Primitive Recursive Presentations of Transducers and their Products

Victor Yodaiken*

FSResearch LLC ,
2718 Creeks Edge Parkway
yodaiken@fsmlabs.com
http://www.yodaiken.com

Abstract. Methods for specifying Moore type state machines (transducers) abstractly via primitive recursive string functions are discussed. The method is mostly of interest as a concise and convenient way of working with the complex state systems found in computer programming and engineering, but a short section indicates connections to algebraic automata theory and the theorem of Krohn and Rhodes. The techniques are shown to allow concise definition of system architectures and the compositional construction of parallel and concurrent systems.

Key words: transducer, Moore machine, primitive recursion, composition, parallel

1 Introduction

The engineering disciplines of programming and computer system design have been handicapped by the practical limitations of mathematical techniques for specifying complex discrete state systems. While finite automata are the natural basis for such efforts, the traditional state-set presentations of automata are convenient for only the simplest systems and for classes of systems, but become awkward as state sets become large and in the presence of partially specified behavior or compositional systems. Furthermore, it would be nice to be able to parameterize automata so that we can treat, for example, an 8bit memory as differing from a 64bit memory in only one or a few parameters. These problems can all be addressed by using a recursive function presentation of automata that is introduced here.

General automata have long been understood to be functions from finite strings of input symbols to finite strings of output symbols[1] but for specifying computer systems it is more useful to consider functions from finite strings of inputs to individual outputs. The intuition is that each string describes a path from the initial state to some "current" state and the value of the function is the output of the system in the "current" state. If A is an alphabet of input events and X is a set of possible outputs, let A^* be the set of finite strings

* This paper replaces multiple earlier rough drafts.

over A including the empty string Λ and then a function $f : A^* \to X$ defines a relationship between input sequences and outputs. These functions can be shown to be strongly equivalent to (not necessarily finite) Moore type automata[5] while abstracting out details that are not interesting for our purposes here. If a is an input and w is a string, wa is the result of appending a to w and by defining $f(\Lambda) = x_0$ and $f(wa) = h(a, f(w))$, we can completely specify the operation of f.

Correspondence between a transducer M and a string function f.

$$\text{Input:}w \Rightarrow \text{Machine:}M \Rightarrow \text{Output:}x$$
$$f(w) = x$$

It turns out that a type of simultaneous recursion can be used to specify automata products that model composition and parallel (and concurrent) state change. Suppose that $f_1, ...f_n$ are previously defined string functions, $f_i : A_i^* \to X_i$ and we wish to combine these into a system where inputs from some alphabet A drive the components forward. At each step an input a to the composite system will be used to generate an input sequence z_i for each component f_i. The input sequence for the component is a function of both a and the feedback, the outputs of the components. The composition builds a new function f from $f_1 \dots, f_n$ plus a communication map g and an output mapping h. Let $f(w) = h(f_i(u_1) \dots, f_n(u_n))$ where the u_i are themselves primitive recursive functions of f and w. I will write u_i when w is clear from context and use functional form $u_i(w)$ otherwise. We always require that $u_i(\Lambda) = \Lambda$ — so that in the initial state of the composite system every component is in its own initial state. Let $w \circ z$ be the string obtained by concatenating w and z. The communication map is used as follows:$u_i(wa) = u_i(w) \circ g(i, a, f(w))$. The idea is that appending a to w causes the string $g(i, a, f(w))$ to be concatenated to u_i.

