinite, if the graph has any cycles) answers the second question.

For example, in figure 2.2 , suppose that the update function for each $\otimes_i$ simply takes the value of $\otimes_{i-1}$, ($\otimes_1$ takes the value of the input N^{in}, and output cell c_5 takes the value $f(\otimes_1, \otimes_4)$ for some function f. If the initial value of cell $\otimes_i$ is v_i, the initial value of c_5 is v_5, and the input node N^{in} supplies values $a_0, a_1, a_2, ...$, the successive states of the automaton are seen in table 2.1.

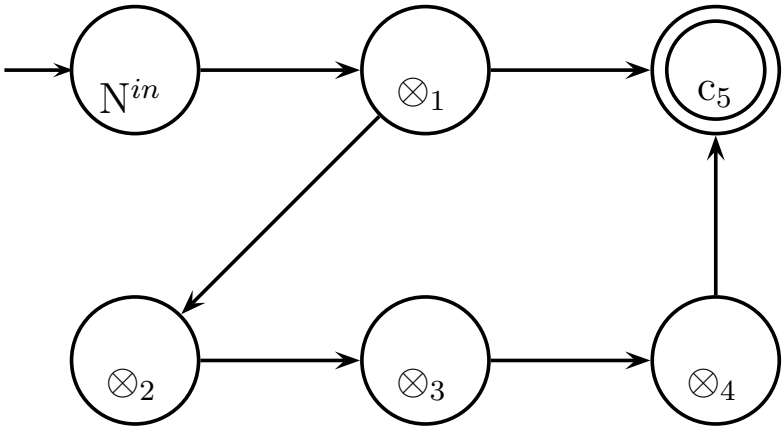

Figure 2.2: Intrinsic width example 1

We use $\otimes$ to denote a *delay line* cell, whose update function simply copies the value of another cell, making that value available to future updates of the automaton. At times, a cell may be labeled c_j and referred to as $\otimes_j$, or vice versa, if it is a delay line cell. The conflation in one direction is for uniform references to cells c_i, rather than cells c_i and $\otimes_i$, and the other is to emphasize that the cell in question is actually a delay line.

Table 2.1: Trace of values in figure 2.2 automaton

N^{in}	$\otimes_1$	$\otimes_2$	$\otimes_3$	$\otimes_4$	c_5
a_0	v_1	v_2	v_3	v_4	v_5
a_1	a_0	v_1	v_2	v_3	$f(v_1, v_4)$
a_2	a_1	a_0	v_1	v_2	$f(a_0, v_3)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_k	a_{k-1}	a_{k-2}	a_{k-3}	a_{k-4}	$f(a_{k-2}, a_{k-5})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

The shortest path from N^{in} to the output cell c_5 is of length 2, and indeed

the output never depends on any input that arrived less than 2 steps before the output was updated. The longest path is of length 5 and, similarly, no output is affected by any input that arrived more than 5 steps previously.

Similarly, in figure 2.3 , suppose the update functions are all the same, ex-

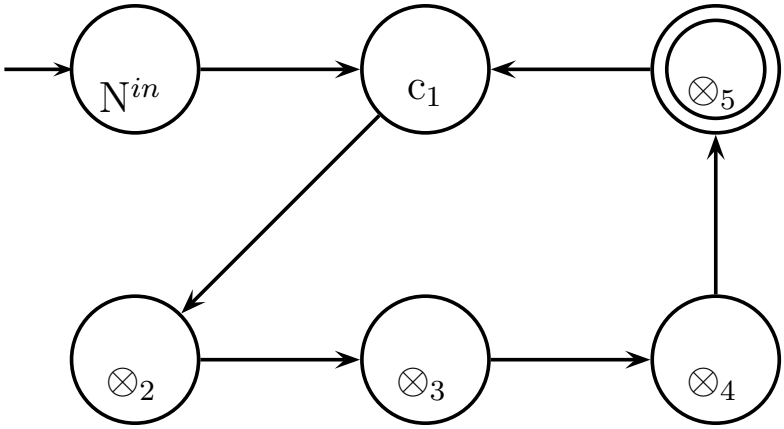

Figure 2.3: Intrinsic width example 2

cept that c_5 is changed to delay line $\otimes_5$ c_1 takes the value $f(N^{in}, \otimes_5)$. We still are interested in outputs taken from cell $\otimes_5$. The trace of values is shown in table 2.2.

In this case, the shortest path is of length 5, and no input appears in any output in less than five steps after its introduction, but now there is a cycle in the graph, and once a value has been introduced, it continues to influence the trace in perpetuity.

While these properties of the input-output map are evident from examination of the update transducer's graph, they are, like intrinsic states, properties of the input-output map and not of the particular implementation chosen. Because of this, we may define the *upper* and *lower intrinsic widths* of an input-output map.

Table 2.2: Trace of values in figure 2.3 automaton

N^{in}	c_1	$\otimes_2$	$\otimes_3$	$\otimes_4$	$\otimes_5$
a_0	v_1	v_2	v_3	v_4	v_5
a_1	$f(a_0, v_5)$	v_1	v_2	v_3	v_4
a_2	$f(a_1, v_4)$	$f(a_0, v_5)$	v_1	v_2	v_3
a_3	$f(a_2, v_3)$	$f(a_1, v_4)$	$f(a_0, v_5)$	v_1	v_2
a_4	$f(a_3, v_2)$	$f(a_2, v_3)$	$f(a_1, v_4)$	$f(a_0, v_5)$	v_1
a_5	$f(a_4, v_1)$	$f(a_3, v_2)$	$f(a_2, v_3)$	$f(a_1, v_4)$	$f(a_0, v_5)$
a_6	$f(a_5, f(a_0, v_5))$	$f(a_4, v_1)$	$f(a_3, v_2)$	$f(a_2, v_3)$	$f(a_1, v_4)$
a_7	$f(a_6, f(a_1, v_4))$	$f(a_5, f(a_0, v_5))$	$f(a_4, v_1)$	$f(a_3, v_2)$	$f(a_2, v_3)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

2.5.5 Intrinsic width and locking inputs

Let $\overline{a}$ be any finite sequence of inputs, and T a causal transducer. Let $\overline{x}_1, \overline{x}_2$ be infinite sequences of inputs. If $T(\overline{a}\overline{x}_1) = T(\overline{a}\overline{x}_2)$, we say that $\overline{x}_1 \sim_{\overline{a}} \overline{x}_2$ ($\overline{x}_1$ is $\overline{a}$-equivalent to $\overline{x}_2$.) Note that for all $\overline{a}$, $\sim_{\overline{a}}$ is an equivalence relation.

Let $\Sigma^\star$ be the set of all infinite sequences of outputs for T. Consider the map $S : \Sigma^\star \times \Sigma^\star \to \mathbb{N} \cup \infty$ where $S(\overline{x}_1, \overline{x}_2) \leq min(|\overline{x}_1|, |\overline{x}_2|)$ is the greatest integer k such that $T(\overline{x}_1)[0..n] = T(\overline{x}_2)([0..n])$ (i.e. $T(\overline{x}_1)$ and $T(\overline{x}_2)$ share an initial segment of length k, and no more.) If $\overline{x}_1 \sim_\emptyset \overline{x}_2$, ($\emptyset$ denotes the empty sequence) then since $T(\overline{x}_1) = T(\overline{x}_2)$, we say $S(\overline{x}_1, \overline{x}_2) = \infty$. Note that S is nonnegative.

For a fixed finite sequence of inputs $\overline{a}$, let $L_{\overline{a}}$ denote the minimum of $S(\overline{a}\overline{x}_1, \overline{a}\overline{x}_2) - |\overline{a}|$ over all pairs of infinite extentions $\overline{x}_1, \overline{x}_2$. That is, if $\overline{a}$ has been

given to the (freshly initialized) transducer, then it will take be at least $L_{\overline{a}}$ time steps before any difference in subsequent outputs will be observed. If $L_{\overline{a}} = \infty$, then all inputs after $\overline{a}$ produce the same result. Call such an $\overline{a}$ a *locking input*.

2.5.6 Upper and lower intrinsic widths

For a given transducer T, let LW_T denote the minimum of $L_{\overline{a}}$ over all $\overline{a}$. Call LW_T the lower intrinsic width. LW_T corresponds to our earlier notion found by the shortest path through update transducer's graph, since no inputs have any impact on the output before LW_T time steps have passed.

Let UW_T be the least k such that there is exactly one k-intrinsic state of the transducer. If there is no such finite k we say $UW_T = \infty$. If $UW_T = k < \infty$, then no matter what inputs the transducer has had in the past, they do not change outputs more than k steps in the future, which corresponds to our earlier notion found by the longest patch through the graph. Note that even if $UW_T = \infty$, this does not imply that there is a single ∞-intrinsic state (though that is possible) but rather there is no fixed bound on how long an input can continue to influence outputs (even if, for any given finite input sequence and infinite continuation, there is a bound, as in the case of a single ∞-intrinsic state.)

If it is the case that $UW_T = LW_T$, we will simply refer to *the* intrinsic width W_T.

Example

Returning to the *Crypt*() transducer in figure 2.1, we calculate the intrinsic widths.

Let l_i denote the ith letter of the alphabet. Suppose $\overline{a} = [l_1, l_2, l_3, l_4, l_5], \overline{x}_1 = [l_6, l_7, l_8, ...], \overline{x}_2 = [l_9, l_{10}, l_{11}, ...]$. We have

$$T(\overline{ax}_1) = [\perp\perp\perp\ l_5, l_{10}, l_{15}, l_{20}, l_{25}, l_4, l_9, l_{14}, ...]$$

and

$$T(\overline{ax}_2) = [\perp\perp\perp\ l_5, l_{10}, l_{15}, l_{20}, l_{25}, l_{19}, l_{24}, l_3, ...]$$

We have $S(\overline{ax}_1, \overline{ax}_2) = 8$ (the sequences match on their first eight entries, and no more) so $L_{\overline{a}}$ is at most 3, since $S(\overline{ax}_1, \overline{ax}_2) - |\overline{a}| = 3$.

A little thought reveals it is exactly 3, and that this number does not depend on choices of $\overline{a}$ or $\overline{x}_i$, and thus we have $LW_T = 3$, since $L_{\overline{a}}$ is constant-valued (at 3) for all $\overline{a}$. Furthermore, as noted before there is only one 3-intrinsic state, but 27 (one for each letter, plus $\perp$) 2-intrinsic states, so $n = 3$ is the least number for which we have just one n-intrinsic state, so we have $UW_T = 3$ as well. With these two being the same, we say that $W_T = LW_T = UW_T$, or that the automaton has intrinsic width of three.

While it would be possible to construct a single-celled, twenty-thousand state update transducer to implement T, the 'natural' size of T (and the size as implemented by the UT originally described) is three, the graph width of the original UT.

Example

We will also look at an example involving Newton's Method for finding roots, but will first point out that the treatment given to this numerical approximation is a specific case of a general class of contraction problems, of which the Picard Existence Theorem for Differential Equations (see, for example, [6]) is another instance, and which are treated extensively in [14]. Since the domains of the cells are arbitrary, we could as well have update functions performing an integration of a given function to yield a new function (the domains being functions spaces) as we could have an update function yielding the value of a function and its derivative at a given point.

Given a function $f : \mathbb{R} \to \mathbb{R}$, let $N_f(n, x)$ denote the nth iteration of Newton's Method for finding roots, starting from initial approximation x. Consider the transducer $T_f : (\mathbb{N}\times\mathbb{R})^\star \to (\{\bot\}\cup\mathbb{R})^\star$ which, given an input $\overline{a}$ where $\overline{a}[k] = (n_k, x_k)$ with $n_k \in \mathbb{N}$ and $x_k \in \mathbb{R}$, produces an output $T_f(\overline{a})$ satisfying

$$T_f(\overline{a})[k+1] = \begin{cases} \bot & \text{if } n_k > k \\ N_f(k - n_k, x_{n_k}) & \text{if } k \geq n_k \end{cases}$$

If $(\cdot, x)$ is given to the transducer at time t, and $(t, \cdot)$ is given to the transducer at time t', then either t' is earlier than t, in which case the output at t' is $\bot$, or it is later, and the output will be a root-approximation derived from applying Newton's Method $t' - t$ times.

For generic initial segments $\overline{a}$ and generic distinct infinite sequences $\overline{x}_1, \overline{x}_2$, $S(\overline{a}\overline{x}_1, \overline{a}\overline{x}_2) = |\overline{a}| + 1$. By generic, we mean that the real parts are pairwise distinct, and pairwise do not generate the same Newton's Method sequence after finitely many steps, and whose second (time) parts do not call for "future" times. The

equation $S(\overline{ax}_1, \overline{ax}_2) = |\overline{a}| + 1$ arises since the genericity conditions imply that $T_f(\overline{ax}_1)[|\overline{a}| + 1] = T_f(\overline{ax}_2)_{|\overline{a}|+1}$ is the last time step where equality is guaranteed.

From that, $L_{\overline{a}} = (|\overline{a}| + 1) - |\overline{a}| = 1$ follows immediately, and is independent of $\overline{a}$, so the lower intrinsic width is $LW_T = 1$.

On the upper end, however, note that an input at time t of $(0, \cdot)$ will always produce an output at the next update whose value depends very clearly on the first input given to the transducer, no matter how large t is. Indeed, since the point of approximation methods is that later iterations produce better results, it would be expected that t would be relatively large when $(0, \cdot)$ was supplied. Though, since a major feature of Newton's method is that it converges very rapidly (if at all) t would not necessarily be all that large after all. Since every input can end up being relevant to any output, given any generic initial sequence $\overline{a}$ it would be unlikely that any generic extensions $\overline{x}_1, \overline{x}_2$ would ever result in $T(\overline{ax}_1) = T(\overline{ax}_2)$ at all but finitely many locations, to say nothing about *every* such extention, so there is certainly no k such that the transducer has only one k-intrinsic state. The upper intrinsic width is ∞ (and this is a case where we do not even have an ∞-intrinsic state.)

As it has different upper and lower intrinsic widths, we do not say that this transducer has an intrinsic width.

Note that if this were implemented as an UT with real or integer domains[1], it would require infinitely many cells in which to store and update the arbitrarily many initial guesses, but the minimum causal path length could still be arranged as in figure 2.4 such that the path length from an input node to an out-

[1]On the other hand, if implemented with domains which could contain arbitrary vectors of real numbers, it could be accomplished with a single cell, but a single cell with very involved values!

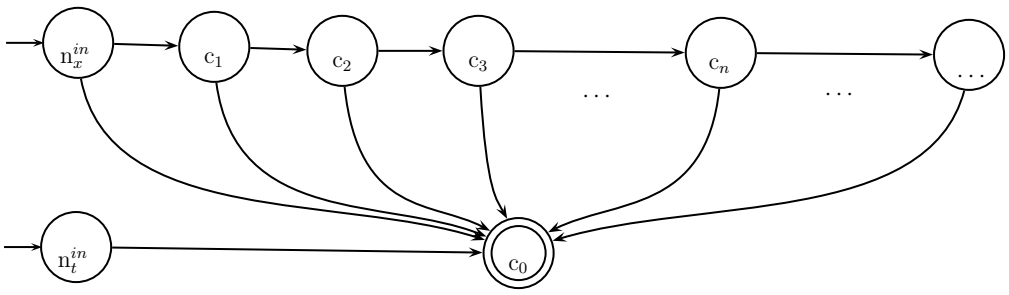

Figure 2.4: An update transducer for Newton's Method

put cell was just 1 (either input to c_0.) The update rules for the graph in figure 2.4 would be $c_i = N_f(1, c_{i-1})$ (apply Newton's Method to the value of the cell to the left) for $1 \leq i$ (with the obvious modification for c_1) and c_0 would update by copying the value in cell c_t, where t was the value read from n_t^{in}.

2.6 Composition and collapsing

Suppose we have two update transducers, T_1 and T_2, where the outputs of T_1 match the inputs of T_2 in number and their domains. For example, let T_1 consume letters of the alphabet $a, b, ..., z$ and produce their index in the alphabet, $1, 2, ..., 26$, and let T_2 perform the inverse operation. Each of T_i could be accomplished with a single input node n_i and a single output cell c_i. For T_1, the domain of n_1 is A, the alphabet, the domain of c_1 is $\mathbb{Z}_{26}$, and the update function for c_1 takes value $indexOf(n_1)$. For T_2, the domains are switched, and the update function is simply $letterWithIndex(n_2)$.

