Primitive Recursion and state machines

Victor Yodaiken
Copyright 2009.[*]
yodaiken@finitestateresearch.com

July 1, 2009

1 Introduction

In this note, I look at how primitive recursion and feedback relate, how to represent state machines as primitive recursive functions, and at a particularly powerful machine product in terms of primitive recursive composition. Although the relationship between maps on sequences and state machines was described by Myhill and Nerode in the late 1950s [Arb68], the relationships between primitive recursion and state transition and between recursive composition and machine products has not been well studied. Sequence functions offer a concise, scalable, and mathematically convenient alternative to the traditional set-of-states presentation of automata for describing the complex state systems encountered in computer engineering. The recursive composition described here offers a means of factoring designs and of constructing specifications compositionally. 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.

Section 2 reviews Moore machines and maps on sequences. The net section describes the ``general product" and some results about recursion and products. The last section looks at relationships between recursion and algebraic view of state machines[Gin68].

2 Basics

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

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

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.

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

[*]Permission granted to make and distribute complete copies for non-commercial use but not for use in a publication. All other rights reserved but fair use encouraged as long as properly cited.

λ. Given M, extend the transition function δ to A^* by:

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

So $\gamma(\delta^*(\textit{start}, w))$ is the output of M in the state reached by following w from M's initial state. Call $f_M(w) = \gamma(\delta^*(\textit{start}, w))$ the *representing function* of M.

The representing function captures the behavior of a Moore machine and, as shown below, *all* the interesting information about the Moore machine from a certain perspective. As we'll see below, for every map $f : A^* \to X$ there is a way to construct both a ``canonical" Moore machine represented by f and the monoid of that Moore machine.

0 **or** 1
Shift Register
value

Although Moore machines are usually limited to finite state sets and, in fact, finite Moore machines are the ones that correspond to digital computer systems and processes, this work contemplates both finite and infinite Moore machines. For example, an unbounded counter

$$C(\lambda) = 0 \text{ and } C(wa) = 1 + C(w)$$

represents an infinite state machine but may be useful in specifying how a finite state machine operates. Say that $f : A^* \to X$ is finite state if and only if f is the representing function of a finite state Moore machine. Given an alphabet $A = \{0, 1\}$ a bounded ``shift-register" can be defined recursively as follows:

$$\begin{array}{l} R_n(\lambda) = 0 \\ R_n(wa) = 2R_n(w) \bmod 2^n + a \end{array}$$

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

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

Both R_n and R'_n are obviously finite state.

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

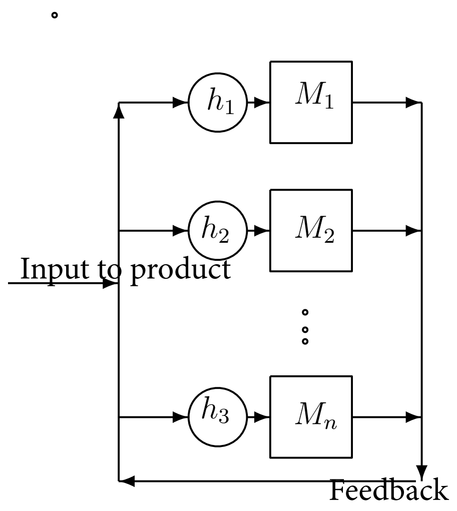
$$Q_n(wa) = \begin{cases} () & \text{if } Q_n(w) = (v) \text{ and } a = \textit{pop} \\ (v_1..., v_{j-1}) & \text{if } Q_n(w) = (v_1...v_j) \text{ for some } j > 1; \\ & \text{and } a = \textit{pop}; \\ (v_1..., v_j, v) & \text{if } Q_n(w) = (v_1...v_j) \text{ for some } j < n; \\ & \text{and } a = push[v];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

If f represents 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. For example, $E_n(w) = R_n(w)/2^{n-1}$ hides the interior state of R_n and only outputs the highest order bit.

Composing these elements into complex systems and factoring them into simpler elements is the subject of the next section.

3 Products

The product, which Gécseg calls a ``general product" [G8⊠6][1] connects the 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 $M_i = (A_i, X_i, S_i, \mathit{start}_i, \delta_i, \lambda_i)(0 < i \leq n)$ 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 the connection maps generate sequences instead of single events so that the factors can run at non-uniform rates. If $h_i(\vec{x}, a) = \lambda$, then M_i skips a turn.

