Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

Synthesis of Mealy machines using derivatives

H.H. Hansen, D. de Oliveira Costa, J.J.M.M. Rutten

REPORT SEN-R0605 MAY 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2006, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Synthesis of Mealy machines using derivatives

ABSTRACT
In Rutten (2005), the theoretical basis was given for the synthesis of binary Mealy machines from specifications in 2-adic arithmetic. This construction is based on the symbolic computation of the coalgebraic notion of stream function derivative, a generalisation of the Brzozowski derivative of regular expressions. In this paper we complete the construction of Mealy machines from specifications in both 2-adic and modulo-2 arithmetic by describing how we decide equivalence of expressions via reduction to normal forms; we present a Haskell implementation of this Mealy synthesis algorithm; and a theoretical result which characterises the (number of) states in Mealy machines constructed from rational 2-adic specifications.

2000 Mathematics Subject Classification: 03D99, 03C13, 18B20, 68Q10, 68Q45, 68Q55, 68Q65
Keywords and Phrases: Mealy machine; synthesis; derivatives; streams; coalgebra; algebraic specification
Note: This work was carried out under the NWO project C-Quattro, and FCT grant 13762 (Portugal).

Synthesis of Mealy Machines Using Derivatives

Helle Hvid Hansen [1]

Free University Amsterdam (VUA) and
Centrum voor Wiskunde en Informatica (CWI)
De Boelelaan 1081a, NL–1081 HV Amsterdam, Netherlands

David de Oliveira Costa [2]

Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, NL–1090 GB Amsterdam, Netherlands

Jan Rutten [3]

Free University Amsterdam (VUA) and
Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, NL–1090 GB Amsterdam, Netherlands

Abstract

In Rutten [16] the theoretical basis was given for the synthesis of binary Mealy machines from specifications in 2-adic arithmetic. This construction is based on the symbolic computation of the coalgebraic notion of stream function derivative, a generalisation of the Brzozowski derivative of regular expressions. In this paper we complete the construction of Mealy machines from specifications in both 2-adic and modulo-2 arithmetic by describing how we decide equivalence of expressions via reduction to normal forms; we present a Haskell implementation of this Mealy synthesis algorithm; and a theoretical result which characterises the (number of) states in Mealy machines constructed from rational 2-adic specifications.

Key words: Mealy machine, synthesis, derivatives, streams, coalgebra.

1 Introduction

Mealy machines are finite state transducers used in the modelling and specification of systems performing synchronous, ongoing computations such as

[1] Email: `hhhansen@few.vu.nl`
[2] Email: `costa@cwi.nl` Supported by FCT grant 13762 – 2003, Portugal.
[3] Email: `janr@cwi.nl`

sequential digital circuits (cf. [9]), and more generally, reactive systems (see e.g. [17]). Synthesis of Mealy machines refers to an automated process of constructing from a formal specification a Mealy machine whose behaviour satisfies or realises the specification.

In Rutten [16] the theoretical basis was given for the synthesis of binary Mealy machines from specifications in 2-adic arithmetic: Mealy machines can be seen as coalgebras for the Set-functor $\mathsf{M}(S) = (B \times S)^A$, and their behaviour as causal stream functions $f : A^\omega \to B^\omega$. The set of causal stream functions becomes a final M-coalgebra under the operations of initial output and stream function derivative, and a minimal Mealy coalgebra with behaviour f is obtained from the subcoalgebra $\langle f \rangle$ generated by f in the final Mealy coalgebra. Based on this result, a synthesis method is sketched for bitstream functions (i.e., $A = B = 2 = \{0, 1\}$) specified in 2-adic arithmetic. The method relies on the symbolic computation of initial output and stream function derivative, and is similar to Brzozowski's construction of finite deterministic automata from regular expressions.

In this paper we complete the construction of Mealy machines from bitstream function specifications in both 2-adic and modulo-2 arithmetic. Our present contributions are (i) a description of how we compute normal forms in the algebras of 2-adic and modulo-2 arithmetic in order to determine whether two expressions specify the same behaviour; this is crucial for the termination of the synthesis algorithm, and was not fully addressed in [16]; (ii) an implementation in the functional programming language Haskell [8] of Mealy synthesis from specifications in both 2-adic and modulo-2 arithmetic; (iii) a characterisation of the states in the minimal Mealy machine constructed from a rational 2-adic specification (Theorem 3.3). This result provides us with an alternative proof of the fact that rational 2-adic bitstream functions have finitely many derivatives, and moreover an upper bound on their number can be expressed in terms of the specification (Corollary 3.4). We point out that these results on rational 2-adic functions were conjectured based on data generated by our Haskell program.

In section 2 we introduce basic notions together with the coalgebraic view on Mealy machines (cf. [15,16]), and in section 3 we present the bitstream algebras that we use as specification languages. Parts of these preliminary sections are already contained in [16], but in the current presentation we are motivated by implementation concerns and hence more careful about the distinction between syntax and semantics. The new results on rational 2-adic functions are found in subsection 3.2. In section 4, we describe how we compute stream function derivatives symbolically, and how we determine equivalence via normal forms. In section 5 the actual construction of Mealy machines using these symbolic derivatives is described. A brief, user-oriented description of our Haskell program is found in section 6, and finally, we discuss related work in section 7.

2 Preliminaries

The natural numbers are denoted by $\mathbb{N}$, the integers by $\mathbb{Z}$, and the rational numbers by $\mathbb{Q}$. The absolute value of a rational number x is written $|x|$, and the sign of rational numbers is given by the function $sgn : \mathbb{Q} \to \{-1, 0, 1\}$ with the usual definition.