T_1 may be composed with T_2 (or vice versa) by choosing the input sequence for n_2 to be given by the outputs of c_1 (or n_1 taking outputs from c_2) and, if one pays no attention to *when* outputs arrive, the result will simply be an identity

transformation. However, each of the transducers individually has an intrinsic width of 1, and the composed transducer has an intrinsic width of 2, given initial values $c_1^{init} = 1, c_2^{init} = a$, an input sequence of $z, y, x, w, v, u, t, s, ...$ would result in an output sequence $a, a, z, y, x, w, v, u, ...$ In general, it is the case that the lower (resp. upper) intrinsic width of a composed transducer is the sum of the lower (resp. upper) intrinsic widths of the transducers being composed.

In every case, one could reduce a composed transducer to a single-cell transducer whose outputs, when properly reindexed, would be equal to the original at all but finitely many times. For example, the one-node, one-cell transducer T_3 where both n_3 and c_3 have domain A and the update rule is simply $c_3 = n_3$ would, given initial value $c_3^{init} = a$, turn the input sequence $z, y, x, w...$ into output $a, z, y, x, ...,$ which is *shift-equivalent* to the above composition of T_1 with T_2.

In most cases, however, such a reduction has three major failings. First, it reduces the entire procedure to a single 'black box', so is unlikely to provide any insight into the process being modeled. Second, if there were constraints on the spaces from which update functions could be selected for the uncollapsed version, those constraints could be violated by the update function for the collapsed transducer. Third, the domain of the collapsed transducer's cell might have to expand substantially in order to track all the history and side-computations that could be involved in the various cells of the uncollapsed transducer.

In some cases, though, shift -equivalent transducers with lower (if not minimally low) intrinsic width can be produced without hitting these problems. The alphabet-index transducers T_1 and T_2 above, for example, lose very little by being collapsed to T_3 with its single update function $c_3 = letterWithIndex(index - of(n_3)) = n_3$. A more general case is when we have two cells in an update trans-

ducer which are adjacent in the transducer's graph, removing that edge would disconnect the graph, and the update functions for the cells come from a domain which is closed under composition. In this case, replacing the two cells with a single cell whose update function is the composition of the original update functions will reduce the intrinsic width without introducing obfuscation.

This kind of procedure cannot, however, be blindly applied. If there are other cells v_1, v_2 such that the edge between our cells of interest lies on some paths from v_1 to v_2 but not on all such paths, then replacing our cells with their composition could change the length of some (and not all) of the paths from v_1 to v_2, which would likely disrupt the timing of when particular values are combined.

For example, consider transducers T_1, T_2, T_3 , each with intrinsic width 1, which perform the following mappings. T_1 takes in input x in $\mathbb{Z}_{15}$ and, at the next time step, produces two outputs, $y_1 \in \mathbb{Z}_3$ and $y_2 \in \mathbb{Z}_5$, where $x \equiv y_1 \ mod\ 3$ and $x \equiv y_2 \ mod\ 5$. T_2 takes two inputs, x_1, x_2, and produces two identical outputs $y_1 = x_1, y_2 = x_2$. T_3 performs the inverse operation of T_1, taking to inputs x_1, x_2 and producing an output $y \in \mathbb{Z}_{15}$ where $y \equiv x_1 \ mod\ 3$ and $y \equiv x_2 \ mod\ 5$. The composition of these is shift-equivalent to an identity map, with intrinsic width of 3. See figure 2.5

If we now collapse the two cells on the lower branch (figure 2.6 to a single cell c_c with update function $c_c = c_1 \ mod\ 5$, the intrinsic width of the transducer is no longer well defined, but it does have lower intrinsic width of 2 and upper intrinsic width of 3. This immediately indicates that, unlike the original transducer, the outputs may depend on the values of multiple inputs (any inputs that arrived between 2 and 3 steps in the past) rather than the single input which ar-

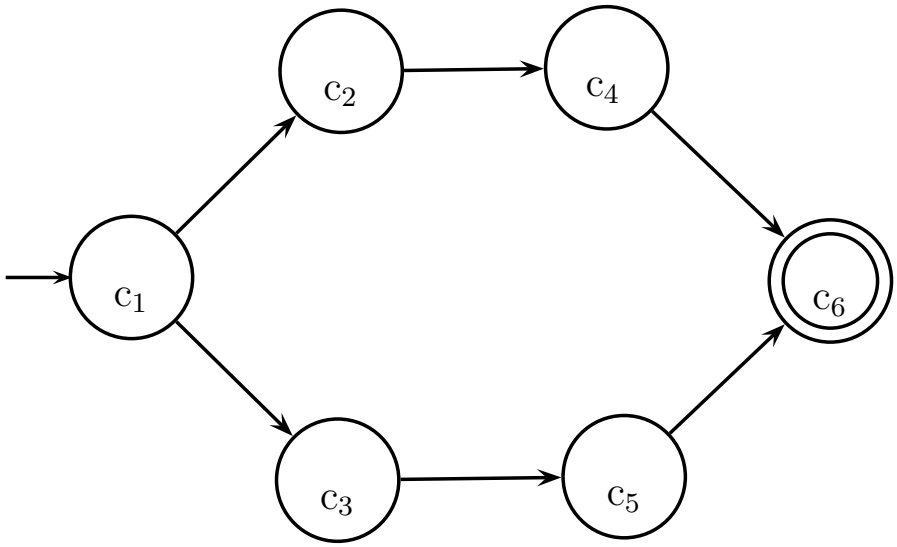

Figure 2.5: 3-wide identity map

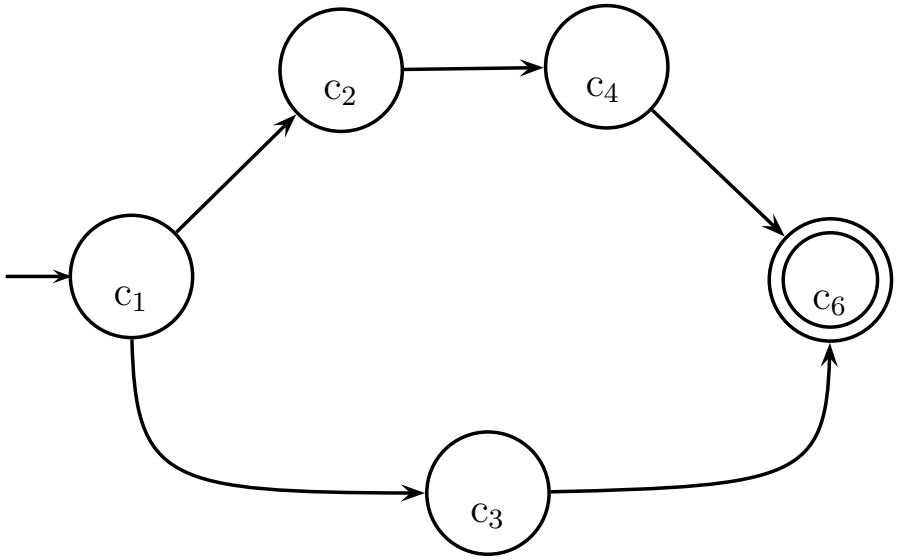

Figure 2.6: Incomplete collapse of figure 2.5: upper width 3, lower width 2

rived 3 steps ago. Indeed, with all cells given initial values of 2, and an input sequence

$$1,1,1,1,1,\ \ 2,2,2,2,2,\ \ 3,3,3,3,3,\ \ 2,2,2,2,2,,...$$

the output sequence for the partially collapsed transducer is

$$2, 11, 1, 1, 1, \quad 1, 7, 2, 2, 2, \quad 2, 8, 3, 3, 3, \quad 3, 12, 2, 2, 2, ...$$

whereas the output sequence for the uncollapsed transducer would be

$$2, 2, 1, 1, 1, \quad 1, 1, 2, 2, 2, \quad 2, 2, 3, 3, 3, \quad 3, 3, 2, 2, 2, ...$$

In the uncollapsed transducer, when the last cell is updating based on information in the upper branch that came from an input at some time t, it is updating based on information in the lower branch based on input from time $t + 1$, and when the inputs at those times were not the same, the output produces unexpected values.

In this simple case, simultaneously collapsing both branches at once would address the timing mismatch, and as with the alphabet-index example, collapsing the entire thing to a single node does not reduce the transparency of the operation, but it is sufficient to illustrate that a naive approach that collapses cells locally is insufficient for reducing to shift-equivalent transducers with reduced intrinsic width.

2.7 Nondeterministic update automata (NDUA)

In some instances, it may be of interest to use a one-to-many update *map* for one or more cells, rather than a one-to-one update *function*. A language recognition transducer, for instance, may nondeterministically produce any of the symbols b, c, d upon consuming symbol a. A general Nondeterministic Update Automaton (or Transducer) is identical to an Update Automaton (resp. Transducer) with

the exception that each cell c is equipped with an update *relation* τ_c where $\tau_c(a, b)$ exactly when b is a potential value for the cell based on inputs a.

We do not explore this avenue deeply in this thesis: our primary concern is with causal (and therefore deterministic) maps. Nonetheless, we will mention a few means of addressing the nondeterministic case, and note that discussions relying on determinism (such as the definition of intrinsic state) may still be relevant with an appropriate application of quantifiers over all possible runs of an NDUA.

2.7.1 Simulating NDUA with UA

Since UA are finite as control structures, but can be controlling arbitrary targets, the introduction of nondeterminism should not add any additional power. We may see this by modifying only the domains and update functions of a given NDUA to make it an UA, without altering its associated graph.

Let N be a given NDUA, let R be a semiring, and let ϕ be a function assigning an element of the semiring to related pairs of inputs and values (So, in a two-cell automaton, if an integer-domain cell c_1 has update relation $\{((x, y), x + y), ((x, y), x - y) : x, y \in \mathbb{Z}\}$ and the other cell c_2 has the functional update relation $\{((x, y), x * y) : x, y \in \mathbb{Z}\}$, then the domain of ϕ would be $\{((x, y), x + y), ((x, y), x - y), ((x, y), x * y) : x, y \in \mathbb{Z}$. We construct an UA D as follows. The cells of D are exactly the cells of N, and the input nodes are the same as well. If c is a cell of N with domain $\mathcal{D}_c$, the domain of c in D is the power set of $R \times \mathcal{D}_n$. For each cell c, if $(x, d), (x, e), (x, f), (y, d), (y, e)$ are in the transition relation for N, the update function in D will map $(r, x), (s, y)$ to

$(r \cdot \phi(x,d),d),(r \cdot \phi(x,e),e),(r \cdot \phi(x,f),f),(s \cdot \phi(y,d),d),(s \cdot \phi(y,e),e)$

In essence, each of the possible nondeterministic choices are given (not necessarily unique) names in r, the product of the 'names' of all choices is carried along with the values that result from those choices, and all choices are concurrently explored.

In particular, if S is the set of all related input-value pairs, R the noncommutative ring of formal products and sums of elements of S, and ϕ the identity map, then the names given to each choice are distinct, and the product of names is simply a list (in order from left to right) of the choices previously made.

2.7.2 Probabilistic Update Automata

Rather than full nondeterminism, it may be of interest to work with automata whose transitions are, while not deterministic, chosen with known frequencies. Here we use $\mathbb{R}$ as our semiring, and impose the condition that for every update function, the sum of the names used is 1. The value in each cell now becomes a weighted formal sum $p_1v_1 + p_2v_2 + p_3v_3 + ...$ of values v_i that the associated NDUA could be displaying, where p_i is the probability that the particular transitions taken would have resulted in v_i.

CHAPTER 3

EQUIVALENCE

We have already seen that different update transducers can implement the same input-output mapping up to shift-equivalence, and even if no reindexing of outputs is allowed, it is not hard to come up with distinct transducers with the exact same input-output behavior. As update automata are labeled state transition machines, the notion of bisimulation arises as a natural candidate for describing equivalence between update transducers. At the most fine-grained level, the state of an update automata may be taken as the tuple of values of each of its cells (including, for a transducer, both the internal and output values of an output cell.) Similarly, every tuple of inputs is a label for a transition from some state to the state obtained by performing an update step.

3.1 Bisimulation

For an update automata A, let $U_i : inputs(A) \times states(A) \rightarrow states(A)$ denote the update function, producing new states based on the previous states and inputs. We say there is a *bisimulation* between two update automata A_1, A_2 if their input nodes are equal in number and have matching domains (i.e. $inputs(A_1) = inputs(A_2)$,) and there is a relation R on $states(A_1) \times states(A_2)$ such that for all inputs $I \in \mathcal{I}(A_i)$ and states $S_1 \in states(A_1), S_2 \in states(A_2)$, it is the case that $R(S_1, S_2) \implies R(U_1(I, S_1), U_2(I, S_2))$. Simply put, if the two automata are in related states, then updating them with any input will result in related states. [1]

[1]Note that in the case of nondeterministic update automata, where $U_i(I, S_i)$ has a variety of possible values, we require that if $R(S_1, S_2)$, then for each state S_1' of $U_1(I, S_1)$ there is a state S_2' of $U_2(I, S_2)$ with $R(S_1', S_2')$ and, vice versa, that for every state S_2' that an update of the second

We say that two states S_1, S_2 are *bisimilar* if a bisimulation R exists with $R(S_1, S_2)$, and that two update automata are bisimilar if for every state of one, there exists a bisimilar state for the other, and vice versa (no matter how one machine is configured, the other can be set up to match it.)

Note that this notion does not, as given, imply that bisimilar transducers will exhibit the same input-output mappings. A single-celled automaton A_1 whose cell updates to 0 if its input reads $\perp$, and which updates to 1 otherwise, and a single-celled automaton A_2 whose cell updates to 1 if its input is $\perp$ and which updates to 0 otherwise, are transparently bisimilar, but produce complimentary outputs given the same input. There does, however, exist an invertible function between outputs of A_1 and outputs of A_2 (namely, $f(x) = 1 - x$) such that, up to translation through that function, the outputs are identical.

In a deterministic update transducer, since the sequence of states is uniquely determined by the initial state and the inputs, and the output sequence is determined uniquely by the sequence of states, it might seem like such a function should always exist. However, the following counterexample shows that, even in the deterministic case, there's not necessarily an invertible function:

Let A_1 and A_2 have cells c_1, c_2 for A_1 and d_1 for A_2. The domains for the c_i are $\mathbb{Z}_3$, and for d_1 is $\mathbb{Z}_2$. The update functions are $c_2 = c_1, c_1 = 2c_2 \ mod\ 3, d_1 = (d_1 + 1) \ mod\ 3$. The output of A_1 is the value of c_1 and the output of A_2 is the value of d_1. Since no external inputs are used, we will omit the input term from the update function U_i. We have $states(A_2) = \{(0), (1)\}$, and $states(A_1) = \{(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)\}$. The relation $R = \{((0), (0,n)), ((1), (n,0)) | n \in \{1,2\}\}$ is a bisimulation, $((U_2(0), U_1(0,n)) =$

machine could produce, there is a state S_1' of the first machine that can result from the same input.

$(1, (\overline{n}, 0)) \in R$ for $n = 1, \overline{n} = 2$ or $n = 2, \overline{n} = 1$, and $(U_2(1), U_1(n, 0)) = (0, (0, \overline{n})) \in R$ for $n = 1, \overline{n} = 2$ or $n = 2, \overline{n} = 1$.)

Although R is a bisimulation, note that if the A_1 is started with initial state $(0, 1)$, it produces outputs $0, 2, 0, 1, 0, 2, 0, 1, ...$ while A_2 with initial state (0) produces $0, 1, 0, 1, 0, 1.....$ Certainly, in this case, there is a function from outputs of A_1 to outputs of A_2 under which the sequences are equal, but it is not invertible. Furthermore, A_1 and A_2 could be combined and augmented into A_1', A_2' such that a similar (but modified) bisimulation existed but which produced runs $0, 2, 0, 1, 0, 1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, ...$ from A_1' and $0, 1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, 0, 2, 0, 1, ...$ from A_2', ensuring there was not even a function in one direction. Granted, there are still visible similarities between the outputs, but clearly the basic notion of bisimulation is insufficient.

3.2 IO equivalence

One approach to reconciling bisimulation with input-output equivalence would be to redefine the notion of 'label'. Rather than treating inputs as labels, we could treat $input : output$ pairs as labels. This would ensure that bisimilar automata implemented identical (no shift-equivalence allowed) input-output maps, though it would not *a priori* guarantee that automata which implemented identical input-output maps were bisimilar.

This leads us to focus on the input-output map, rather than on the states of a particular implementation, which brings us back to the notion of intrinsic states. Recall that two finite sequences of inputs σ_1, σ_2 place a transducer in the same intrinsic state if, when supplied with identical inputs, their outputs are identi-

cal, in which case we say $\sigma_1 \sim_0 \sigma_2$. Moreover, for all finite sequences ρ, it is the case that $\sigma_1 \sim_0 \sigma_2 \implies \sigma_1\rho \sim_0 \sigma_2\rho$. Without reference to any particular implementation, we consider bisimulation on the level of intrinsic states, returning for the moment to labels being inputs (not coupled with outputs.)

