Symbolic Synthesis of Mealy Machines from Arithmetic Bistream Functions

Hansen, H.H.; Rutten, J.J.M.M.

published in
Scientific Annals of Computer Science
2010

document version
Publisher's PDF, also known as Version of record

Scientific Annals of Computer Science vol. 20, 2010
"Alexandru Ioan Cuza" University of Iaşi, Romania

Symbolic Synthesis of Mealy Machines from Arithmetic Bitstream Functions

Helle Hvid HANSEN[1] and Jan RUTTEN[2]

Abstract

In this paper, we describe a *symbolic* synthesis method which given an algebraic expression that specifies a bitstream function f, constructs a (minimal) Mealy machine that realises f. The synthesis algorithm can be seen as an analogue of Brzozowski's construction of a finite deterministic automaton from a regular expression. It is based on a coinductive characterisation of the operators of 2-adic arithmetic in terms of stream differential equations.

1 Introduction

A (binary) Mealy machine is a deterministic automaton which in each step reads an input bit, produces an output bit and moves to a next state. The induced mapping of input streams to output streams is a causal bitstream function, which we call the bitstream function realised by the Mealy machine. In this paper, we describe a *symbolic* synthesis method which given an algebraic expression that specifies a bitstream function f, constructs a minimal Mealy machine that realises f.

The inputs to our synthesis algorithm are called function expressions and they define bitstream functions in the algebra of 2-adic numbers. Here we use that the formal power series representation of a 2-adic integer can be seen as the bitstream of its coefficients. We describe the 2-adic algebra

[1]Technische Universiteit Eindhoven and Centrum Wiskunde & Informatica, P.O.Box 513, 5600 MB Eindhoven, The Netherlands, email: h.h.hansen@tue.nl.

[2]Centrum Wiskunde & Informatica and Radboud University Nijmegen, P.O.Box 94079, 1090 GB Amsterdam, The Netherlands, email: janr@cwi.nl.

below, but for now such a function expression can be thought of as a specification of a function on the rational numbers. The interesting property of 2-adic arithmetic is that it allows us to calculate with bitstream representations of rational numbers in an easy manner similar to how one computes with integers.

The synthesis algorithm described here can be seen as an analogue of Brzozowski's construction in [3] of a finite deterministic automaton from a regular expression. In particular, we show that the set of function expressions carries the structure of a Mealy machine by giving an inductive definition of derivative and output of function expressions. This Mealy machine of expressions is defined in such a way that the algebraic semantics coincides with the *behavioural* semantics of Mealy machines, which will be defined in terms of *causal stream functions*. Now given a function expression `F` specifying a function f, we obtain a realisation of f by the symbolic computation of the (sub)machine generated by `F`. (The generated submachine is in general not minimal, but we can ensure minimality by reducing expressions to normal form.) The language of 2-adic arithmetic allows the specification of functions that *cannot* be realised by any finite Mealy machine. But we shall identify a subclass of so-called *rational function expressions* for which a finite realisation exists.

In the design of digital hardware, Mealy machines specify the behaviour of sequential circuits, and there exist algorithms which construct from a (finite) Mealy machine, a sequential circuit which exhibits the specified behaviour. Combining these algorithms with our synthesis algorithm we thus obtain a complete construction from algebraic specification to sequential circuit.

In summary, the main contributions of the present paper are: (1) The elementary but useful observation that the set of all causal stream functions constitutes a final Mealy machine, which forms the basis for a behavioural semantics of Mealy machines in terms of their minimisation. (Although minimisation of Mealy automata is well known [4], its characterisation here by means of finality is new.) (2) The insight that given a rational function expression `F`, we can effectively construct by symbolic computation a (minimal) Mealy machine that realises the bitstream function specified by `F`.

The basis for the present synthesis algorithm was given in [18]. These ideas were developed into a proper algorithm in [8] (for an implementation, see [7]). Further results on complexity and size of realisations were included in the first author's PhD thesis [6, Ch. 3]. This paper contains a

short, but improved presentation of the basic results in the abovementioned work. In particular, the presentation of the Mealy machine of expressions, the algebraic semantics of function expressions, and the proof that algebraic semantics coincides with behavioural semantics for function expressions are new with respect to [6, 8, 18]. Moreover, the synthesis algorithm and complexity analysis have been simplified by making use of particular properties of rational function expressions. Finally, we mention that the theory underlying the present work is essentially coalgebraic (cf. [15]), but we have deliberately chosen for a presentation which does not require any familiarity with coalgebra.

Structure of the paper: In Section 2 we give the basic definitions regarding Mealy machines. In Section 3 we describe the 2-adic algebra of bitstreams and define the function expressions which serve as our specification language. In Section 4 we show how function expressions can be turned into a Mealy machine, and in Section 5 we present the synthesis algorithm for rational function expressions and analyse its time complexity. Finally, we discuss related and future work in Section 6.

2 Mealy Machines

We give the basic definitions on Mealy machines, streams and causal stream functions. We introduce the notion of stream function derivative and show how it can be used to turn the set of causal stream functions into a final Mealy machine, thus providing a characterisation of minimal Mealy machines.

Let A and B be arbitrary sets. A *Mealy machine* $\langle S, m\rangle$ with inputs in A and outputs in B consists of a set of states S and a transition function

$$m\colon S \to (B \times S)^A$$

This function maps a state $s_0 \in S$ to a function $m(s_0)\colon A \to (B \times S)$, which produces for every input $a \in A$ a pair $\langle b, s_1\rangle$, consisting of the output b and the next state s_1. We call a Mealy machine *binary* if inputs and outputs are taken from the set $2 = \{0, 1\}$. We will write

$$s \xrightarrow{a|b} t \quad \text{iff} \quad m(s)(a) = \langle b, t\rangle.$$

For an arbitrary set A, we denote by A^ω the set of *streams over* A, i.e., $A^\omega = \{\alpha \mid \alpha\colon \mathbb{N} \to A\}$. An element $\alpha \in A^\omega$ will be denoted by $\alpha =$

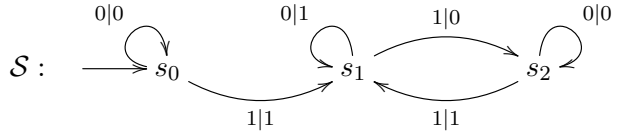
$(\alpha(0), \alpha(1), \alpha(2), \ldots)$. The head and tail maps on streams are denoted by hd and tl, respectively, that is, for $\alpha \in A^\omega$, $hd(\alpha) = \alpha(0)$ and $tl(\alpha) = (\alpha(1), \alpha(2), \alpha(3), \ldots)$. Later, in the context of stream differential equations, we will also refer to $hd(\alpha)$ and $tl(\alpha)$ as the *initial value* and *stream derivative* of α, respectively, and write α' instead of $tl(\alpha)$. Moreover, for $\alpha \in A^\omega$ and $n \in \mathbb{N}$, we write $\alpha\restriction_n$ for the finite prefix $(\alpha(0), \ldots, \alpha(n))$.

We define the (input-output) *behaviour* of a state s_0 in $\mathcal{S} = \langle S, m\rangle$ as the stream function $beh_{\mathcal{S}}(s_0)\colon A^\omega \to B^\omega$ which maps an input stream $(a_0, a_1, a_2, \ldots) \in A^\omega$ to the output stream $(b_0, b_1, b_2, \ldots) \in B^\omega$ given by the unique sequence of transitions

$$s_0 \xrightarrow{a_0|b_0} s_1 \xrightarrow{a_1|b_1} \cdots \xrightarrow{a_k|b_k} s_{k+1} \xrightarrow{a_{k+1}|b_{k+1}} \cdots$$

If $\mathcal{S}$ is clear from the context, we will often leave out the subscript and simply write $beh(s)$. We say that a state s in $\langle S, m\rangle$ *realises* a stream function f if $beh(s) = f$.

Example 1 The figure below shows an example of a binary Mealy machine which starting in state s_0 counts the number of ones in the input modulo 2. Formally, s_0 realises the function f defined on input stream $\alpha \in 2^\omega$ by $f(\alpha)(n) = \sum_{i=0}^{n} \alpha(i) \mod 2$ for all $n \in \mathbb{N}$.



Note that, for all $\alpha \in 2^\omega$ and $n \in \mathbb{N}$, $f(\alpha)(n)$ is determined by $\alpha(0), \ldots, \alpha(n)$.

We call a stream function $f\colon A^\omega \to B^\omega$ *causal* if the n-th element of the output stream $f(\alpha)$ depends only on the first n elements of the input stream. More formally, f is causal if for all $n \geq 0$ and all $\alpha, \beta \in A^\omega$:

$$\alpha\restriction_n = \beta\restriction_n \qquad \Longrightarrow \qquad f(\alpha)(n) = f(\beta)(n).$$

It is straightforward to prove that for any Mealy machine $\langle S, m\rangle$ and any $s \in S$, $beh(s)\colon A^\omega \to B^\omega$ is causal. Let Γ denote the set of all causal stream functions, i.e.,

$$\Gamma = \{f \mid f\colon A^\omega \to B^\omega \text{ and } f \text{ is causal }\}.$$

Next, we will define a function $\gamma\colon \Gamma \to (B \times \Gamma)^A$ such that $\langle \Gamma, \gamma \rangle$ is a Mealy machine. For $\alpha \in A^\omega$ and $a \in A$, we denote by $a : \alpha$ the stream $(a, \alpha(0), \alpha(1), \alpha(2), \ldots)$, and given $f\colon A^\omega \to B^\omega$, we write $f(a : -)$ for the stream function that maps α to $f(a : \alpha)$. Now let $f \in \Gamma$ and $a \in 2$. We define

$$\begin{array}{lcll} f[a] & := & hd \circ f(a:-) & \in \quad A^\omega \to B \\ f_a & := & tl \circ f(a:-) & \in \quad A^\omega \to B^\omega \end{array} \tag{1}$$

Since f is causal, it follows that also f_a is causal (hence $f_a \in \Gamma$), and that $f[a]$ is constant, hence $f[a]$ can be considered an element of B. We call $f[a]$ the *initial output of f on input a*, and f_a is the *stream function derivative of f on input a*. We define the transition function

$$\gamma\colon \Gamma \to (B \times \Gamma)^A \qquad \text{by} \qquad \gamma(f)(a) = \langle f[a], f_a \rangle \text{ for all } f \in \Gamma, a \in A.$$

This gives us an (infinite) Mealy machine $\mathbf{\Gamma} = \langle \Gamma, \gamma \rangle$ with transitions

$$f \xrightarrow{a|f[a]} f_a$$

Next we characterise $\langle \Gamma, \gamma \rangle$ using the following notion. A *homomorphism of Mealy machines* from $\langle S, m \rangle$ to $\langle S', m' \rangle$ is a function $h\colon S \to S'$ that preserves transitions: if $m(s)(a) = \langle b, t \rangle$ then $m'(h(s))(a) = \langle b, h(t) \rangle$; in other words,

$$s \xrightarrow{a|b} t \quad \Rightarrow \quad h(s) \xrightarrow{a|b} h(t)$$

Theorem 2 *For any Mealy machine $\mathcal{S} = \langle S, m \rangle$, the map $beh_{\mathcal{S}}\colon S \to \Gamma$ is the unique homomorphism from $\mathcal{S}$ to $\mathbf{\Gamma}$. In other words, $\mathbf{\Gamma}$ is a final Mealy machine.*

Proof: Let $\langle S, m \rangle$ be an arbitrary Mealy machine. We denote the output and next state functions defined by m at state $s \in S$ by o_s and d_s, respectively, that is, $m(s)(a) = \langle o_s(a), d_s(a) \rangle$. To see that $beh\colon S \to \Gamma$ is a homomorphism, let $s \in S$, $a \in A$ and $\alpha \in A^\omega$ be arbitrary. We have:

$$beh(s)[a] = hd(beh(s)(a:\alpha)) = o_s(a),$$

and by letting $s_0 = d_s(a)$ and $s_{i+1} = d_{s_i}(\alpha(i))$ for all $i \geq 0$, we have

$$\begin{array}{rcl} (beh(s)_a)(\alpha) & = & tl(beh(s)(a:\alpha)) \\ & = & tl(o_s(a), o_{s_0}(\alpha(0)), o_{s_1}(\alpha(1)), \ldots) \\ & = & (o_{s_0}(\alpha(0)), o_{s_1}(\alpha(1)), \ldots) \\ & = & beh(d_s(a))(\alpha). \end{array}$$

Hence $beh(s)_a = beh(d_s(a))$. The proof that $beh\colon S \to \Gamma$ is the unique homomorphism from $\mathcal{S}$ to $\mathbf{\Gamma}$ is left to the reader. □