We denote by Set the category of sets and functions, and a Set-functor F is then a functor from Set to Set. Given a Set-functor F, an F-*coalgebra* (X, γ) consists of a set X together with a function $\gamma : X \to \mathsf{F}(X)$. A function $f : X \to Y$ is an F-*coalgebra homomorphism* between F-coalgebras (X, γ) and (Y, δ) if $\delta \circ f = \mathsf{F}(f) \circ \gamma$. An F-coalgebra (Z, ϕ) is *final* if for any F-coalgebra (X, γ) there is a unique F-coalgebra homomorphism $h : (X, \gamma) \to (Z, \phi)$, also referred to as the *final map*.

2.1 Streams and stream differential equations

Let A be an arbitrary set, then A^* is the set of (finite) words over A, ϵ is the empty word, and $A^\omega = \{\alpha \mid \alpha : \mathbb{N} \to A\}$ is the set of streams over A. For $\alpha \in A^\omega$ we will also write $\alpha = (\alpha(0), \alpha(1), \alpha(2), \ldots)$, and for $a \in A$ and $\alpha \in A^\omega$, we will use the notation $a\,{:}\,\alpha$ for the stream $(a, \alpha(0), \alpha(1), \ldots)$. The *initial value of* $\alpha \in A^\omega$ is defined as $\alpha(0)$, and the *stream derivative of* α is the stream $\alpha' = (\alpha(1), \alpha(2), \alpha(3), \ldots)$. Defining the map $\gamma : A^\omega \to A \times A^\omega$ by $\alpha \mapsto \langle \alpha(0), \alpha' \rangle$, it is well-known that (A^ω, γ) is a coalgebra for the Set-functor S_A defined by $\mathsf{S}_A(X) = A \times X$. We will also refer to S-coalgebras as *stream automata*. Moreover, (A^ω, γ) is a final S-coalgebra which means that for any S-coalgebra $(X, \chi : X \to A \times X)$ there is a unique S-coalgebra homomorphism $h_\mathsf{S} : (X, \chi) \to (A^\omega, \gamma)$. This can easily be proved if we write $\chi(x) = \langle o(x), t(x) \rangle$, and then let $h_\mathsf{S}(x) = (o(x), o(t(x)), o(t(t(x))), \ldots)$ for all $x \in X$. The map h_S can be seen to assign each $x \in X$ with its (stream) behaviour.

In particular, the behaviour of a stream is simply the stream itself, thus if two streams have the same behaviour, then they must be equal. This proof principle is called *stream coinduction*, and is formally defined via stream bisimulations: A relation $R \subseteq A^\omega \times A^\omega$ is a *stream bisimulation* if for all $(\alpha, \beta) \in R$, (i) $\alpha(0) = \beta(0)$ and (ii) $(\alpha', \beta') \in R$.

Theorem 2.1 (Stream coinduction) *For all $\alpha, \beta \in A^\omega$, if there is a bisimulation $R \subseteq A^\omega \times A^\omega$ such that $(\alpha, \beta) \in R$, then $\alpha = \beta$.*

We omit the straightforward proof of the above theorem, which may be found in [15]. One way of applying the coinduction proof principle is to specify streams or stream operations by defining their behaviour in terms of initial value and stream derivative. For example, we can define the bitstream representations of rational numbers with odd denominator. Let $\hat{Q} = \{n/2m + 1 \mid n, m \in \mathbb{Z}\}$ denote the set of rational numbers with odd denominator, and let 2^ω be the set of bitstreams (i.e., streams over $2 = \{0, 1\}$), then we can turn $\hat{Q}$

into a bitstream automaton as follows:

$$q(0) = odd(q) \quad \text{and} \quad q' = \frac{q - odd(q)}{2}, \tag{1}$$

where $odd(n/2m+1) = n \mod 2$. The final (bitstream) map $B : \hat{Q} \to 2^\omega$ now associates a bitstream with every $q \in \hat{Q}$, and using stream coinduction one can show that $B(n/2m+1) = B(k/2l+1)$ iff $n(2l+1) = (2m+1)k$. In particular, it can easily be checked that for a natural number $n \in \mathbb{N}$, $B(n)$ is simply the binary expansion of n (followed by a tail of zeros). We will return to this example in section 3.1. Equations such as those used in (1) are called *stream differential equations* (cf. [15]), and under certain well-formedness conditions they have a unique stream solution given by the final map. In section 3, we will define operations on bitstreams using stream differential equations.

2.2 Mealy machines and coalgebras

Assume the sets A and B are given. A *Mealy coalgebra* with input in A and output in B is a coalgebra for the Set-functor M defined by $\mathsf{M}(X) = (B \times X)^A$. If Q is a set (of states) and (Q, ϕ) is a Mealy coalgebra, then the transition structure $\phi : Q \to (B \times Q)^A$ associates with every state $q \in Q$ an output function $o_q : A \to B$ and a next-state function $d_q : A \to Q$ defined by $\phi(q)(a) = (o_q(a), d_q(a))$ for all $a \in A$. For $\phi(q)(a) = (b, r)$ we will also use the notation:

$$\begin{tikzcd} q \arrow[r, "a|b"] & r \end{tikzcd}.$$

If (Q, ϕ) and (T, ψ) are Mealy coalgebras, then a function $g : Q \to T$ is a *Mealy homomorphism* if g respects the transition structure: for all $q \in Q$, and all $a \in A$, $o_q(a) = o_{g(q)}(a)$ and $d_{g(q)}(a) = g(d_q(a))$:

$$\begin{tikzcd} q \arrow[r, "a|b"] & r & \Longleftrightarrow & g(q) \arrow[r, "a|b"] & g(r) \end{tikzcd}.$$