Suppose that T_1, T_2 are transducers, $states(T_i)$ is the set of intrinsic states of T_i, $inputs(T_i)$ are the inputs, and $U_i(S, I) \mapsto S'$ encodes the transitions between intrinsic states. As our notion of intrinsic state becomes difficult to apply if a transducer's outputs are not determined by its inputs, we will assume that the T_i are deterministic. Now, if S_1, S_2 are bisimilar states, then while the outputs associated with $U_1(S_1, I)$ and $U_2(S_2, I)$ may be different, they are each determined by I.

Considering the counterexample above, note that even though A_1 and A_2 used no inputs, with the given initial conditions they had 4 and 2 distinct intrinsic states, respectively. A_1's intrinsic states s_1 corresponded to "subsequent outputs will be $1, 0, 2, 0, 1, 0, 2, 0, ...$", state s_2 was "subsequent outputs will be $0, 2, 0, 1, 0, 2, 0, 1, ...$, s_3 led to $2, 0, 1, 0, 2, 0, 1, 0, ...$ and s_4 to $0, 1, 0, 2, 0, 1, 0, 2, ...$ These also corresponded to actual implementation states $(1, 0), (0, 1), (2, 0)$, and $(0, 2)$ respectively. A_2 had two intrinsic states: s_5 corresponding to $0, 1, 0, 1, ...$ and actual state (1), and s_6 to $1, 0, 1, 0, ...$ and (0). We are now in position to translate the bisimulation relation on actual states to one on intrinsic states: $R = \{(s_6, s_2), (s_6, s_4), (s_5, s_1), (s_5, s_3)\}$, and note that multiple intrinsic states of A_1 are being related to the same intrinsic state of A_2. s_2 and s_4, for example, are different intrinsic states, and as such there are input sequences (in this case, all (empty) input sequences) which result in distinct outputs. When we imposed the labeling based on input:output pairs, rather than simply inputs, that

amounted to restricting the bisimulation relation on intrinsic states to a single-valued map.

3.3 $\delta - \epsilon$ equivalence

Suppose we have a family of transducers which common input and output domains, and we have a metric on sequences in those domains. For example, let the domains be real vector spaces such as humidity/temperature/pressure inputs and control voltage outputs, and the transducers implement some desired control, such as an HVAC (Heating, Ventilation, Air Conditioning) system applying homeostatic regulation to a building's climate based on a variable user-defined set-point. In such a setting, there may be a single ideal input-output map which all HVAC controllers aim to implement, at least to within a given tolerance level. Demanding exact reproduction of the target mapping would likely be infeasible, so we only ask for near-equivalence. That is, given a tolerance ϵ, there should be a sequence of input readings that, if they are sufficiently close (within some δ depending on ϵ) to specified standard inputs, will produce outputs (control signals) which are within ϵ of the target 'ideal output'. We assume that the reference mapping and the system being compared are in the same intrinsic state, or at least in intrinsic states that are sufficiently close, in the sense that if the reference system initialized in one state were compared to the reference system initialized in the other state, all identical inputs would produce outputs which remained within some sufficiently small distance of each other for a sufficiently long time. The particular thresholds determining 'sufficiently' would depend on design specifications.

3.3.1 Thévenin equivalence

Consider that when the values of input nodes in one transducer are determined by the values in output nodes (with matching domains) of another transducer, the result is another transducer: networks of update transducers are themselves update transducers. And, conversely, update automata and transducers are (sometimes trivial) networks of update transducers. Given an update automaton A, one may look at its graph, select any subset A_r of its cells, and then separate that subset and its complement, A_l, into two update transducers T_r and T_l. The cells of T_r are exactly the cells of A_r. Every cell in A_r which is referenced by one or more update functions in A_l is an output node with output domain the same as its internal domain, and output value equal to its internal value. For every cell c_l which is referenced by an update function in one or more cells of A_r, the transducer T_r also has an input node n_{c_l}. The update functions are identical, save for replacing any references to cells c_l in A_l with references to input node n_{c_l}, and the modification for output nodes to put the same results in both internal and output values. The construction for T_l is entirely symmetric. The result is two transducers which, when connected in the obvious way, exactly reconstruct the original automaton (or transducer.)

Recall that for any update transducer there is a single-cell transducer which has an identical input-output mapping: take the domain of the cell to be the cartesian product of domains of all the original transducer's cells, and make the update function be defined component-wise by using the appropriate update function from the original, modified to read a state-tuple from the single cell's value, rather than referencing separate states from each of the original cells. Since the original transducer may have had multiple input and/or output

nodes, we must use the more general notion of update automata (a 'proper' one-celled input transducer would only have one input node and one output node.) Since the compressed single-cell transducer retains the same input-output mapping, it has the same intrinsic states and intrinsic widths as the original.

Combining these two points, we may take any update automaton, select some set of its cells, factor the automaton into two transducers corresponding to that set of cells and its complement, compress one of the transducers to a single cell with the same IO mapping, and then reconnect them to reconstruct an automaton with the same input-output behavior, but a different graph. We call this a collapsing Thévenin replacement, as this process is reminiscent of replacing a subcircuit of an electrical circuit with a Thévenin equivalent component[13] whose behavior at its terminals is indistinguishable from the original subcircuit.

This sort of replacement may hide more about the process being executed than it reveals, but there is nothing explicitly preventing the opposite, an expanding Thévenin replacement, from being done instead. While we leave for the future the subject of general algorithms for expanding a single cell transducer (essentially, an input-output specification,) do note that the above process is entirely symmetric if such an expansion is available. Given a transducer T that is decomposed into a single cell transducer and its complement, and given another transducer with the same input-output map as the single celled one, we may make the replacement and reconstruct an expanded transducer T' with the same behavior as T. Indeed, this process can be iterated, reducing the problem of meeting one IO specification to the problem of meeting two (or more) potentially simpler ones, and then reducing any of those to even more presumably simpler specifications.

3.4 Domain collapsing

Recall the width-4 transducer in figure 2.5 which took a number in $\mathbb{Z}_{15}$, split it into its equivalents mod 3 and mod 5, delayed these values, and used the Chinese Remainder Theorem to recombine them into the original number. Suppose we modified this to, in the end, be a transducer from $\mathbb{Z}$ to $\mathbb{Z}_{15}$, and did so by changing the domains of c_1, c_2, c_3 to $\mathbb{Z}$ (as well as the input to c_1,) making c_4 and c_5 perform the appropriate mod operation (3 or 5) on c_2 and c_3, respectively, and set c_2 and c_3 to be delay lines taking their value from c_1. The graph remains the same, and the end result almost the same, but now half the cells are using superfluously infinite domains. Note that we could have achieved the same result by having only the input domain be $\mathbb{Z}$, and then having c_1 update by taking the input mod 15.

By performing a collapsing Thévenin replacement on c_2 and c_3, and then another replacement on c_4 and c_5, we can get a four-celled transducer (figure 3.1) in which the domain of c_{23} is $\mathbb{Z} \times \mathbb{Z}$, the domain of c_{45} is $\mathbb{Z}_3 \times \mathbb{Z}_5$, the update func-

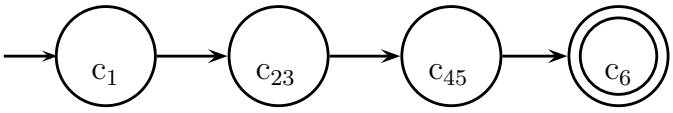

Figure 3.1: A collapsing Thévenin replacement

tion for c_{23} produces the value (c_1, c_1), the update function for c_{45} takes the value $(c_{23}[left] \ mod \ 3, c_{23}[right] \ mod \ 5)$, and the update function for c_6 now applies the Chinese Remainder Theorem to $c_{45}[left]$ and $c_{45}[right]$ instead of c_4 and c_5.

There are some rather poor domain choices here, given that this all could be done with four cells, all sharing domain $\mathbb{Z}_{15}$, and the update rules all being

delay lines except for the first, which took the input mod 15. While c_{45}'s domain is isomorphic to $\mathbb{Z}_{15}$, the domains of c_1 and c_{23} are not. They are, however, collapsible in the following sense:

Let T be the four-cell $(c_1, c_{23}, c_{45}, c_6)$ transducer with poor domain choices, and let T' be the four-cell $(c'_1, c'_{23}, c'_{45}, c'_6)$ transducer with uniform domains of $\mathbb{Z}_{15}$. Suppose the values in the cells of the transducers are such that the transducers are in the same intrinsic state. Let $i \in \{1, 23, 45, 6\}$ and consider the two player game in which player one selects either c_i or c'_i and changes its value (to something else in the domain of that cell, even if it could not usually arise in from an update) and player two tries to change the value of the cell not selected such that the two transducers are once again in the same intrinsic state. If player two cannot do so, the game terminates and player one wins. If the game does not terminate, player two wins.

Note that, regardless of the initial state the machine is in, and regardless of which i the game is played on, player two has a winning strategy. For cell c_1 and c'_1, it is sufficient to set a value such that $c'_1 \equiv c_1 \ mod \ 15$, which is always possible. If player one set the value of cell c'_{23}, then player two sets $c_{23} = (c'_{23}, c'_{23})$. If player one set the value of cell c_{23} as (n, m) then player two sets c'_{23} according to the Chinese Remainder Theorem such that $c'_{23} \equiv n \ mod \ 3, c'_{23} \equiv m \ mod \ 5$ (note that the transducer would only update c_{23} such that $n = m$, but even 'illegal values' with $n \neq m$ can be handled.) The strategy for cells c_{45} and c'_{45} is essentially the same as for c_{23} and c'_{23}, and cells c_6 and c'_6 should be set equal.

Player two's winning strategy induces an equivalence relation on the joint domain $D_{c_i} \cup D_{c'_i}$ under which the input-output maps are well defined on equivalence classes. Specifically, in this case, this gives a function which collapses

each of the domains of T (especially the poorly chosen ones) to $\mathbb{Z}_{15}$. In this example, another transducer with a well chosen domain was provided, but note that this works even when the game is played with $T' = T$: if player one changes the value of c_1 to 17, player two follows essentially the same winning strategy to set c_1' (which now has the same domain as c_1, since we are using $T' = T$) to $2 = c_1 \ mod \ 15$, and that winning strategy will induce the equivalence relation $n \sim m \iff n \equiv m \ mod \ 15$. The equivalence classes are simply $\mathbb{Z}_{15}$. Similarly, for c_{23}, the winning strategy is, given (n, m), pick (n', m') such that $n' \equiv n \ mod \ 3$ and $m' \equiv m \ mod \ 5$, and the equivalence classes are $\mathbb{Z}_3 \times \mathbb{Z}_5$, which is isomorphic to $\mathbb{Z}_{15}$.

3.4.1 F-collapsible domains

Note that player two's winning strategy could be implemented with access only to the ability to compute moduli and (in the first case where $T \neq T'$) apply the Chinese Remainder Theorem. A subject for further investigation is F-collapsibility: given some space F of functions, does player two have a winning strategy *which can be produced with access only to* F, and what equivalence relation does that strategy induce? For example, if F is recursive functions, it may be possible to reduce some domains more than if F were, say, finite automata recognizable functions.

CHAPTER 4

APPLICATIONS

4.1 Ledger transducer

Consider the following scenario: A household contains four people, p_1, p_2, p_3, and p_4, two of whom (p_3 and p_4) maintain their finances jointly. Most household expenses, such as food staples, utilities, repairs, and supplies, are shared equally between each of the four members. Some expenses, such as rent or favorite foods, are shared in proportion to room size or consumption rate. Splitting all expenses at the time the payment is made would be burdensome, so the household wishes to keep an ongoing record of who has paid how much for what kind of expense, and use this record roughly once per month to balance contributions: those who have spent less than their share transfer an appropriate amount of money to those who have spent more than their share.

The record's role is a transducer: it takes as inputs information about expenses and transfers, and outputs a running account of what transfers should be made to balance accounts.

The inputs consist of a: the amount of a transaction (in dollars,) p: the *primary person* in the transaction, and t: the type of transaction. The types of transactions include transferring money to another person p (denoted X_p,) paying for an expense that is to be shared equally among individuals (S_e,) paying for an expense which is to be shared unequally such as rent (S_r,) and claiming responsibility (undoing) for all or part of previous shared expense (C.) For example, if p_1 pays for a 100 dollar grocery bill, which includes 70 dollars of shared

house expenses, 10 dollars of personal items for p_1, and 20 dollars of personal expenses for p_2, the inputs to the transducer would be $(100, 1, S_e)$ for the initial bill, $(10, 1, C)$ for p_1's personal expenses, and $(20, 2, C)$ for p_2's personal expenses.

The output of the transducer consists of b_1, b_2, and b_{34}, which are the dollar amounts owed by p_1, p_2, or p_{34} to other members of the house (if negative) or owed to that person by other members (if positive.) The sum of these three values should always be zero.

We may make a first pass at modeling this as an Update Transducer as follows:

There are three input nodes a, p, t and nine cells, $s_1, s_2, s_{34}, c_1, c_2, c_{34}, b_1, b_2, b_{34}$ of which b_1, b_2, b_{34} are output cells. The domain for node p is the set of 'financial parties' in the household: $D_p = \{1, 2, 34\}$. The domain for node t is the set of transaction types, $D_t = \{X_1, X_2, X_{34}, C, S_e, S_r\}$. The domain for a and for all cells is the set of dollar amounts (positive, negative, and zero.) The graph for this UA is presented in figure 4.1.

Suppose that r_1, r_2, r_{34} are the amounts of rent paid by each of the parties. For $i \in D_p$, cells b_i updates to take value $b_i - s_i + c_i$. Cell s_1 takes the value $a/4$ if the previous value of t was S_e, the value $-a/4$ if the previous value of t was C, the value 0 if t was X_k (for any $k \in D_p$,) and r_1 if t was S_r. Similarly, s_2 takes the same values in the same conditions, except using r_2 if t was S_r. As p_{34} consists of two people, their joint share is double that of any other person, so if the type was S_e, the cell s_{34} takes value $a/2$ and value $-a/2$ if the type of transaction was C. A transfer type (X_i) still is associated with s_{34} taking a value of 0, and a rent-share type (S_r) with a value of r_3.

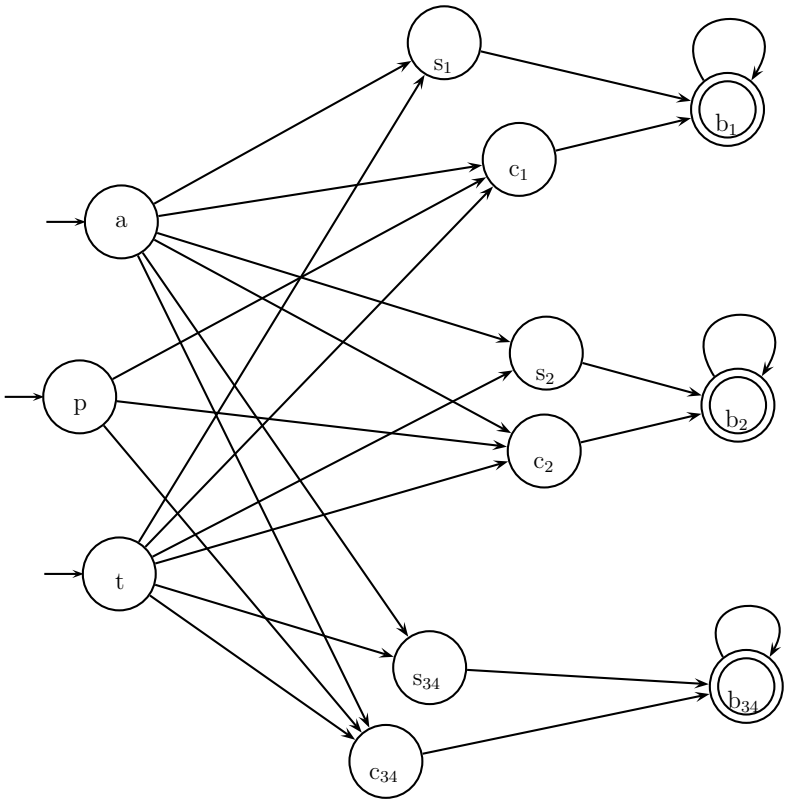

Figure 4.1: Naive ledger automaton

If i is the value of p, then cell c_i's update depends on the type t. For a claim C, the value of c_i becomes $-a$. For a payment S_e or S_r, the value is set to a. For a self-transfer X_i, the value is 0, while for a transfer X_j where $j \neq i$, the value is $-a$. If i is not the value of p, then cell c_i updates to zero unless the type is X_i (a transfer from the primary person to p_i) in which case it updates to a.