The universality of $\langle \Gamma, \gamma \rangle$ can be expressed in yet another way. We need the following notions. For a state $s \in S$ of a Mealy machine $\langle S, m \rangle$, let

$$\langle\!\langle s \rangle\!\rangle \subseteq S$$

denote the smallest subset that contains s and is closed under transitions (for any inputs). Clearly, $\langle\!\langle s \rangle\!\rangle$ is also a sub*machine* of $\langle S, m \rangle$, by taking as its transition function the restriction of m to the set $\langle\!\langle s \rangle\!\rangle$. We call $\langle\!\langle s \rangle\!\rangle$ the *submachine generated by* s *in* $\langle S, m \rangle$. We call $\langle\!\langle s \rangle\!\rangle$ a *realisation* of f, if $beh_{\mathcal{S}}(s) = f$, i.e., s realises f (in $\mathcal{S}$). We say that a Mealy machine $\langle S, m \rangle$ is *minimal* if $beh\colon \langle S, m \rangle \to \langle \Gamma, \gamma \rangle$ is injective. From the finality of $\mathbf{\Gamma}$, we get:

Corollary 1 *For all $f \in \Gamma$, $\langle\!\langle f \rangle\!\rangle$ is a minimal realisation of f.*

Proof: Follows from the fact that the inclusion map $\langle\!\langle f \rangle\!\rangle \to \Gamma$ is an injective homomorphism of Mealy machines. □

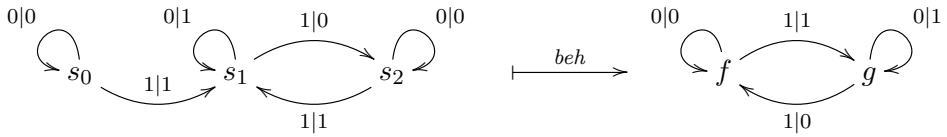
The final Mealy machine $\mathbf{\Gamma}$ thus consists of all the behaviours that can be realised by some Mealy machine, and we therefore refer to $beh(s)$ as the *behavioural semantics* of a state s.

For a causal stream function $f\colon A^\omega \to B^\omega$, we use the following notation for repeated stream function derivatives: for $w \in A^*$ and $a \in A$, we define $f_\varepsilon = f$ (where ε is the empty word) and $f_{wa} = (f_w)_a$. Hence the state set of $\langle\!\langle f \rangle\!\rangle$ equals the set $\{f_w \mid w \in A^*\}$ of all stream function derivatives of f.

Example 3 We illustrate the notions and results above with the binary counter from Example 1. Let $f = beh(s_0)$, $g = beh(s_1)$ and $h = beh(s_2)$. It can easily be seen that $f = h$ and $g(\alpha)(k) = 1 - f(\alpha)(k)$ for all $\alpha \in 2^\omega$ and $k \in \mathbb{N}$. Moreover, we have the following initial outputs and stream function derivatives:

$$f[0] = 0, \;\; f[1] = 1, \;\; g[0] = 1, \;\; g[1] = 0, \;\; f_0 = f, \;\; f_1 = g, \;\; g_0 = g, \;\; g_1 = f$$

and thus beh maps $\mathcal{S} = \langle\!\langle s_0 \rangle\!\rangle$ to its minimisation $\langle\!\langle f \rangle\!\rangle \subseteq \Gamma$, on the right:



In algebra, the behaviour of Mealy machines is typically described in terms of functions of type $A^+ \to B$. Although this set is isomorphic to Γ, we prefer to work with the latter because of the rich algebraic structure on streams. The notion of stream function derivative already occurs in [13], and is called *state* there. It is also a variation on the classical notion of derivative (or inverse) of functions from A^* to B^* (cf. [4]). Also Theorem 2 and Corollary 1 are essentially reformulations of classical results.

The main contribution of the present paper consists of the observation that $\langle\!\langle f \rangle\!\rangle$ can often be constructed by a symbolic computation of stream function derivatives starting from an algebraic specification of f. In Section 5, we shall describe such a symbolic algorithm for bitstream functions specified in 2-adic arithmetic.

3 The 2-Adic Bitstream Algebra

We will specify bitstream functions in the algebra of 2-adic integers (cf. [5]). A 2-adic integer is usually written as a (formal) power series of the form $\sum_{i=0}^{\infty} a_i 2^i$ where $a_i \in 2$ for all $i \in \{0, 1, 2, \ldots\}$. We identify such a power series with the bitstream $(a_0, a_1, a_2, \ldots)$, so for us the set of 2-adic integers is simply the set 2^ω of bitstreams.

There is a (strict) inclusion of the set of rational numbers with odd denominator

$$\mathbb{Q}_{odd} = \{p/q \mid p, q \in \mathbb{Z}, q \text{ odd } \}$$

into the 2-adic integers by taking infinitary base 2 expansions (see e.g. [10]). For a positive integer n, this is just the binary representation of n (least significant bit on the left) padded with a tail of zeros; for instance,

$$\begin{array}{lcl} \mathrm{Bin}(2) & = & (0,1,0,0,0,\ldots) = 010^\omega \\ \mathrm{Bin}(5) & = & (1,0,1,0,0,\ldots) = 1010^\omega \end{array}$$

Binary representations of negative integers end with an infinite sequence of ones and rational numbers (with odd denominator) have binary representations that are eventually periodical; for instance,

$$\begin{array}{lcl} \mathrm{Bin}(-1) & = & (1,1,1,1,\ldots) = 1^\omega \\ \mathrm{Bin}(-5) & = & (1,1,0,1,1,1,\ldots) = 1101^\omega \\ \mathrm{Bin}(1/5) & = & (1,0,1,1,0,0,1,1,0,0,1,1,0,\ldots) = 1(0110)^\omega. \end{array}$$

(Rationals with even denominator require formal power series representations $\sum_{i=k}^{\infty} a_i 2^i$ where $k < 0$, and are not treated here.)

Below we shall define the binary representation $\mathrm{Bin}(q)$ of any rational $q \in \mathbb{Q}_{odd}$ by means of a *stream differential equation*. Such equations specify streams σ, in analogy with traditional differential calculus, in terms of their initial value $\sigma(0)$ and derivative σ' (which are defined as $hd(\sigma)$ and $tl(\sigma)$). For instance, the differential equation

$$\sigma' = \sigma \qquad \sigma(0) = 1$$

clearly defines the stream $(1, 1, 1, \ldots)$. We refer to [16] for an overview on stream differential equations.

Now let $odd(n/2m + 1) = n \mod 2$ for $n, m \in \mathbb{Z}$. We define the inclusion map

$$\mathrm{Bin} : \mathbb{Q}_{odd} \to 2^\omega$$

by the following system of stream differential equations (one for each $q \in \mathbb{Q}_{odd}$):

$$\mathrm{Bin}(q)(0) = odd(q), \;\; \mathrm{Bin}(q)' = \mathrm{Bin}((q - odd(q))/2) \tag{2}$$

The reader is invited to compute the examples of binary expansions given above, using this definition of Bin.

We recall that the set of rational numbers (with odd denominator) is an *integral domain*: a commutative ring in which multiplication has no zero-divisors, i.e., for all p and q, if $p \times q = 0$ then $p = 0$ or $q = 0$. Next we shall introduce operations of addition and multiplication (as well as minus and inverse) on the set of 2-adic integers such that they reflect the operations on the rationals. More formally, we shall turn the set of 2-adic integers into an integral domain

$$\mathcal{A}_{2adic} = \langle 2^\omega, +, -, \times, /, [0], [1] \rangle$$

such that the inclusion map $\mathrm{Bin} : \mathbb{Q}_{odd} \to 2^\omega$ is a homomorphism of integral domains.

In the literature, the operators on the 2-adic integers are often defined by explicitly specifying how they work on binary representations, see Figure 1 for examples of addition and multiplication. Their definitions are sometimes also given in terms of their representation as power series, typically in the form of some recurrence relation on their coefficients. Here we shall give, instead, a definition by means of stream differential equations, which we saw already examples of above, and which can be seen as a generalisation of definitions by recurrence, cf. [16]. Our choice to use stream differential equations is not just a matter of taste: they form the basis for

addition:

$$\begin{array}{rcr} {\scriptstyle 1\ 1\ 1\ \cdots} & & \text{(carry)} \\ 1\,0\,1\,0\,0\,0\,\cdots & = & 5 \\ +\,0\,1\,1\,1\,1\,1\,\cdots & = & -2 \\ \hline 1\,1\,0\,0\,0\,0\,\cdots & = & 3 \end{array}$$

multiplication:

$$\begin{array}{l} 1\,1\,0\,0\,0\,\cdots \times 0\,1\,1\,1\,1\,\cdots \quad = 3 \times (-2) \\ \hline 0\,0\,0\,0\,0\,0\,\cdots \\ \quad 1\,1\,0\,0\,0\,\cdots \\ \quad\quad 1\,1\,0\,0\,0\,\cdots \\ \quad\quad\quad 1\,1\,0\,0\,0\,\cdots \\ \quad\quad\quad\quad 1\,1\,0\,0\,0\,\cdots \\ +\quad\quad\quad\ \vdots \\ \hline 0\,1\,0\,1\,1\,1\,\cdots \quad\quad\quad\quad = -6 \end{array}$$

Figure 1: Examples of 2-adic addition and multiplication.

the definition of the Mealy machine of *stream function expressions*, to be introduced in Section 4.

First we define the constants zero and one by

$$[0] = (0, 0, 0, \ldots) \qquad [1] = (1, 0, 0, 0, \ldots)$$

Below we shall use the following Boolean operators on $2 = \{0, 1\}$, which are defined, for all $a, b \in 2$, as usual: $a \wedge b = \min\{a, b\}$ and $a \oplus b = 1$ iff $a = 0$, $b = 1$ or $a = 1$, $b = 0$ (exclusive or).

Definition 1 *We define the operators of addition, minus, multiplication and inverse of 2-adic integers by the following system of stream differential equations. For $\alpha, \beta \in 2^\omega$,*

derivative:			*initial value:*		
$(\alpha+\beta)'$	$=$	$(\alpha'+\beta')+[\alpha(0)\wedge\beta(0)]$	$(\alpha+\beta)(0)$	$=$	$\alpha(0)\oplus\beta(0)$
$(-\alpha)'$	$=$	$-(\alpha'+[\alpha(0)])$	$(-\alpha)(0)$	$=$	$\alpha(0)$
$(\alpha\times\beta)'$	$=$	$(\alpha'\times\beta)+([\alpha(0)]\times\beta')$	$(\alpha\times\beta)(0)$	$=$	$\alpha(0)\wedge\beta(0)$
$(1/\alpha)'$	$=$	$-(\alpha'\times(1/\alpha))$	$(1/\alpha)(0)$	$=$	1

There is a side condition for the definition of inverse: $1/\alpha$ is defined only for α with $\alpha(0) = 1$.

We briefly explain the intuition behind the equations above. (For the fact that the operators above are *uniquely* defined by their defining stream differential equations, we refer to [16].) The equation for sum shows that a carry term must be added in case the two initial values are both 1. The equation for minus is obtained from the requirement that $(-\alpha)+\alpha = [0]$ for all $\alpha \in 2^\omega$. By taking initial value and derivative on both sides and using the equation for $+$, we find that $(-\alpha)(0) \oplus \alpha(0) = 0$, hence $(-\alpha)(0) = \alpha(0)$, and that $(-\alpha)' + \alpha' + [\alpha(0)] = [0]$, hence $(-\alpha)' = -(\alpha' + [\alpha(0)])$. The equation for the product states that for all $\alpha, \beta \in 2^\omega$, $\alpha \times \beta$ can be calculated using the base 2 version of shift-add-multiplication known from the multiplication in decimal notation. Finally, the multiplicative inverse of $\alpha \in 2^\omega$ is defined only if $\alpha(0) = 1$, since there is no bit $a \in 2$ such that $a \wedge \alpha(0) = 1$. If $1/\alpha$ is defined then it satisfies $(1/\alpha) \times \alpha = [1]$. The equation for $1/\alpha$ can be derived in a similar way as for $-\alpha$.

The operations on the 2-adic numbers have been devised in such a way that $\mathrm{Bin}\colon \mathbb{Q}_{odd} \to \mathcal{A}_{2adic}$ is a homomorphism of integral domains: one can easily show that, for all $p, q \in \mathbb{Q}_{odd}$,

$$\mathrm{Bin}(p \times q) = \mathrm{Bin}(p) \times \mathrm{Bin}(q)$$

(with on the left multiplication of rationals and on the right multiplication of bitstreams), and similarly for the other operators.