An *initialised Mealy coalgebra* is a triple (Q, ϕ, q) where (Q, ϕ) is a Mealy coalgebra and $q \in Q$ is the initial state. A *Mealy machine* is an initialised Mealy coalgebra (Q, ϕ, q_0) in which the set of states Q, and the input/output sets A and B are finite. Mealy machines are also referred to as sequential machines (cf. [4]), and rather than being a language recognition device, a Mealy machine is a so-called *deterministic transducer*, i.e., it transform input streams to output streams in a deterministic manner. This input-output behaviour coincides with the coalgebraic notion of behaviour. Given a Mealy coalgebra (Q, ϕ), the *(Mealy) behaviour of a state* $q_0 \in Q$ is the stream function $Beh(q_0) : A^\omega \to B^\omega$ which maps $\alpha \in A^\omega$ to the stream $(b_0, b_1, b_2, \ldots) \in B^\omega$ of outputs observed on input α starting in q_0:

$$\begin{tikzcd} q_0 \arrow[r, "\alpha(0)|b_0"] & q_1 \arrow[r, "\alpha(1)|b_1"] & \ldots \arrow[r, "\alpha(k-1)|b_k"] & q_k \arrow[r, "\alpha(k)|b_k"] & q_{k+1} \ \ldots \end{tikzcd}$$

More precisely, $Beh(q_0)(\alpha)(k)$ is defined inductively for all $k \geq 0$ by:

$$Beh(q_0)(\alpha)(k) = o_{q_k}(\alpha(k)), \quad \text{where } q_{k+1} = d_{q_k}(\alpha(k)). \tag{2}$$

It is well-known (and easy to check) that Mealy behaviours $f : A^\omega \to B^\omega$ have the property of being causal, meaning that the n-th element of $f(\alpha)$ only depends on the first n elements of the input α. Formally, $f : A^\omega \to B^\omega$ is *causal* if for all $\alpha, \beta \in A^\omega$ and for all $n \in \mathbb{N}$,

$$\text{if } \forall k \leq n.\, \alpha(k) = \beta(k) \text{ then } f(\alpha)(n) = f(\beta)(n).$$

Thus with every state in a Mealy coalgebra we can associate a causal stream function via the behaviour map Beh. Moreover, the set of causal stream functions can itself be viewed as a Mealy coalgebra via the notion of initial output and stream function derivative.

Definition 2.2 Let $f : A^\omega \to B^\omega$ be a causal stream function and $a \in A$. The *initial output of f (on input a)* is defined (for arbitrary $\alpha \in A^\omega$) by

$$f[a] := f(a{:}\alpha)(0).$$

The *stream function derivative of f (on input a)* is the function $f_a : A^\omega \to B^\omega$ defined by

$$f_a(\alpha) := f(a{:}\alpha)'.$$

We extend the above notions from letters to words over A, in the expected manner. Let $w \in A^*$, $a \in A$ and $f_\epsilon = f$, then

$$f[wa] := (f_w)[a], \quad \text{and} \quad f_{wa} := (f_w)_a.$$

Note that in Definition 2.2, $f[a]$ is well-defined because f is causal, and it is easy to show that the derivative of a causal stream function is again causal. When we speak of the *derivatives of a stream function* $f : A\omega \to B^\omega$, then we generally refer to the set $\{f_w \mid w \in A^*\}$.

The operations of initial output and stream function derivative are universal in the sense that they make the set of causal stream functions into a final Mealy coalgebra.

Theorem 2.3 *Let $\Gamma = \{f : A^\omega \to B^\omega \mid f \text{ is causal}\}$ and define $\pi : \Gamma \to (B \times \Gamma)^A$ by $\pi(f)(a) = (f[a], f_a)$. Then (Γ, π) is a final Mealy coalgebra: for every Mealy coalgebra (Q, ϕ), the behaviour map $Beh : Q \to \Gamma$ as defined in (2) is the unique Mealy homomorphism from (Q, ϕ) to (Γ, π).*

Proof. The proof is straightforward, but we include it for completeness' sake. Let (Γ, π) be as stated, and let (Q, ϕ) be an arbitrary Mealy coalgebra. We first verify that $Beh : Q \to \Gamma$ is a Mealy homomorphism. So let $q \in Q$ and $\alpha \in A^\omega$ be arbitrary. Then by definition

$$o_{Beh(q)}(a) = Beh(q)[a] = Beh(q)(a{:}\alpha)(0) = o_q(a),$$

and by letting $q_0 = d_q(a)$ and $q_{i+1} = d_q(\alpha(i))$ for all $i \geq 0$, we have

$$\begin{aligned} d_{Beh(q)}(a)(\alpha) &= Beh(q)_a(\alpha) = Beh(q)(a\!:\!\alpha)' \\ &= (o_q(a), o_{d_q(a)}(\alpha(0)), o_{q_1}(\alpha(1)), \ldots)' \\ &= (o_{d_q(a)}(\alpha(0)), o_{q_1}(\alpha(1)), \ldots) \\ &= Beh(d_q(a))(\alpha). \end{aligned}$$

To see that Beh is unique suppose $g : Q \to \Gamma$ is also a Mealy homomorphism. That is, for all $q \in Q$ and all $\alpha \in A^\omega$,

$$\begin{aligned} o_{g(q)}(a) &= \quad o_q(a) \quad = o_{Beh(q)}(a) \\ d_{g(q)}(a) &= g(d_q(a)) = d_{Beh(q)}(a). \end{aligned}$$

We show by coinduction (on streams in B^ω) that for all $q \in Q$ and all $\alpha \in A^\omega$, $g(q)(\alpha) = Beh(q)(\alpha)$. Let $R \subseteq B^\omega \times B^\omega$ be defined by

$$R := \{\langle g(q)(\alpha), Beh(q)(\alpha)\rangle \mid q \in Q, \alpha \in A^\omega\}.$$

