Primitive Recursion and State Machines

Victor Yodaiken

yodaiken@finitestateresearch.com

July 23, 2009

Abstract

Methods for specifying Moore type state machines (transducers) abstractly via primitive recursive functions and for defininining parallel composition via simultaneous primitive recursion 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.

1 Introduction

A transducer or Moore type automata is readily seen to define a function on sequences: each sequence of inputs determines a value that is the output of the transducer in the state reached by following the sequence from the initial state. Primitive recursion offers a convenient method for defining these *sequence functions* and working with the large, multi-level, interacting, and often partially specified state machines encountered in computer engineering. While computer scientists have resorted to many exotic mathematical objects in an effort to evade the perceived limits of state machines, many of those limitations can be removed by using the sequence function presentation.

A simple primitive recursive definition of a sequence function consists of a pair of rules $f(\lambda) = x_0$ where λ is the empty sequence (which leads to the initial state), and $f(wa) = h(a, f(w))$ where wa is the sequence obtained by appending a to w. Such a function completely defines the input/output behavior of a transducer — which is often all we care about. By defining automata in terms of functions we can avoid the enumeration of state and distinctions based on artifacts of the representation - such as the names of states and the presence or absence of unreachable or duplicated states. Further, the sequence function representation is convenient when transducers are only partially specified or depend on parameters such as size of memory or are known only via constraints on behavior.

Parallel composition can be attacked using simultaneous recursion. Given $f_1 \ldots, f_n$, we simultaneously define f and f^* so $f(w,i) = f_i(f^*(w,i))$ where f^* computes an input sequence for the "factor" state machine represented by f_i. The idea is that when a is appended to w, the next inputs for each factor i are computed by f^* from both the input a and the *feedback* — $f(w,1) \ldots, f(w,n)$. If $f^*(w,i) = z$ then $f^*(wa) = z \circ u$ where $\circ$ denotes concatenation of sequences and u is some function of $(a, i, f(w,1) \ldots, f_n(w,n))$. For example we might define $u_i = \langle getmessage[m,j] \rangle$ if $f(w,i)$ indicates i is willing to accept a message, $f(w,j)$ indicates j wants to sent m to i and perhaps $f(w,k)$ indicates that communication between j and i is permitted. Results of classical automata theory can be carried over to this type of function composition to show a relationship between the feedback, the underlying group structure of the monoid induced by f, and the extent to which a function can be factored into simpler functions.

In what follows, the correspondence between sequence functions and transducers is made clear, the correspondence between the simultaneous recursion scheme given above to a "general product" of automata is proven, some illustrations of the practical utility of the method are provided, and some implications are drawn for the study of automata structure and algebraic automata theory.

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.

The set A^* contains all finite sequences over A including the empty sequence λ. Let wa denote the sequence obtained by appending $a \in A$ to $w \in A^*$ and let $w \circ z$ denote the sequence obtained by concatenating $z \in A^*$ to $w \in A^*$.

2.1 Representations

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

$$\delta^*(s,\lambda) = s \text{ and } \delta^*(s,wa) = \delta(\delta^*(s,w),a). \tag{1}$$

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.

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.

Nerode[Arb68] showed that there is a construction of a Moore machine $\mathscr{M}(f)$ from any $f : A^* \to X$ via a left equivalence relation. Given f, say $w \sim_f u$ if and only if $f(wz) = f(uz)$ for all $z \in A^*$. The relation $\sim_f$ is readily seen to be an equivalence relation. The set A^* is partitioned by $\sim_f$ into disjoint classes of equivalent sequences: $[w]_f = \{u : u \sim_f w, u \in A^*\}$. The set of these equivalence classes $A^*/\sim_f = \{[u]_f : u \in A^*\}$ can be the state set of $\mathscr{M}(f)$ and the transition and output functions are given by $\delta_f([w]_f, a) = [wa]_f$ and $\gamma_f([w]_f) = f(w)$.

$$\mathscr{M}(f) = \{A, X, A^*/\sim_f, [\lambda]_f, \delta_f, \gamma_f\}.$$

Since $f(w) = \gamma_f(\delta_f^*([\lambda]_f, w)$ by definition f is the representing function of $\mathscr{M}(f)$.