We save an example of verification for later on, when discussing difference equations, as this case is more tedious than enlightening, and will simply make note of the properties to check. First, the sum $b_1 + b_2 + b_{34}$ should always be zero. Second, transactions of the form $(0, \cdot, \cdot)$ (that is, zero-dollar transactions) should leave all the b_i unchanged. Beyond those, there are constraints on the effects of particular types of transactions. Any transaction of the form (a, p, S_e)

followed by a transaction (a, p, S_c) (or vice versa) should leave the system in the same intrinsic state as two transactions $(0, \cdot, \cdot)$. A transaction (a, p_i, X_j) should decrease i's balance by a, increase j's balance by the same amount, and leave the third person's balance unchanged. Starting from the balanced-accounts intrinsic state, a (a, p, S_e) transaction should result in a state where person p is owed a dollars, and the amount owed by p_{34} (if any) is twice the amount owed by p_1 or p_2.

The outputs in the b_i cells should fall into one of three classes: All b_i are zero (in which case all accounts are balanced,) b_i is positive while b_k and b_j are negative (in which case parties k and j should transfer $-b_k$ and $-b_j$ dollars, respectively, to b_i in order to balance the accounts,) or b_i is negative while b_k and b_j are positive (in which case p_i should transfer b_k dollars to p_k and b_j dollars to p_j to balance accounts.)

An immediate annoyance with this automaton is that it has an lower intrinsic width of 2: after an input has been supplied, it will require multiple updates before the content of that input is visible in the output. If we fold all the 'side computations' for updating s_i and c_i into the update function for the corresponding b_i, we can reduce the lower intrinsic width to its minimum of 1. Note that the upper intrinsic width is, and should be, ∞, since the current balance is determined by all historical transactions- there is no 'statute of limitations' after which someone's contribution is to be ignored! In this case, we will present the transition table 4.1 describing the update function for the transducer as a whole (inputs in the left column, current values in the middle column, new values in the rightmost column.)

Using the spreadsheet view of this update automaton, a slightly more elab-

Table 4.1: Transitions for Refined Ledger UA, sorted by type input t

a	p	t	b_1	b_2	b_3	b'_1	b'_2	b'_3
d	1	X_1	b_1	b_2	b_{34}	b_1	b_2	b_{34}
d	2	X_1	b_1	b_2	b_{34}	$b_1 + d$	$b_2 - d$	b_{34}
d	34	X_1	b_1	b_2	b_{34}	$b_1 + d$	b_2	$b_{34} - d$
d	1	X_2	b_1	b_2	b_{34}	$b_1 - d$	$b_2 + d$	b_{34}
d	2	X_2	b_1	b_2	b_{34}	b_1	b_2	b_{34}
d	34	X_2	b_1	b_2	b_{34}	b_1	$b_2 + d$	$b_{34} - d$
d	1	X_3	b_1	b_2	b_{34}	$b_1 - d$	b_2	$b_{34} + d$
d	2	X_3	b_1	b_2	b_{34}	b_1	$b_2 - d$	$b_{34} + d$
d	34	X_3	b_1	b_2	b_{34}	b_1	b_2	b_{34}
d	1	C	b_1	b_2	b_{34}	$b_1 + d/4 - d$	$b_2 + d/4$	$b_{34} + d/2$
d	2	C	b_1	b_2	b_{34}	$b_1 + d/4$	$b_2 + d/4 - d$	$b_{34} + d/2$
d	34	C	b_1	b_2	b_{34}	$b_1 + d/4$	$b_2 + d/4$	$b_{34} + d/2 - d$
d	1	S_e	b_1	b_2	b_{34}	$b_1 - d/4 + d$	$b_2 - d/4$	$b_{34} - d/2$
d	2	S_e	b_1	b_2	b_{34}	$b_1 - d/4$	$b_2 - d/4 + d$	$b_{34} - d/2$
d	34	S_e	b_1	b_2	b_{34}	$b_1 - d/4$	$b_2 - d/4$	$b_{34} - d/2 + d$
d	1	S_r	b_1	b_2	b_{34}	$b_1 - r_1 + d$	$b_2 - r_2$	$b_{34} - r_{34}$
d	2	S_r	b_1	b_2	b_{34}	$b_1 - r_1$	$b_2 - r_2 + d$	$b_{34} - r_{34}$
d	34	S_r	b_1	b_2	b_{34}	$b_1 - r_1$	$b_2 - r_2$	$b_{34} - r_{34} + d$

orate version of this transducer has been successfully used by the author for the past two years to keep the accounts of multiple households balanced.

4.1.1 Ledger-balancer transducer

The Ledger UT takes transaction events as inputs, and produces status updates as outputs. The user of the Ledger UT can then take those status updates as inputs, and subsequently issues outputs in the form of instructions to members of the household to perform certain transaction events (specifically, money transfers.) As such, the Ledger UT user could also be viewed as a transducer.

Note that while there are 'recognizable events' of interest: a person owes money to others, or is owed money, or accounts are balanced, there is no point at which it is desirable for the transducer to enter a halting, accepting, or otherwise *finished* state (The end of the household as a group of people may, however, be an appropriate point to stop supplying inputs.)

4.2 Database network

Consider a network of agents a_i which each maintain a database. For this case, suppose the databases are monotonic nondecreasing collections of "things the agent has heard" - once a thing enters the database, it remains there. Suppose that each agent accepts inputs from a source unique to them, as well as from the databases of some collection of their peers. At each time step, an agent adds to its database everything in its peers databases, as well as anything its unique source provides (the source may not always provide an input.)

We may model this scenario as an update transducer as follows: For each agent a_i, we have an input node n_i and a cell c_i. Letting F be the set of all expressible hearings, the domain of every node is $F \cup \emptyset$ and the domain of every

cell is the power set $\mathcal{P}(F)$. Let $peers(a_i) \subseteq \mathbb{N}$ be the set of $j \in \mathbb{N}$ such that agent a_j's database is one of agent a_i's sources. The update function for cell c_i is then $c_i = c_i \cup n_i \bigcup_{j \in peers(a_i)} c_j$

4.2.1 Asymptotic connectivity consequences

The monotonicity of the databases and the uniformity of the update rule permits some conclusions to be drawn about the asymptotic content of various agents databases.

Note that if agent a_i has $f \in F$ in its database, then we are ensured that every agent a_j such that $a_i \in peers(a_j)$ will have f in their database at the next time step (if it wasn't there already.) That is to say, if there is an edge from c_i to c_j, then the database of c_j at time $t+1$ will be a superset of the database of c_i at time t. And, transitively, if there is a path from c_i to c_j, c_j's database at time $t+m$ will be a superset of c_i's database at time t, where m is the length of the path.

Recall that a *strongly connected component* in a directed graph is a maximal subgraph in which every pair of vertices v_1, v_2 are connected along a directed path. If no external inputs are supplied, then it will be the case that every c_i in a strongly connected component will eventually have the same contents in their databases, and in general the $\mathcal{V}(c_i) \subseteq \mathcal{V}(c_j)$ exactly when there is a path from c_i to c_j. If external inputs are included, then this relation still holds when restricted to elements of F whose most recent introduction from an input node was sufficiently many time steps in the past.

4.2.2 Provenance

A number of refinements of the original network are possible. It may be desirable, for example, to maintain information on the provenance[11] of elements of F: It might be the case that a_i may not draw values from a_j's database because there is a prohibitive cost to maintaining a communication line, but it might also be because agent a_i does not trust a_j's private information source, or a_j's judgement in other agents to draw values from. In this case, in the network as given, to avoid data from a_j, a_i would have to avoid drawing data from any other agents who, transitively, drew data from a_j. If data were accompanied by provenance information, however, a_i could potentially filter out data which originated in an untrusted source (blacklisting,) accept only data which had passed through a trusted source (whitelisting,) or apply more complicated selection rules on incoming data or even data already in its database.

A downside of adding such control is the loss of the straightforward connectivity consequences, and in the case of allowing revisions to the agent's database, the additional loss of monotonicity. These can be mitigated by closing F under a predicate $r : F \to F$, where $r(f)$ indicates that f has been retracted (and $r(r(f))$ indicates that the retraction has been retracted,) so a database which actually contained $\{f_1, f_2, r(f_2)\}$ could be viewed as effectively containing just $\{f_1\}$.

4.3 Mohri-style weighted transducers

Mohri et al. [1],[2] have shown that weighted finite-state transducers can be used to great benefit in speech recognition systems. To review, a *weighted finite-state transducer* $T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$ over a weight semiring $\mathbb{K}$ consists of a finite input alphabet Σ, a finite output alphabet Δ, a finite set of states Q, a set of initial states $I \subseteq Q$, a set of final states $F \subseteq Q$, a finite set of transitions $E \subseteq Q \times \Sigma \times \Delta \times \mathbb{K} \times Q$, an initial weight function $\lambda : I \to \mathbb{K}$, and a final weight function $\rho : F \to \mathbb{K}$. Unless otherwise specified, ϵ-transitions (and ϵ-outputs) are allowed, changing state without consuming (or, respectively, producing) any letters. Unless otherwise specified, the set of transitions need not define a function on input letters (i.e. the transducer may be nondeterministic.)

Figure 4.2 shows a weighted transducer over the *tropical semiring* $(\mathbb{R}_+ \cup \{\infty\}, \ min, +, \infty, 0)$. It has three states $Q = \{0, 1, 2\}$, a two-letter input (and output) alphabet $\Sigma = \Delta = \{a, b\}$, initial state $I = \{0\}$, final state $F = \{2\}$, final weight $\rho : 2 \mapsto 4$, and transitions

$$(0, \ \epsilon, \ a, \ 1, \ 2)$$
$$(1, \ a, \ b, \ 2, \ 1)$$
$$(1, \ b, \ \epsilon, \ 3, \ 2)$$

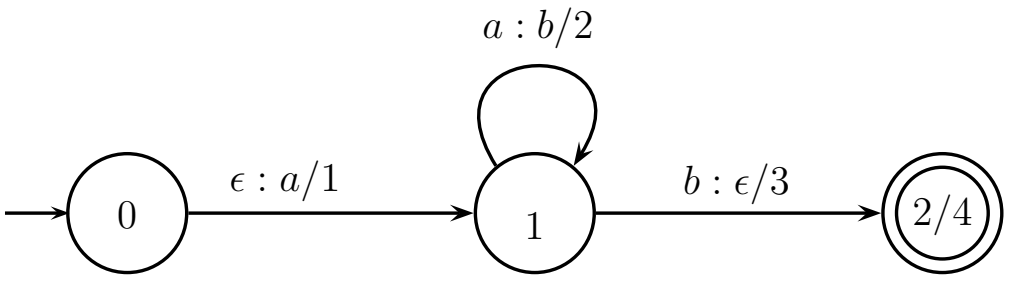

Figure 4.2: A weighted transducer

This automaton accepts words of the form $a^n b$ for $n \in \mathbb{N}$, and produces words of the form ab^n (for the same value of n) with weight $4 + 2n$.

A number of optimizations [1] may be applied to obtain an equivalent transducer with various desirable properties, such as having no ϵ-transitions or synchronizing (as much as possible) consumption with production or (if inequivalence is allowed) determinizing the transducer's input-output mapping. For example, the transducer in figure 4.3 is equivalent to the one in figure 4.2, but produces outputs synchronously with consuming inputs.

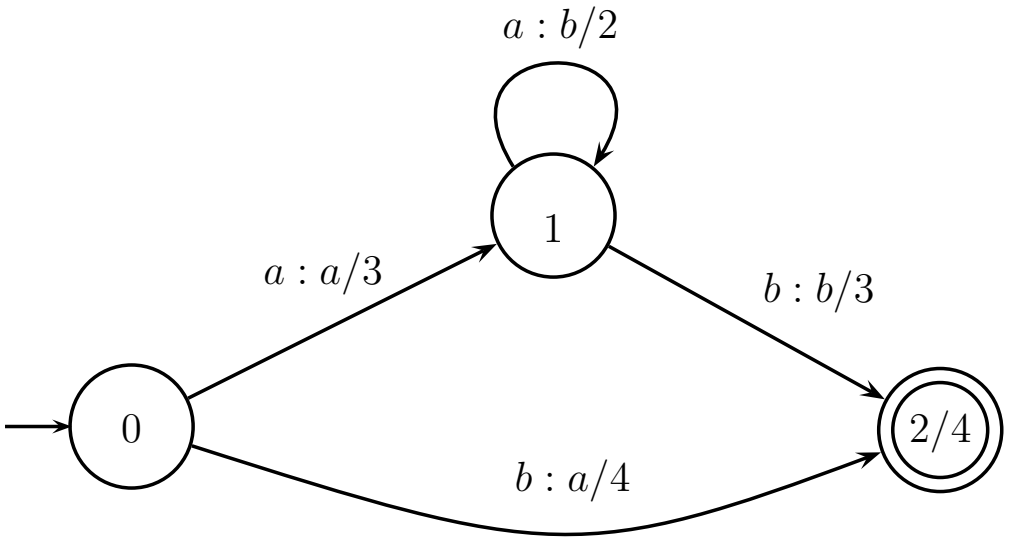

Figure 4.3: A weighted transducer equivalent to that in figure 4.2

Implementing a determinized, fully synchronized weighted transducer as an Update Transducer is straightforward: Two cells are used, one with a finite domain (the 'state' cell) and one whose domain depends on the semiring of the transducer (the 'weight' cell.) Both cells receive a letter from the input alphabet as input. The weight cell also reads the state cell, updates to a new weight value (such as the sum of its current weight and the weight of the current transition) determined by the current state and the input letter, and outputs its current

value. The state cell updates to the appropriate 'next' state and outputs the appropriate output letter. All internal and output domains are augmented with $\perp$, which is produced whenever a transition is not defined by the original weighted transducer.

As the focus of Update Transducers is processing (transduction, copying) and not recognition, the notion of "accepting state" does not translate directly, and would require some other machine which monitored the output of the transducer. If desired, however, recognition of an accepted string could be simplified by modifying the state cell to update to its current value and output $\top$ instead of $\perp$ as long as its last value corresponded to an accepting state and its input was empty ($\perp$.) The external machine could then simply watch for a $\top$ symbol to recognize that an input had ended in an accepting state.

One can imagine incorporating additional inputs to allow nondeterministic choices to be selected and determinized, or forming buffers to handle asynchronous transducers, but keeping in mind the motivating premise of Update Automata as finite controls over arbitrary domains, we will now turn our focus to using UA to organize weighted transducers, rather than simply implement them.

4.3.1 Control of deterministic synchronized transducers

Consider a collection of single-input, single-output transducers (suppose they share a common input and output alphabet,) some of whose inputs are the outputs of others. Assuming no self-feeding loops, the connectivity graph of the transducers will be a directed forest (two different transducers may be fed by a

single transducer, but the shared alphabet and single-input mean no transducer will be fed by multiple other transducers.) If the transducers are deterministic (present outputs are uniquely determined by past inputs) and synchronized (at each time step, one letter is consumed as input and one is produced as output,) the collection can be naturally expressed as an Update Transducer.

The Update Transducer has a cell for each transducer in the collection. Input nodes associated with cells whose transducer is fed by another transducer (non-root cells) are ignored, and output nodes associated with cells whose transducer feeds at least one other (non-leaf cells) are essentially ignored as well. The internal domain for each cell is the set of states for the corresponding transducer, and the output domain is the output alphabet. The update function for a given cell takes σ, the letter of the parent cell's output value, and s, the state of the current cell's internal value, and updates the internal value to t and the output value to τ, where t is the state the associated transducer would transition to, and τ is the letter which would be output.

Composition and width reduction

Note that the Update Transducer T_1 so produced is generally *not* the Update Transducer T_2 associated with the parallel composition of the transducer collection. For example, if there are any cells which are not both a root and a leaf, then

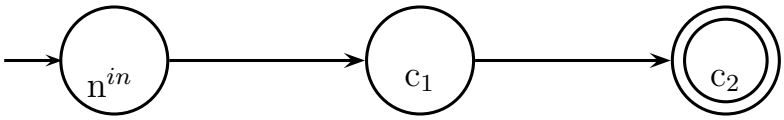

Figure 4.4: Composed translation T_1

the intrinsic width of T_1 will be greater than 1, while the intrinsic width of T_2 is

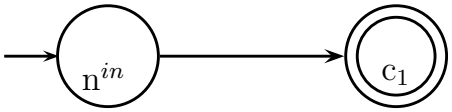

Figure 4.5: Translated composition T_2

always 1. However, in the case where the connectivity graph is connected and has no branching, the difference between the I/O maps of T_1 and T_2 are simply that T_1's outputs are delayed by a number of time steps equal to the width of the connectivity graph.

