Primitive Recursive Presentations of Transducers and 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.

For a set A, let A^* be, as usual, the set of finite strings over A, including Λ the empty string. Arbib[2] defines automata as functions $f : A^* \to X^*$ with "finite support" where A is a finite set called the input

* March-2-2010 version of paper sent to LATA2010 with more up to date references, minor edits

alphabet, and X is a set called the output alphabet. That "finite support" is traditionally a state table or state-set/transition function presentation. This work presents automata in terms of primitive recursive functions $f : A^* \to X$ where 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. Primitive recursive algorithms for computing the output from the input sequence take the place of state-sets and tables. The result is a method for describing Moore type automata[9] while abstracting out details that are not interesting for current purposes.

If a is an input and w is a string, wa is the result of appending a to w. By defining $f(\Lambda) = x_0$ and $f(wa) = h(a, f(w))$, we can completely specify the operation of f. Although this is a very simple variation of well known techniques, it produces excellent results for describing very large scale and partially constrained state machines and also for describing state machines constructed as a system of interconnected component state machines.

Correspondence between a transducer M and a string function f.
Input:$w \Rightarrow$ Machine:$M \Rightarrow$ Output:x
$f(w) = x$

For example, a memory bank with n cells each holding a value between 0 and k so that $Mem(w, loc)$ is the contents of location loc in the state determined by w, could have an alphabet of inputs

$$\{write[l, j] : 0 \leq l < n, 0 \leq j \leq k\}$$

and be defined by:

$$Mem(wa, loc) = \begin{cases} j & \text{if } a = write[loc, j] \\ Mem(w, loc) & \text{otherwise} \end{cases}$$

The condition $0 \leq Mem(w, loc) \leq k$ might be all we want to specify about the initial value.

A second flavor of primitive recursion can be used to specify automata products that model composition and parallel (and concurrent) state change. The basic idea is to construct F from $f_1 \ldots, f_n$ and a "glue" function ϕ that defines the connections between components. Write

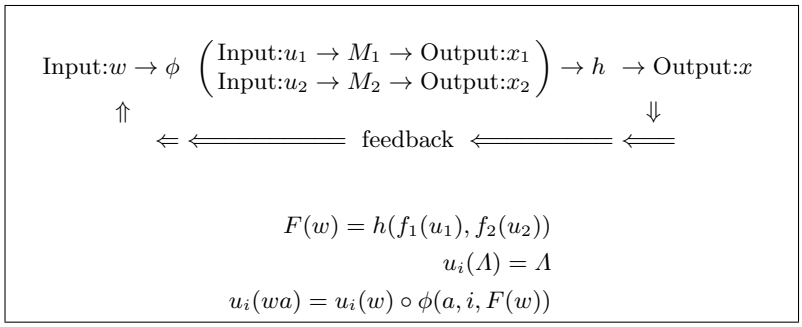
$$F(w) = h(f_1(u_1) \ldots f_n(u_n))$$

where each u_i is a primitive recursive function of F and w and ϕ. Writing $u_i(w)$ to make the functional dependency clear, $u_i(\Lambda) = \Lambda$ for the base case, and

$$u_i(wa) = u_i(w) \text{ concatenated with } \phi(a, i, F(w)).$$

This simultaneous recursive definition[10] corresponds to a very general form of automata product. The intuition is that $\phi(a, i, F(w))$ is the input sequence generated for component i when a is applied to the composite system.

The two factors case is illustrative. Write $w \circ u$ to indicate concatenation.



While the corresponding automata product has been described in the literature since the 1960s (see Gcseg's monograph[4] and see Domosi[3] for a more recent and more "algebraic" treatment) the presentation in terms of simultaneous primitive recursion is original to this line of work as far as I know[1]

For concreteness, consider a parallel implementation of a k element shift register. The components are copies of function L_V that are simple storage cells over a set V so that $L_V(\Lambda) = 0$ and $L_V(wb) = b$ for $b \in V$. The input alphabet is the set

$$\{RotateLeft, Left[b], Right[b] : b \in V\}$$

$$R(w) = \underbrace{(L_V(u_1) \ldots, L_V(u_k))}_{k} \quad (1)$$

$$u_i(\Lambda) = \Lambda \quad (2)$$

[1] I began using the general product as a semantics for a primitive recursive modal logic[12, 11] but eventually discovered that the formal logic was just an impediment to understanding.