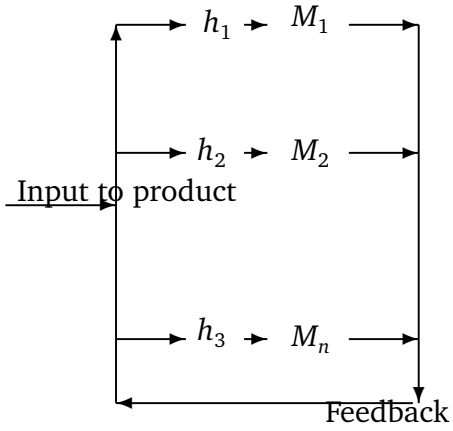
Any M_2 that has f as a representing function can differ from $M_1 = \mathscr{M}(f)$ only in names of states and by including unreachable and/or duplicative states. That is, there may be some w so that $\delta_1^*(start_1, w) \neq \delta_2^*(start_2, w)$ but since $w \sim_f w$ it must be the case that 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 $\mathscr{M}(f)$ as *the* Moore machine represented by f.

Say that f is finite if $\mathscr{M}(f)$ is finite state. While finite sequence functions are the only ones that can directly model digital computer devices or processes[1], infinite ones are often useful in describing system properties.

2.2 Products

The product, which Gécseg [G86] calls a "general product" [2] connects factor state machines so that the input to each factor is a function of the input to the composite machine and the outputs of some or all of the factors (this is the "feedback"). Pictorially, the Gécseg product is straightforward: n machines are connected via n maps that determine communication among the machines.



Suppose we have a collection of (not necessarily distinct) Moore machines $M_i = (A_i, X_i, S_i, start_i, \delta_i, \lambda_i)(0 < i \leq n)$ that are to be connected to construct a new machine with alphabet A. The intuition is that when an input a is applied to the system, the connection map computes a sequence of inputs for M_i from the input a and the outputs of the factors (*feedback*). I have made

[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] I'm using a slight modification.

the connection maps generate sequences instead of single events so that the factors can run at non-uniform rates. If $h_i(a,\vec{x}) = \lambda$, then M_i skips a turn.

Definition 2.1 General product of automata
Given $M_i = (A_i, X_i, S_i, start_i, \delta_i, \gamma_i)$ and $h_i : A \times \Pi_{i=1}^n X_n \to A_i^*$ for $0 < i \leq n$ define the Moore machine: $M = \mathscr{A}_{i=1}^n[M_i, h_i] = (A, X, S, start, \delta, \gamma)$

- $S = \Pi_{i=1}^n S_i$ and $start = (start_1 \ldots start_n)$
- $X = \{(x_1 \ldots, x_n) : x_i \in X_i\}$ and $\gamma((s_1 \ldots, s_n)) = \ldots \gamma_i(s_i) \ldots)$.
- $\delta((s_1 \ldots, s_n), a) = (\delta_1^*(s_1, h_1(a, \gamma(s))) \ldots \delta_n^*(s_n, h_n(a, \gamma(s))))$.

Theorem 2.1 **If** each f_i represents M_i and $f(w,i) = f_i(f^*(w,i))$
and $f^*(wa,i) = f^*(w,i) \circ h_i(a, f(w,1) \ldots, f(w,n))$
and $f^*(\lambda, i) = \lambda$
and $M = \mathscr{A}_{i=1}^n[M_i, h_i]$ **then** $f'(w) = (f(w,1) \ldots, f(w,n))$ represents M

Proof: Note that $f(w,i) = f_i(f^*(w,i))$ by definition and each f_i represents M_i so

$$f_i(z) = \gamma_i(\delta_i^*(start_i, z)) \tag{2}$$

and what we have to show is that $f^*(w,i)$ is correct so that

$$\delta^*(start, w) = (\ldots \delta_i^*(start_i, f^*(w,i)) \ldots) \tag{3}$$

The theorem follows directly from 3 because: $f'(w) = (\ldots f(w,i) \ldots) = (\ldots f_i(f^*(w,i)) \ldots)$
but $\gamma(\delta^*(st, w)) = \gamma(\ldots (\delta_i^*(start_i, f^*(w,i)) \ldots))$
by the definition of M and 3 so
$\gamma(\delta^*(st, w)) = (\ldots \gamma_i(\delta_i^*(start_i, f^*(w,i))) \ldots)$
$= (\ldots f_i(f^*(w,i)) \ldots) = f'(w)$.