We claim that R is a bisimulation. The initial values of $g(q)(\alpha)$ and $Beh(q)(\alpha)$ agree, since

$$g(q)(\alpha)(0) = o_{g(q)}(\alpha(0)) = o_{Beh(q)}(\alpha(0)) = Beh(q)(\alpha)(0).$$

Similarly, by the assumption that g and h are Mealy homomorphisms, we see that the derivatives are again R-related:

$$\begin{aligned} g(q)(\alpha)' \quad &= \quad g(q)_{\alpha(0)}(\alpha') = d_{g(q)}(\alpha(0))(\alpha') \\ &= \quad g(d_q(\alpha(0)))(\alpha') \\ &R \quad Beh(d_q(\alpha(0)))(\alpha') = d_{Beh(q)}(\alpha(0))(\alpha') \\ &= \quad Beh(q)(\alpha)'. \end{aligned}$$

QED

The final Mealy coalgebra (Γ, π) thus characterises all behaviours of Mealy coalgebras via the behaviour map Beh. We will need the following notions. Two states $q, r \in Q$ in some Mealy coalgebra (Q, ϕ) are called *(behaviourally) equivalent* if $Beh(q) = Beh(r)$. An initialised Mealy coalgebra (Q, ϕ, q) *implements*, or is an *(abstract) implementation* of, a causal stream function f if $Beh(q) = f$, and f is called a *realisable (or finite-state)* behaviour, if f can be implemented by a Mealy machine. Given a state q in a Mealy coalgebra (Q, ϕ), we denote by $\langle q\rangle$ the *Mealy subcoalgebra of* (Q, ϕ) *generated by* q. That is, $\langle q\rangle$ is the restriction of (Q, ϕ) to the least subset of Q which contains

q and is closed under the transition map ϕ. It should be clear that if (Q, ϕ, q) implements $f : A^\omega \to B^\omega$, then so does $\langle q \rangle$.

As it was shown in [16](Corollary 2.3), the existence of the final Mealy coalgebra now guarantees that any $f \in \Gamma$ has an (abstract) implementation, namely $\langle f \rangle$, and $\langle f \rangle$ is the minimal Mealy coalgebra to implement f, in the sense that any other implementation will have at least as many states as $\langle f \rangle$. In general $\langle f \rangle$ can have infinitely many states, but for synthesis purposes we are mainly interested in realisable behaviours.

Remark 2.4 The notion of stream function derivative was considered already in 1958 by Raney [11], but Raney simply referred to the derivatives of a stream function f as the *states of f*. This notion of derivative of stream functions was later adapted by Brzozowski [2] to become the better known derivative of regular languages. Other work by Raney [12,13] on formal power series, generating functions, and automata is closely related to the stream calculus described in Rutten [15].

3 Bitstream Algebras

We will now describe the two algebraic structures we use in specifying causal bitstream functions. The semantic domain of both algebras is the set of bitstreams 2^ω, i.e., streams over $2 = \{0, 1\}$, and the operations are defined using stream differential equations and Boolean operations on bits. The Boolean operations on 2 of $\vee$ (*or*), $\wedge$ (*and*), $\neg$ (*negation*) and $\oplus$ (*exclusive-or*) are defined as usual: for $a, b \in 2$, $a \vee b = \max\{a, b\}$, $a \wedge b = \min\{a, b\}$, $\neg a = 1 - a$ and $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$. In what follows, we will use the notation 2 to denote both the set $\{0, 1\}$ as well as the integer 2. The context should make clear which reading is intended.

The first bitstream algebra to be represented is based on the arithmetic operations on 2-adic numbers [5]. The motivation for studying this structure is its relevance for sequential binary arithmetic and digital circuits. Not much literature seems to be available on this subject, with the exception of the work by Vuillemin, see e.g. [19]. The other bitstream algebra is based on addition modulo-2, and it is also motivated by its connection to digital circuits, and switching theory [9], in particular, to the theory and design of linear circuits.

In section 4 we will describe the syntax in more detail, and explain how we determine equivalence of expressions. Before we describe the two bitstream algebras, we first introduce some notation and conventions. Syntactically, the expressions of each bitstream algebra are generated over a single bitstream variable σ by an arithmetic signature which contains binary function symbols for addition, multiplication and division, a unary function symbol for minus, and constants $[0]$, $[1]$, X^n for $n \in \mathbb{N}$. The constants are in both cases interpreted as the following bitstreams:

$$[0] = (0, 0, 0, \ldots), \quad [1] = (1, 0, 0, \ldots), \quad X^n = (\underbrace{0, \ldots, 0}_{n \text{ times}}, 1, 0, 0, \ldots).$$

Thus as is standard, X^0 and $[1]$ denote the same object. To save notation, we do not explicitly include the constants X^n, $n \in \mathbb{N}$, in the signature, but they are tacitly assumed to be part of it. We note that the notation X^n is not ambiguous in the presence of the two definitions of multiplication $\times$ (2-adic) and $\otimes$ (mod-2). After seeing the definitions of $\times$ and $\otimes$ in the subsections below, the reader can easily prove that for all $\alpha \in 2^\omega$, $X \times \alpha = X \otimes \alpha = 0 : \alpha$. Hence

$$\underbrace{X \times \ldots \times X}_{n \text{ times}} = \underbrace{X \otimes \ldots \otimes X}_{n \text{ times}} = X^n.$$

We will use a number of standard arithmetic conventions in our meta-notation. For the sake of illustration, suppose $\{+, -, \cdot, /, [0], [1]\}$ is an arithmetic signature. We will write X instead of X^1, and sometimes write $x - y$ instead of $x + (-y)$, and $\frac{x}{y}$ instead of x/y. The $\frac{x}{y}$-notation will be used for both 2-adic and mod-2 expressions, but only as meta-notation, and it will always be clear from the other operations whether the expression is a 2-adic fraction or a mod-2 fraction. Due to the fact that fractions, rather than inverses, are the basic form of expression manipulated by our algorithm, we have chosen to treat $/$ as a primitive constructor, rather than define x/y as shorthand for $x \cdot (1/y)$ (as in [16]) where $1/y$ is defined as the (multiplicative) inverse of y. The inverse $1/y$ can be defined as the fraction $[1]/y$, and when we consider the algebraic properties, of the two bitstream algebras, we implicitly use this definition of inverse.