3.1 Definitions of the product

The sequence product composition is based on recursive concatenation of sequences to construct a sequence for each factor from the product sequence. Use $\circ$ to indicate concatenation so that if $w = \langle a_1 \ldots, a_n \rangle$ and $u = \langle b_1 \ldots, b_k \rangle$ we have $w \circ u = \langle a_1 \ldots, a_n, b_1 \ldots, b_k \rangle$.Note that $w \circ \lambda = \lambda \circ w = w$.

Definition 3.1 General product of sequence functions

$$f = \mathcal{F}_{i=1}^{n}[f_i, h_i]$$

defines:

$$f(w) = (f_1(h_1^*(w)) \ldots f_n(h_n^*(w)))$$

where $h_i^*(\lambda) = \lambda$ and $h_i^*(wa) = h_i^*(w) \circ h_i(f(w), a)$.

This definition scheme is equivalent to the automata product.

Definition 3.2 General product of automata
Given $M_i = (A_i, X_i, S_i, \mathit{start}_i, \delta_i, \gamma_i)$ and $h_i : \Pi_{i=1}^{n} X_n \times A \to A_i^*$ for $0 < i \leq n$

[1] I'm using a slight modification.

define the Moore machine:

$$M = \mathcal{A}_{i=1}^{n}[M_i, h_i] = (A, X, S, start, \delta, \gamma)$$

so that:

- $S = \Pi_{i=1}^{n} S_i$
- $start = (start_1 ... start_n)$
- $X = \{(x_1 ..., x_n) : x_i \in X_i\}$.
- If $s = (s_1 ..., s_n)$, then $\gamma(s) = (\gamma_1(s_1) ... \gamma_n(s_n))$.
- If $s = (s_1 ..., s_n)$, then $\delta(s, a) = (\delta_1^*(s_1, h_1(\gamma(s), a)) ... \delta_n^*(s_n, h_n(\gamma(s), a)))$.

Theorem 3.1 *If each f_i represents M_i and $f = \mathcal{F}_{i=1}^{n}[f_i, h_i]$ and $M = \mathcal{A}_{i=1}^{n}[M_i, h_i]$ then f represents M*

Before going through the proof, some examples may help clarify how the product works.

3.2 Example of a shift register

The shift register defined above can be constructed as a product of simpler machines.

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

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

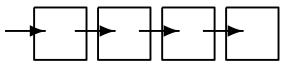
Define $G = \mathcal{F}_{i=1}^{n}[B_i, h_i]$ where h_i is to be given below. Note that $G(w) = (b_1 ..., b_n)$ where $b_i = B_i(h_i^*(w))$. For any $\vec{x} = (b_1 ..., b_n)$ write $\vec{x}_i = b_i$ to index the elements. The connection maps are written so that input b is given directly to factor 1 and ignored for all higher index factors where the input is determined by the output of the adjacent factor.

$$h_1(\vec{x}, b) = \langle b \rangle \text{ and for } 1 < i \leq n, h_i(\vec{x}, b) = \langle \vec{x}_{i-1} \rangle.$$

To see how the product works: $G(wb)_1 = b$ since $h_1(x, b) = \langle b \rangle$ and so if $z = h_1^*(w)$ then $G(wb)_i = B_i(z \circ \langle b \rangle) = b$. If $G(w) = (b_1 ..., b_n)$ then for $i > 1$ $h_i(G(w), b) = \langle b_{i-1} \rangle$. So for $i > 1$, $G(wb)_i = B_i(z \circ \langle b_{i-1} \rangle) = b_{i-1}$.

Define $G'(w) = \Sigma_{i=1}^{n} 2^{i-1} * G(w)_i$ $G'(\lambda) = 0$. Suppose $G'(w) = R_n(w)$. Then consider $G'(wb)$ which is equal to $b + \Sigma_{i=2}^{n} b2^i * G(wb)_{i-1} = 2 * G'(w) + b$. 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 2-state state machines: 2^n reduced to $2n$ and G' just modifies the output map.



Construction of a queue of bits Q_n from copies of B and a bounded counter is also straightforward. Here's the counter- note it has a different alphabet so we are dealing here with the queue alphabet of the product, the alphabet of the counter and the bit store alphabet. Let $C_n(\lambda) = 0$ and $C_n(wa) = min(n, 1 + C_n(w))$ if

$a = increment$ and $C_n(wa) = max(0, C_n(w) - 1)$ if $a = decrement$. Define $f_1...f_{n+1}$ so that $f_1...f_n$ are copies of B and f_{n+1} is a copy of C_n.

