Primitive Recursive Transducers

Victor Yodaiken
Copyright 2009.*
yodaiken@finitestateresearch.com

August 9, 2009

Abstract

Methods for specifying Moore type state machines (transducers) abstractly via primitive recursive 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.

1 Introduction

The traditional state-set presentations of automata are hard to use when state sets are large, when systems are composed, or when systems are only partially known or specified. 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. Fortunately, state machines can be effectively presented as recursive functions for computing outputs from input sequences. These *transducer functions* can be shown to be strongly equivalent to Moore type states machines but abstract out some details that are not particularly interesting. Write λ for the empty string and wa for sequence addition - that is for appending element a to sequence w. A definition of the

*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.

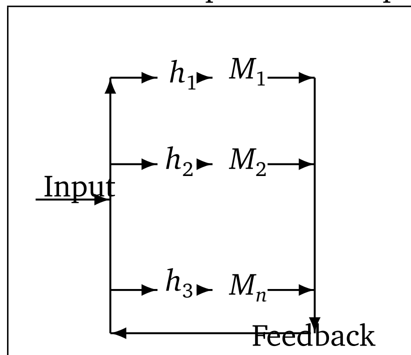
following type completely determines the input-output behavior of a state machine so that $f(w)$ is the output of the state machine in the state reached by following w from the initial state.

$$f(\lambda) = x_0, \quad f(wa) = h(a, f(w)) \tag{1}$$

The output can be modified by standard function composition

$$f(w) = h(gw) \tag{2}$$

But the interesting type of composition is simultaneous recursion[Pet67]. We can define f and f^* simultaneously so that f "encapsulates" a collection of previously defined transducer functions and f^* generates input sequences for the encapsulated transducer functions. This type of definition creates transducer functions that correspond to composite state machines like the one shown here:



Let $w \circ z$ be the sequence obtained by concatenating sequence z onto the end of sequence w[1]. The simultaneous definitions look like this:

$$\text{For: } g_1, \ldots g_n \text{ and } h_1 \ldots, h_n$$

$$f(w,i) = g_i(f^*(w,i)) \tag{3}$$

$$f^*(\lambda, i) = \lambda \tag{4}$$

$$f^*(wa,i) = f^*(w,i) \circ h_i(a, f(w,1) \ldots, f(w,n)) \tag{5}$$

If each g_i corresponds to an M_i, the transducer function f corresponds to the product of the M_i as shown in the drawing above in a way that will be made precise below.

[1] $w \circ \lambda = w$, $w \circ ua = (w \circ u)a$.

To illustrate, define mod k counters.

$$C_k(\lambda) = 0, \tag{6}$$

$$C_k(wa) = \begin{cases} 0 & \text{if } a = reset; \\ C_k(w) + 1 \bmod k & \text{if } a = increment; \\ C_k(w) & \text{otherwise} \end{cases} \tag{7}$$

and a device created from an array of C_2 counters that can increment, reset, and shift left (divide by 2). Note that each input a induces a sequence of inputs for each factor transducer — and that input sequence is λ if we want to leave the factor in an unchanged state.

$$\text{For } 0 < i \leq n, \quad D(w,i) = C_2(D^*(w,i)) \tag{8}$$

$$D^*(\lambda, i) = \lambda \tag{9}$$

$$D^*(wa,i) = D^*(w,i) \circ z_i \tag{10}$$

where

$$z_i = \begin{cases} \langle increment \rangle & \text{if } a = increment \\ & \quad \text{and } (i = 1 \text{ or } \Pi_{j=1}^{j<i} D(w,j) = 1); \\ & \text{or if } a = LShift \text{ and } i < n \\ & \quad \text{and } D(w,i) = 0 \text{ and } D(w,i+1) = 1; \\ \langle reset \rangle & \text{if } a = reset \\ & \quad \text{or if } a = LShift \text{ and} (i = n \\ & \qquad \text{or } (i < n \text{ and } D(w,i+1) = 0)) \\ \lambda & \text{otherwise} \end{cases} \tag{11}$$

In what follows, the correspondence between transducer functions and transducers is made clear, 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[Yod].

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:

$$\delta^*(s, \lambda) = s \text{ and } \delta^*(s, wa) = \delta(\delta^*(s, w), a). \tag{12}$$

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 $\mathcal{M}(f)$ from any $f : A^* \to X$ via a left equivalence relation. Given f, say $w \sim_f u$ if and only if $f(w \circ z) = f(u \circ z)$ 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 $\mathcal{M}(f)$. A similar construction can be used to produce a monoid from a transducer 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 $w \sim_f w$ 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 when $\mathscr{M}(f)$ is finite state. While finite sequence functions are the only ones that can directly model digital computer devices or processes[2], infinite ones are often useful in describing system properties.