When calculating with the 2-adic operations it is convenient to have a constant denoting the bitstream Bin(2). We define

$$X := \mathrm{Bin}(2) = (0, 1, 0, 0, 0, \ldots),$$

The constant X can be used to express some identities on bitstreams that will be useful later: For all $\alpha \in 2^\omega$,

$$\begin{array}{rcl} \alpha & = & [\alpha(0)] + (X \times \alpha') \\ \alpha + \alpha & = & X \times \alpha \end{array} \tag{3}$$

4 Mealy Machine of Expressions

We will specify bitstream functions in the language of $\mathcal{A}_{2adic}$ over a single variable `s`. Formally, the set of *function expressions* `FExpr` is generated by the following grammar:

$$\mathtt{F}, \mathtt{G} \quad ::= \quad \mathtt{s} \mid \mathtt{0} \mid \mathtt{1} \mid \mathtt{X} \mid -\mathtt{F} \mid \mathtt{F} + \mathtt{G} \mid \mathtt{F} \times \mathtt{G} \mid \mathtt{1}/(\mathtt{1} + (\mathtt{X} \times \mathtt{F}))$$

We use the symbols $-, +, \times, /$ to denote the operations on bitstreams as well as the corresponding syntax constructors. The typing should always be clear from the context. We will use standard notational conventions: we write $\mathtt{F}^n$ for the n-fold product of $\mathtt{F}$ with itself, (in particular, $\mathtt{F}^0 = \mathtt{1}$) and $\mathtt{F}/(\mathtt{1} + (\mathtt{X} \times \mathtt{G}))$ or $\frac{\mathtt{F}}{\mathtt{1}+(\mathtt{X}\times\mathtt{G})}$ instead of $\mathtt{F} \times (\mathtt{1}/(\mathtt{1} + (\mathtt{X} \times \mathtt{G})))$. The algebraic semantics of function expressions is given by the expected interpretation in $\mathcal{A}_{2adic}$. The reason we only allow division by terms of the form $\mathtt{1} + (\mathtt{X} \times \mathtt{F})$ is to ensure that the inverse operation is always defined when evaluating function expressions in $\mathcal{A}_{2adic}$.

Definition 2 *We define the* algebraic 2-adic semantics $[\![\mathtt{F}]\!] \colon 2^\omega \to 2^\omega$ *of a function expression* $\mathtt{F} \in \mathtt{FExpr}$ *by the following inductive clauses. Let* σ *be a bitstream,*

$$\begin{array}{rclcrcl}
[\![\mathtt{s}]\!](\sigma) & = & \sigma & & [\![-\mathtt{F}]\!](\sigma) & = & -([\![\mathtt{F}]\!](\sigma)) \\
[\![\mathtt{0}]\!](\sigma) & = & [0] & & [\![\mathtt{F}+\mathtt{G}]\!](\sigma) & = & [\![\mathtt{F}]\!](\sigma) + [\![\mathtt{G}]\!](\sigma) \\
[\![\mathtt{1}]\!](\sigma) & = & [1] & & [\![\mathtt{F}\times\mathtt{G}]\!](\sigma) & = & [\![\mathtt{F}]\!](\sigma) \times [\![\mathtt{G}]\!](\sigma) \\
[\![\mathtt{X}]\!](\sigma) & = & X & [\![\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F})]\!](\sigma) & = & 1/(1 + (X \times [\![\mathtt{F}]\!](\sigma))
\end{array}$$

We say that a function expression $\mathtt{F}$ *specifies* the bitstream function $[\![\mathtt{F}]\!]$, and two function expressions $\mathtt{F}$ and $\mathtt{G}$ are *equivalent* (notation: $\mathtt{F} \equiv \mathtt{G}$) if $[\![\mathtt{F}]\!] = [\![\mathtt{G}]\!]$.

One easily shows (by structural induction on function expressions) that for all $\mathtt{F} \in \mathtt{FExpr}$, $[\![\mathtt{F}]\!]$ is a causal bitstream function. In other words, $[\![-]\!]$ is a map from $\mathtt{FExpr}$ to Γ. We will now define a transition function $\xi \colon \mathtt{FExpr} \to (2 \times \mathtt{FExpr})^2$ such that $\langle \mathtt{FExpr}, \xi \rangle$ is a binary Mealy machine and the algebraic semantics $[\![-]\!] \colon \mathtt{FExpr} \to \Gamma$ is a Mealy homomorphism. That is, for each $\mathtt{F} \in \mathtt{FExpr}$ and $a \in 2$, we want to define $\mathtt{F}[a] \in 2$ and $\mathtt{F}_a \in \mathtt{FExpr}$ such that $\mathtt{F}[a] = [\![\mathtt{F}]\!][a]$ and $[\![\mathtt{F}_a]\!] = [\![\mathtt{F}]\!]_a$. In order to do so, first recall how we defined $\gamma(f)(a)$ in terms of $f(a:-)$, hd and tl (cf. equation (1)) for $f \in \Gamma$ and $a \in 2$. We will "mimic" γ in the syntax.

First, we need a syntactic version of the map $f(a:-)$ for $f \in \Gamma$ and $a \in 2$. More precisely, given $\mathtt{F} \in \mathtt{FExpr}$ and $a \in 2$, we want to find a function expression which specifies $[\![\mathtt{F}]\!](a:-)$. Note that for all $\sigma \in 2^\omega$, $[\![\mathtt{0} + (\mathtt{X} \times \mathtt{s})]\!](\sigma) = 0{:}\sigma$ and $[\![\mathtt{1} + (\mathtt{X} \times \mathtt{s})]\!](\sigma) = 1{:}\sigma$.

Definition 3 *We define* $\iota \colon 2 \to \mathtt{FExpr}$ *by* $\iota(0) = \mathtt{0}$ *and* $\iota(1) = \mathtt{1}$*, and for* $\mathtt{F} \in \mathtt{FExpr}$ *and* $a \in 2$*, we let* $a{:}\mathtt{s} := \iota(a) + (\mathtt{X} \times \mathtt{s})$*. We define* $\mathtt{F}(a{:}\mathtt{s})$ *to be the function expression obtained by uniformly substituting* $a{:}\mathtt{s}$ *for* $\mathtt{s}$ *in* $\mathtt{F}$*.*

We now show that $\mathtt{F}(a{:}\mathtt{s})$ indeed specifies $[\![\mathtt{F}]\!](a{:}-)$.

Lemma 1 *For all* $\mathtt{F} \in \mathtt{FExpr}$*, all* $a \in 2$ *and all* $\sigma \in 2^\omega$*:*

$$[\![\mathtt{F}(a{:}\mathtt{s})]\!](\sigma) = [\![\mathtt{F}]\!](a{:}\sigma).$$

Proof: The lemma is proved by a straightforward induction on the structure of $\mathtt{F}$. We only show a few example cases:

$$\begin{aligned}
[\![\mathtt{s}(a{:}\mathtt{s})]\!](\sigma) &= [\![\iota(a) + (\mathtt{X} \times \mathtt{s})]\!](\sigma) \\
&= [\![\iota(a)]\!](\sigma) + ([\![\mathtt{X}]\!](\sigma) \times [\![\mathtt{s}]\!](\sigma)) \\
&= [a] + (X \times \sigma) \\
\text{(cf. eqn. (3))} &= a{:}\sigma = [\![\mathtt{s}]\!](a{:}\sigma)
\end{aligned}$$

$$\begin{aligned}
[\![(\mathtt{F} + \mathtt{G})(\mathtt{a}{:}\mathtt{s}))]\!](\sigma) &= [\![\mathtt{F}(a{:}\mathtt{s}) + \mathtt{G}(a{:}\mathtt{s})]\!](\sigma) \\
&= [\![\mathtt{F}(a{:}\mathtt{s})]\!](\sigma) + [\![\mathtt{G}(a{:}\mathtt{s})]\!](\sigma) \\
&\overset{\text{IH}}{=} [\![\mathtt{F}]\!](a{:}\sigma) + [\![\mathtt{G}]\!](a{:}\sigma) \\
&= [\![\mathtt{F} + \mathtt{G}]\!](a{:}\sigma)
\end{aligned}$$

□

The transition function ξ on function expressions is defined inductively over the syntactic structure. In order to motivate the definition of ξ, consider, for example, a product expression $\mathtt{F} \times \mathtt{G}$, and suppose that $[\![\mathtt{F}]\!] = f$ and $[\![\mathtt{G}]\!] = g$. We then want $[\![(\mathtt{F} \times \mathtt{G})_a]\!](\sigma) = tl(f(a:\sigma) \times g(a:\sigma))$ for all $\sigma \in 2^\omega$. By the stream differential equation for product, this means that

$$\begin{aligned}
[\![(\mathtt{F} \times \mathtt{G})_a]\!](\sigma) &= (tl(f(a{:}\sigma)) \times g(a{:}\sigma)) + ([hd(f(a{:}\sigma))] \times tl(g(a{:}\sigma))) \\
&= (f_a(\sigma) \times g(a{:}\sigma)) + ([f[a]] \times g_a(\sigma) \\
&= ([\![\mathtt{F}]\!]_a(\sigma) \times [\![\mathtt{G}]\!](a{:}\sigma)) + ([[\![\mathtt{F}]\!][a]] \times [\![\mathtt{G}]\!]_a(\sigma).
\end{aligned}$$

Comparing this equation with the definition of $(\mathtt{F} \times \mathtt{G})_a$ below it is easy to see that an induction argument and Lemma 1 will show that $[\![(\mathtt{F} \times \mathtt{G})_a]\!] = [\![\mathtt{F} \times \mathtt{G}]\!]_a$.

Definition 4 *Let* $\xi\colon \mathtt{FExpr} \to (2 \times \mathtt{FExpr})^2$ *be the map given by* $\xi(\mathtt{F})(a) = \langle \mathtt{F}[a], \mathtt{F}_a \rangle$ *where* $\mathtt{F}[a]$ *and* $\mathtt{F}_a$ *are defined for all* $\mathtt{F} \in \mathtt{FExpr}$ *and* $a \in 2$ *by:*

syntactic initial output					
$\mathtt{s}[a]$	$=$	a	$(-\mathtt{F})[a]$	$=$	$\mathtt{F}[a]$
$\mathtt{0}[a]$	$=$	0	$(\mathtt{F}+\mathtt{G})[a]$	$=$	$\mathtt{F}[a] \oplus \mathtt{G}[a]$
$\mathtt{1}[a]$	$=$	1	$(\mathtt{F}\times\mathtt{G})[a]$	$=$	$\mathtt{F}[a] \wedge \mathtt{G}[a]$
$\mathtt{X}[a]$	$=$	0	$(\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F})))[a]$	$=$	1

syntactic stream function derivative					
$\mathtt{s}_a$	$=$	$\mathtt{s}$	$(-\mathtt{F})_a$	$=$	$-(\mathtt{F}_a + \iota(\mathtt{F}[a]))$
$\mathtt{0}_a$	$=$	$\mathtt{0}$	$(\mathtt{F}+\mathtt{G})_a$	$=$	$(\mathtt{F}_a + \mathtt{G}_a) + \iota(\mathtt{F}[a] \wedge \mathtt{G}[a])$
$\mathtt{1}_a$	$=$	$\mathtt{0}$	$(\mathtt{F}\times\mathtt{G})_a$	$=$	$(\mathtt{F}_a \times \mathtt{G}(a\!:\!\mathtt{s})) + (\iota(\mathtt{F}[a]) \times \mathtt{G}_a)$
$\mathtt{X}_a$	$=$	$\mathtt{1}$	$(\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F})))_a$	$=$	$-(\mathtt{F}(a\!:\!\mathtt{s}))/(\mathtt{1}+(\mathtt{X}\times\mathtt{F}(a\!:\!\mathtt{s})))$

We call $\langle \mathtt{FExpr}, \xi \rangle$ *the* Mealy machine of expressions*, and we refer to* $\mathtt{F}[a]$ *and* $\mathtt{F}_a$ *as the* syntactic initial output*, respectively the* syntactic stream function derivative*, of* $\mathtt{F}$ *on input* a*.*

We now show that ξ indeed ensures that the algebraic semantics concides with the behavioural semantics of function expressions.