$$V_n = \mathcal{F}_{i=1}^{n+1}[f_i, h_i]$$

$$h_i(\vec{q}, a) = \begin{cases} \langle v \rangle & \text{if } i = 1 \text{ and } a = push[v] \text{ and } \vec{q} < n \\ \langle \vec{q}_{i-1} \rangle & \text{if } 1 < i \le n \text{ and } a = push[v] \text{ and } \vec{q} < n \\ \langle increment \rangle & \text{if } i = n+1 \text{ and } a = push[v] \text{ and } \vec{q} < n \\ \lambda & \text{if } i = n \text{ and } a = pop \\ \langle \vec{q}_{i+1} \rangle & \text{if } i < n \text{ and } a = pop \\ \langle decrement \rangle & \text{if } i = n+1 \text{ and } a = pop \end{cases}$$

Now put $Size(w) = V_n(w)_{n+1}$ and define $NQ(w) = ()$ if $Size(w) = 0$ and $NQ(w) = (V_n(w)_1...V_n(w)_{Size(w)})$ otherwise.

3.3 Proof of the product theorem and some results

Each f_i represents M_i so $f_i(z) = \gamma_i(\delta_i^*(start_i, z))$. We prove the stronger assertion:

$$(A) \quad f(w) = \gamma(\delta^*(start, w)) \text{ and } \delta^*(start, w) = (...\delta_i^*(start_i, h_i^*(w))...)$$

For λ, $h_i^*(\lambda) = \lambda$ so $\delta^*(start, \lambda) = (...\delta_i^*(start_i, \lambda)...)$ is obviously correct. And we have $f(\lambda) = (...f_i(\lambda)...) = \gamma(\delta^*(start, \lambda))$.

Now suppose (A) holds for w and consider wa.

$h_i^*(wa) = h_i^*(w) \circ h_i(\vec{x}, a)$ where $\vec{x} = f(w) = (...f_i(h_i^*(w)...)$. Note that $\delta^*(start, wa) = \delta(\delta^*(start, w), a)$.
Let $s = (s_1, ...s_n) = \delta^*(start, w)$. Then $\delta(s, a) = (...\delta_i^*(s_i, h_i(\vec{x}, a))...)$ where $\vec{x} = f(w)$ by the recursive hypothesis.
It follows that: $\delta_i(s_i, h_i(\vec{x}, a)) = \delta_i^*(\delta_i^*(start_i, h_i^*(w)), h_i(\vec{x}, a)) = \delta_i^*(start_i, h_i^*(w) \cdot h_i(f(w), a)) = \delta_i^*(start_i, h_i^*(wa))$ as claimed.

A number of results follow.

Theorem 3.2 *For M and f as above.*

- *There are an infinite number of distinct products $M' = \mathcal{F}_{i=1}^k[N_i, g_i]$ 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 $M' = \mathcal{F}_{i=1}^k r[Z_i, g_i]$ 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 3.3 *If g has a finite image and each f_i is finite state and $F(\lambda) = x_0$ and:*
$F(wa) = g((F(w), f_1(w), ...f_n(w)), a)$
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 $i \le n$ 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 Algebraic view

4.1 The left equivalence

Nerode showed that there is a construction of a Moore machine $\mathcal{M}(f)$ 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 $\mathcal{M}(f)$ and the transition and output functions are given by $\delta_f([w]_f, a) = [wa]_f$ and $\gamma_f([w]_f) = f(w)$.

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

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

f is finite state if and only if $\mathcal{M}(f)$ is finite state. In fact, it is easy to show that any M' that has f as a representing function can differ from $\mathcal{M}(f)$ only in names of states and by including unreachable and/or duplicative states (if $\gamma(\delta^*(s, w)) = \gamma(\delta^*(s', w))$ for all w, then s and s' are duplicative). 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.

4.2 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 $f = \mathcal{F}_i^k[f_i, h_i]$ where each h_i only depends on the feedback from factors indexed by $j < i$. That is, there are $g_1 \ldots g_n$ so that $h_1(x, a) = g_1(a)$ and for each $i > 1$, we have $h_i((x_1 \ldots x_n), a) = g_i((x_1 \ldots x_{i-1}), a)$. So f is constructed in cascade where information flows only in one direction. The function G defined above 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 D_n where $D_n(\lambda) = 0$ and $D_n(wa) = D_n(w) + 1 \bmod n$. For D_8 there is an easy factorization into the D_2's. Let $Z_i = C_2$ so we don't run into indexing conflicts, let $h_1(x, a) = \langle a \rangle$ and let $h_i(x, a) = \lambda$ if $\Pi_{j=1}^n x_{i-1} = 0$ and $h_i(x, a) = \langle a \rangle$ otherwise. 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. Then $G_n = \mathcal{F}_i^n[Z_i, h_i]$ and works as a mod n counter only when n is a power of 2. Otherwise, the underlying group of D_n has simple group factors 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.

[G8☒6] 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.