The two "factors" case is illustrative.

$$\text{Input:}w \to g \begin{pmatrix} \text{Input:}w_1 \to M_1 \to \text{Output:}x_1 \\ \text{Input:}w_2 \to M_2 \to \text{Output:}x_2 \end{pmatrix} \to h \to \text{Output:}x$$

$\Uparrow$ $\Downarrow$

$\Leftarrow \Longleftarrow$ feedback $\Longleftarrow \Longleftarrow$

$$F(w) = h(f_1(u_1), f_2(u_2))$$
$$u_i(\Lambda) = \Lambda$$
$$u_i(wa) = u_i(w) \circ g(a, i, F(w))$$

Example: Stack By way of illustration consider a parallel implementation of a stack.

$$Stack_n(w) = (S(u_1) \dots, S(u_n)) \tag{1}$$

where each $S(za) = a$ so that the n factors are are simple storage cells. Let's have a special value so we can spot empty cells $S(\Lambda) = EMPTY$ and have some $a = EMPTY$ in the storage cell alphabet. The alphabet of the stack is $PUSH[v] : v \in A_{storage}$ and POP. Then define the u_i

$$u_i(\Lambda) = \Lambda \tag{2}$$

$$u_i(wa) = u_i(w) \circ \begin{cases} \langle v \rangle & \text{if } i = 1 \text{ and } a = PUSH[v] \\ \langle EMPTY \rangle & \text{if } i = n \text{ and } a = POP \\ \langle S(u_{i-1}(w)) \rangle & \text{if } i > 1 \text{ and } a = PUSH[v] \\ \langle S(u_{i+1}(w)) \rangle & \text{if } i < n \text{ and } a = POP \end{cases} \tag{3}$$

Then define $Top(w) = S(u_1)$ and

$$Empty(w) = \begin{cases} 1 \text{ if } S(u_1) = EMPTY \\ 0 \text{ otherwise.} \end{cases}$$

and

$$Full(w) = \begin{cases} 1 \text{ if } S(u_n) \neq EMPTY \\ 0 \text{ otherwise.} \end{cases}$$

Example: Network A computer on a network might, from the outside, appear to have an alphabet consisting of $RECV[m]$, $TRANSMIT[m]$, for m in a set of possible messages and $TICK$ to indicate passage of time. Say D is a networked computer if $D(w) \in \{(m, c) : m \in Messages \cup \{NULL\}, c \in \{ready, busy\})\}$ where $D(w) = (x, y)$ tells us that D is trying to send message x (or not sending any message if $x = NULL$) and that D is or is not ready to accept a message. For simplicity assume a broadcast network and then define

$$N(w) = (D_1(u_1) \ldots, D_n(u_n), R(v))$$

where each D_i is a network node and R is an arbiter we can define to pick which, if any, node gets to send a message next. Each D_i may be distinct as long as it satisfies the specifications of output values.

$$R(z) \in \{1 \ldots n\}$$

The alphabet of N can just consist of the single symbol $TICK$. Let $u_i(wa) = u_i(w) \circ \langle RECV[m], TICK \rangle$ if $R(v(w)) = j$ and $D_j(u_j(w)) = (m, c)$ and $D_i(u_i(w)) = (k, ready)$. Otherwise, just append $TICK$ to u_i.

If D_j is itself a product, say $D_j(w) = (OS(r_{os}), APP(r_{app})$ then if w is the string parameter to N, we can look inside at the value of $OS(r_{os}(u_i(w)))$.

Outline In what follows, the correspondence between string functions and transducers is made precise, the correspondence between the simultaneous recursion scheme given above to a "general product" of automata is proven and some implications are drawn for the study of automata structure and algebraic automata theory. Companion technical reports describe practical use.

2 Basics

A Moore machine or transducer is usually given by a 6-tuple

$$M = (A, X, S, start, \delta, \gamma)$$

where A is the alphabet, X is a set of outputs, S is a set of states, $start \in S$ is the initial state, $\delta : S \times A \to S$ is the transition function and $\gamma : S \to X$is the output function.

Given M, use primitive recursion on sequences to extend the transition function δ to A^* by:

$$\delta^*(s, \Lambda) = s \text{ and } \delta^*(s, wa) = \delta(\delta^*(s, w), a). \tag{4}$$

So $\gamma(\delta^*(start, w))$ is the output of M in the state reached by following w from M's initial state. Call $f_M(w) = \gamma(\delta^*(start, w))$ the *representing function* of M. It's easy to go from $f : A^* \to X$ to $f^* : A^* \to X^*$, but that does not suit the purposes of this work.