$$u_i(za) = u_i(z) \circ \langle v \rangle \quad (3)$$

where

$$v = \begin{cases} b & \text{if } i = 1 \text{ and } a = \mathit{Right}[b] \\ L_V(u_{i-1}(z)) & \text{if } i > 1 \text{ and } a = \mathit{Right}[b] \\ b & \text{if } i = k \text{ and } a = \mathit{Left}[b] \\ L_V(u_{i+1}(z)) & \text{if } i < k \text{ and } a = \mathit{Left}[b] \text{ or } a = \mathit{RotateLeft} \\ B(u_k(z)) & \text{if } i = 1 \text{ and } a = \mathit{RotateLeft} \end{cases} \quad (4)$$

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 Automata and string functions

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, $\mathit{start} \in S$ is the initial state, $\delta : S \times A \to S$ is the transition function and $\gamma : S \to X$is the output function. In general, I am not requiring S to be finite although finite machines are the key subclass when we describe actual computing devices/programs. Sometimes it is useful to use, for example, an unbounded counter or an unlimited memory in a proof or construction so the generality is convenient.

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

$$\delta^*(s, \Lambda) = s \text{ and } \delta^*(s, wa) = \delta(\delta^*(s, w), a). \quad (5)$$

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

If f_M is the representing function of M, then $f'(w) = g(f_M(w))$ represents M' obtained by replacing γ with $\gamma'(s) = g(\gamma(s))$, but it may be that M' is no longer minimal.

The transformation from string function to transducer is also simple and depends on the well known congruence relation. Given $f : A^* \to X$

define function $(f+w) : A^* \to X$ so that $(f+w)(u) = f(w \circ u)$. Intuitively, if w_1 and w_2 lead to the "same state" then following any sequence of inputs from that state produces the same output, no matter whether we began by following w_1 or w_2. In this case, $(f + w_1) = (f + w_2)$ as functions even though w_1 and w_2 may be different strings. So we can consider each $(f + w)$ to represent a state. Let S_f be the set of all the states of f.

$$S_f = \{(f + w) : w \in A^*\} \tag{6}$$

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

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

and, by construction f is the representing function for $\mathcal{M}(f)$. If S_f is finite with cardinality k, say that the state set size of f is k

A similar construction can be used to produce a monoid from a string function as discussed below in section 3.

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 ϕ and an output map Γ. The output map simply computes a composite output from the tuple of factor outputs. 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 I use here is a variant of the product described in Gcseg's monograph[4]. In contrast to [4], I have made the connection maps generate string instead of single events so that the factors can run at non-uniform rates. If $\phi(a, i, \boldsymbol{x}) = \Lambda$, 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 Γ and ϕ define the Moore machine: $M = \mathcal{A}_{i=1}^{n}[M_i, \Gamma, \phi] = (A, X, S, start, \delta, \gamma)$

- S is the set of n-tuples over the S_i, $S = \{(s_1 \ldots, s_n) : s_i \in S_i\}$
- $start = (start_1 \ldots, start_n)$
- $X = \{h(x_1 \ldots, x_n) : x_i \in X_i\}$

- $\gamma((s_1 \ldots, s_n)) = \Gamma(\gamma_1(s_1) \ldots, \gamma_n(s_n))$.
- $\delta((s_1 \ldots, s_n), a) = (\delta_1^*(s_1, \phi(a, 1, \gamma(s))) \ldots, \delta_n^*(s_n, \phi(a, n, \gamma(s))))$.

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. There is no analogous problem in function domain

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 $u_i(wa) = u_i(w) \circ \phi(a, i, F(w))$
and $M = \mathcal{A}_{i=1}^n[M_i, h, \phi]$ **then** F represents M