Equation 3 can be proved by induction on w. Since $f^*(\lambda, i) = \lambda$ the base case is obvious. Now suppose that equation 3 is correct for w and consider wa.

Let $\delta(start, w) = s = (s_1 \ldots, s_n)$ and let $f^*(w,i) = z_i$. 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:
$\delta^*(start, wa) = (\ldots \delta_i^*(\delta_i^*(start, z_i), h_i(a, f'(w))) \ldots)$
$= (\ldots \delta_i^*(start, z_i \circ h_i(a, f'(w)) \ldots)$
$= (\ldots \delta_i^*(start, f^*(wa, i) \ldots)$
proving 3 for wa.

3 Examples

This section begins with a variety of sequence functions defined using simple primitive recursion and then shows how to define products.

3.1 Simple machines

Consider a single bit store machine over an alphabet $A = \{0, 1\}$.

$$B(\lambda) = 0 \text{ and } B(wb) = b \tag{4}$$

If we defined the storage state machine over an infinite alphaphet then it would be infinite state. It's trivial to create a store over any set.

Definition 3.1 *For any set X and $x_0 \in X$, say Y* ***is a store over*** *X with initial value x_0 if and only if:*

$$Y(\lambda) = x_0 \textit{ and } Y(wx) = x \textit{ for } x \in X \tag{5}$$

It's sometimes useful not to specify the initial state value if the device or system needs to be first brought to a known state for proper use.

An unbounded counter

$$T(\lambda) = 0 \text{ and } T(wa) = 1 + T(w) \tag{6}$$

represents an infinite state machine but may be useful in specifying how a finite state machine operates. A finite counter:

$$T_n(\lambda) = 0 \text{ and } T_n(wa) = (1 + T_n(w)) \bmod n \tag{7}$$

A finite counter with explicit increment and decrement and reset operations specified to ignore other inputs:

$$C_n(\lambda) = 0 \tag{8}$$

$$C_n(wa) = \begin{cases} min(n, 1 + C_n(w)) & \text{if } a = increment \\ max(0, C_n(w) - 1) & \text{if } a = decrement \\ 0 & \text{if } a = reset \\ C_n(w) & \text{otherwise.} \end{cases} \tag{9}$$

Given an alphabet $A = \{0,1\}$ a bounded "shift-register" can be defined recursively in a purely arithmetic way as:

$$R_n(\lambda) = 0 \tag{10}$$

$$R_n(wa) = (2 * R_n(w)) \bmod 2^n + a \tag{11}$$

Or we could expand the alphabet to $A = \{0,1,reset\}$ and define

$$R'_n(\lambda) = 0 \tag{12}$$

$$R'_n(wa) = \begin{cases} (2 * R'_n(w) \bmod 2^n + a) & \text{if } a \in \{0,1\} \\ 0 & \text{if } a = reset; \end{cases} \tag{13}$$

Both R_n and R'_n are obviously finite state. Defining, $E_n(w) = R_n(w)/2^{n-1}$ hides the interior state of R_n and only outputs the highest order bit.

Let $\vec{O}^n$ be the n-bit tuple of all zeros. To make the bits visible define S_n

$$\text{For } n > 1$$

$$S_n(\lambda) = \vec{0}^n \tag{14}$$

$$S_n(wa) = \begin{cases} (a, b_1, \ldots b_{n-1}) & \text{if } a \in \{0,1\} \\ & \text{and } S_n(w) = (b_1 \ldots, b_n) \\ \vec{0}^n & \text{if } a = reset; \end{cases} \tag{15}$$