2.2 Products

Suppose we have a collection of (not necessarily distinct) Moore machines $M_i = (A_i, X_i, S_i, start_i, \delta_i, \lambda_i)i$ for $(0 < i \leq n)$ 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 $h_i(a, \vec{x}) = \lambda$, 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: $M = \mathscr{A}_{i=1}^{n}[M_i, h_i] = (A, X, S, start, \delta, \gamma)$

- $S = \{(s_1 \ldots, s_n) : s_i \in X_i\}$ and $start = (start_1 \ldots, start_n)$
- $X = \{(x_1 \ldots, x_n) : x_i \in X_i\}$ and $\gamma((s_1 \ldots, s_n)) = \gamma_1(s_1) \ldots, \gamma_n(s_n))$.
- $\delta((s_1 \ldots, s_n), a) = (\delta_1^*(s_1, h_1(a, \gamma(s))) \ldots, \delta_n^*(s_n, h_n(a, \gamma(s))))$.

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 transducer 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 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

[2]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.

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

$$f_i(z) = \gamma_i(\delta_i^*(start_i, z)) \tag{13}$$

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

$$\delta^*(start, w) = (\dots \delta_i^*(start_i, f^*(w,i))\dots) \tag{14}$$

The theorem follows directly from 14 because: $f'(w) = (\dots f(w,i)\dots) = (\dots f_i(f^*(w,i))\dots)$
but $\gamma(\delta^*(st,w)) = \gamma(\dots(\delta_i^*(start_i, f^*(w,i))\dots))$ by the definition of M and 14 so
$\gamma(\delta^*(st,w)) = (\dots \gamma_i(\delta_i^*(start_i, f^*(w,i)))\dots)$
$= (\dots f_i(f^*(w,i))\dots) = f'(w)$.

Equation 14 can be proved by induction on w. Since $f^*(\lambda, i) = \lambda$ the base case is obvious. Now suppose that equation 14 is correct for w and consider wa.

Let $\delta(start, w) = s = (s_1 \dots, s_n)$ 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:
$\delta^*(start, wa) = (\dots \delta_i^*(\delta_i^*(start, z_i), h_i(a, f'(w)))\dots)$
$= (\dots \delta_i^*(start, z_i \circ h_i(a, f'(w))\dots)$
$= (\dots \delta_i^*(start, f^*(wa,i)\dots)$
proving 14 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.

3 More on representation and some algebra

A number of results follow from theorem 2.1.

Theorem 3.1 *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 3.2 *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 $T(\lambda) = \vec{x_0}$ where $\vec{x_0}$ is the vector of initial outputs of the f_i, and $T(wx) = x$. Clearly, T is finite state if its alphabet is restricted to X. Define $E(w, 1) = T(E^*(w, i))$ and $E(w, i + 1) = f_i(E^*(w, i + 1))$. Set $E^*(wa, 1) = \langle g(E(w, 1), E(w, 2), \ldots E(w, n + 1)) \rangle$ and $E^*(wa, i + 1) = \langle a \rangle$. Clearly $E(w, 1) = F(w)$ and since E is finite, F must be.

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 $z, y \in A^*$. 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 $[w]_{/f} \cdot [z]_{/f} = [w \circ z]_{/f}$. 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 $z_1 = r_1(a)$ 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. In this case the results of Krohn-Rhodes theory[Hol83, Gin68] will apply.

Consider some $F(w) = g(f(w, 1) \ldots, f(w, n))$ where $f(w, i) = g_i(f^*(w, i))$ is defined by simultaneous recursion. Then F is combining the outputs of the encapsulated factors of f. Note that if a transducer function is finite, its monoid is also finite (since the monoid can be considered a subset of the maps from states to states). If F is finite and represents a state machine with k states and each of the g_i are finite with k_i states in the represented state machine, then if $\Sigma_{j=1}^{j \le n} k_j < k$ then the factorization is an implementation of f by essentially simpler transducer 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:

$$G_n(w,i) = T_2(G^*(w,i)) \tag{15}$$

$$G^*(wa,1) = wa \tag{16}$$

$$G^*(wa,i+1) = \begin{cases} \langle a \rangle & \text{if } \Pi_{j=1}^{j\leq i} G(w,j) = 1 \\ \lambda & \text{otherwise} \end{cases} \tag{17}$$

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 $G'(w) = \Sigma_{i=1}^{i\leq n} G_n(w,i) \times 2^{i-1}$. Then $G' = T_k$ if and only if $k = 2^n$. 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

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

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

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

[Pet67] Rozsa Peter. *Recursive functions*. Academic Press, 1967.

[Yod] Victor Yodaiken. Technical reports. Technical Report http://www.yodaiken.com/papers/.