4.3.2 Tokenizer and detokenizer automata

While considering nondeterministic or asynchronous transducers is itself beyond the scope of this work, two particular tools for such consideration are not: transducers which aggregate sequences of letters into sequences of words and, inversely, transform sequences of words into sequences of their constituent letters. We introduce two auxiliary UA, the *tokenizer* and *detokenizer*.

Given an alphabet Σ, the tokenizer's behavior, as a transducer, maps sequences from $\Sigma \cup \{\perp\}$ to $\Sigma^{<\omega} \cup \{\perp\}$. Given an input $\sigma_1, \sigma_2, ..., \sigma_n, \perp, \sigma_{n+1}, ..., \sigma_{n+m}, \perp, ...$ where $\sigma_i \in \Sigma$, the tokenizer will output $\perp$ n times, then output the concatenation of σ_1 through σ_n, then output $\perp$ m times, then output the concatenation of σ_{n+1} through σ_{n+m}, and so forth (i.e. $\underbrace{\perp, \perp, ..., \perp}_{n \text{ times}}, \underbrace{\sigma_1\sigma_2...\sigma_n}_{\text{concatenated}}, \underbrace{\perp, ..., \perp}_{m \text{ times}}, \underbrace{\sigma_{n+1}...\sigma_{n+m}}_{\text{concatenated}}, ...$)

The detokenizer is essentially the reverse of this. Given an alphabet Σ, the input $w_1, w_2, ...$ with $w_i \in \Sigma^{<\omega}$ is transduced into $w_{1,1}, w_{1,2}, ..., w_{1,|w_1|}, \perp, w_{2,1}, w_{2,2}, ..., w_{2,|w_2|}, \perp, ...$ where $w_{i,j}$ is the j^{th} letter of w_i. Note that while the tokenizer buffers its inputs until they are complete before producing an output,

the detokenizer immediately begins producing outputs, but must buffer inputs because its output (one letter at a time) carries less information than its input (one word at a time.)

Implementing these transducers as update automata can be done with one cell apiece (an indication that the domain and the update function will do the heavy lifting.) For both automata, the domain of the inputs and outputs are determined above, while the domain of the cell is $(\Sigma \cup \{\perp\})^{<\omega}$. The update function for the tokenizer maps input letter σ_{in} and previous value $\perp\ \sigma_1\sigma_2...\sigma_n$ to new value $\perp\ \sigma_1\sigma_2...\sigma_n\sigma_{in}$ with output value $\perp$, and maps input letter $\perp$ and previous value $\perp\ \sigma_1\sigma_2...\sigma_n$ to new value $\perp$ and output value $\sigma_1\sigma_2...\sigma_n$. The update function for the detokenizer maps input word w_{in} and previous value $\sigma_1\sigma_2...\sigma_k \perp w_2 \perp w_3... \perp w_n$ to new value $\sigma_2\sigma_3...\sigma_k \perp w_2 \perp w_3... \perp w_n \perp w_{in}$ and output value σ_1 (if the previous value was $\perp w_2 \perp w_3...$ the new value would be $w_2 \perp w_3...$ and the output would be $\perp$.)

4.4 Zero knowledge proof

Interactive proofs involve two agents, a *prover* who wishes to convince a verifier that it has certain information, and a *verifier* who wishes to be convinced that Prover has that information (but with a high probability to avoid being convinced if Prover does not have it!) When logging into a computer, for example, the user wishes to prove they are actually an authorized user, while the computer (on behalf of its administrators) wants to verify that the user really is authorized. An interactive proof may result in complete certainty on Verifier's part of their acceptance or rejection of Prover's claim (An example in [15] is

Boolean satisfiability. A prover claims a Boolean formula is satisfiable. Verifier asks for the satisfying assignment of truth values. Prover supplies it. Verifier checks it. If it works, the formula is clearly satisfiable, and if it doesn't work then Prover has clearly failed to back up their claim.) An interactive proof may also result only in probabilistic certainty which can be increased by repeated passes of the proving protocol, but never results in complete certainty ([15] cites an example in [7] (more recently [8]) of proving graph nonisomorphism.)

A particular subset of interactive proofs, of which boolean satisfiability as presented above is not an example and graph nonisomorphism proof referenced happens to be a particularly interesting example, are *zero knowledge proofs*. Here, Prover still wishes to convince Verifier and Verifier still wishes to be (with high probability) only convinced when appropriate, but now Prover also wishes to convey no information beyond the fact that they have what they claim. For example, an authorized user may prove to an unauthorized user that they are authorized by handing the unauthorized user their username and password, which the unauthorized user may then verify are valid, but a much better plan would be to show the unauthorized user a machine which was not logged in, have them leave the room while the authorized user logs in, and then bring them back in and display that the authorized user could cause a successful login. The former method conveys more information than simply being authorized, while the latter (barring windows, surveillance equipment, key loggers, and so forth) does not. As the solution given for boolean satisfiability is a demonstration by transmitting the entire solution, it is not (in that form) a zero-knowledge proof.

We will work through an example of a zero-knowledge proof, and then

model it as an interaction between two update transducers.

4.4.1 Zero knowledge proof of graph n-colorability

Suppose some graph G is publicly known, and Prover wishes to demonstrate to Verifier that they have an n-coloring for a G (an assignment of n distinct colors to the vertices of G such that no edge of G has the same color on both ends,) and to do so without giving away what the coloring actually is. That is to say, to prove that an n-coloring on G exists. Since a well-intentioned Verifier wouldn't do anything that a malicious Verifier might do, let us suppose that Verifier is actively trying to either discover Prover's coloring for G, or reveal that Prover is a fraud.

We assume the existence of cryptographic functions which are easy to compute and whose inverse is not feasible to compute without a secret key (in which case the inverse is easy to compute.) For the sake of intuition, we will use the informal notion of a "lock box" inside which information can be hidden, and which can only be opened by someone in possession of an appropriate key. We assume that no two lock boxes have the same key, even if they have the same contents.

The protocol begins with Verifier secretly selecting a sequence of edges in G, and then proceeds by repeating the following round:

1: Prover begins (presumably) with a function $p : V_G \to \{c_1, c_2, ..., c_n\}$ which maps vertices of G to colors c_1 through c_n. Prover holds (or tries to hold) p in secret. 2: Prover randomly selects a permutation $\sigma : \{c_1, c_2, ..., c_n\} \to \{c_1, c_2, ..., c_n\}$

and applies it to the coloring function to get $p \circ \sigma$. Note that this is a valid coloring if and only if p was a valid coloring. Prover holds σ in secret. 3: For each $v \in V_G$, Prover places $\sigma \circ p(v)$ in a lock box L_v which is handed over to Verifier. Prover keeps the key K_v held in secret. 4: After receiving all the L_v, Verifier hands the next edge (v_1, v_2) in their secretly selected sequence over to Prover. 5: Prover retrieves keys K_{v_1} and K_{v_2} and hands them back to Verifier. 6: Verifier uses the keys to unlock L_{v_1} and L_{v_2}, revealing their permuted colorings in order to compare them. 7: If Verifier sees colors $\sigma(p(v_1)) = \sigma(p(v_2))$ are the same, then the original colors $p(v_1) = p(v_2)$ of Prover must have been the same, and Prover is revealed as a fraud. If the colors are different, then there is a nonzero chance $0 < \delta \leq$, bounded away from zero, that Prover really does have an n-coloring, and the process is repeated.[1]

After one (successful) pass, there is a $(1 - \delta)$ chance that Prover got lucky and hid a same-colored pair of vertices somewhere that Verifier didn't check. After k passes, that chance decreases to $(1 - \delta)^k$, which becomes arbitrarily close to zero, so given any certainty $\epsilon < 1$, there is some number k of passes after which $\epsilon < 1 - (1 - \delta)^k$ and the chance of Prover not actually having a proper coloring is less than Verifier's desired certainty. Note that δ is bounded away from zero, since if Prover doesn't have a coloring, there are at least two adjacent vertices of the same color, so there's at least a $\delta \geq \frac{1}{|E_G|}$ chance of randomly selecting a bad edge. Note that, by putting all the colors in lock boxes, Prover has to commit to a particular coloring each round, and cannot fabricate results by waiting until Verifier has selected an edge and then producing two conveniently dissimilar colors for the vertices on that edge.

So, Verifier can reduce the chance of a false positive below any threshold

[1] Equality occurs in the trivial case when the G has exactly two vertices.

$\epsilon < 1$, and never gets a false negative. Why is it that even after a successful proof, Verifier can't reconstruct the coloring from this information? Well, consider it this way: If Prover were replaced by Conspirator, and Verifier and Conspirator wished to produce a transcript that looked like it had come from a successful run of the protocol, could they do so without having in hand an actual coloring of the graph, or even an actual transcript of a successful run? If they can conspire to produce such a transcript without having a coloring, and since Verifier's only information is only what is visible in a successful transcript, Verifier hasn't learned anything new.

So, what is produced in a successful transcript? Plenty of unopenable locked boxed, and a sequence of edges whose associated lock boxes, when unlocked, contain different colors. (Note that, because Prover chooses a (potentially different) random permutation of colors on each pass, if Verifier picked the same edge over and over, the colors might be Red-Blue one time, Red-Green the next, Green-Blue the time after, and so forth.) If, before the protocol begins, Verifier randomly selects the sequences of edges to test and reveals that sequence to Conspirator, then on the pass when Conspirator knows Verifier will pick edge (v_i, v_j), the graph can be colored such that $p(v_i) = c_1$ and for all $v \neq v_i$, $p(v) = c_2$. The rest of the procedure remains unchanged, and the resulting transcript will be indistinguishable from the transcript for a real successful run.

So, being Verifier in a successful run of the protocol yields no more information (just a transcript that looks correct) about how to color the graph than conspiring based on no knowledge of the graph does (which produces a transcript that looks correct) so Verifier learns nothing about Prover's graph coloring.

4.4.2 Implementation in update transducers

As the graph is common knowledge, we introduce one cell g_0 which is initialized with the graph, and which updates by taking its own value. This effectively makes the graph a constant that any cell may refer to at any time.

We will give Prover four cells, one of which is a trivial delay line. The domain of p_1 is "colored graphs". It updates by applying a randomly permuted coloring to the value found in g_0 (if Prover is fraudulent, or if Prover is honest but knows of multiple colorings, this coloring may be fundamentally different each step, but generally we assume it applies an arbitrary permutation of the same coloring each time.) The domain of p_2 is "graphs whose vertices are colored by (lock box, key) pairs". p_2 updates by looking at the coloring in p_1 and applying the locking routine on the color at at each vertex. p_2 is an output node, and its visible output domain is "graphs whose vertices are colored by a lock box", which is simply accomplished by a projection from the cell's internal domain. A delay-line cell $\otimes$ updates to take the value of p_2. The domain of cell p_3 is "(key, key) pairs". p_3 uses an input cell p_3^{in} whose domain is "edges of G". p_3 updates by looking up the keys stored in p_2 (as presented in the delay line $\otimes$) which are on the vertices at either end of the edge given in p_3^{in}. p_3 is an output node, and its visible output domain is the same as its internal domain: it simply displays the keys associated with the chosen edge.

Verifier gets five cells, one of which are trivial delay lines.

The domain of cell v_1 is "edges of G". It updates by randomly selecting an edge from the graph in g_0. v_1 is an output node, and its visible output domain is identical to its internal domain. The domain of cell v_2 is "graphs whose vertices

are colored by a lock box". v_2 uses an input cell v_2^{in} with the same domain, and updates by copying the value of the input cell. A delay-line cell $\otimes$ updates to take the value of v_2. The domain of cell v_3 is "graphs whose vertices are colored by lock boxes or colors". It uses an input cell v_3^{in} whose domain is "(key,key) pairs". It updates by reading the value of v_2 (as presented through the delay line) and unlocking the boxes whose keys are given in v_3^{in}. The domain of v_4 is real numbers $x \in [0, 1]$. If it updates to 0 if the $v_4 = 0$ or if two colors in v_3 match, and otherwise to $1-(1-v_4)(1-\frac{1}{|E_G|})$, the chance of the coloring being valid[2]. Also, Verifier has two acceptance criteria: if $v_4 = 0$, accept that the coloring is a fraud, and if $v_4 \in (\epsilon, 1]$ for some predetermined confidence threshold ϵ, accept that the coloring is probably correct.

The Prover and Verifier interact as follows: The values of v_2^{in} are determined by the output of p_2 (as available through p_2's output domain, which only displays lock boxes, and not its internal domain, which also includes all the keys!) The values of p_3^{in} are determined by the output of v_1. The values of v_3^{in} are determined by the output of p_3. This is illustrated in figure 4.6.

Note that, as in figure 4.6, all the cells can be arranged in such a way that every edge (except those from the constant cell g_0 and in the aggregator v_4) begins in one vertical column and ends in column immediately adjacent to the right (note also that this arrangement requires the presence of a couple delay lines, marked $\otimes$.) This allows to check the system's behavior by trancing the progression of information from one column to the next:

Beginning with the column containing p_1, Prover applies a permuted color-

[2] v_4 represents the chance of the coloring being valid, $1 - v_4$ is the chance that a fraudulent coloring has so far escaped detection, $1 - \frac{1}{|E_G|}$ is a lower bound on the chance that a fraudulent coloring escapes detection on this step, so $1 - (1 - v_4)(1 - \frac{1}{|E_G|})$ is the chance of the coloring being valid after yet another step.

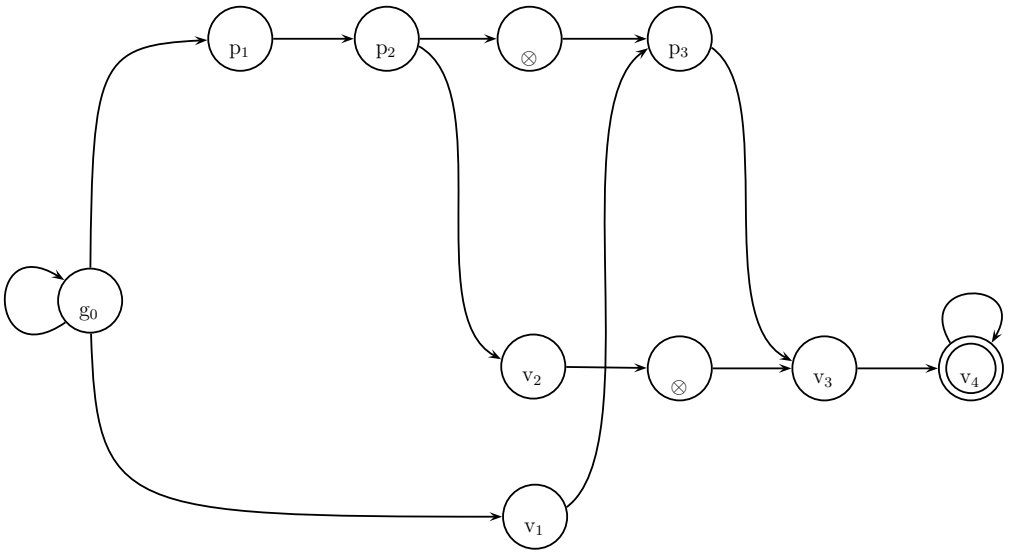

Figure 4.6: Prover (p_i) and verifier (v_i) automata for zero-knowledge proof

ing to the graph.

In the next column (p_2,) Prover locks all the colors, and those locked colors are output.

Next, (v_2, v_1, $\otimes$) Prover does nothing (save for remembering the keys) while Verifier stores the lock boxes (input from previous column) and selects an edge (output to next column.)

Next, (p_3, $\otimes$) Verifier does nothing (save for remembering the lock boxes) while Prover looks up the appropriate keys for the edge (input from previous column) and outputs those keys (output to next column.)

Penultimately, (v_3) Verifier unlocks the edges.

Finally, (v_4) the current chance that Prover is correct based on this one pass is aggregated into the running estimate based on all previous passes.

We can see that, with an external process monitoring v_4 for acceptance, this

system does implement the essence of the n-coloring zero-knowledge proof. The main difference is that, whereas the protocol given above performs all steps before repeating, the transducer presented here lays the steps out in and assembly-line fashion and can have a number of executions in the pipeline simultaneously.

4.5 Difference equations

Difference equations, whether singly or in systems, with constant or variable coefficients, and homogeneous or with constraint functions, are particularly amenable to being expressed and (given appropriate inputs) solved with update transducers. At the most general level, a system of difference equations relating functions $f_1, ..., f_n$, with non-homogeneous constraints $g_1, ..., g_k$, and variable coefficients, may be reduced to a form which reveals which (if any) of the f_i are unconstrained, and from which an update transducer may be constructed which, when equipped with the desired initial conditions and given as inputs the successive values of the g_i, the free f_i, and the coefficients, will produce (after some delay) the successive values of the remaining f_i.