Proof: Each f_i represents M_i so

$$f_i(z) = \gamma_i(\delta_i^*(start_i, z)) \tag{7}$$

Note that each s in the state set of M is a tuple $s = (s_1 \ldots s_n)$. We know $\gamma(\delta^*(start, w)) = \Gamma(\gamma(s)) = \Gamma(\ldots \gamma_i(\delta_i^*(start_i, w_i)) \ldots)$ for some w_i. All we have to show is that

$$\delta^*(start, w) = (\ldots \delta_i^*(start_i, u_i(w)) \ldots) \tag{8}$$

and then we have

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

It follows immediately that

$$\gamma(\delta^*(start, w)) = \Gamma(\ldots f_i(u_i(w))) \ldots) = F(w)$$

Equation 8 can be proved by induction on w. Since $u_i(\Lambda) = \Lambda$ the base case is obvious. Now suppose that equation 8 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:

$$\begin{aligned}
\delta^*(start, wa) &= \delta(\delta^*(start, w), a) && (9)\\
&= \delta(s, a) && (10)\\
&= (\ldots \delta_i^*(s_i, g(i, a, \gamma(s))) \ldots) && (11)\\
&= (\ldots \delta_i^*(\delta_i^*(start, u_i(w)), \phi(a, i, F(w))) \ldots) && (12)\\
&= (\ldots \delta_i^*(start, u_i(w) \circ \phi(a, i, F(w))) \ldots) && (13)\\
&= (\ldots \delta_i^*(start, u_i(wa)) \ldots) && (14)
\end{aligned}$$

proving 8 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 Structure

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.*

False modularity is a common problem in programming and computer engineering. We can almost always divide a complex system into simpler parts but the result may be more complex than the original,particularly if have just moved complexity into the communications from the computation. Suppose $F(w) = (f_1(u_1) \dots f_n(u_n))$ is finite where each f_i has a smaller state set than F. Consider the function $f_i'(w) = f_i(u_i(w))$. It may well be that the state set size of f_i' is not smaller than the state set of F because u_i implictly smuggles in a large state set.

If f is finite where $\mathcal{M}(f)$ has a state set with cardinality k, there is always some $F(w) = h(B(u_1) \dots B(u_n))$ where $n = \lceil \log_2 k \rceil$, and B is a binary store, and so that $F(w) = f(w)$ for all w. For this construction, we number the states of $\mathcal{M}(f)$ from $0, \dots K-1$ and encode this number as a binary string into the component B factors. The glue function $\phi(a, i, F(w))$ computes the encoded number c from $F(w)$, uses that to index the elements of S_f, computes $\delta(a, s_c)$, looks up the index c' of the result, and assigns the i^{th} bit of the result to the i^{th} component. The sum of the number of states in the construction of F is $2n \leq 2\log_2(k) + 1$ for an apparent vast simplification. But it's clear that this brute force reduction basically moves all the hard work to the glue function and, not

surprisingly $f'(w) = f_i(u_i(w))$ may turn out to have a state set as large as the state set of F.

There are a number of ways to limit the "complexity" of glue that might turn out to be interesting. One way is to use the "cascade" products that are studied in classical algebraic automata theory.

3.1 Cascade Decompositions

Say that a glue function $\phi(a, i, (x_1 \dots x_n))$ is loop-free if it does not depend on x_j for $j \geq i$. That is, ϕ is loop-free if the outputs of only lower numbered components can affect the input to each component. In particular $\phi(a, 1, \boldsymbol{x})$ does not depend on $\boldsymbol{x}$ at all. Say $F(w) = (f_1(u_1) \dots, f_n(u_n))$ is loop-free if and only if the glue function of $F(w)$ is loop-free.

The Myhill/Nerode congruence produces a monoid from any $f : A^* \to X$ consisting of the set of functions $\sigma_w : S_f \to S_f$ where $\sigma_w((f + z)) = (f{+}z{\circ}w)$. Note that if $(f{+}z_1) = (f{+}z_2)$ then $\sigma_w((f{+}z_1)) = (f{+}z_1{\circ}w) = (f + z_2 \circ w) = \sigma_w((f + z_2))$ by definition, preserving independence of representative. And σ_Λ is clearly an identity $\sigma_w\sigma_\Lambda(s) = \sigma_w(s) = \sigma_\Lambda\sigma_w(s)$. Finally, associativity is inherited from the associativity of concatentation since in this case function composition is equivalent to concatenation of subscripts. The set of maps $\sigma_w : w \in A^*$ is therefore a monoid under composition of fuctions. The monoid is finite if S_f is finite. So we can now apply all the results from algebraic automata theory although in a functional framework some things are simpler.