Proposition 1 *The map* $[\![-]\!]\colon \mathtt{FExpr} \to \Gamma$ *is a homomorphism of Mealy machines from* $\langle \mathtt{FExpr}, \xi \rangle$ *to* $\langle \Gamma, \gamma \rangle$*.*

Proof: We must show that for all $\mathtt{F} \in \mathtt{FExpr}$ and all $a \in 2$:

$$[\![\mathtt{F}]\!][a] = \mathtt{F}[a] \quad \text{and} \quad [\![\mathtt{F}_a]\!] = [\![\mathtt{F}]\!]_a.$$

The proof is by induction on the structure of $\mathtt{F}$. We only show the case for $\mathtt{s}$ and product; the others can be shown along the same lines (the full proof details are found in the Appendix). In the identities below, IH refers to the induction hypothesis. Note also that for any $b \in 2$ and $\sigma \in 2^\omega$: $[\![\iota(b)]\!](\sigma) = [b]$. Let $a \in 2$ and $\sigma \in 2^\omega$.

$$\begin{array}{lcllcll} [\![\mathtt{s}]\!][a] & = & ([\![\mathtt{s}]\!](a\!:\!\sigma))(0) & = & (a\!:\!\sigma)(0) = a & = & \mathtt{s}[a]. \\ [\![\mathtt{s}]\!]_a(\sigma) & = & ([\![\mathtt{s}]\!](a\!:\!\sigma))' & = & (a\!:\!\sigma)' = \sigma & = & [\![\mathtt{s}_a]\!](\sigma). \end{array}$$

$$\begin{array}{lcl} [\![\mathtt{F}\times\mathtt{G}]\!][a] & = & ([\![\mathtt{F}\times\mathtt{G}]\!](a\!:\!\sigma))(0) = ([\![\mathtt{F}]\!](a\!:\!\sigma) \times [\![\mathtt{G}]\!](a\!:\!\sigma))(0) \\ & = & [\![\mathtt{F}]\!][a] \wedge [\![\mathtt{G}]\!][a] \\ & \overset{\text{IH}}{=} & \mathtt{F}[a] \wedge \mathtt{G}[a] = (\mathtt{F}\times\mathtt{G})[a]. \end{array}$$

$$
\begin{aligned}
[\![\mathtt{F}\times\mathtt{G}]\!]_a(\sigma) &= ([\![\mathtt{F}\times\mathtt{G}]\!](a{:}\sigma))' = ([\![\mathtt{F}]\!](a{:}\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma))' \\
&= (([\![\mathtt{F}]\!](a{:}\sigma))'\times[\![\mathtt{G}]\!](a{:}\sigma)) + ([([\![\mathtt{F}]\!](a{:}\sigma))(0)]\times([\![\mathtt{G}]\!](a{:}\sigma))') \\
&= ([\![\mathtt{F}]\!]_a(\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma)) + ([[\![\mathtt{F}]\!][a]]\times[\![\mathtt{G}]\!]_a(\sigma)) \\
&\overset{\text{IH}}{=} ([\![\mathtt{F}_a]\!](\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma)) + ([\mathtt{F}[a]]\times[\![\mathtt{G}_a]\!](\sigma)) \\
&\overset{\text{Lem.1}}{=} [\![(\mathtt{F}_a(\sigma)\times\mathtt{G}(a{:}\mathtt{s})) + (\iota(\mathtt{F}[a])\times\mathtt{G}_a)]\!](\sigma) \\
&= [\![(\mathtt{F}\times\mathtt{G})_a]\!](\sigma).
\end{aligned}
$$

□

Since $\langle\Gamma,\gamma\rangle$ is a final Mealy machine, the map $[\![-]\!]$ coincides with the unique homomorphism $beh\colon\langle\mathtt{FExpr},\xi\rangle\to\langle\Gamma,\gamma\rangle$. In other words, we have shown that for all function expressions $\mathtt{F}$, $[\![\mathtt{F}]\!] = beh(\mathtt{F})$.

5 Synthesis

A consequence of Proposition 1 is that the generated submachine $\langle\!\langle\mathtt{F}\rangle\!\rangle$ is a realisation of $[\![\mathtt{F}]\!]$. Conceptually, we can construct $\langle\!\langle\mathtt{F}\rangle\!\rangle$ by computing the transition closure of $\{\mathtt{F}\}$ in $\langle\mathtt{FExpr},\xi\rangle$. However, in general, $\langle\!\langle\mathtt{F}\rangle\!\rangle$ is neither finite nor minimal. In order to obtain a minimal realisation of $[\![\mathtt{F}]\!]$ we compute $\langle\!\langle\mathtt{F}\rangle\!\rangle$ modulo equivalence. In practice, this is achieved by reducing function expressions to normal form, which we briefly describe now.

5.1 Normal forms

We call a function expression $\mathtt{F}$ *integral* if it does not contain the inverse operation, and $\mathtt{F}$ is *closed* if it does not contain $\mathtt{s}$. The *polynomial normal form* $pnf(\mathtt{F})$ of an integral expression $\mathtt{F}$ is an analogue of the distributed normal form of polynomials (in the variable $\mathtt{s}$). It can be computed in the expected manner by applying identities of commutative rings: (i) distribute $\times$ over $+$, (ii) reduce using identities for $\mathtt{0}$ and $\mathtt{1}$, (iii) collect terms on powers of $\mathtt{s}$, and finally (iv) reduce sums of closed expressions using the identity $\mathtt{X}^n + \mathtt{X}^n = \mathtt{X}^{n+1}$, for all $n \geq 0$. This last identity holds since for all $\alpha\in 2^\omega$, $\alpha+\alpha = X\times\alpha$, cf. equation (3). For example, $pnf((\mathtt{1}+\mathtt{X})\times(\mathtt{1}+\mathtt{s})+\mathtt{1}) = \mathtt{X}^2 + (\mathtt{1}+\mathtt{X})\times\mathtt{s}$, and this polynomial normal form is "computed" in the

following series of identities:

$$\begin{array}{lcl}((\mathtt{1}+\mathtt{X})\times(\mathtt{1}+\mathtt{s}))+\mathtt{1} & \equiv & ((\mathtt{1}+\mathtt{X})\times\mathtt{1})+((\mathtt{1}+\mathtt{X})\times\mathtt{s})+\mathtt{1}\\ & \equiv & ((\mathtt{1}\times\mathtt{1})+(\mathtt{X}\times\mathtt{1}))+((\mathtt{1}\times\mathtt{s})+(\mathtt{X}\times\mathtt{s}))+\mathtt{1}\\ & \equiv & (\mathtt{1}+\mathtt{X})+(\mathtt{s}+(\mathtt{X}\times\mathtt{s}))+\mathtt{1}\\ & \equiv & (\mathtt{1}+\mathtt{1}+\mathtt{X})+(\mathtt{1}+\mathtt{X})\times\mathtt{s}\\ & \equiv & \mathtt{X}^2+(\mathtt{1}+\mathtt{X})\times\mathtt{s}.\end{array}$$

Note that in the last step we used that,

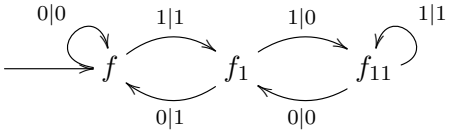
$$\mathtt{1}+\mathtt{1}+\mathtt{X}\ \equiv\ \mathtt{X}^0+\mathtt{X}^0+\mathtt{X}^1\ \equiv\ \mathtt{X}^1+\mathtt{X}^1\ \equiv\ \mathtt{X}^2.$$

For any integral expression $\mathtt{F}$, it should be clear that $\mathtt{F}\equiv \mathit{pnf}(\mathtt{F})$ and that $\mathit{pnf}(\mathtt{F})$ is unique: $\mathit{pnf}(\mathtt{F})=\mathit{pnf}(\mathtt{G})$ if and only if $\mathtt{F}\equiv\mathtt{G}$. Given arbitrary function expressions $\mathtt{F}$ and $\mathtt{G}$, we can therefore decide whether $\mathtt{F}\equiv\mathtt{G}$ by first rewriting (using the identities of integral domains) $\mathtt{F}$ and $\mathtt{G}$ into fractions $\mathtt{P}/\mathtt{Q}$ and $\mathtt{R}/\mathtt{S}$, respectively, where $\mathtt{P},\mathtt{Q},\mathtt{R},\mathtt{S}$ are integral, and then checking whether $\mathit{pnf}(\mathtt{P}\times\mathtt{S})=\mathit{pnf}(\mathtt{R}\times\mathtt{Q})$. These normal forms are treated in more detail in [6]. In fact, in our synthesis algorithm we only need to compute normal forms of *closed*, integral expressions, as we will see in Section 5.3.

Example 4 We illustrate by computing (a representation of) $\langle\!\langle[\![\mathtt{F}]\!]\rangle\!\rangle$ for the function expression $\mathtt{F}=(\mathtt{1}+\mathtt{X})\times\mathtt{s}$. For the transition on input 1 we find that:

$$\begin{array}{lcl}((\mathtt{1}+\mathtt{X})\times\mathtt{s})[1] & = & (\mathtt{1}[1]\oplus\mathtt{X}[1])\wedge\mathtt{s}[1]=(1\oplus 0)\wedge 1=1.\\ ((\mathtt{1}+\mathtt{X})\times\mathtt{s})_1 & = & ((\mathtt{1}+\mathtt{X})_1\times\mathtt{s}(1{:}\mathtt{s}))+(\iota((\mathtt{1}+\mathtt{X})[1])\times\mathtt{s}_1)\\ & = & (((\mathtt{1}_1+\mathtt{X}_1)+\iota(\mathtt{1}[1]\wedge\mathtt{X}[1]))\times(\mathtt{1}+(\mathtt{X}\times\mathtt{s})))\\ & & \qquad\qquad +(\iota(\mathtt{1}[1]\oplus\mathtt{X}[1])\times\mathtt{s}_1)\\ & = & ((\mathtt{0}+\mathtt{1})+(\iota(1\wedge 0)\times(\mathtt{1}+(\mathtt{X}\times\mathtt{s}))))+(\iota(1\oplus 0)\times\mathtt{s})\\ & = & (((\mathtt{0}+\mathtt{1})+\mathtt{0})\times(\mathtt{1}+(\mathtt{X}\times\mathtt{s}))+(\mathtt{1}\times\mathtt{s})\\ & \equiv & \mathtt{1}+((\mathtt{1}+\mathtt{X})\times\mathtt{s})=\mathtt{1}+\mathtt{F}\end{array}$$

Computing further derivatives and initial output, we find the following minimal realisation of $[\![\mathtt{F}]\!]$:



where $f=[\![\mathtt{F}]\!]$, $f_1=[\![\mathtt{1}+\mathtt{F}]\!]$ and $f_{11}=[\![\mathtt{X}+\mathtt{F}]\!]$.

The normal forms essentially allow us to compute submachines in the final Mealy machine. Still, for arbitrary $\mathtt{F} \in \mathtt{FExpr}$, the least fixed point construction of $\langle\langle [\![\mathtt{F}]\!] \rangle\rangle$ is not guaranteed to terminate as it is easy to specify functions that have no finite realisation. For example, if we take $\mathtt{F} = \mathtt{s} \times \mathtt{s}$ and we compute the derivatives $[\![\mathtt{F}]\!]_0, [\![\mathtt{F}]\!]_{00}, [\![\mathtt{F}]\!]_{000}, \ldots$ we get the sequence $[\![\mathtt{X} \times \mathtt{F}]\!]$, $[\![\mathtt{X}^2 \times \mathtt{F}]\!]$, $[\![\mathtt{X}^3 \times \mathtt{F}]\!], \ldots$ which are all distinct. A similar argument shows that if $\mathtt{F} = \mathtt{1}/(\mathtt{1} + (\mathtt{X} \times \mathtt{s}))$, then $[\![\mathtt{F}]\!]$ has infinitely many distinct stream function derivatives.

5.2 Rational functions

We now define a class of expressions that specify functions with finite realisations. A *rational function expression* is a function expression of the form

$$\mathtt{F} = \frac{\mathtt{D} + (\mathtt{C} \times \mathtt{s})}{1 + (\mathtt{X} \times \mathtt{E})}$$

where $\mathtt{D}, \mathtt{C}, \mathtt{E} \in \mathtt{FExpr}$ are closed, integral function expressions. In other words, $\mathtt{D}, \mathtt{C}$ and $\mathtt{E}$ specify constant bitstream functions whose value is of the form $\mathrm{Bin}(x)$ for some integer $x \in \mathbb{Z}$. An example of a rational function expression is

$$\mathtt{F} = \frac{(-\mathtt{X}) + ((\mathtt{1} + \mathtt{X}) \times \mathtt{s})}{\mathtt{1} + (\mathtt{X} \times (\mathtt{X} + \mathtt{X}))}$$

which specifies the bitstream function

$$f(\sigma) = \frac{\mathrm{Bin}(-2) + (\mathrm{Bin}(1+2) \times \sigma)}{\mathrm{Bin}(1 + 2 \times (2+2))} = \frac{\mathrm{Bin}(-2) + (\mathrm{Bin}(3) \times \sigma))}{\mathrm{Bin}(9)} \tag{4}$$

A function $f\colon 2^\omega \to 2^\omega$ is called *rational* if there is a rational function expression $\mathtt{F}$ such that $[\![\mathtt{F}]\!] = f$. Hence the function $f(\sigma) = \mathrm{Bin}(3) \times \sigma$ specified in Example 4 is also rational by taking $\mathtt{D} = \mathtt{E} = \mathtt{0}$ and $\mathtt{C} = \mathtt{1} + \mathtt{X}$. The numeric interpretation of closed, integral expressions will be convenient below.