We will begin by working with a single difference equation, then move on to systems of difference equations.

Given a function $f : \mathbb{N} \to D$ where D is a ring, the shift operator E produces $Ef(t) = f(t + 1)$. Any difference equation $a_0\Delta^k f + a_1\Delta^{k-1} f + ... + a_k f = g$ may be written in terms of the shift operator rather than the difference operator[5], as $\Delta^k f = E^k f - f$.

4.5.1 Linear difference equation with constant coefficients

Given a kth-order difference equation $a_0E^kf + a_1E^{k-1}f + ... + a_kf = g$ with $a_0 \neq 0$, we may solve for $f(n)$ in terms of earlier values as

$$f(n) = \frac{g(n-k) - (a_1f(n-1) + a_2f(n-2) + ... + a_kf(n-k))}{a_0}$$

Given initial conditions for $f(0), g(0), f(1), g(1), ..., f(k-1), g(k-1)$, we construct an update transducer that consumes inputs $g(k), g(k+1), g(k+2), ...$ and produces outputs $f(k), f(k+1), f(k+2), ...$

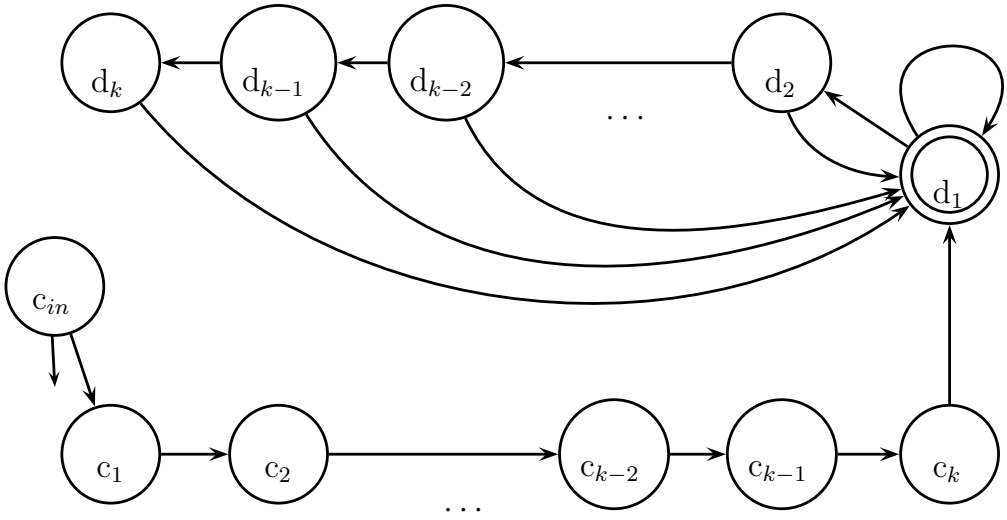

Figure 4.7: Automaton for single difference equation

The automaton (see figure 4.7 for graph) has one is an input node, c^{in}, and $2k$ cells, c_1 through c_k and d_1 through d_k with d_1 an output cell.

Input node c^{in} supplies values $g(k), g(k+1), ...$ The initial value for cell c_i is $g(k-i)$. The initial value for cell d_i is $f(k-i)$. The update function for c_1 takes the value of c^{in}. The update function for other c_i takes the value of c_{i-1}. The update function for d_1 takes the value $\frac{c_k-(a_1d_1+a_2d_2+...+a_kd_k)}{a_0}$. The update function for other d_i takes the value of d_{i-1}.

Verification: To see that this automaton does, indeed, perform as expected, assume that it is in a state where, for some integer m, the value of cells c_i are $g(m-i)$ and the values of cells d_i are $f(m-i)$. We wish to find that updating puts us in a state where the values are $g((m+1)-i)$ and $f((m+1)-i)$ respectively.

Since the last update made c_1 become $g(m-1)$, that must have been the value of c^{in}, and since c^{in} takes consecutive values of g, the current value of c^{in} is $g(m-1+1) = g(m)$, so this update will make c_1 become $g(m)$. All other c_i will take the previous value of c_{i-1}, which is to say the new values of c_i are $g(m-(i-1)) = g((m+1)-i)$. Similarly, except for d_1, the new values of the d_i will be the previous values of d_{i-1}, which were $f(m-(i-1)) = f((m+1)-i)$. d_1 will update to $\frac{c_k-(a_1d_1+a_2d_2+...+a_kd_k)}{a_0}$, which by assumption is $\frac{g(m)-(a_1f(m-1)+a_2f(m-2)+...+a_kf(m-k))}{a_0}$. This is exactly $f(m)$, as solved above, so the new values of the d_i are each $f((m+1)-i)$, and the values of c_i are $g((m+1)-i)$, as desired.

4.5.2 Linear difference equation with variable coefficients

The proceeding construction implicitly assumed that the coefficients $a_0, ..., a_k$ were constant, but modifying the automaton to accept varying coefficients is straightforward.

The addition of variable coefficients to the automaton faces the same problem as the inclusion of a non-homogeneous constraint g: the homogeneous part (the d_i cells) makes use of k delay cells, and when $g(n)$ is being input, $f(n-k)$ is being output. The update to produce $f(n-k)$ requires information about $g(n-k)$, rather than the 'current' $g(n)$.

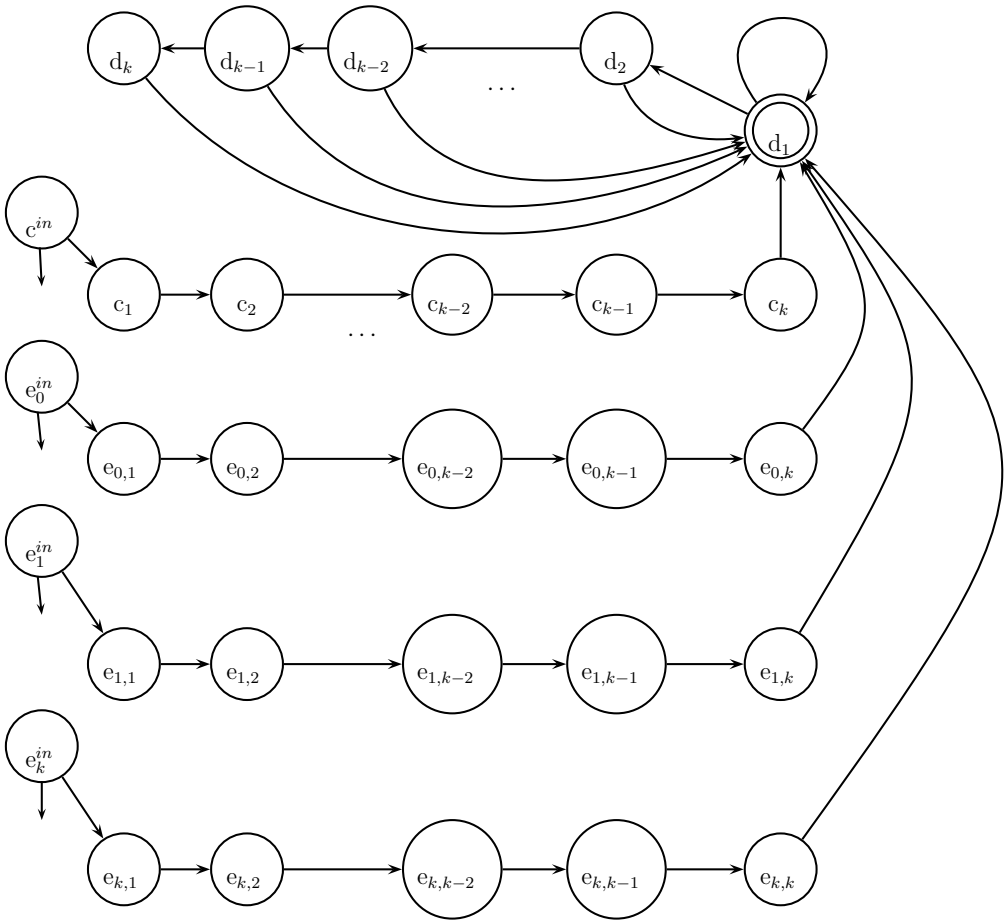

Figure 4.8: Automaton for single difference equation with variable coefficients

For the non-homogeneous constraint, this was addressed by adding a k-wide delay line where the action of the update was simply to move values from c_{i-1} to c_i, and this same procedure can be applied to each of the variable coefficients $a_0(t)$ through $a_k(t)$. In figure 4.8, for each variable coefficient $a_j(t)$ we add an extra input e_j^{in} and k cells $e_{j,1}$ through $e_{j,k}$ arranged in a delay line. That is, the update rule for $e_{j,i}$ is to take the value of $e_{j,i-1}$, and the update for $e_{j,1}$ is to take the value of the input e_j^{in}.

Also, the update rule for the d_1 cell is modified to produce the value $\frac{c_k - (e_{1,k}d_1 + e_{2,k}d_2 + \ldots + e_{k,k}d_k)}{e_{0,k}}$. As long as $a_0(t) \neq 0$ for all t, no other changes are required to

produce a transducer from inputs of the form $(g(n+k), a_0(n+k), a_1(n+k), ..., a_k(n+k))$ to an output of $f(n)$.

4.5.3 Single equation as a system of linear equations

To introduce systems of difference equations, we will recast a single equation as a system. Returning to constant coefficients, consider the equation $a_0E^kf + a_1E^{k-1}f + ... + a_kf = g$. We introduce new functions $f_0 = f,\ f_1 = Ef = Ef_0,\ f_2 = E^2f = Ef_1,\ ...,\ f_{k-1} = E^{k-1}f = Ef_{k-2}$. Note that we also have

$$Ef_{k-1} = E^kf = \frac{g - (a_1E^{k-1}f + ... + a_kf)}{a_0} = \frac{g - (a_1f_{k-1} + a_2f_{k-2}... + a_{k-1}f_1 + a_kf_0)}{a_0}$$

Treating these as a system of equations, we are looking for solutions to $f_0, ..., f_{k-1}$ which satisfy

$$\begin{bmatrix} E & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & E & -1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & E & -1 \\ a_k & a_{k-1} & a_{k-2} & a_{k-3} & \dots & a_2 & a_0E + a_1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{k-2} \\ f_{k-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ g \end{bmatrix} \tag{4.1}$$

or, alternately,

$$\begin{bmatrix} E & -1 & 0 & 0 & \dots & 0 & 0 \\ E^2 & 0 & -1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ E^{k-1} & 0 & 0 & 0 & \dots & 0 & -1 \\ a_0E^k + a_k & a_{k-1} & a_{k-2} & a_{k-3} & \dots & a_2 & a_1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{k-2} \\ f_{k-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ g \end{bmatrix} \tag{4.2}$$

Which is related to equation (4.1) by row reductions as follows:

Add $-E$ times first row to second row, $-E^2$ times first row to third row, ... , add $-E^{k-2}$ times first row to $k-1$st row, add $-a_0E^{k-1}$ times first row to last row...

$$\begin{bmatrix} E & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & E & -1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & E^{k-2} & 0 & 0 & \dots & 0 & -1 \\ a_k & a_{k-1} + a_0E^{k-1} & a_{k-2} & a_{k-3} & \dots & a_2 & a_1 \end{bmatrix} \tag{4.3}$$

Add $-E$ times the second row to the third, $-E^2$ times the second row to the fourth, ... , $-E^{k-3}$ times the second row to the $k-1$st row, and add $-a_0E^{k-2}$ times the second row to the last...

$$\begin{bmatrix} E & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & E & -1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & E^{k-3} & 0 & \dots & 0 & -1 \\ a_k & a_{k-1} & a_{k-2} + a_0E^{k-2} & a_{k-3} & \dots & a_2 & a_1 \end{bmatrix} \tag{4.4}$$

Repeating this procedure $k-3$ more times will result in the matrix in equation (4.1.)

Now, pretending for the moment that the non-homogeneous constraint g was zero, consider the d_i cells from the single difference automaton in figure 4.7. Their update functions read either "d_i updates to the previous value of d_{i-1}" when $i \neq 1$ or "d_1 updates to $-\frac{(a_1 d_1 + a_2 d_2 + \ldots + a_k d_k)}{a_0}$". In all cases, the updated value is a linear function of the previous values of d_1 through d_k, and taken together, all the updates comprise a system of linear equations from (previous values of) d_1 through d_k to (updates values of) d'_1 through d'_k. Expressed as a matrix, we have

$$\begin{bmatrix} -\frac{a_1}{a_0} & -\frac{a_2}{a_0} & -\frac{a_3}{a_0} & \ldots & -\frac{a_{k-2}}{a_0} & -\frac{a_{k-1}}{a_0} & -\frac{a_k}{a_0} \\ 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{k-1} \\ d_k \end{bmatrix} = \begin{bmatrix} d'_1 \\ d'_2 \\ d'_3 \\ \vdots \\ d'_{k-1} \\ d'_k \end{bmatrix} \tag{4.5}$$

Keeping in mind that at any given time, there is an m such that the value in cell d_i is supposed to be $f(m - i)$, and the updated value d'_i is hence $f(m - i + 1)$, and using the labeling $f_0 = f, f_j = E^j f$, we could rewrite this as

$$\begin{bmatrix} -\frac{a_1}{a_0} & -\frac{a_2}{a_0} & -\frac{a_3}{a_0} & \ldots & -\frac{a_{k-2}}{a_0} & -\frac{a_{k-1}}{a_0} & -\frac{a_k}{a_0} \\ 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} f_{k-1} \\ f_{k-2} \\ f_{k-3} \\ \vdots \\ f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} Ef_{k-1} \\ Ef_{k-2} \\ Ef_{k-3} \\ \vdots \\ Ef_1 \\ Ef_0 \end{bmatrix} \tag{4.6}$$

Or, more naturally, we can move everything to the left hand side of each

equation and multiply each equation (row) by -1 or $-a_0$ as appropriate, and we have

$$\begin{bmatrix} a_0E + a_1 & a_2 & a_3 & \dots & a_{k-2} & a_{k-1} & a_k \\ -1 & E & 0 & \dots & 0 & 0 & 0 \\ 0 & -1 & E & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -1 & E & 0 \\ 0 & 0 & 0 & \dots & 0 & -1 & E \end{bmatrix} \begin{bmatrix} f_{k-1} \\ f_{k-2} \\ f_{k-3} \\ \vdots \\ f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{4.7}$$

Or, as seen before in equation (4.1),

$$\begin{bmatrix} E & -1 & 0 & \dots & 0 & 0 & 0 \\ 0 & E & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & E & -1 & 0 \\ 0 & 0 & 0 & \dots & 0 & E & -1 \\ a_k & a_{k-1} & a_{k-2} & \dots & a_3 & a_2 & a_0E + a_1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{k-2} \\ f_{k-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{4.8}$$

So, viewing a single difference equation as a system of first-order difference equations is strongly related to constructing an update automaton which produces successive values of a solution to the equation. We now turn our attention to systems (not necessarily first-order) of difference equations which do not arise directly from a single difference equation.

4.5.4 System of difference equations

Given a system of difference equations, we wish to construct a transducer from values of any constraint functions to values of solutions. If the system is homogeneous (i.e. there are no constraint functions) then there are no inputs, so our end product is an update automaton instead of an update transducer.

Here is a simple such system.

$$\begin{aligned}(E^2-1)f_1 &+ (E^2+1)f_2 &= 0\\ (E-1)f_1 &+ (E-1)f_2 &= 0\end{aligned}$$

Here $f_1(t), f_2(t)$ are to be determined. The letter E is still used to denote the step function ($Ef(t) = f(t+1)$.)

Generally, equations are of the form $P(E)F = G$

where we are trying to determine all solutions $F = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}$ from $G = \begin{bmatrix} g_1 \\ \vdots \\ g_n \end{bmatrix}$ and

$P(E) = \begin{bmatrix} P_{11}(E) & \dots & P_{1n}(E) \\ \vdots & \ddots & \vdots \\ P_{m1}(E) & \dots & P_{mn}(E) \end{bmatrix}$, the given constraint and polynomial matrix.

We shall be sloppy and use E the variable in the polynomials. For readability, we will assume the coefficients of the polynomials are constant, but that the automaton treatment allows for them to be variables just as was the case for the single equation. For simplicity, we will also assume there are as many equations as there are variables.

We give a general algorithm for finding all solutions to such systems by an

analogue of row reduction for systems of linear algebraic equations. It is based on a a row reduction algorithm algorithm for matrices of polynomials in E, unimodular row reduction. This presentation is adapted from [17].

4.5.5 Unimodular row reduction

This is row reduction for matrices with polynomial entries.