Finally, brackets are used to disambiguate expressions in the text, but they are not part of the syntax, and in order to minimise the use of them we assume that the binding strength of the operations in descending order is $-, \cdot, /, +$, and that $+$ and $\cdot$ associate to the right. We also point out that we use σ strictly to denote a bitstream variable, i.e., a syntactic object, whereas α and β will be used as our meta-notation for bitstreams or expressions.

3.1 The 2-adic operations

The 2-adic bitstream algebra is the structure

$$\mathbb{A}_{2adic} = (2^\omega, +, -, \times, /, [0], [1])$$

where the operations are interpreted as addition, minus, multiplication and division of bitstreams viewed as *2-adic integers*. Briefly described, for any prime number p, the p-adic integers are obtained as power series of the form $\Sigma_{i=0}^{\infty} a_i p^i$, where $a_i \in \{0, \ldots, p-1\}$ for all $i \in \mathbb{N}$, and the limit of such a series is defined with respect to the p-adic norm. The p-adic integers $\mathbb{Z}_p$ form a subring of the field $\mathbb{Q}_p$ of p-adic numbers [5] in which the index of the power series may start at some negative integer rather than at 0. The usual integers $\mathbb{Z}$ are (strictly!) included in $\mathbb{Z}_p$ by writing a positive integer in its finite base p expansion; negative integers are represented by taking the infinitary version of p's complement of its positive counterpart. A bitstream $\alpha = (a_0, a_1, a_2, \ldots)$ thus represents the 2-adic integer $\Sigma_{i=0}^{\infty} a_i 2^i$, and in particular, positive integers are represented by bitstreams which contain only finitely many 1's, and negative integers are represented by bitstreams containing finitely many 0's.

The addition of 2-adic integers is an infinitary version of binary addition, that is, carry bits may be propagated indefinitely. For example, $(1,1,1,\ldots) + (1,0,0,\ldots) = (0,0,0,\ldots)$, which shows that $-[1] = (1,1,1,\ldots)$. 2-adic multiplication is the Cauchy product of 2-adic addition, i.e., for bitstreams α and β, $(\alpha \times \beta)(n) = \Sigma_{i=0}^{n} \alpha(i) \wedge \beta(n-i)$, where Σ denotes 2-adic summation, and it may be computed in the add-and-shift manner, analogously to how one multiplies the usual integers. The 2-adic integers form a commutative ring and integral domain, as there are no zero divisors, i.e., $\alpha \times \beta = [0]$ implies $\alpha = [0]$ or $\beta = [0]$. This means that the arithmetic operations have the familiar properties of associativity, commutativity and distributivity, and we will use these properties freely in what follows. However $\mathbb{Z}_2$ is not a field, since 2-adic integers with initial value $a_0 = 0$ do not have a (multiplicative) inverse. This is clear since $\alpha \times \beta = [1]$ implies $\alpha(0) = \beta(0) = 1$, and in fact, it suffices that a_0 has an inverse in the underlying structure, which in the 2-adic case means that $a_0 = 1/a_0 = 1$. The 2-adic operations are defined on bitstreams by the stream differential equations in Figure 1.

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

Fig. 1. 2-adic operations

The stream differential equations for $+$ and $\times$ should be easy to understand from the above description of the 2-adic integers. The defining equation for $-$ is simply derived from that of $+$ and the requirement that $\alpha + (-\alpha) = [0]$ by taking initial value and derivative on both sides. For the initial value we get that $\alpha(0) \oplus (-\alpha)(0) = 0$, hence $(-\alpha)(0) = \alpha(0)$. By taking derivatives we get $\alpha' + (-\alpha)' + [\alpha(0)] = [0]$, and hence $(-\alpha)' = -([\alpha(0)] + \alpha')$. The stream differential equations for α/β can be derived similarly.

In section 2.1 we have already seen how members of the set $\hat{Q}$ of rationals with odd denominator can be viewed as bitstreams, and in fact, the final stream map $B : \hat{Q} \to \mathbb{A}_{2adic}$ is a homomorphism of integral domains, thus the 2-adic operations applied to bitstreams that represent integers and rationals with odd denominator correspond with the usual arithmetic operations. In particular, since the constant stream $X = (0, 1, 0, 0, \ldots)$ represents the base 2, we have the identity of bitstreams $\alpha + \alpha = X \times \alpha$.

3.2 Rational 2-adic stream functions

In this section we will use the correspondence induced by the homomorphism $B : \hat{Q} \to \mathbb{A}_{2adic}$ to provide a numeric interpretation of taking derivatives of rational 2-adic stream functions (to be defined shortly), which in turn leads to a result on the size of their minimal implementing automata (Corollary 3.4), and an alternative proof of the fact that rational 2-adic stream functions have finite implementations (cf. [16]).