Lemma 2 *A function $f\colon 2^\omega \to 2^\omega$ is rational iff there are $d, m, n \in \mathbb{Z}$ such that n is odd, and for all $\sigma \in 2^\omega$*

$$f(\sigma) = \frac{\mathrm{Bin}(d) + (\mathrm{Bin}(m) \times \sigma)}{\mathrm{Bin}(n)}$$

Proof: This follows essentially from the fact that $\mathrm{Bin}\colon \mathbb{Q}_{odd} \to \mathcal{A}_{2adic}$ is a homomorphism of integral domains. □

To simplify notation, we will leave out the Bin-part, and just write $f(\sigma) = (d + (m \times \sigma))/n$ whenever $f(\sigma) = (\mathrm{Bin}(d) + (\mathrm{Bin}(m) \times \sigma)))/\mathrm{Bin}(n)$. Similarly, for $x \in \mathbb{Z}$, we write $x(0)$ and x' instead of $\mathrm{Bin}(x)(0)$ and $\mathrm{Bin}(x)'$, respectively. The following technical lemmas will be used to prove that rational functions have finite realisations. Their proofs can be found in the Appendix. The first of these lemmas uses the numeric interpretation of rational functions to characterise the immediate derivatives.

Lemma 3 *Let f be a rational bitstream function of the form:*

$$f(\sigma) = \frac{d + (m \times \sigma)}{n}$$

for integers d, m and n with n odd. For $a \in 2$, the stream function derivative f_a is given by:

$$(f_a)(\sigma) = \frac{\delta(a) + (m \times \sigma)}{n} \tag{5}$$

where (in the numeric interpretation)

$$\delta(0) = \begin{cases} \frac{1}{2}\, d & \text{if } d \text{ even} \\ \frac{1}{2}(d-n) & \text{if } d \text{ odd} \end{cases} \qquad \delta(1) = \begin{cases} \frac{1}{2}(d+m) & \text{if } d+m \text{ even} \\ \frac{1}{2}(d+m-n) & \text{if } d+m \text{ odd} \end{cases}$$

Hence Lemma 3 already tells us that the derivatives of a rational function are again rational. The next lemma uses the numeric interpretation to give a bound on the range of δ-values that can occur in the stream function derivatives of a rational function.

Lemma 4 *Let $f(\sigma) = \frac{d+(m\times\sigma)}{n}$ be a rational function with $n > 0$ odd. For all $w \in 2^*$, the stream function derivative f_w is of the form*

$$(f_w)(\sigma) = \frac{\delta(w) + (m \times \sigma)}{n} \tag{6}$$

where $\delta(w)$ is an integer such that

$$\min\{d, -n+1, -n+m+1\} \;\leq\; \delta(w) \;\leq\; \max\{d, m-1, 0\}.$$

The crucial properties that make it possible to perform synthesis from rational function specifications are stated in the following proposition.

Proposition 2 *For all rational bitstream functions $f\colon 2^\omega \to 2^\omega$,*

1. *all stream function derivatives of f are rational,*
2. *$\langle\langle f \rangle\rangle$ is finite.*

Proof: Let $f(\sigma) = \frac{d+(m\times\sigma)}{n}$ be a rational function. Item 1 of the proposition follows from Lemmas 2 and 3. To see that item 2 holds, first note that we can always assume that $n > 0$ since f is equal to the function $g(\sigma) = \frac{-d+(-m\times\sigma)}{-n}$. The number of states in $\langle\langle f \rangle\rangle$ equals the number of distinct $\delta(w)$-values in the derivatives of f. By Lemma 4 this number is finite. □

5.3 Algorithm

We construct $\langle\langle [\![\mathtt{F}]\!] \rangle\rangle$ for a rational function expression $\mathtt{F}$ by computing states and transitions in the final Mealy machine. Stream function derivatives of $[\![\mathtt{F}]\!]$ are denoted by rational function expressions, and in order to efficiently determine when two expressions denote the same derivative, we normalise the closed integral subexpressions of $\mathtt{F}$ at the beginning of the computation. More precisely, if $\mathtt{F} = (\mathtt{D} + (\mathtt{C} \times \mathtt{s}))/(\mathtt{1} + (\mathtt{X} \times \mathtt{E}))$ is a rational function expression, then we define the *reduced form of* $\mathtt{F}$ as

$$\mathit{red}(\mathtt{F}) = (\mathit{pnf}(\mathtt{D}) + (\mathit{pnf}(\mathtt{C}) \times \mathtt{s}))/(\mathtt{1} + (\mathtt{X} \times \mathit{pnf}(\mathtt{E}))),$$

and we call $\mathtt{F}$ *reduced*, if $\mathtt{F} = \mathit{red}(\mathtt{F})$. Starting from a reduced expression, all derivatives computed during the synthesis algorithm are represented in a similar, reduced form. This is achieved by using the function next.

Definition 5 *Given a rational function expression*

$$\mathtt{F} = \frac{\mathtt{D} + (\mathtt{C} \times \mathtt{s})}{1 + (\mathtt{X} \times \mathtt{E})}$$

we define

$$\begin{aligned} \mathrm{next}_\mathrm{D}(\mathtt{F}, 0) &= \mathtt{D}_0 + -(\mathtt{D}[0] \times \mathtt{E}), \text{ and} \\ \mathrm{next}_\mathrm{D}(\mathtt{F}, 1) &= ((\mathtt{D}_1 + \mathtt{C}_1) + \iota(\mathtt{D}[1] \wedge \mathtt{C}[1])) + -(\iota(\mathtt{D}[1] \oplus \mathtt{C}[1]) \times \mathtt{E}), \end{aligned}$$

and for $a \in 2$ we define

$$\mathrm{next}(\mathtt{F}, a) = \frac{\mathit{pnf}(\mathrm{next}_\mathrm{D}(\mathtt{F}, a)) + (\mathtt{C} \times \mathtt{s})}{1 + (\mathtt{X} \times \mathtt{E})}.$$

It should be clear that starting from a reduced function expression $\mathtt{F}$, any two expressions computed with the next-function are equivalent if and only if they are (syntactically) equal. The next lemma shows that the next-function can be identified with the syntactic derivative function.

Lemma 5 *For all rational function expressions* $\mathtt{F}$ *and all* $a \in 2$,

$$\mathrm{next}(\mathtt{F}, a) \equiv \mathtt{F}_a.$$

Proof: The lemma follows easily by writing out the details of the definition of $\mathtt{F}_a$, and applying integral domain identities. See also the proof of Lemma 3 (in the Appendix). □

The synthesis algorithm is shown in Figure 2 (and we explain it in some more detail below). We denote the empty list by $[\,]$, the concatenation of two lists x and y by $x.y$, and we let *remdup* be a function that removes duplicates from a list. If S is a list of states (i.e. reduced rational function expressions) and T is a list of transitions, then $\mathit{transitions}(S)$ is the list of transitions with source state in S, and $\mathit{targets}(T)$ is the list of target states of transitions in T. Using list comprehension, we can write this more precisely as:

$$\begin{array}{lcl} \mathit{transitions}(S) & := & [\,\mathtt{G} \xrightarrow{0|\mathtt{G}[0]} \mathrm{next}(\mathtt{G},0), \mathtt{G} \xrightarrow{1|\mathtt{G}[1]} \mathrm{next}(\mathtt{G},1)) \mid \mathtt{G} \in S\,], \\ \mathit{targets}(T) & := & [\,\mathtt{G} \mid \mathtt{G} \xrightarrow{a|b} \mathtt{H} \in T\,]; \end{array}$$

```
1.  NewStates := [ red(F) ]; States := [ ]; Trans := [ ];
2.  do until NewStates = ∅ {
3.        NewTrans  :=  transitions(NewStates);
4.        Trans     :=  Trans.NewTrans;
5.        States    :=  States.NewStates;
6.        NewStates :=  remdup(targets(NewTrans)) \ States;
7.  }
8.  return Trans;
```

Figure 2: Synthesis algorithm

The synthesis algorithm is a standard least fixed point algorithm. We represent a (partially constructed) Mealy machine as a list *Trans* of transitions. A list *States* is also used to keep track of which states have been

processed, and a list *NewStates* contains the states that were newly found in the previous iteration. Initially, *NewStates* only contains the initial specification in reduced form. In each iteration, we compute the list *NewTrans* of transitions starting from *NewStates*, and add these to *Trans*. The new states for the next iteration are the target states of *NewTrans* except for the ones that have already been processed. At the end of iteration number k, all states and transitions up to depth k have been created. Hence when no new states are found, *Trans* represents $\langle\langle [\![\mathtt{F}]\!] \rangle\rangle$.

We can now finally state our synthesis result for rational function specifications.

Theorem 5 *For any rational function expression* $\mathtt{F}$*, we can effectively construct a finite, minimal realisation of* $[\![\mathtt{F}]\!]$ *using the algorithm in Figure 2.*

Proof: By Corollary 1 and Proposition 2, $\langle\langle [\![\mathtt{F}]\!] \rangle\rangle$ is a finite, minimal realisation of $[\![\mathtt{F}]\!]$. The algorithm in Figure 2 constructs a Mealy machine $\langle S, m \rangle$ whose states are (rational) function expressions. Due to Lemma 5, and the fact that all rational function expressions in S are reduced and have the same "denominator" expression, two function expressions $s_1, s_2 \in S$ are equivalent if and only if they are (syntactically) identical. Hence $\langle S, m \rangle$ is minimal. □

We illustrate our synthesis algorithm with a couple of examples.

Example 6 We construct the minimal realisation of the function f specified by $\mathtt{F} = \mathtt{X}^3 \times \mathtt{s}$, i.e., in numeric notation, $f(\sigma) = 8 \times \sigma$ for all $\sigma \in 2^\omega$. For the sake of readability, we will write derivative expressions in their numeric interpretation. For example, the derivative $(\mathtt{X}^2 + (\mathtt{X}^3 \times \mathtt{s}))$ will be denoted $4 + 8\sigma$. The diagram in Figure 3 shows the Mealy machine obtained from $\mathtt{F}$ using our algorithm. At the beginning of the computation, the only state is 8σ. In the first iteration, the immediate derivatives of 8σ are computed. These are 8σ and $4 + 8\sigma$, so $4 + 8\sigma$ is a new state. In the second iteration, we compute the immediate derivatives of $4 + 8\sigma$ and find $2 + 8\sigma$ and $6 + 8\sigma$, both of which are new. In the third iteration, we find the new states $1 + 8\sigma$, $5 + 8\sigma$, $3 + 8\sigma$ and $7 + 8\sigma$. In the fourth iteration, we find that all derivatives of the states $1 + 8\sigma$, $5 + 8\sigma$, $3 + 8\sigma$ and $7 + 8\sigma$ have already been visited, hence there are no new states after this round, and the algorithm terminates.

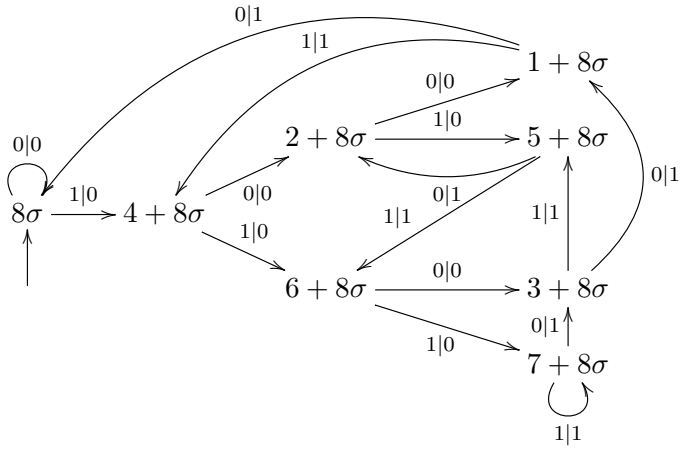


Figure 3: Mealy machine constructed from $\mathtt{F} = \mathtt{X}^3 \times \mathtt{s}$.