The unimodular row operations are:

I. Interchange two rows.

II. Add a polynomial multiple of one row to another.

III. Multiply a row by a non-zero constant.

Note: Multiplication of a row by a non-constant polynomial is NOT allowed. In the linear difference equations application, such a multiplication would raise the degree of the polynomial and introduce extraneous solutions. These would then have to be eliminated afterwards by substitution of the purported solution into the original equations.

Definition. A unimodular (polynomial) matrix M is a matrix M whose entries are polynomials and which has a determinant which is a non-zero *constant*.

Each of these three row operations Ω has a corresponding row matrix ΩI which is a *unimodular matrix.*

1) Products of unimodular matrices are unimodular since the determinant of a product is the product of the determinants, and the product of non-zero

constants is a non-zero constant.

2) The inverse of a unimodular matrix is a polynomial unimodular matrix since the determinant of the inverse is the inverse of the determinant.

Corollary. Every square unimodular matrix is the product of elementary unimodular row matrices.

The proof is the same as for matrices with real or complex entries, one has to clear the whole column above and below the diagonal as with Gauss Jordan normal form (reduced row echelon form.) The fact that the matrix is unimodular means that it has a non-zero determinant and so it reduce to the identity. The row operations used are all unimodular, and their product in reverse order is the matrix reduced.

4.5.6 Unimodular row reduction algorithm

This algorithm is for square matrices of polynomials.

First mark (as completed) all columns whose entries are all zero. We now iterate the following procedure, beginning with stage $n = 1$:

Stage n:

(a) Let j_n be the leftmost unmarked column with a non-zero entry.

(b) If the degree of entry a_{n,j_n} is not minimal among the degrees of non-zero entries below it in column j_n, interchange row n with a row below it whose entry in column j_n is of minimal degree.

(c) For each row i below row n in which the degree of a_{i,j_n} is d or greater, add a suitable polynomial multiple of row n to row i to make the degree of the a_{i,j_n} entry less than d.

(d) If all entries in column j_n below row n are zero, mark column j_n as completed and stage n is complete. Otherwise, return to substage (b.)

Continue with successive stages until all columns are marked, at which point the result is an upper triangular matrix.

Note that this algorithm marks one column per stage, and any matrix will have finitely many columns, so we are guaranteed termination as long as substage (d) does not repeat indefinitely. Since substage (c) causes a strict decrease in the degree of entries found below row n, and substage (b) ensures that row n has minimal degree among the non-zero entries below it, and all our entries will have finite degree, substage (d) can arise only finitely many times.

When used in the context of difference equations, this gives rise to all solutions by back substitution. We immediately apply these operations to systems of difference equations, where the polynomials are polynomials in E.

Example

We begin with a system of two difference equations in two unknowns:

$$\begin{aligned} (E^2-1)f_1 &+ (E^2+1)f_2 &= g(t) \\ ((E-1)f_1 &+ (E-1)f_2 &= 0 \end{aligned}$$

We introduce the augmented matrix

$$\left[\begin{array}{cc|c} (E^2-1) & (E^2+1) & g \\ (E-1) & (E-1) & 0 \end{array}\right]$$

The last column consists of functions to which the polynomial operators apply. The other columns are polynomials in E. The last column does *NOT* consist of difference operators, it consists of functions to which difference operators are applied in the course of the reduction. So we are really applying row reduction in a slightly more general case than described above.

We apply unimodular row reduction.

Stage 1: No columns are zero, so none are initially marked, so (a) the leftmost unmarked column j_1 is the first column. We (b) interchange the rows to put the lowest degree element of the first column in the first row:

$$\left[\begin{array}{cc|c} (E-1) & (E-1) & 0 \\ (E^2-1) & (E^2+1) & g \end{array}\right]$$

We now (c) add $-(E+1)$ times the first row to the second to reduce the degree of the second row (all the way down to zero)

$$\begin{aligned} &\left[\begin{array}{cc|c} (E-1) & (E-1) & 0 \\ (E^2-1)-(E+1)(E-1) & (E^2+1)-(E+1)(E-1) & g-(E-1)0 \end{array}\right] \\ =&\left[\begin{array}{cc|c} (E-1) & (E-1) & 0 \\ (E^2-1)-(E^2-1) & (E^2+1)-(E^2-1) & g-0 \end{array}\right] \\ =&\left[\begin{array}{cc|c} (E-1) & (E-1) & 0 \\ (0) & 2 & g \end{array}\right] \end{aligned} \tag{4.9}$$

There is only one row below the first to deal with, so we're done with (c) for now, and since all the lower entries are zero, we (d) mark the first column as complete and move to the second column.

Stage 2:

The leftmost unmarked column j_2 is the second column. There are (b) no lower rows to interchange. All lower rows are (c) as reduced as they can be, and they're all zero, so (d) we mark the second column as complete.

All non-augmenting columns are now marked, the matrix is in triangular form, so we are done reducing and may solve by back substitution.

The second row says $2f_2 = g$, or $f_2 = \frac{g}{2}$, so

$$f_2(t) = g(t)/2$$

The first equation says $(E-1)f_1 + (E-1)f_2 = 0$, or that $(E-1)(f_1 + \frac{g}{2}) = 0$, so

$$f_1(t+1) = f_1(t) + \frac{g(t) - g(t+1)}{2}$$

While we do have recurrence relations for f_i entirely in terms of previous values of f_i and values of a constraint g, note that f_1 now calls for multiple different values of g, and not simply $g(t)$ as in our single equation setting. Recall in figure 4.7, however, that the delay line that was necessary for saving the value of $g(t)$ for k stages until it was time to calculate the value of $f(t+k)$) contained not just the value of $g(t)$, but the values of $g(t+1), g(t+2), ...g(t+(k-1)$ as well, so using those values presents no great obstacle. In this case, we have two automata seen in figures 4.9 and 4.10. The update functions are $n^{in}_{f_2} = g(t)$ and $c_1 = \frac{n^{in}_{f_2}}{2}$ for the f_2 transducer, with output read from c_1, and $n^{in}_{f_1} = g(t)$, $c_3 = (n^{in}_{f_1})$, $c_4 = c_4 + \frac{c_3 - n^{in}_{f_1}}{2}$ for

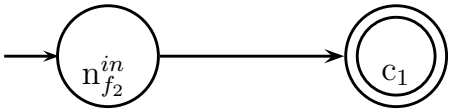

Figure 4.9: f_2 transducer

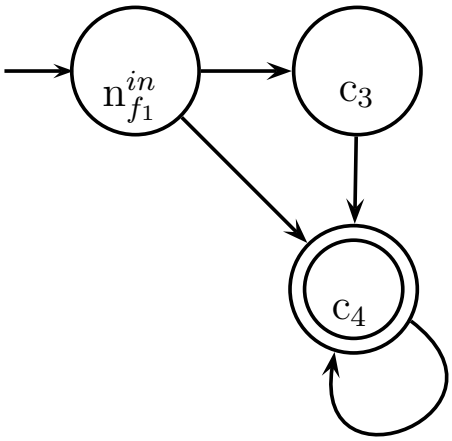

Figure 4.10: f_1 transducer

the f_1 transducer, with output read from c_4. These take values of g as inputs, and produce values of the appropriate f_i as outputs. Note that while both automata have a lower intrinsic width of one, the f_2 automaton, dealing with a zero degree recurrence relation, has an upper intrinsic width of one while the f_1 automaton, which works with a first degree recurrence relation, has an upper width of two.

Also, rather than reducing f_1 algebraically, we may use the already constructed f_2 transducer and build on top of it, to produce a combined "f_1 and f_2 transducer", as in figure 4.11, where we have the following update rules: $n^{in} = g(t)$, $c_1 = \frac{n_{f_2}^{in}}{2}$, $c_4 = c_4 + (c_1 - \frac{n^{in}}{2})$, with c_1's output being the value for f_2, and c_4 the value for f_1.

To verify the behavior of the automaton, we will check that if (n^{in}, c_1, c_4) are in states whose values are $g(t), f_2(t-1), f_1(t-1)$, then after an update the states will have values $g(t+1), f_2(t), f_1(t)$, and that combined with the obvious initial

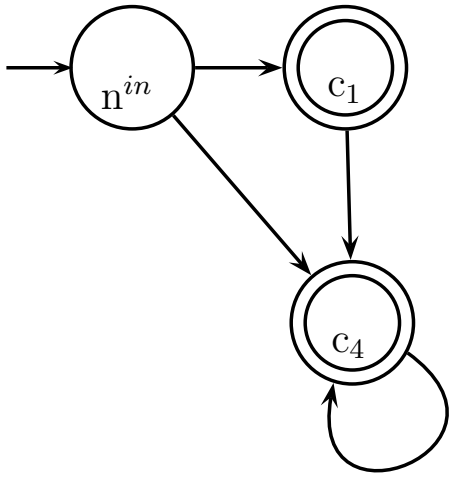

Figure 4.11: f_1, f_2 combined transducer

conditions inductively proves the desired behavior.

That n^{in} takes the desired values is obvious, since it is an input node whose values are successive values of g. So if its value is $g(t)$ at one step, it will be $g(t+1)$ at the next step. The update rule for c_1 will put it into state $n^{in}/2 = g(t)/2$, as desired. The update rule for c_1 gives it value $c_1 = \frac{g(t)}{2} = f_2(t)$. The update rule for c_4 produces value

$$c_4 + (c_1 - \frac{n^{in}}{2}) = f_1(t-1) + (f_2(t-1) - \frac{g(t)}{2}) = f_1(t)$$

Thus, each time step preserves the property that the values of the cells are $g(t), f_2(t-1), f_1(t-1)$ for the current t, so reading off the values of c_1, c_4 to obtain the values of f_2, f_1 does indeed work.

Example

Suppose g is a function such that $g(a+b) = g(a)g(b)$ so, in particular, $Eg = g(t+1) = g(t)g(1)$.

$$\begin{aligned}(E+1)f_1 \quad &+Ef_2 \quad &= 1\\ (E^2+E+1)f_1 \quad &+(E^2+1)f_2 \quad &= g\end{aligned}$$

The augmented matrix is

$$\left[\begin{array}{cc|c} (E+1) & E & 1 \\ (E^2+E+1) & (E^2+1) & g \end{array}\right]$$

Add $-E$ times the first row to the second row.

$$\left[\begin{array}{cc|c} (E+1) & E & 1 \\ (E^2+E+1)-(E^2+E) & (E^2+1)-E^2 & g-E1 \end{array}\right]$$

$$= \left[\begin{array}{cc|c} (E+1) & E & 1 \\ 1 & 1 & g-1 \end{array}\right]$$

Interchange the rows

$$\left[\begin{array}{cc|c} 1 & 1 & g-1 \\ (E+1) & E & 1 \end{array}\right]$$

Add $-(E+1)$ times the first row to the second row.

$$\left[\begin{array}{cc|c} 1 & 1 & g-1 \\ (E+1)-(E+1)1 & E-(E+1)1 & 1-(E+1)(g-1) \end{array}\right]$$

$$= \left[\begin{array}{cc|c} 1 & 1 & g-1 \\ 0 & -1 & 3-(1+g(1))g \end{array}\right]$$

This is now in triangular form. The second equation says $f_2 = g(t)(1+g(1))-3$, and the first equation says $f_1 = (g(t)-1) - f_2 = 2 - g(t)g(1)$. We assemble a single transducer which finds both values (figure 4.12,) with update rules $n^{in} = g(t)$,

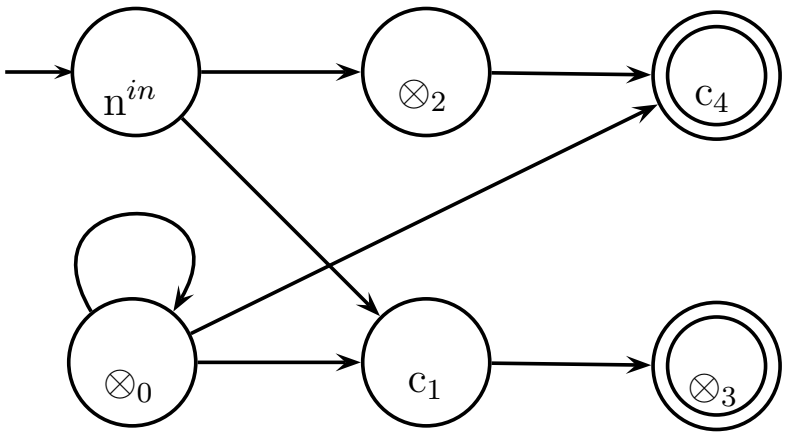

Figure 4.12: Transducer for both f_1 and f_2

$c_0 = c_0$ (we initialize c_0 to value $g(1)$) $c_1 = 2 - c_0 \cdot n^{in}$, $c_2 = n^{in}$, $c_3 = c_1$ (a delay line,) $c_4 = c_2(1 + c_0) - 3$. The output of c_4 is the value of f_2, and the output of c_3 is the value of f_1. Note that since $g(t) = g(t-1)(g(1)$, this could also be set up as an automaton with no inputs other than its initial conditions, as in figure 4.13 where we replace the input node with a cell c_0' whose update function is $c_0' = c_0' * c_0$, which corresponds to updating the value of $g(t)$ to be equal to $g(t-1)g(1)$.

For verification (we will do the automaton from figure 4.13) we wish to show that if $c_0, c_0', c_1, c_2, c_3, c_4$ are respectively holding the values $g(1), g(t), f_1(t-1), g(t-1), f_1(t-2), f_2(t-2)$, then an application of the update rules produces the values $g(1), g(t+1), f_1(t), g(t), f_1(t-1), f_2(t-1)$.

For c_0, the update takes the value $c_0 = g(1)$, as desired. For c_0', the update rule appropriately produces $c_0'c_0 = g(t)g(1) = g(t+1)$. c_2 is simply a delay line, and

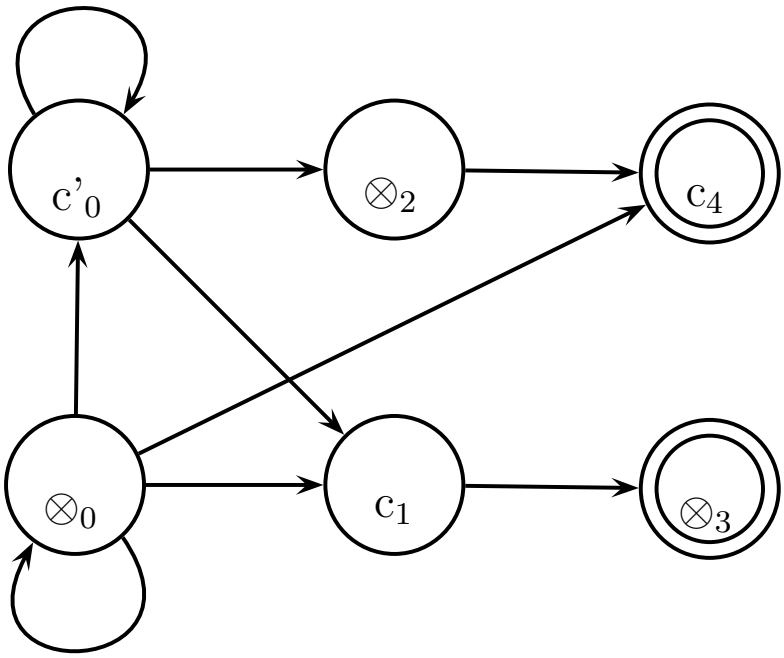

Figure 4.13: Automaton for both f_1 and f_2.

the update to $c_0' = g(t)$ does the right thing. The update rule for c_1 is, as desired,

$$2 - c_0 c_0' = 2 - g(1)g(t) = (g(t) - 1) - f_2(t) = f_1(t)$$

c_3 is another delay line, updating to $c_1 = f_1(t-1)$, which checks, and c_4 becomes

$$c_2(1 + c_0) - 3 = g(t-1)(1 + g(1)) - 3 = f_2(t-1)$$

Thus, the contract is preserved, so with appropriate initial values, the outputs will be correct.