If f_M is the representing function of M, then $f'(w) = g(f(w))$ represents M' obtained by replacing γ with $\gamma'(s) = g(\gamma(s))$. The state set of M and transition map remain unchanged.

The transformation from string function to transducer is also simple. Given $f : A^* \to X$ define $f_w(u) = f(w \circ u)$. Let $S_f = \{f_w : w \in A^*\}$. Say f is finite if and only if S_f is finite. Define $\delta_f(f_w, a) = f_{wa}$ and define $\gamma(f_w) = f_w(\Lambda) = f(w)$. Then with $start_f = f_\Lambda$ we have a Moore machine

$$\mathcal{M}(f) = \{S_f, start_f, \delta_f, \gamma_f\}$$

and, by construction f is the representing function for $\mathcal{M}(f)$.

A similar construction can be used to produce a monoid from a string function as discussed below in section 3.1.

Any M_2 that has f as a representing function can differ from $M_1 = \mathcal{M}(f)$ only in names of states and by including unreachable and/or duplicative states. That is, there may be some w so that $\delta_1^*(start_1, w) \neq \delta_2^*(start_2, w)$ but since $f_w = f_w$ it must be the case that the states are identical in output and in the output of any states reachable from them. If we are using Moore machines to represent the behavior of digital systems, these differences are not particularly interesting and we can treat $\mathcal{M}(f)$ as *the* Moore machine represented by f.

While finite string functions are the only ones that can directly model digital computer devices or processes[1], infinite ones are often useful in describing system properties. For example, we may want $L(\Lambda) = 0$ and $L(wa) = L(w)+1$ and then seek to prove for some P that there is a t_0 so that whenever $L(w \circ z) \geq L(w)+t_0$ there is a prefix v of z so that $P(w \circ v) = 0$. In this case, L is an ideal measuring device, not necessarily something we could actually build.

[1] There is a lot of confusion on this subject for reasons I cannot fathom, but processes executing on real computers are not Turing machines because real computers do not have infinite tapes and the possibility of removeable tapes doesn't make any difference.

2.1 Products

Suppose we have a collection of (not necessarily distinct) Moore machines $M_i = (A_i, X_i, S_i, start_i, \delta_i, \lambda_i)$ for $(0 < i \leq n)$ that are to be connected to construct a new machine with alphabet A using a connection map g. The intuition is that when an input a is applied to the system, the connection map computes a string of inputs for M_i from the input a and the outputs of the factors (*feedback*). The general product here is described by Gcseg [2]. I have made the connection maps generate strings instead of single events so that the factors can run at non-uniform rates. If $g(i, a, \boldsymbol{x}) = \Lambda$, then M_i skips a turn.

Definition 21 General product of automata
Given $M_i = (A_i, X_i, S_i, start_i, \delta_i, \gamma_i)$ and h and g define the Moore machine: $M = \mathcal{A}_{i=1}^{n}[M_i, g, h] = (A, X, S, start, \delta, \gamma)$

- $S = \{(s_1 \ldots, s_n) : s_i \in X_i\}$ and $start = (start_1 \ldots, start_n)$
- $X = \{h(x_1 \ldots, x_n) : x_i \in X_i\}$ and $\gamma((s_1 \ldots, s_n)) = h(\gamma_1(s_1) \ldots, \gamma_n(s_n))$.
- $\delta((s_1 \ldots, s_n), a) = (\delta_1^*(s_1, g(1, a, \gamma(s))) \ldots, \delta_n^*(s_n, g(n, a, \gamma(s))))$.