For a vector $\vec{b} = (b_1 \ldots b_n)$ let $\vec{b}[i] = b_i$ indicate indexing. Consider $S'_n(w) = \Sigma_{i=1}^{n} 2^{i-1} * S_n(w)[i]$. Note that $S'_n(\lambda) = 0 = S'(w \circ \langle reset \rangle)$. And for $b \in 0,1$, $S'(wb) = b + \Sigma_{i=1}^{n-1} 2^i * S_n(w)[i]$ since $S_n(w)[i] = S_n(wb)[i+1]$.Thus, $S'_n(w) = R'_n(w)$

$0 \xrightarrow{\textbf{or}} 1$ **Shift Register** $\xrightarrow{\textbf{value}}$

A bounded queue can be defined to ignore pushes when it is full. If $A = \{pop\} \cup \{push[v] : v \in V\}$ define:

$$Q_n(\lambda) = () \tag{16}$$

$$Q_n(wa) = \begin{cases} () & \text{if } Q_n(w) = (v) \text{ and } a = pop \\ (v_1 \ldots, v_{j-1}) & \text{if } Q_n(w) = (v_1 \ldots v_j) \text{ for some } j > 1; \\ & \text{and } a = pop; \\ (v_1 \ldots, v_j, v) & \text{if } Q_n(w) = (v_1 \ldots v_j) \text{ for some } j < n; \\ & \text{and } a = push[v]; \\ Q_n(w) & \text{otherwise;} \end{cases} \tag{17}$$

3.2 Product machines

Since it is always the case here that $f^*(\lambda, i) = \lambda$, I'll just leave it implicit in what follows.

The shift register defined above can be constructed as a product of simpler machines – the bit store defined in equation 4.

$$\text{Define } G(w,i) = B(G^*(w,i)) \tag{18}$$
$$G^*(w,1) = w \tag{19}$$
$$G^*(wa, i+1) = G^*(w, i+1) \circ \langle G(w,i) \rangle \tag{20}$$
$$\text{Define } G'(w) = \Sigma_{i=1}^{n} 2^{i-1} G(w,i) \tag{21}$$

To see how the product works: $G(wb,1) = b$ since $G^*(wb,1) = wb$ and then $G(wb,1) = B(wb) = b$. And $G(wb, i+1) = G(w,i)$ since $G^*(wb, i+1) = z \circ \langle G(w,i) \rangle$ for some z and if we let $b = G(w,i)$ then $G^*(wa, i+1) = zb$.

$G'(\lambda) = 0$ since $G(\lambda, i) = B_i(\lambda) = 0$. Suppose $G'(w) = R_n(w)$. Then consider $G'(wb)$ which is equal to $\Sigma_{i=1}^{n} 2^i * G(wb, i)$ which is $2^0 b + 2^1 * G(w,2) \ldots + 2^{n-1} G(w,n)$ which is $2G'(w) \bmod 2^n + b$. So $G'(w) = R_n(w)$.

Note that the obvious realization of R_n is a state machine with 2^n states where $\gamma(x) = x$ and $\delta(x,b) = 2 * x \bmod 2^n + b$. But G replaces that with n state state machines which each have 2 states, reducing the total number of states to $2n$. G' just modifies the output map.

Construction of a queue from copies of a bit store and a counter can be done using the counter to track how many elements are in the queue. Note that there are three different alphabets: the queue alphabet of the product, the alphabet of the counter and the bit store alphabet.

$$\text{Let } Y \text{ be a store over} V$$
$$\text{And } C_n \text{ be as defined in equation 9}$$
$$\text{For } 0 < i \leq n, U_n(w,i) = Y(U_n^*(w,i)) \tag{22}$$
$$\text{and } U_n(w, n+1) = C_n(U_n^*(w, i+1)) \tag{23}$$
$$\text{where } U_n^*(wa, i) \tag{24}$$

$$= U_n^*(w,i) \circ \begin{cases} \langle v \rangle & \text{if } i = 1 \text{ and } a = push[v] \\ & \text{and } U_n(w, n+1) < n \\ \langle U(w, i-1) \rangle & \text{if } 1 < i \leq n \text{ and } a = push[v] \\ & \text{and } U_n(w, n+1) < n \\ \langle U_n(i-1) \rangle & \text{if } i < n \text{ and } a = pop \\ \langle increment \rangle & \text{if } i = n+1 \text{ and } a = push[v] \\ & \text{and } U_n(w, n+1) < n \\ \langle decrement \rangle & \text{if } i = n+1 \text{ and } a = pop \\ \lambda & \text{otherwise} \end{cases} \tag{25}$$