Since we are interested in the numeric semantics of bitstreams and 2-adic operations, we will identify q with $B(q)$ for $q \in \hat{Q}$, and simply use integers and rationals in our meta-notation.

Definition 3.1 A bitstream function $f : 2^\omega \to 2^\omega$ is a *rational 2-adic stream function*, if f is of the form:

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

where m and n are integers, n is odd, and σ is a stream variable.

From the definition of the 2-adic operations, it should be clear that rational 2-adic stream functions are causal. As an example, the rational 2-adic stream function $f(\sigma) = \frac{6}{-9} \times \sigma$ can be expressed in 2-adic notation by $f(\sigma) = \frac{X+X^2}{-[1]-X^3} \times \sigma$. Moreover, we note that $f(\sigma)$ is equivalent with $\frac{-2}{3} \times \sigma$, but not with $\frac{-12}{18} \times \sigma$, since only fractions with odd denominator are well-defined.

Lemma 3.2 (Numeric 2-adic derivatives) *Let $f : 2^\omega \to 2^\omega$ be a 2-adic stream function of the form*

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

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

$$f_a(\sigma) = \frac{d_a + m \times \sigma}{n}$$

where (in the numeric interpretation)

$$d_0 = \begin{cases} \frac{1}{2} d & \text{if } d \text{ even} \\ \frac{1}{2}(d - n) & \text{if } d \text{ odd} \end{cases} \qquad d_1 = \begin{cases} \frac{1}{2}(d + m) & \text{if } d(0) = m(0) \\ \frac{1}{2}(d + m - n) & \text{if } d(0) \neq m(0) \end{cases}$$

Proof. Applying the inductive definitions in Figure 1 on page 9 to determine $f(0{:}\sigma)'$ and $f(1{:}\sigma)'$, we get

$$\begin{aligned} d_0 &= d' - [d(0)] \times n' \text{ and} \\ d_1 &= d' + m' + [d(0) \wedge m(0)] - [d(0) \oplus m(0)] \times n'. \end{aligned}$$

The rest of the proof is now straightforward using (1), and we only give the details of some cases. In the case d is odd, we get for d_0:

$$d_0 = d' - n' = \frac{1}{2}(d-1) - \frac{1}{2}(n-1) = \frac{1}{2}(d-n).$$

When d and m are both odd, i.e. $d(0) = m(0) = 1$, we have

$$d_1 = d' + m' + 1 = \frac{1}{2}(d-1) + \frac{1}{2}(m-1) + 1 = \frac{1}{2}(d+m).$$

Finally, if d is odd, and m is even, then

$$d_1 = d' + m' - n' = \frac{1}{2}(d-1) + \frac{1}{2}m - \frac{1}{2}(n-1) = \frac{1}{2}(d+m-n).$$

QED

Using the numeric interpretation of taking derivatives we can prove that rational 2-adic functions have a finite number of derivatives.

Theorem 3.3 (Derivatives of rational functions) *Let $f(\sigma) = \frac{m}{n} \times \sigma$ be a rational 2-adic stream function where $m \neq 0$ and $n > 0$ is odd, then for all stream function derivatives f_w, $w \in 2^*$, of f, f_w is of the form*

$$f_w(\sigma) = \frac{d_w + m \times \sigma}{n} \tag{3}$$

where d_w is an integer such that

$$\begin{array}{llllll} 1) & -n+1 & \leq & d_w & \leq & m-1 & \textit{if } m > 0, \\ 2) & -n+m+1 & \leq & d_w & \leq & 0 & \textit{if } m < 0. \end{array}$$

Proof. It is a direct consequence of Lemma 3.2 that the derivatives of f have the format (3), since f is itself of the form required in Lemma 3.2 (take $d_\epsilon = 0$), and hence so are all derivatives of f. We prove by induction on the length of $w \in 2^*$ that the numeric value d_w is in the given range. We distinguish between the cases $m > 0$ and $m < 0$. The base case ($w = \epsilon$) is clear To prove the inductive step we use the numeric interpretation of derivatives of rational 2-adic functions given in Lemma 3.2.

Case $m > 0$: Assume that 1) holds for d_w and consider d_{w0}. Suppose first that d_w is even. Then $d_{w0} = \frac{1}{2}d_w$, and in the case $d_w > 0$, we have by induction hypothesis and $n > 0$ that $-n+1 \leq 0 \leq \frac{1}{2}d_w \leq d_w \leq m-1$. If $d_w \leq 0$, then we have $-n+1 \leq d_w \leq \frac{1}{2}d_w \leq 0 \leq m-1$. Now if d_w is odd, then $d_{w0} = \frac{1}{2}(d_w - n)$. For the upper bound, we note that if $d_w < n$ then $\frac{1}{2}(d_w - n) < 0 \leq m-1$. If $d_w \geq n$ then $\frac{1}{2}(d_w - n) \leq d_w - n$, and from the induction hypothesis and $n > 0$, it follows that $d_w - n \leq d_w \leq m-1$, and hence that $\frac{1}{2}(d_w - n) \leq m-1$. To see that $d_{w0} \geq -n+1$, we have by induction hypothesis that $-n+1-n = -2n+1 \leq d_w - n$, and since $d_w - n$ is even it follows that $-2(n-1) \leq d_w - n$, and hence $-n+1 \leq \frac{1}{2}(d_w - n) = d_{w0}$.