One thing to note is that the general product, in fact any product of automata, is likely to produce a state set that contains unreachable states. The string function created by simultaneous recursion represents the minimized state machine as well. The possible "blow up" of unreachable and duplicate states is not a problem for composite recursion.

Theorem 1 If each f_i represents M_i and $f(w) = h(f_1(u_1) \ldots, f_n(u_n))$
and $u_i(\Lambda) = \Lambda$
and $u_i(wa) = u_i(w) \circ g(i, a, f(w))$
and $M = \mathcal{A}_{i=1}^{n}[M_i, h, g]$ **then** f represents M

Proof: Each f_i represents M_i so

$$f_i(z) = \gamma_i(\delta_i^*(start_i, z)) \tag{5}$$

But $\gamma(\delta^*(start, w)) = h(\gamma(s)) = h(\ldots \gamma_i(\delta_i^*(start_i, w_i)) \ldots)$ for some w_i. All we have to show is that

$$\delta^*(start, w) = (\ldots \delta_i^*(start_i, u_i(w)) \ldots) \tag{6}$$

and then we have

$$\gamma(\delta^*(start, w)) = h(\ldots \gamma_i(\delta_i^*(start_i, u_i(w))) \ldots).$$

It follows immediately that

$$\gamma(\delta^*(start, w)) = h(\ldots f_i(u_i(w))) \ldots) = f(w)$$

Equation 6 can be proved by induction on w. Since $u_i(\Lambda) = \Lambda$ the base case is obvious. Now suppose that equation 6 is correct for w and consider wa.

Let $\delta(start, w) = s = (s_1 \ldots, s_n)$ and let $u_i(w) = z_i$. Then, by the induction hypothesis $s_i = \delta_i^*(start_i, z_i)$, and, by the argument above $\gamma(\delta^*(start, w)) = f(w)$. So:

$$\delta^*(start, wa) = \delta(\delta^*(start, w), a) \tag{7}$$
$$= \delta(s, a) \tag{8}$$
$$= (\ldots \delta_i^*(s_i, g(i, a, \gamma(s))) \ldots) \tag{9}$$
$$= (\ldots \delta_i^*(\delta_i^*(start, u_i(w)), g(i, a, f(w))) \ldots) \tag{10}$$
$$= (\ldots \delta_i^*(start, u_i(w) \circ g(i, a, f(w))) \ldots) \tag{11}$$
$$= (\ldots \delta_i^*(start, u_i(wa)) \ldots) \tag{12}$$

proving 6 for wa.

It follows directly that if M is represented by f, and f is defined by simultaneous recursion, then f can also be defined by single recursion — although such a definition may be impractical because of the large state set size.

3 More on Representation and Some Algebra

A number of results follow from theorem 1.

Theorem 2 *For M and f constructed as products as above in theorem 1.*

- *There are an infinite number of distinct products $M' = \mathcal{A}_{i=1}^{k}[N_i, g_i]$ so that f represents M' as well as M.*
- *If all of the M_i are finite state, M is finite state (by construction).*
- *If all of the f_i are finite state, f is finite state (since it represents a finite state Moore machine).*
- *If f is finite state then there is some $M' = \mathcal{A}_{i=1}^{k} r[Z_i, g, h]$ where f represents M' and each Z_i is a 2 state Moore machine. In fact $k = \lceil \log_2(|S_{M'}|) \rceil$. This is simple binary encoding.*

3.1 Monoids

If $f : A^* \to X$ then say $w \equiv_f u$ iff $f(z \circ w \circ y) = f(z \circ u \circ y)$ for all $z, y \in A^*$. Let $[w]_{/f} = \{u \in A^*, u \equiv w\}$. Then define $[w]_{/f} \cdot [z]_{/f} = [w \circ z]_{/f}$. The set of these classes with $\cdot$ comprises a monoid where $[w]_{/f} \cdot [\Lambda]_{/f} = [w]_{/f}$ for the required identity. Say that this monoid is the monoid determined by f. Recall the construction of states from string functions above and the set S_f consisting of all the functions f_w so that $f_w(z) = f(w \circ z)$. Note that if $v, z \in [w]_{/f}$ it must be the case that for any string r $f_{r \circ z} = f_{r \circ v}$. So it is possible to associate each $[w]_{/f}$ with a map from $S_f \to S_f$ where $f_r \mapsto f_{r \circ z}$ for any z in $[w]_{/f}$. As a result, whenever S_f is finite, there are only a finite number of maps $S_f \to S_f$ so the monoid determined by f must also be finite.