Example 7 As yet another example, in Figure 4 we give a minimal realisation of the rational function from equation (4), i.e., $f(\sigma) = ((-2)+(3\times\sigma))/9$. For a compact presentation, the states are labelled only by the δ-value. For example, the state which realises the function $(1+(3\times\sigma))/9$ is labelled just with 1.

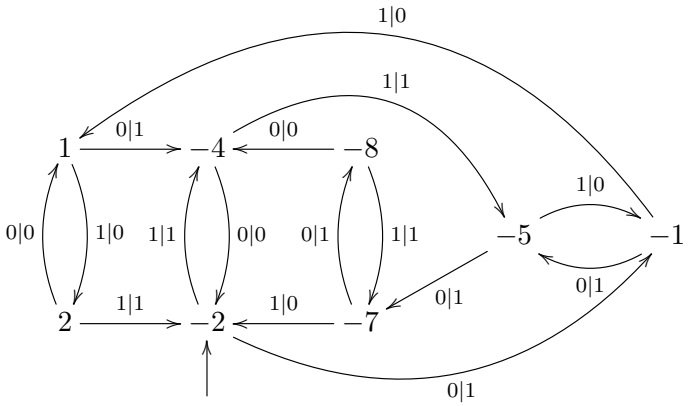
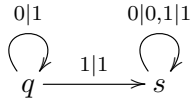


Figure 4: Mealy machine realising $f(\sigma) = ((-2) + (3 \times \sigma))/9$.

Knowing that rational functions have finite realisations, it is natural to ask whether the converse holds, that is, whether all causal bitstream functions that are realised by a finite Mealy machine can be specified by a rational function expression. This question is answered in the negative by the following example.

Example 8 Consider the Mealy machine depicted in the following diagram:



We will show that there is no rational function expression $\mathtt{F}$ such that $[\![\mathtt{F}]\!] = \mathit{beh}(q)$. To this end, we first observe that the behaviour of the state s is simply the identity function on bitstreams, hence $\mathit{beh}(s) = [\![\mathtt{s}]\!]$. Suppose, for the sake of arriving at a contradiction, that $\mathtt{F}$ is a function expression such that $[\![\mathtt{F}]\!] = \mathit{beh}(q)$, and $\mathtt{F}$ is of the form $\mathtt{F} = (\mathtt{B} + (\mathtt{C} \times \mathtt{s}))/(\mathtt{1} + (\mathtt{X} \times \mathtt{E}))$ where $\mathtt{B}, \mathtt{C}, \mathtt{E} \in \mathtt{FExpr}$ are closed, integral function expressions. By Proposition 1, $[\![\mathtt{F}_0]\!] = [\![\mathtt{F}]\!]_0 = \mathit{beh}(q)_0 = \mathit{beh}(s) = [\![\mathtt{s}]\!]$, hence $\mathtt{F}_0 \equiv \mathtt{s}$. On the other hand, Lemma 3 tells us that $\mathtt{F}_0 \equiv (\mathtt{D} + (\mathtt{C} \times \mathtt{s}))/(\mathtt{1} + (\mathtt{X} \times \mathtt{E}))$ for some closed, integral function expression $\mathtt{D}$, and hence $(\mathtt{D} + (\mathtt{C} \times \mathtt{s}))/(\mathtt{1} + (\mathtt{X} \times \mathtt{E})) \equiv \mathtt{s}$. Consequently, $\mathtt{D} \equiv \mathtt{0}$, $\mathtt{C} \equiv \mathtt{1} + (\mathtt{X} \times \mathtt{E})$, which implies that $\mathtt{C}[0] = \mathtt{C}[1] = 1$. From Definition 4 of the Mealy machine of expressions it follows that for all $a \in 2$:

$$\begin{aligned} \mathtt{F}[a] &= (\mathtt{B} + (\mathtt{C} \times \mathtt{s}))[a] = \mathtt{B}[a] \oplus (\mathtt{C}[a] \wedge \mathtt{s}[a]) = \mathtt{B}[a] \oplus (1 \wedge a) \\ &= \mathtt{B}[a] \oplus a, \end{aligned}$$

and hence,

$$\mathtt{F}[0] = \mathtt{B}[0] \quad \text{and} \quad \mathtt{F}[1] = 1 - \mathtt{B}[1]$$

But since $\mathtt{B}$ is a closed function expression, $\mathtt{B}[0] = \mathtt{B}[1]$, and hence by the above $\mathtt{F}[0] \neq \mathtt{F}[1]$. But $\mathit{beh}(q)[0] = \mathit{beh}(q)[1] = 1$ which contradicts the assumption that $[\![\mathtt{F}]\!] = \mathit{beh}(q)$, and we conclude that no such $\mathtt{F}$ exists.

Given the numeric nature of rational function expressions and their high level of abstraction, we do not find it surprising that they cannot specify all finite state Mealy behaviours.

5.4 Complexity

In this section, we analyse the time complexity of our synthesis algorithm. Given a rational function expression $\mathtt{F}$, we denote the Mealy machine constructed by the synthesis algorithm by $\langle S_{\mathtt{F}}, m_{\mathtt{F}} \rangle$. We also need the following definitions. The length $len(\mathtt{E})$ of an integral function expression $\mathtt{E}$ is the number of symbol occurrences in $\mathtt{E}$. In particular, $len(\mathtt{s}) = 1$, $len(\mathtt{E} + \mathtt{G}) = 1 + len(\mathtt{E}) + len(\mathtt{G})$, and $len(\mathtt{X}^k) = 2k - 1$. If $\mathtt{E}$ is a closed, integral function expression, then $val(\mathtt{E})$ is the integer $x \in \mathbb{Z}$ such that $[\![\mathtt{E}]\!] = \mathrm{Bin}(x)$. For example, $val(\mathtt{X}^2 + \mathtt{1}) = 5$. For an integer $x \in \mathbb{Z}$, we denote by $\mathtt{C}_x$ the unique closed, integral function expression in polynomial normal form such that $val(\mathtt{C}_x) = x$.

We consider the computation of $red(\mathtt{F})$ from $\mathtt{F}$ in line 1 of the algorithm as a preprocessing step. Consequently, we ignore its time cost in the overall analysis, and in the rest of this section, we assume that the initial specification is already reduced and of the form

$$\mathtt{F} = \frac{\mathtt{C}_d + (\mathtt{C}_m \times \mathtt{s})}{\mathtt{C}_n} \tag{7}$$

for integers $d, m, n \in \mathbb{Z}$ with $n > 0$ odd. In particular, $[\![\mathtt{F}]\!]$ is the function $f(\sigma) = \frac{d+(m\times\sigma)}{n}$. Moreover, we let $K := |d| + |m| + |n|$.

As a first step, we give a high-level description of the complexity in terms of the subcomponents of the algorithm.

Proposition 3 *The synthesis algorithm which constructs from initial specification* $\mathtt{F}$ *the Mealy machine* $\langle S_{\mathtt{F}}, m_{\mathtt{F}} \rangle$ *runs in time* $O(RM + EM^3)$ *where* M *is the number of states in* $S_{\mathtt{F}}$*,* R *is the time cost of the function* next*, and* E *is the time cost of checking syntactic equality of two states.*

Proof: For every state $\mathtt{G} \in S_{\mathtt{F}}$, we make exactly two calls to next (line 3). This yields a factor $M2R$. The number of iterations is bounded by M, since at least one new state must be added in each iteration. Adding elements to a list can be done in time proportional to the length of the list. The length of *Trans* is bounded by $2M$, and the length of *NewStates* by M. Hence the list operations in lines 4 and 5 can be carried out in time $O(M)$. The list *NewTrans* has length at most $2M$, hence computing $targets(NewTrans)$ can be done in time $O(M)$ resulting also in a list of length at most $2M$. Removing duplicates from a list of length l requires at most l^2 comparisons. Hence $remdup(targets(NewTrans))$ can be computed

in time $O(EM^2)$. Finally, with a similar argument, removing elements from $remdup(targets(NewTrans))$ that occur in *States* can also be done in time $O(EM^2)$. Summing up, we obtain an overall complexity of $O(M2R + M(M + M + EM^2 + EM^2)) \in O(RM + EM^3)$. □

We now relate M, R and E to the length of the initial specification $\mathtt{F}$. The numeric interpretation of $\mathtt{F}$, specifically the value K, will be used in the complexity analysis. The next lemma relates K and $len(\mathtt{F})$.

Lemma 6 *For the initial specification* $\mathtt{F}$, $\log(K) \in O(len(\mathtt{F}))$.

Proof: We first make the following observation. A closed, integral expression which maximises value with respect to length is of the form $\mathtt{X}^n$ for some $n \in \mathbb{N}$. Since $len(\mathtt{X}^n) = 2n-1$ and $val(\mathtt{X}^n) = 2^n$ it follows that for all closed, integral function expressions $\mathtt{C}$, $val(\mathtt{C}) \leq 2^{len(\mathtt{C})}$. It now follows that

$$K = |d| + |m| + |n| \leq 2^{len(\mathtt{C}_d)} + 2^{len(\mathtt{C}_m)} + 2^{len(\mathtt{C}_n)} \leq 3 \cdot 2^{len(\mathtt{F})}$$

and hence that $\log(K) \in O(len(\mathtt{F}))$. □

Next we show that the length of function expressions in $S_{\mathtt{F}}$ can be bounded in terms of the length of $\mathtt{F}$.

Lemma 7 *For all function expressions* $\mathtt{G} \in S_{\mathtt{F}}$, $len(\mathtt{G}) \in O(len(\mathtt{F})^2)$.

Proof: We first show that for all closed, integral function expressions $\mathtt{C}$,

$$len(\mathtt{C}) \in O(\log(|val(\mathtt{C})|)^2). \tag{8}$$

A closed, integral $\mathtt{C} \in \mathtt{FExpr}$ which maximises $len(\mathtt{C})$ with respect to $|val(\mathtt{C})|$ is of the form $\mathtt{C} = -(\mathtt{1} + \mathtt{X} + \mathtt{X}^2 + \ldots + \mathtt{X}^n)$ for some $n \in \mathbb{N}$. One can easily prove (by induction on n) that $len(-(\mathtt{1} + \mathtt{X} + \mathtt{X}^2 + \ldots + \mathtt{X}^n)) = n^2 + n + 2$. Since $n < \log(|val(\mathtt{C})|)$, it follows that $len(\mathtt{C}) \in O(\log(|val(\mathtt{C})|)^2)$.

By the definition of the next-function, any $\mathtt{G}$ in $S_{\mathtt{F}}$ is of the form $\mathtt{G} = (\mathtt{D} + (\mathtt{C}_m \times \mathtt{s}))/\mathtt{C}_n$, hence $len(\mathtt{G}) \leq len(\mathtt{D}) + len(\mathtt{F})$. By (8) and Lemma 4, $len(\mathtt{D}) \in O(\log(K)^2)$, and hence by Lemma 6, $len(\mathtt{D}) \in O(len(\mathtt{F})^2)$. It follows that $len(\mathtt{G}) \in O(len(\mathtt{F})^2) + O(len(\mathtt{F})) = O(len(\mathtt{F})^2)$. □

In order to express R in terms of $len(F)$, we analyse the length of expressions produced by the function $\mathrm{next}_{\mathrm{D}}$. As a first step, we show that derivatives of a closed, integral expression $\mathtt{C}$ are linearly bounded by $len(\mathtt{C})$.

Lemma 8 *If* $\mathtt{C}$ *is a closed, integral function expression in polynomial normal form and* $a \in 2$*, then* $len(\mathtt{C}_a) \leq 3 \cdot len(\mathtt{C})$*.*

Proof: We first show that for all $k \geq 1$:

$$len(\mathtt{X}_a^k) = 6k - 5 \tag{9}$$

The proof is by straightforward induction on k. For $k = 1$, $len(\mathtt{X}_a) = len(\mathtt{1}) = 1 = 6 \cdot 1 - 5$. Now let $k > 1$.

$$\mathtt{X}_a^k = (\mathtt{X}^{k-1} \times \mathtt{X})_a = (\mathtt{X}_a^{k-1} \times \mathtt{X}) + (\mathtt{0} \times \mathtt{1}).$$

Hence

$$len(\mathtt{X}_a^k) = len(\mathtt{X}_a^{k-1}) + 6 =_{\mathrm{IH}} 6(k-1) - 5 + 6 = 6k - 5$$

To prove the lemma, we first consider the case where $val(\mathtt{C}) \geq 0$. Since $\mathtt{C}$ is in polynomial normal form, $\mathtt{C}$ is a sum of terms of the form $\mathtt{X}^k$, $k \in \mathbb{N}$. For all $k \geq 1$ and $a \in 2$, we have from (9):

$$len(\mathtt{X}_a^k) = 6k - 5 \leq 6k - 3 = 3 \cdot len(\mathtt{X}^k).$$

From this observation it follows easily that $len(\mathtt{C}_a) \leq 3 \cdot len(\mathtt{C})$.