We now show that d_{w1} is in the given range. In the case that d_w and m have the same parity, i.e., $d_w(0) = m(0)$, we have $d_{w1} = \frac{1}{2}(d_w + m)$. By induction hypothesis, $d_w + m \leq (m-1) + m = 2m-1$, and since $d_w + m$ is even, and $m > 0$, it follows that $d_w + m \leq 2(m-1)$ and hence $\frac{1}{2}(d_w + m) \leq m-1$. For the lower bound, we distinguish between $d_w \geq 0$ and $d_w < 0$. If $d_w \geq 0$ then $-n+1 \leq 0 \leq \frac{1}{2}(d_w + m)$, since $m > 0$. If $d_w < 0$ then by induction hypothesis $-n+1 \leq d_w < \frac{1}{2}(d_w + m)$. Now suppose $d_w(0) \neq m(0)$, then $d_{w1} = \frac{1}{2}(d_w + m - n)$. From the induction hypothesis it follows that,

$$-n+1+m-n \leq d_w + m - n \leq (m-1) + m - n,$$

and since $m > 0$ and $n \geq 1$

$$-2n+1 \leq d_w + m - n \leq 2(m-1).$$

As $d_w + m - n$ is even, we obtain

$$-2(n-1) \leq d_w + m - n \leq 2(m-1).$$

and hence

$$-n+1 \leq \frac{1}{2}(d_w + m - n) \leq (m-1).$$

Case $m < 0$: Assume now that 2) holds for d_w. It is easy to see that $d_w \leq 0$, $m < 0$ and $n > 0$ imply that both $d_{w0} \leq 0$ and $d_{w1} \leq 0$. It remains to show that $-n+m+1 \leq d_{w0}, d_{w1}$. For d_{w0} in the case d_w is even, it follows immediately from $d_w \leq 0$ that $d_w \leq \frac{1}{2}d_w = d_{w0}$, and hence by induction hypothesis that $-n+m+1 \leq d_{w0}$. If d_w is odd, then $d_{w0} = \frac{1}{2}(d_w - n)$. Now if $d_w \leq -n$, we have $2d_w \leq d_w - n$ and hence $-n+m+1 \leq d_w \leq \frac{1}{2}(d_w - n)$. If on the other hand $-n < d_w$, then $-2n < d_w - n$ and $-n < \frac{1}{2}(d_w - n)$. Since $m \leq -1$, we get $-n+m+1 \leq -n$, and hence $-n+m+1 < \frac{1}{2}(d_w - n)$.

Now consider d_{w1}. If $d_w(0) = m(0)$, then $d_{w1} = \frac{1}{2}(d_w + m)$. With a similar reasoning as above we find that $d_w \leq m$ implies $d_w \leq \frac{1}{2}(d_w + m)$, whence by induction hypothesis $-n+m+1 < \frac{1}{2}(d_w - m)$; and $d_w > m$ implies $m < \frac{1}{2}(d_w + m)$, which together with $-n+m+1 \leq m$ yields the desired lower bound. If $d(0) \neq m(0)$, then $d_{w1} = \frac{1}{2}(d_w + m - n)$. From the induction hypothesis $(-n+m+1 \leq d_w)$ we obtain,

$$d_w + m - n \geq -2n + 2m + 1 = 2(m-n) + 1,$$

and since $d_w + m - n$ is even and negative,

$$d_w + m - n \geq 2(m-n) + 2,$$

and hence

$$\frac{1}{2}(d_w + m - n) \geq -n + m + 1.$$

QED

The derivatives of a rational 2-adic function $f(\sigma) = \frac{m}{n} \times \sigma$ are thus characterised by the different values of d_w, which implies that from the above theorem, we can deduce an upper bound on the number of states in $\langle f \rangle$ for rational 2-adic f.

Corollary 3.4 (Automaton size) *Let $\langle \frac{m}{n} \times \sigma \rangle$ be the minimal Mealy coalgebra implementing the rational 2-adic function $f(\sigma) = \frac{m}{n} \times \sigma$, where m and n are integers such that $m \neq 0$ and n is odd, and let $NumStates(\langle \frac{m}{n} \times \sigma \rangle)$ denote the number of states in $\langle \frac{m}{n} \times \sigma \rangle$. Then*

$$NumStates(\langle \frac{m}{n} \times \sigma \rangle) \leq \begin{cases} \frac{|m|+|n|}{\gcd(m,n)} - 1 & \text{\it if } \frac{m}{n} > 0 \\ \frac{|m|+|n|}{\gcd(m,n)} & \text{\it if } \frac{m}{n} < 0 \end{cases}$$

Proof. Let m and n be as stated, and let $\tilde{m} := sgn(\frac{m}{n}) \cdot |m| / \gcd(m,n)$ and $\tilde{n} := |n| / \gcd(m,n)$, then numerically $\frac{\tilde{m}}{\tilde{n}} = \frac{m}{n}$, and $\langle \frac{\tilde{m}}{\tilde{n}} \times \sigma \rangle$ is isomorphic with $\langle \frac{m}{n} \times \sigma \rangle$. Hence $NumStates(\langle \frac{\tilde{m}}{\tilde{n}} \times \sigma \rangle) = NumStates(\langle \frac{m}{n} \times \sigma \rangle)$. As $\tilde{m}$ and $\tilde{n}$ satisfy the condition of Theorem 3.3 it follows that $NumStates(\langle \frac{\tilde{m}}{\tilde{n}} \times \sigma \rangle)$ equals the number of distinct d_w-values ocurring in the stream function derivatives of f. Hence if $\tilde{m} > 0$, i.e., $sgn(\frac{m}{n}) = 1$, then

$$NumStates(\langle \frac{\tilde{m}}{\tilde{n}} \times \sigma \rangle) \leq (\tilde{m}-1) - (-(\tilde{n}-1)) + 1 = \tilde{m} + \tilde{n} - 1 = \frac{|m| + |n|}{\gcd(m,n)} - 1,$$

and if $\tilde{m} < 0$, i.e., $sgn(\frac{m}{n}) = -1$, then

$$NumStates(\langle \frac{\tilde{m}}{\tilde{n}} \times \sigma \rangle) \leq -(-\tilde{n} + \tilde{m} + 1) + 1 = \tilde{n} - \tilde{m} = |\tilde{n}| + |\tilde{m}| = \frac{|m| + |n|}{\gcd(m,n)}.$$

QED

Experimental results strongly indicate that the inequality in Corollary 3.4 is in fact an equality. I.e., the number of states in $\langle \frac{m}{n} \times \sigma \rangle$ is exactly given by the above formula. However, at present we have no formal proof of this.