Suppose $f(w) = h(f_1(u_1)\ldots, f_n(u_n))$ so that $u_i(wa) = u_i(w) \circ z_i$ where z_i only depends on the feedback from factors indexed by $j < i$. That is, there are $r_1 \ldots r_n$ so that $z_1 = r_1(a)$ and $z_{i+1} = r_{i+1}(a, f(w,1)\ldots, f(w,i))$. In this case f is constructed in cascade where information flows only in one direction and the results of Krohn-Rhodes theory[4, 3] will apply.

If f is finite and represents a state machine with k states and each of the f_i are finite with k_i states in the represented state machine, then if $\Sigma_{j=1}^{j\leq n} k_j < k$ the factorization is an implementation of f by essentially simpler string functions — and it corresponds to a factorization of the monoid of f into simpler monoids.

Let $T_n(\Lambda) = 0$ and $T_n(wa) = T(w) + 1 \bmod n$. Now define G_n as a cascade of T_2's as follows:

$$G_n(w) = (T_2(u_1)\ldots, T_2(u_n)) \tag{13}$$

$$u_1(wa) = u_1(w) \circ \langle a \rangle = wa \tag{14}$$

$$u_{i+1}(wa) = u_{i+1}(w) \circ \begin{cases} \Lambda & \text{if } \exists j < i, T_2(u_j(w)) = 0 \\ \langle a \rangle & \text{otherwise} \end{cases} \tag{15}$$

This is called a "ripple carry adder" in digital circuit engineering: each counter increments only if the "carry" is propagating through all lower order counters. Put $H_n(w) = \Sigma_{i=1}^{i\leq n} T_2(u_i) \times 2^{i-1}$ where the u_i are as defined for G_n. Then $H_n = T_{2^n}$ and you cannot make a G_n which counts mod any number other then 2^n. Otherwise, the underlying monoid of T_k has a simple group factor (a prime cyclic group) and those cannot be factored into smaller elements without some feedback.

While the cascade decompositions may simplify the interconnect in one way, they do not necessarily indicate the most efficient or interesting decomposition in practice. Cascades are good designs for "pipelined" execution but may be slow if we have to wait for the data to propagate to the terminal element. And group qualities in data structures can correspond to "undo" properties. For example, consider a circular buffer - like those commonly used for UNIX type fifos/pipes. The idea is that "write" operations push data into the pipe and "read" operations remove data in order of the "writes". The memory used to hold the data is allocated in a cycle. One way to implement such a buffer is to decompose it into an array of k memory locations and a mod k counter. A write operation causes an increment of the counter and a store of data in the appropriate memory location. The increment has an inverse, the write does not. But the result is that a write can be "forgotten". Perhaps factoring off group-like components will reveal other possibilities for this type of partial inverse.

References

1. Michael A. Arbib. *Theories of Abstract Automata.* Prentice-Hall, 1969.
2. Ferenc Gecseg. *Products of Automata.* Monographs in Theoretical Computer Science. Springer Verlag, 1986.
3. A. Ginzburg. *Algebraic theory of automata.* Academic Press, 1968.
4. W.M.L. Holcombe. *Algebraic Automata Theory.* Cambridge University Press, 1983.

5. E.F. Moore, editor. *Sequential Machines: Selected Papers*. Addison-Welsey, Reading MA, 1964.