In case $val(\mathtt{C}) < 0$, then $\mathtt{C} = -\mathtt{E}$ for some $\mathtt{E}$, and $\mathtt{C}_a = -(\mathtt{E}_a + \iota(\mathtt{E}[a]))$, hence

$$len(\mathtt{C}_a) = 3 + len(\mathtt{E}_a) \leq 3 + 3 \cdot len(\mathtt{E}) = 3 \cdot (1 + len(\mathtt{E})) = 3 \cdot len(\mathtt{C})$$

where the inequality follows from the previous case. □

Now we can give a bound on the length of expressions that must be reduced to polynomial normal form.

Lemma 9 *For all* $\mathtt{G} \in S_{\mathtt{F}}$ *and* $a \in 2$*,* $len(\mathrm{next}_{\mathrm{D}}(\mathtt{G}, a)) \in O(len(\mathtt{F})^2)$*.*

Proof: Let $\mathtt{G} = (\mathtt{D} + (\mathtt{C} \times \mathtt{s}))/(\mathtt{1} + (\mathtt{X} \times \mathtt{E}))$. Writing out the definition of $\mathrm{next}_{\mathrm{D}}(\mathtt{G}, a)$, we find that

$$\begin{array}{rcl} len(\mathrm{next}_{\mathrm{D}}(\mathtt{G}, a)) & \leq & len(\mathtt{D}_a) + len(\mathtt{C}_a) + len(\mathtt{E})) + 7 \\ \text{(Lemma 8)} & \leq & 3(len(\mathtt{D}) + len(\mathtt{C})) + len(\mathtt{E}) + 7 \\ & \in & O(len(\mathtt{G})) \\ \text{(Lemma 7)} & \in & O(len(\mathtt{F})^2). \end{array}$$

□

We can now state the time complexity of our synthesis algorithm expressed in terms of the length of the initial specification.

Theorem 9 *The synthesis algorithm which constructs from initial specification* $\mathtt{F}$ *the Mealy machine* $\langle S_{\mathtt{F}}, m_{\mathtt{F}}\rangle$ *runs in time* $2^{O(len(\mathtt{F})^2)}$.

Proof: We first express R in terms of $len(\mathtt{F})$. Let G be in $S_{\mathtt{F}}$. The time complexity of computing $\mathrm{next}(\mathtt{G}, a)$ is dominated by the time complexity of computing $pnf(\mathrm{next}_{\mathrm{D}}(\mathtt{G}, a))$. Computing the polynomial normal form of an integral expression $\mathtt{E}$ is in the worst case exponential in $len(\mathtt{E})$ due to the duplication of subexpressions when applying the distributive law. E.g. $\mathtt{P} \times (\mathtt{Q} + \mathtt{R}) = \mathtt{P} \times \mathtt{Q} + \mathtt{P} \times \mathtt{R}$. All other manipulations in the computation of $pnf(\mathtt{E})$ are polynomial in the length of the expression they are applied to. By Lemma 9, $len(\mathrm{next}_{\mathrm{D}}(\mathtt{G}, a)) \in O(len(\mathtt{F})^2)$, hence

$$R \in 2^{O(len(\mathtt{F})^2)} \tag{10}$$

Checking whether two expressions in $S_{\mathtt{F}}$ are syntactically equal is linear in the length of the two expressions. Hence by Lemma 7,

$$E \in O(len(\mathtt{F})^2) \tag{11}$$

Lemma 4 can be used to show that the number of derivatives of $[\![\mathtt{F}]\!]$ is at most $K = |d| + |m| + |n|$, and in many cases an even tighter upper bound can be given. A detailed proof of this result can be found in Theorem 3.3.10 of [6]. Consequently, $M \leq K$ and by Lemma 6,

$$M \in O(2^{len(\mathtt{F})}) \tag{12}$$

Combining Proposition 3 with equations (10), (11) and (12) we find that the overall complexity of the synthesis algorithm is

$$O(RM + EM^3) = O(2^{O(len(\mathtt{F})^2)} \cdot 2^{len(\mathtt{F})} + len(\mathtt{F})^2 \cdot 2^{len(\mathtt{F})}) \in 2^{O(len(\mathtt{F})^2)}.$$

□

Remark 1 *The complexity result stated in Theorem 3.5.10 of [6] differs from Theorem 9 above due to the following. Firstly, the synthesis algorithm in [6] is based on computing actual syntactic derivativatives rather than the function* next *which only applies to rational function expressions. In particular, the algorithm described in [6] also allows the partial construction of Mealy machines from non-rational function expressions. Secondly, the syntax of function expressions here differs slightly from the 2-adic expressions in [6] where terms of the form* $\mathtt{X}^n$ *are atomic grammar entities. The implementation in [7] is based on the algorithm presented in [6].*

6 Discussion and Related Work

Brzozowski [3] showed how to construct a deterministic finite automaton for a rational expression by computing its finitely many *derivatives*, herewith lifting the well-known fact that rational languages have a finite number of (left) quotients, to the symbolic level of expressions. Since then, various applications and generalisations have been studied. In [1], Antimirov introduced the notion of *partial derivative* and used it to construct nondeterministic finite automata. In [14, 16], we reformulated Brzozowski's original approach in coalgebraic terms and generalised it to formal power series over arbitrary semirings, providing at the same time a generalisation of Antimirov's results. A similar generalisation to formal power series and rational expressions with multiplicities was found, independently, by Lombardy and Sakarovitch [12].

We have shown how to construct (in exponential quadratic time) a finite Mealy machine realisation of rational 2-adic functions by a symbolic computation of derivatives. The same principles can be used to perform Mealy synthesis of functions specified in mod-2 arithmetic (cf. [6, Ch. 3]). We expect that the method can also be extended to include bitstream functions specified in the Boolean bitstream algebra, respectively Kleene bitstream algebra, described in [17] alongside the 2-adic and mod-2 bitstream algebras. It would be interesting to combine the operators of these bitstream algebras into "mixed specifications". Although a Mealy machine of "mixed expressions" can be defined (using the stream differential equations), and various "mixed identities" are proved in [17], it is not clear whether there exists an equivalence on mixed expressions with finite index, which is effectively decidable. Such an equivalence is necessary to generalise symbolic synthesis to mixed specifications.

Closely related to the work presented here is the work in [19] where coalgebras are synthesised from a kind of generalised recursive process specifications. The method is similar to ours in the sense that it relies on a symbolic computation of generated subcoalgebras. This construction is parametric in the functor T which defines the coalgebra type as well as the syntax of the specifications, and so covers many different automaton-like system types (including Mealy machines) in a uniform framework. Instantiating the results of [19] to Mealy machines, we point out the following differences with our work. The syntax of the specification language in [19] is in close correspondence with the semantic structure and specifying rational 2-adic functions would be inconvenient to say the least. On the other hand, the

close connection between syntax and semantics ensures that any behaviour of a finite Mealy machine can be specified in their language and vice versa, all specifications have finite realisations. As we have seen in Example 8, rational function expressions are not expressively complete with respect to finite Mealy machines. However, function expressions can specify infinite-state behaviours such as $f(\sigma) = \sigma \times \sigma$ which are not expressible in the language of [19]. Finally, we mention that the synthesis algorithm in [19] does not necessarily produce a minimal Mealy machine as is the case with our synthesis algorithm.

The idea of generating behaviour from syntax is possible at a very general level. Such interplay between algebra (syntax) and coalgebra (behaviour) can often be captured by so-called bialgebras for a distributive law (cf. [20, 2]). For example, the stream differential equations for the 2-adic operators define a distributive law of the 2-adic signature over stream behaviour. Other examples of distributive laws are rules in structural operational semantics (cf. [2]), and also regular expressions and deterministic automata form an example of this much more abstract setup (cf. [11]). Such a bialgebraic picture also exists for function expressions and Mealy machines, although in a slightly less direct way than in the examples just mentioned. This result can be found in [9].

References

[1] V. Antimirov. Partial derivatives of regular expressions, and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[2] F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[3] J.A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[4] S. Eilenberg. *Automata, Languages and Machines (Vol. A)*. Academic Press, 1974.

[5] F.Q. Gouvêa. *p-adic Numbers: An Introduction*. Springer, 1993.

[6] H.H. Hansen. *Coalgebraic Modelling: applications in automata theory and modal logic*. PhD thesis, Vrije Universiteit Amsterdam, 2009.

[7] H.H. Hansen and D. Costa. DIFFCAL. Tool webpage (source code, documentation, executable) currently available at: `http://homepages.cwi.nl/~costa/projects/diffcal`, 2005.

[8] H.H. Hansen, D. Costa, and J.J.M.M. Rutten. Synthesis of Mealy machines using derivatives. In *Proceedings of CMCS 2006*, volume 164(1) of *Electronic Notes in Theoretical Computer Science*, pages 27–45. Elsevier Science Publishers, 2006.

[9] H.H. Hansen and B. Klin. Pointwise extensions of GSOS-defined operations. To appear in *Mathematical Structures in Computer Science*.

[10] E.C.R. Hehner and R.N. Horspool. A new representation of the rational numbers for fast easy arithmetic. *SIAM Journal on Computing*, 8:124–134, 1979.

[11] B. Jacobs. A bialgebraic review of deterministic automata, regular expressions and languages. In *Algebra, Meaning and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 375–404. Springer, 2006.

[12] S. Lombardy and J. Sakarovitch. Derivatives of rational expressions with multiplicity. *Theoretical Computer Science*, 332:141–177, 2005.

[13] G.N. Raney. Sequential functions. *Journal of the ACM*, 5(2):177–180, April 1958.

[14] J.J.M.M. Rutten. Automata, power series, and coinduction: taking input derivatives seriously (extended abstract). In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP 1999)*, volume 1644 of *Lecture Notes in Computer Science*, pages 645–654, 1999.

[15] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[16] J.J.M.M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata and power series. *Theoretical Computer Science*, 308(1):1–53, 2003.

[17] J.J.M.M. Rutten. Algebra, bitstreams, and circuits. In *Proceedings of AAA68 Workshop on General Algebra*, volume 16 of *Contributions to General Algebra*, pages 231–250. Verlag Johannes Heyn, Klagenfurt, 2005.

[18] J.J.M.M. Rutten. Algebraic specification and coalgebraic synthesis of Mealy machines. In *Proceedings of FACS 2005*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 305–319, 2006.

[19] A.M. Silva, M.M. Bonsangue, and J.J.M.M. Rutten. Kleene coalgebras. Technical Report SEN-1001, Centrum Wiskunde & Informatica, 2010.

[20] D. Turi and G.D. Plotkin. Towards a mathemathical operational semantics. In *Proceedings of LICS 1997*, pages 280–291. IEEE Computer Society, 1997.