Example (Homogeneous system)

We consider the following system of equations

$$\begin{aligned}
(E-1)f_1 \quad &+2f_2 \quad &-(E-1)f_3 \quad &= 0 \\
-(E^2-1)f_1 \quad & \quad &+(2E^2-1)f_3 \quad &= 0 \\
(2E-2)f_1 \quad &+(5)f_2 \quad &+(E^2+E)f_3 \quad &= 0
\end{aligned}$$

and express it as an augmented matrix in preparation for applying unimodular row reduction:

$$\left[\begin{array}{ccc|c} (E-1) & (2) & -(E-1) & 0 \\ -(E^2-1) & (0) & (2E^2-1) & 0 \\ (2E-2) & (5) & (E^2-E+2) & 0 \end{array}\right]$$

Stage 1: Add $(E+1)$ times the first row to the second, and (-2) times the first row to the third.

$$\left[\begin{array}{ccc|c} (E-1) & (2) & -(E-1) & 0 \\ (0) & (2E+2) & (E^2) & 0 \\ (0) & (1) & (E^2+E) & 0 \end{array}\right]$$

The first column is taken care of, so stage 2 moves on to the second column:

Switch the second and third row,

$$\left[\begin{array}{ccc|c} (E-1) & (2) & -(E-1) & 0 \\ (0) & (1) & (E^2+E) & 0 \\ (0) & (2E+2) & (E^2) & 0 \end{array}\right]$$

Add $-(2E+2)$ times the second row to the third row,

$$\left[\begin{array}{ccc|c} (E-1) & (2) & -(E-1) & 0 \\ (0) & (1) & (E^2+E) & 0 \\ (0) & (0) & (-2E^3-3E^2-2E) & 0 \end{array}\right]$$

and the second column is now done. The matrix is in upper triangular form, so the reduction is done.

The third equation is $(-2E^3 - 3E^2 - 2E)f_3 = 0$, so $E^3 f_3 = -3E^2 f_3 - 2E f_3$, or

$$f_3(t+1) = -(3f_3(t) + 2f_3(t-1)) \tag{4.10}$$

The second equation is by back substitution: $f_2 + (E^2 + E)f_3 = 0$, or

$$\begin{aligned} f_2(t+1) &= -f_3(t+3) - f_3(t+2) \\ &= -(-(3f_3(t+2) + 2f_3(t+1))) - f_3(t+2) \\ &= 2(f_3(t+2) + f_3(t+1)) \\ &= 2(-(3f_3(t+1) + 2f_3(t)) + f_3(t+1)) \\ &= -4(f_3(t+1) + f_3(t)) \\ &= -4(-(3f_3(t) + 2f_3(t-1)) + f_3(t)) \\ &= 8(f_3(t) + f_3(t-1)) \end{aligned} \tag{4.11}$$

This gives us

$$f_2(t+1) = 8(f_3(t) + f_3(t-1))$$

Note that, to ensure we have a causal transducer whose present values for f_2 do not depend on future values of f_3, we applied 4.10 on the second, fourth, and sixth lines.

Finally, the first equation is $(E-1)f_1 + 2f_2 - (E-1)f_3 = 0$, from which we get

$$\begin{aligned} f_1(t+1) &= f_1(t) - 2f_2(t) + f_3(t+1) - f_3(t) \\ &= f_1(t) - 2f_2(t) - (3f_3(t) + 2f_3(t-1)) - f_3(t) \\ &= f_1(t) - 2f_2(t) - 4f_3(t) - 2f_3(t-1) \end{aligned} \tag{4.12}$$

So, all together, these give us

$$f_1(t+1) = f_1(t) - 2(f_2(t) + 2f_3(t) + f_3(t-1))$$

$$f_2(t+1) = 8(f_3(t) + f_3(t-1))$$

$$f_3(t+1) = -(3f_3(t) + 2f_3(t-1))$$

These may, in turn, be represented as automata as follows: First, f_3 depends only on f_3 (i.e. we just need the initial conditions, and we are in the homogeneous case so there is no constraining function g,) and we have the automaton in figure 4.14. Update functions are $c_2 = c_1$ (initialized to $f(1)$) and $c_1 = -(3c_1 + 2c_2)$

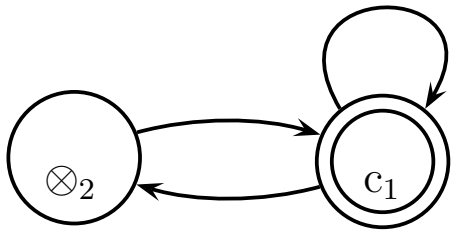

Figure 4.14: Automaton producing f_3

(initialized to $f(2)$.) The output can be taken from either cell, depending on whether $f(t)$ or $f(t-1)$ is desired.

Next, we may either construct a transducer for f_2 which takes values of f_3 as inputs, or an automaton which solved both f_3 and f_2. The former is shown in figure 4.15, where n^{in} is given values of f_3, c_2 is a delay line, and c_4 updates as $c_4 = 8(n^{in} + c_2)$. The value of f_2 is read from the output of c_4. Alternately, we may construct an automaton for f_2 and f_3 (figure 4.16) where c_1 and c_2 are the same cells as in figure 4.14, c_4's rule is the same aside for renaming in c_4's update rule: $c_4 = 8(c_1 + c_2)$. The value of f_3 is read off cell c_2, and the value of f_2 is still read off c_4.

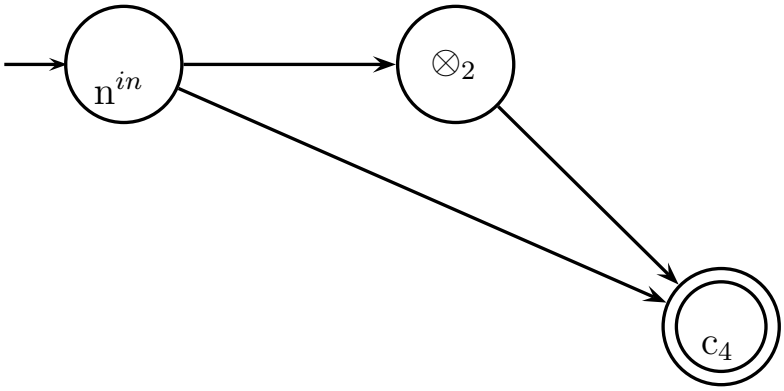

Figure 4.15: Building solutions in stages, f_2 transducer

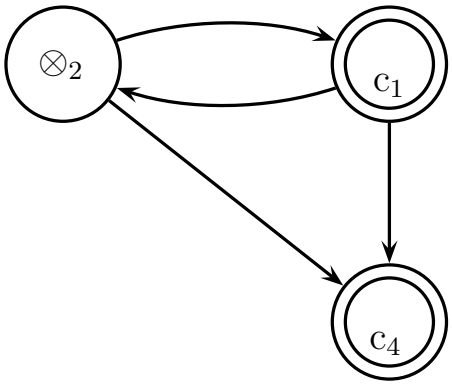

Figure 4.16: Building solution cumulatively, f_2 and f_3 automaton

Finally, we follow the same paths for f_1, first giving it as a separate transducer which requires values of f_3 and f_2 as inputs, and then giving a combined automaton which produces values for all three functions. First, as a transducer depending only on inputs whose values are f_2 and f_3, we have figure 4.17, where n_1^{in} is fed the value of f_3, n_4^{in} is fed the value of f_2, c_2 is a delay lines so that we can retain the value of $f_3(t-1)$, and c_5 updates according to $c_5 = c_5 - 2(n_4^{in} + 2n_1^{in} + c_2)$.

On the other hand, we can construct all three functions together in figure 4.18. Again, cells c_1 through c_4 are identical to their earlier definitions for figure 4.16 with c_1 the output cell for f_3 and c_4 the output for f_2. As for the additional cells, we have $c_5 = c_5 - 2(c_4 + 2c_1 + c_2)$ (derived from $f_1(t+1) = f_1(t) - 2(f_2(t) +$

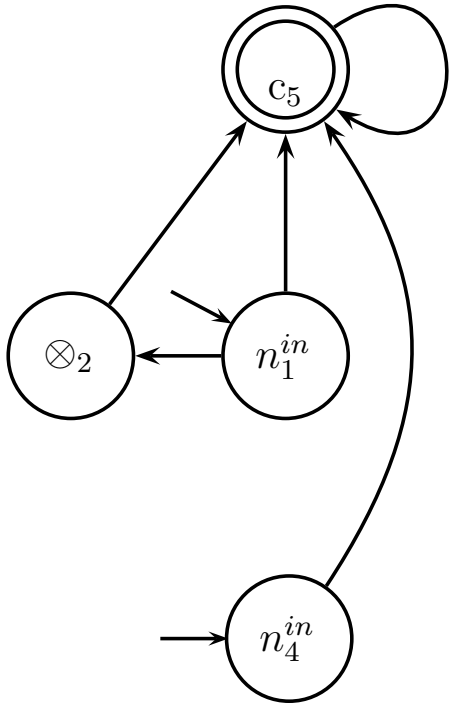

Figure 4.17: Building solutions in stages, f_1 transducer

$2f_3(t) + f_3(t-1)))$ whose value is the output for f_1.

Finally, we verify that the values update properly for the presentation in figure 4.18. We have cells

$$c_1, c_2, c_4, c_5$$

which should begin in states

$$f_3(t), f_3(t-1), f_2(t), f_1(t)$$

and then become

$$f_3(t+1), f_3(t), f_3(t-1), f_2(t+1), f_1(t+1)$$

Since c_2 is a delay lines, it takes the previous values of c_1 so works correctly.

The updated value of c_1 is $-(3c_1 + 2c_2) = -(3f_3(t) + 2f_3(t-1)) = f_3(t+1)$, as desired.

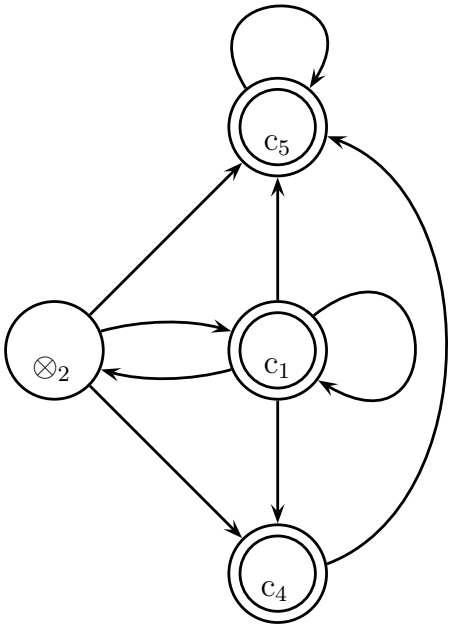

Figure 4.18: Building solution cumulatively, f_1, f_2 and f_3 automaton

Similarly, the updated value of c_4 is $8(c_1+c_2) = 8(f_3(t)+f_3(t-1)) = f_2(t+1)$, and the updated value of c_5 is $c_5-2(c_4+2c_1+c_2) = f_1(t)-2(f_2(t)+2f_3(t)+f_3(t-1) = f_1(t+1)$. This confirms that all the updates preserve the desired state of the automaton and produce the expected values from the output cells.

CHAPTER 5

AREAS OF FURTHER INTEREST

5.1 Continuous time update automata

We have restricted our consideration to strictly causal transducer maps between sequences: output values at time t depend only on inputs at times less than t. On the surface, this seems require discrete time, but a notion of "weakly causal" and an appropriate application of nonstandard analysis could allow for time to be taken to be continuous. We say that a map $F : h \mapsto g$ is *weakly causal* if the values of $h(s)$ for $s \leq t$ are sufficient to determine the value of $g(t)$. In particular, note that $g(t)$ may depend on $h(t)$. This is not, *a priori*, a tenable situation, since one can easily imagine trying to have a cell c which, at time t, takes the negative of the value of cell c at time t.

If, however, we constrain our attention to smooth functions, the difference between causal and weakly causal becomes less problematic, as access to all $f(s)$ with $s < t$ is sufficient to determine $f(t)$. In the case of automata with cycles in their graphs, some fixed-point operation may still be required. While we will not explore this in depth here, it is possible that nonstandard analysis could be made use of. Given some infinite integer n, if the domains of cells were allowed to use $\frac{1}{n}$ and the automaton were run for n steps, the effective result would be to transform differential equations into difference equations with negligibly small differences between time steps.

5.2 Systems of difference equations with variable coefficients

Unlike in Carta's thesis[3], we did not investigate the case of systems of difference equations with variable coefficients. Vartiable coefficients may, at some time, be zero or be related in such a way that there are dependencies among the equations. For a fixed set of equations, an inductive argument shows that coefficient functions can be chosen which do not give rise to such dependencies, in which case row reduction (keeping in mind that the shift operator does not commute with time-varying coefficients) can be performed and used to construct a transducer from constraint functions and coefficients to solutions. In the general case, it may be possible to divide the problem into a number of cases $1, 2, ..., n$ corresponding to all the combinations of zeroed coefficients and dependencies, perform row reduction on each of these n cases separately, construct transducers $T_1, ..., T_n$ for each of these cases which use a common collection of nodes, cells, and domains, and then merge these into a single transducer T in which the update function for some cell applies the update function from the corresponding cell in T_i, where i is the case which applies to the current collection of coefficients. We do not expect that such a 'switching' solution would lend itself to a neat algebraic formulation, however.

5.3 Other directions

Yuri Gurevich's notion of an Evolving Algebra, now called an Abstract State Machine[12], may be a particularly well suited to the task of producing expanding Thévenin replacements. Parallels between update automata and Blum, Smale, and Shub's register machine for computation on real numbers for dy-

namical systems, specifically as they apply to the automata treatment of difference equations, bear further investigation.

Recursive analysis as represented in Pour-El Richards [19] and the Computability and Complexity in Analysis website (http://cca-net.de/) deals with recursive functions on arbitrary real numbers. These maps are induced by partial recursive functions on function spaces. They induce causal transducers mapping any sequence of approximations to an input functions to a sequence of approximations to the output function. Many from standard analysis are given by primitive recursive maps on function spaces. These and many others can be represented as update transducers. This subject remains to be developed.

APPENDIX A

DOCUMENT PREPARATION

This document was typeset in LaTeX 2e (http://www.latex-project.org/.) The figures were constructed in IDES 2.1 (http://www.ece.queensu.ca/hpages/labs/discrete/software.html.)

BIBLIOGRAPHY

[1] C. Allauzen, M. Mohri, M. Riley, and B. Roark. A generalized construction of integrated speech recognition transducers. *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, 1, 2004.

[2] C. Allauzen, M. Mohri, and B. Roark. THE DESIGN PRINCIPLES AND ALGORITHMS OF A WEIGHTED GRAMMAR LIBRARY. *International Journal of Foundations of Computer Science*, 16(3):403–421, 2005.

[3] Edoardo Carta. *Update Transducers and Linear Recurrence Equations over Semirings*. PhD thesis, Cornell University, August 2008.

[4] P.P. Choudhury, B.K. Nayak, S. Sahoo, and S.P. Rath. Theory and Applications of Two-dimensional, Null-boundary, Nine-Neighborhood, Cellular Automata Linear rules. *Arxiv preprint arXiv:0804.2346*, 2008.

[5] S. Elaydi. *An Introduction to Difference Equations*. Undergraduate Texts in Mathematics. Springer, 2005.

[6] Richard R. Goldberg. *Methods of Real Analysis*. John Wiley and Sons, Inc., second edition, 1976.

[7] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. *Proceedings of 27th Annual Symposium on Foundations of Computer Science*, pages 174–187, 1986.

[8] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[9] E. Goles and S. Martínez. *Neural and automata networks: dynamical behavior and applications*, volume 58 of *Mathematics and Its Applications*. Kluwer Academic Publishers, 1990.

[10] T.J. Green, G. Karvounarakis, Z.G. Ives, and V. Tannen. Update exchange with mappings and provenance. *Proceedings of the 33rd international conference on Very large data bases*, pages 675–686, 2007.

[11] T.J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[12] J.K. Huggins and C. Wallace. An Abstract State Machine Primer. *Computer Science Technical Report CS-TR-02-04 Michigan Technological University*, 2002.

[13] D.H. Johnson. Origins of the Equivalent Circuit Concept. *Rice University, MS366, Houston, TX77251*, 2001.

[14] E.W. Kaucher and W.L. Miranker. *Self-validating numerics for function space problems. Computation with guarantees for differential and integral equations*. Notes and Reports in Computer Science and Applied Mathematics. Notes and Reports in Computer Science and Applied Mathematics, 9, 1984.

[15] D. Kozen. *Theory of Computation*. Texts in Computer Science. Springer, 2006.

[16] A. Nerode. Linear automaton transformations. *Proc. Amer. Math. Soc*, 9:541–544, 1958.

[17] Anil Nerode. Differential equations. Book in progress from lecture notes, 2006.

[18] Anil Nerode and J.B. Remmel. Input transducers. In proceedings of Topological Methods of Logic (June 3-5, 2008) Tbilisi, Georgia, June 2008.

[19] M.B. Pour-El and J.I. Richards. *Computability in Analysis and Physics*. Perspectives in Mathematical Logic. Springer Verlag, 1989.

[20] F.F. Soulié, Y. Robert, and M. Tchuente. *Automata networks in computer science: theory and applications*. Nonlinear science: theory and applications. Princeton University Press Manchester: Manchester University Press, Princeton, NJ, 1987.