Now put $Size(w) = U_n(w, n+1)$ and define $NQ(w) = ()$ if $Size(w) = 0$ and $NQ(w) = (U_n(w,1) \ldots U_n(w, Size(w)))$ otherwise. Showing that $NQ(w) = Q_n(w)$ is straightforward.

4 More on representation and some algebra

A number of results follow from theorem 2.1.

Theorem 4.1 *For M and f constructed as products as above in theorem 2.1.*

- *There are an infinite number of distinct products $M' = \mathscr{A}_{i=1}^{k}[N_i, g_i]$ 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 $M' = \mathscr{A}_{i=1}^{k} r[Z_i, g_i]$ 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.*

Theorem 4.2 *If g has a finite image and each f_i is finite state and $F(\lambda) = x_0$ and: $F(wa) = g((F(w), f_1(w), \ldots f_n(w)), a)$*
then F is finite state

Proof. Let X be the image of g and define $T(\lambda) = x_0$ and $T(wx) = x$. Clearly, T is finite state if its alphabet is restricted to X. Define $E = \prod_{i=1}^{n+1}[f_i', h_i]$ so that $f_i' = f_i$ for $i \leq n$ and $f_{n+1} = T$. Let $h_{n+1}(\vec{y}, a) = g(\vec{y}, a)$ and let $h_i(\vec{y}, a) = \langle a \rangle$ for $i < n+1$. Since $E(w)_{n+1} = F(w)$ and E must be finite state, the result follows.

4.1 Monoids

If $f : A^* \to X$ then say $w \equiv_f u$ iff $f(zwy) = f(zuy)$ for all $z, y \in A^*$. Then $A^*/\equiv_f$ is a monoid under the operation of concatenation of representative elements. Let $[w]_{/f} = \{u \in A^*, u \equiv w\}$. Then define $[w]_{/f} \cdot [z]_{/f} = [wz]_{/f}$. The set of these classes with $\cdot$ is a monoid where $[w]_{/f} \cdot [\lambda]_{/f} = [w]_{/f}$ for the required identity.

Suppose $f(w,i)$ is defined from $f_1 \ldots, f_n$ so that $G^*(wa,i) = G^*(w,i)z_i$ 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 $z_i = r_1(a)$ and $z_{i+1} = r_i(a, f(w,1) \ldots f(w,i-1))$. In this case f is constructed in cascade where information flows only in one direction. The function G defined above by equation 18 is an example of such a system. In this case the results of Krohn-Rhodes theory[Hol83, Gin68] will apply: and R_n can be reduced to a cascade product of flip-flops because the monoid induced by $\equiv_{R_n}$ is "group free".

Consider C_n defined in equation 9. If $n = 2^k$ then C_2's.

$$G_n(w,i) = C_2(G^*(w,i)) \tag{26}$$

$$G^*(wa,1) = D_2(wa) \tag{27}$$

$$G^*(wa,i+1) = \begin{cases} \langle a \rangle & \text{if } \Pi_{j=1}^{j<i} G(w,j) = 1 \\ \lambda & \text{otherwise} \end{cases} \tag{28}$$

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. Note that G_n is a cascade and that G_n counts properly if and only if $n = 2^k$ for some $k > 1$. Otherwise, the underlying group of C_n has simple group factor and those cannot be factored into smaller elements without some feedback.

References

[Arb68] Michael A. Arbib. *Algebraic theory of machines, languages, and semigroups*. Academic Press, 1968.

[Gécseg] Ferenc Gécseg. *Products of Automata*. Monographs in Theoretical Computer Science. Springer Verlag, 1986.

[Gin68] A. Ginzburg. *Algebraic theory of automata*. Academic Press, 1968.

[Hol83] W.M.L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1983.