Appendix

Proof of Proposition 1. Let $a \in 2$ and $\sigma \in 2^\omega$.

$$\begin{array}{llllll}
[\![\mathtt{0}]\!][a] & = & [\![\mathtt{0}]\!](a{:}\sigma)(0) & = & \mathtt{0}(0) = 0 & = & \mathtt{0}[a] \\
[\![\mathtt{0}]\!]_a(\sigma) & = & [\![\mathtt{0}]\!](a{:}\sigma)' & = & [0]' = [0] & = & [\![\mathtt{0}_a]\!](\sigma) \\
[\![\mathtt{1}]\!][a] & = & [\![\mathtt{1}]\!](a{:}\sigma)(0) & = & \mathtt{1}(0) = 1 & = & \mathtt{1}[a] \\
[\![\mathtt{1}]\!]_a(\sigma) & = & [\![\mathtt{1}]\!](a{:}\sigma)' & = & [1]' = [0] & = & [\![\mathtt{1}_a]\!](\sigma) \\
[\![\mathtt{X}]\!][a] & = & [\![\mathtt{X}]\!](a{:}\sigma)(0) & = & X(0) = 0 & = & \mathtt{X}[a] \\
[\![\mathtt{X}]\!]_a(\sigma) & = & [\![\mathtt{X}]\!](a{:}\sigma)' & = & X' = [1] & = & [\![\mathtt{X}_a]\!](\sigma) \\
[\![\mathtt{s}]\!][a] & = & ([\![\mathtt{s}]\!](a{:}\sigma))(0) & = & (a{:}\sigma)(0) = a & = & \mathtt{s}[a] \\
[\![\mathtt{s}]\!]_a(\sigma) & = & ([\![\mathtt{s}]\!](a{:}\sigma))' & = & (a{:}\sigma)' = \sigma & = & [\![\mathtt{s}_a]\!](\sigma)
\end{array}$$

Note that for any $b \in 2$ and $\sigma \in 2^\omega$: $[\![\iota(b)]\!] = [b]$.
Minus:

$$\begin{array}{lll}
[\![-\mathtt{F}]\!][a] & = & ([\![-\mathtt{F}]\!](a{:}\sigma))(0) = (-[\![\mathtt{F}]\!](a{:}\sigma))(0) = ([\![\mathtt{F}]\!](a{:}\sigma))(0) = [\![\mathtt{F}]\!][a]. \\
 & \stackrel{\text{IH}}{=} & \mathtt{F}[a] = -\mathtt{F}[a] \\
[\![-\mathtt{F}]\!]_a(\sigma) & = & ([\![-\mathtt{F}]\!](a{:}\sigma))' = (-[\![\mathtt{F}]\!](a{:}\sigma))' \\
 & = & -([\![\mathtt{F}]\!](a{:}\sigma)' + [([\![\mathtt{F}]\!](a{:}\sigma))(0)]) = -([\![\mathtt{F}]\!]_a + [[\![\mathtt{F}]\!][a]])(\sigma) \\
 & \stackrel{\text{IH}}{=} & [\![-(\mathtt{F}_a + \iota(\mathtt{F}[a]))]\!](\sigma) = [\![(-\mathtt{F})_a]\!](\sigma).
\end{array}$$

Sum:

$$\begin{array}{rcl}
[\![\mathtt{F}+\mathtt{G}]\!][a] & = & ([\![\mathtt{F}+\mathtt{G}]\!](a{:}\sigma))(0) = ([\![\mathtt{F}]\!](a{:}\sigma)+[\![\mathtt{G}]\!](a{:}\sigma))(0) \\
& = & [\![\mathtt{F}]\!](a{:}\sigma)(0) \oplus [\![\mathtt{G}]\!](a{:}\sigma)(0) = [\![\mathtt{F}]\!][a] \oplus [\![\mathtt{G}]\!][a] \\
& \stackrel{\text{IH}}{=} & \mathtt{F}[a] \oplus \mathtt{G}[a] = (\mathtt{F}+\mathtt{G})[a]. \\
[\![\mathtt{F}+\mathtt{G}]\!]_a(\sigma) & = & ([\![\mathtt{F}+\mathtt{G}]\!](a{:}\sigma))' = ([\![\mathtt{F}]\!](a{:}\sigma)+[\![\mathtt{G}]\!](a{:}\sigma))' \\
& = & ([\![\mathtt{F}]\!]_a(\sigma)+[\![\mathtt{G}]\!]_a(\sigma))+[[\![\mathtt{F}]\!][a] \wedge [\![\mathtt{G}]\!][a]] \\
& \stackrel{\text{IH}}{=} & ([\![\mathtt{F}_a]\!](\sigma)+[\![\mathtt{G}_a]\!](\sigma))+[\mathtt{F}[a] \wedge \mathtt{G}[a]] \\
& = & [\![(\mathtt{F}_a+\mathtt{G}_a)+\iota(\mathtt{F}[a] \wedge \mathtt{G}[a])]\!](\sigma) \\
& = & [\![(\mathtt{F}+\mathtt{G})_a]\!](\sigma).
\end{array}$$

Product:

$$\begin{array}{rcl}
[\![\mathtt{F}\times\mathtt{G}]\!][a] & = & ([\![\mathtt{F}\times\mathtt{G}]\!](a{:}\sigma))(0) = ([\![\mathtt{F}]\!](a{:}\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma))(0) \\
& = & [\![\mathtt{F}]\!][a] \wedge [\![\mathtt{G}]\!][a] \\
& \stackrel{\text{IH}}{=} & \mathtt{F}[a] \wedge \mathtt{G}[a] = (\mathtt{F}\times\mathtt{G})[a]. \\
[\![\mathtt{F}\times\mathtt{G}]\!]_a(\sigma) & = & ([\![\mathtt{F}\times\mathtt{G}]\!](a{:}\sigma))' = ([\![\mathtt{F}]\!](a{:}\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma))' \\
& = & (([\![\mathtt{F}]\!](a{:}\sigma))'\times[\![\mathtt{G}]\!](a{:}\sigma))+ \\
& & \qquad ([([\![\mathtt{F}]\!](a{:}\sigma))(0)]\times([\![\mathtt{G}]\!](a{:}\sigma))') \\
& = & ([\![\mathtt{F}]\!]_a(\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma))+([[\![\mathtt{F}]\!][a]]\times[\![\mathtt{G}]\!]_a(\sigma)) \\
& \stackrel{\text{IH}}{=} & ([\![\mathtt{F}_a]\!](\sigma)\times[\![\mathtt{G}]\!](a{:}\sigma))+([\mathtt{F}[a]]\times[\![\mathtt{G}_a]\!](\sigma)) \\
& \stackrel{\text{Lem.1}}{=} & [\![(\mathtt{F}_a(\sigma)\times\mathtt{G}(a{:}\mathtt{s}))+(\iota(\mathtt{F}[a])\times\mathtt{G}_a)]\!](\sigma) \\
& = & [\![(\mathtt{F}\times\mathtt{G})_a]\!](\sigma).
\end{array}$$

Inverse:

$$\begin{array}{rcl}
[\![\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F}))]\!][a] & = & 1 = (\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F})))[a]. \\
[\![\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F}))]\!]_a(\sigma) & = & ([\![\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F}))]\!](a{:}\sigma))' \\
& = & (1/(1+(X\times[\![\mathtt{F}]\!](a{:}\sigma))))' \\
& = & -([\![\mathtt{F}]\!](a{:}\sigma)/(1+(X\times[\![\mathtt{F}]\!](a{:}\sigma)))) \\
& \stackrel{\text{Lem.1}}{=} & [\![-(\mathtt{F}(a{:}\mathtt{s})/(\mathtt{1}+(\mathtt{X}\times\mathtt{F}(a{:}\mathtt{s}))))]\!](\sigma) \\
& = & [\![(\mathtt{1}/(\mathtt{1}+(\mathtt{X}\times\mathtt{F})))_a]\!](\sigma).
\end{array}$$

□

Proof of Lemma 3. Let $f(\sigma) = \frac{d+m\times\sigma}{n}$ for integers d, m and n with n odd, and let $a \in 2$. First, using the stream differential equations of Definition 1 and the identities of commutative rings, we find by a straightforward calculation that the initial value and stream derivative of a bitstream quotient σ/τ (with $\tau(0) = 1$) are given by:

$$(\sigma/\tau)(0) = \sigma(0) \quad \text{and} \quad (\sigma/\tau)' = (\sigma' - [\sigma(0)] \times \tau')/\tau \tag{13}$$

We now use (13) to compute $f_a(\sigma)$ for $a \in 2$. The steps marked with (†) use commutativity of $\times$ and $+$.

$$\begin{aligned} f_a(\sigma) &= \left(\frac{d + (m \times (a{:}\sigma))}{n}\right)' \overset{(\dagger)}{=} \left(\frac{d + ((a{:}\sigma) \times m)}{n}\right)' \\ &= \frac{(d + ((a{:}\sigma) \times m))' - ([d(0) \oplus (a \wedge m(0))] \times n')}{n} \\ &= \frac{\begin{array}{r} d' + ((\sigma \times m) + ([a] \times m')) + [d(0) \wedge (a \wedge m(0))] - \\ ([d(0) \oplus (a \wedge m(0))] \times n') \end{array}}{n} \\ &\overset{(\dagger)}{=} \begin{cases} \dfrac{(d' - ([d(0)] \times n')) + (m \times \sigma)}{n} & \text{if } a = 0 \\ \dfrac{\begin{array}{r} ((d' + m' + [d(0) \wedge m(0)]) - ([d(0) \oplus m(0)] \times n') \\ +(m \times \sigma) \end{array}}{n} & \text{if } a = 1 \end{cases} \end{aligned}$$

The initial values are computed in a similar manner. Hence we obtain the following equations for the $\delta(a)$-value in (5):

$$\begin{aligned} \delta(0) &= d' - [d(0)] \times n' \text{ and} \\ \delta(1) &= d' + m' + [d(0) \wedge m(0)] - [d(0) \oplus m(0)] \times n'. \end{aligned}$$

The rest of the proof is now straightforward using (2) (p. 104). If d is even then $\delta(0) = d' = \frac{1}{2}d$, and if d is odd, we get:

$$\delta(0) = d' - n' = \frac{1}{2}(d-1) - \frac{1}{2}(n-1) = \frac{1}{2}(d-n).$$

When $d + m$ is even with d and m both odd, then

$$\delta(1) = d' + m' + 1 = \frac{1}{2}(d-1) + \frac{1}{2}(m-1) + 1 = \frac{1}{2}(d+m).$$

If $d+m$ is odd with d odd, and m even, then

$$\delta(1) = d' + m' - n' = \frac{1}{2}(d-1) + \frac{1}{2}\,m - \frac{1}{2}(n-1) = \frac{1}{2}(d+m-n).$$

The remaining cases are proved similarly, details are left to the reader. □

Proof of Lemma 4. It is a consequence of Lemma 3 that the derivatives of f have the given format (6), since f is itself of the form required in Lemma 3, and hence so are all derivatives of f. We prove by induction on the length of $w \in 2^*$ that the numeric value $\delta(w)$ is in the given range. The base case ($w = \varepsilon$) is clear. To prove the inductive step, we use the numeric interpretation of derivatives of rational 2-adic functions given in Lemma 3. To ease notation, let $l = \min\{d, -n+1, -n+m+1\}$ and $u = \max\{d, m-1, 0\}$. Note that $l \le 0 \le u$. Assume as induction hypothesis (IH) that $l \;\le\; \delta(w) \;\le\; u$. Inequalities obtained from the induction hypothesis will be denoted by $\le_{IH}$.

Induction step for $\delta(w0)$: We first consider the case where $\delta(w)$ is even, and thus $\delta(w0) = \frac{1}{2}\delta(w)$. We have the following cases:

$$\begin{array}{ll} \text{if } \delta(w) \ge 0\text{:} & l \;\le\; 0 \;\le\; \frac{1}{2}\delta(w) \;\le\; \delta(w) \le_{IH} u. \\ \text{if } \delta(w) < 0\text{:} & l \;\le_{IH}\; \delta(w) \;<\; \frac{1}{2}\delta(w) \;<\; 0 \;\le\; u. \end{array}$$

Now if $\delta(w)$ is odd, then $\delta(w0) = \frac{1}{2}(\delta(w) - n)$. To prove the lower bound, we have

$$l \;\le\; -n+1 \;\Rightarrow\; 2l \;\le\; l-n+1 \;\le_{IH}\; \delta(w) - n + 1,$$

and since $\delta(w) - n + 1$ is odd, it follows that $2l \le \delta(w) - n$ and hence $l \le \frac{1}{2}(\delta(w) - n)$. The upper bound follows easily from $\delta(w) - n \;<\; \delta(w) \;\le_{IH}\; u$ which implies $\frac{1}{2}(\delta(w) - n) \;\le\; u$, since $u \ge 0$.

Induction step for $\delta(w1)$: If $\delta(w)+m$ is even, then $\delta(w1) = \frac{1}{2}(\delta(w)+m)$. We first prove the lower bound. We have (since $n > 0$),

$$l \;\le\; -n+m+1 \;\le\; m \quad \text{and} \quad l \;\le_{IH}\; \delta(w)$$

whence $2l \;\le\; \delta(w) + m$, and $l \le \frac{1}{2}(\delta(w) + m)$. For the upper bound, we have

$$m - 1 \;\le\; u \quad \text{and} \quad \delta(w) \le_{IH} u$$

whence $\delta(w) + m - 1 \;\le\; 2u$, and $\delta(w) + m - 1$ must be odd, since $\delta(w) + m$ is even. It follows that $\delta(w) + m \le 2u$, which in turn implies $\frac{1}{2}(\delta(w) + m) \le u$.

If $\delta(w) + m$ is odd, then $\delta(w1) = \frac{1}{2}(\delta(w) + m - n)$. We know that $l \leq -n + m + 1$ and hence $2l \leq l - n + m + 1 \leq_{IH} \delta(w) + m - n + 1$. Since $\delta(w) + m - n + 1$ is odd, it follows that $2l \leq \delta(w) + m - n$, and hence $l \leq \frac{1}{2}(\delta(w) + m - n)$.

The upper bound is proven in a similar fashion. We have $m - 1 \leq u$ which implies $\delta(w) + m - n \leq_{IH} u + m - n \leq u + m - 1 \leq 2u$, and it follows that $\frac{1}{2}(\delta(w) + m - n) \leq u$. □