Call the monoid defined above, the characteristic monoid of a string function. Note that for a glue-free composition, $u_1(w \circ z) = u_1(w) \circ u_1(u)$. In fact, u_1 is a homomorphism from the free monoid A^* to the monoid $\{u_1(w) : w \in A^*\}$ with concatenation as the operator of both monoids. This carries over so that the characteristic monoid of the first factor machine must be a homomorphic image of the characteristic monoid of the composite machine. At this point we know, for example, that if the characteristic monoid of F is a simple group, than f must either be trivial or the characteristic monoid of f is isomorphic to the characteristic monoid of F.

Let $T_n(\Lambda) = 0$ and $T_n(wa) = T(w) + 1 \bmod n$. Now define G_n as follows:

$$G_n(w) = (T_2(u_1) \dots, T_2(u_n)) \tag{15}$$

$$u_1(wa) = u_1(w) \circ \langle a \rangle = wa \tag{16}$$

$$u_{i+1}(wa) = u_{i+1}(w) \circ \begin{cases} \Lambda & \text{if } \exists j < i, T_2(u_j(w)) = 0 \\ \langle a \rangle & \text{otherwise} \end{cases} \tag{17}$$

This is called a "ripple carry incrementer" 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.

More comprehensive treatments of Krohn-Rhodes theory can be found elsewhere[3, 7, 5, 8] but Hartmanis and Stearns still provide the best engineering perspective on classical machine decomposition[6]. All of that work is done using state-table or state diagram or state-set presentations of automata. It is somewhat puzzling to see repeated references to the functional presentation of state machines, and to recursion, without any further development in works such as [1, 2]. Among the advantages of working in the functional space is that state sets keep getting automatically minimized (by the equivalence in the construction of S_f), state names are in a canonical form if they need to be referenced at all.

For example, Michael Arbib notes the following:

> The reader may share the disconcertion I felt on first realizing the irreducibility of machines was not equivalent to irreducibility of semigroups. The reason for this inequivalence is simply that the output maps of M_1 and M_2 may make their output sets so small that neither M_1 nor M_2 alone can simulate M although one of S_1^M and S_2^M is big enough to simulate M. (Chapter 3 p. 46 of [1])

But using functional presentations, much of this goes away. The state sets of some f and of $f'(w) = g(f(w))$ may be radically different. The state set is generated by the operation of the function and is automatically minimized.

3.2 Decompositions with 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.

As a final note indicating an unexplored realm, consider limitations of feedback to one level so that $\phi(a, i, (x_1 \ldots, x_n)$ may depend on x_j for $j \leq i+1$. This limitation allows pipeline stages to, for example, stall until a computation completes in the next stage. For example, if process P_1 is sending data over a UNIX type pipeline to P_2, we would represent it as a product in which a third machine representing the buffering action of the pipe mechanism was placed in between the two process machines and P_1 stalls when the buffer is full and P_2 stalls when the buffer is empty. In this case, P_1 must get feedback from the buffering component and the buffering component must get feedback from P_2, but P_2 only receives downstream data.

References

1. Michael A. Arbib. *Algebraic theory of machines, languages, and semi-groups.* Academic Press, 1968.
2. Michael A. Arbib. *Theories of Abstract Automata.* Prentice-Hall, 1969.
3. Pal Domosi and Chrystopher L. Nehaniv. *Algebraic Theory of Automata Networks (SIAM Monographs on Discrete Mathematics and Applications, 11).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.
4. Ferenc Gecseg. *Products of Automata.* Monographs in Theoretical Computer Science. Springer Verlag, 1986.
5. A. Ginzburg. *Algebraic theory of automata.* Academic Press, 1968.
6. J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, 1966.
7. W.M.L. Holcombe. *Algebraic Automata Theory.* Cambridge University Press, 1983.
8. Chrystopher L. Nehaniv John Rhodes. *Applications of Automata Theory and Algebra by John Rhodes: Book Cover Applications of Automata Theory and Algebra : Via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games.* World Scientific Publishing Company, Incorporated, 2009.
9. E.F. Moore, editor. *Sequential Machines: Selected Papers.* Addison-Welsey, Reading MA, 1964.
10. Rozsa Peter. *Recursive functions.* Academic Press, 1967.
11. Victor Yodaiken. The algebraic feedback product of automata. In *DIMACS Workshop on Computer Aided Verification*, 1989.
12. Victor Yodaiken. Modal functions for concise representation of finite automata. *Information Processing Letters*, Nov 20 1991.