3.3 The modulo-2 operations

The mod-2 bitstream algebra

$$\mathbb{A}_{mod2} = (2^\omega, \oplus, \ominus, \otimes, \oslash, [0], [1])$$

is based on *addition modulo 2*, in other words taking the elementwise exclusive-or of bitstreams. In the text, we will use the symbol $\oplus$ to denote the modulo-2 addition on bitstreams as well as on bits. The typing should be clear from the context.

The mod-2 bitstream operations have no numeric interpretation; rather, they correspond to the operations on bitstreams seen as formal power series over the $\mathbb{M}$od2 ring (and integral domain) $(2, \oplus, \wedge, id, 0, 1)$. Note that addition modulo 2 is nilpotent, i.e., for any $a \in 2$, $a \oplus a = 0$ and hence $-a = a = id(a)$. A bitstream $\alpha = (a_0, a_1, a_2 \ldots)$ is now interpreted as the coefficients of the formal power series $a_0 + a_1 x + a_2 x^2 + \ldots$. From the theory of formal power series, it follows that $\mathbb{A}_{mod2}$ is also a commutative ring and integral domain in which $\oplus$ is nilpotent, and $\ominus$ is the identity. In particular, multiplication $\otimes$ is the Cauchy product with respect to $\oplus$; division $\oslash$ is defined to be an inverse to $\otimes$, and we again require that $\beta(0) = 1$ for fractions $\alpha \oslash \beta$. The modulo-2 operations are defined by the stream differential equations in Figure 2.

derivative	initial value	condition
$(\alpha \oplus \beta)' = \alpha' \oplus \beta'$	$(\alpha \oplus \beta)(0) = \alpha(0) \oplus \beta(0)$	
$(\ominus\alpha)' = \ominus(\alpha')$	$(\ominus\alpha)(0) = \alpha(0)$	
$(\alpha \otimes \beta)' = (\alpha' \otimes \beta) \oplus [\alpha(0)] \otimes \beta'$	$(\alpha \otimes \beta)(0) = \alpha(0) \wedge \beta(0)$	
$(\alpha \oslash \beta)' = (\alpha' \oplus [\alpha(0)] \otimes \beta') \oslash \beta$	$(\alpha \oslash \beta)(0) = \alpha(0)$	$\beta(0) = 1$

Fig. 2. mod-2 operations

We mention that rational mod-2 functions can be defined analogously to rational 2-adic functions, and one can show that rational mod-2 functions are causal and realisable.

4 Syntax

4.1 Derivatives of expressions

In section 3 we presented the semantics of the two bitstream algebras we use to specify causal bitstream functions. We now turn to their syntax, and describe in detail how we compute initial value and stream function derivatives symbolically using the expressions generated by the respective signatures. The idea is to provide the term algebras with Mealy structure; the Mealy behaviour of an expression θ is then obtained via the behaviour map into the final Mealy coalgebra, and we will say that θ is a *specification* of the causal function $Beh(\theta)$.

This construction applies to both bitstream algebras of section 3, so we will illustrate using a generic integral domain signature $\Sigma = \{+, \cdot, -, /\}$. Let

$Term_\Sigma(\sigma)$ be the expressions generated by Σ over the variable σ. Recall Definition 2.2 of the initial output and stream function derivative of a causal function $f(\sigma)$ with respect to a bit $a \in 2$:

$$f[a] = f(a\!:\!\sigma)(0) \quad \text{and} \quad f_a = f(a\!:\!\sigma)'$$

We wish to mimic this semantic definition in the syntax, meaning that given an expression θ which specifies a causal function f_θ, and a bit $a \in 2$, we are looking for a systematic procedure to obtain a bit $\theta[a]$ and an expression θ_a such that $\theta[a] = f_\theta[a]$ and $Beh(\theta_a) = (f_\theta)_a$. We see that the definition of initial value and derivative consists of two parts. The first being the *instantiation of the bitstream variable σ with the bit a*, which turns σ into $a\!:\!\sigma$. The second is the taking of initial value and stream derivative of $f(a\!:\!\sigma)$.

The syntactic equivalent of instantiating the bitstream variable σ, is based on the observation that for any bitstream $\alpha \in 2^\omega$:

$$\begin{aligned} 0\!:\!\alpha &= \phantom{[1] + {}} X \times \alpha = \phantom{[1] \oplus {}} X \otimes \alpha \\ 1\!:\!\alpha &= [1] + X \times \alpha = [1] \oplus X \otimes \alpha \end{aligned}$$

Thus if $\theta \in Term_\Sigma(\sigma)$, then we denote by $\theta(0\!:\!\sigma)$ the expression obtained from θ by substituting all occurrences of σ with $X \cdot \sigma$; similarly $\theta(1\!:\!\sigma) \in Term_\Sigma(\sigma)$ is obtained by substituting $[1] + X \cdot \sigma$ for σ in θ. The expressions $\theta(0\!:\!\sigma)$ and $\theta(1\!:\!\sigma)$ are called *instantiated expressions*. It remains to define the initial value and stream derivative of instantiated expressions. The initial value should be a bit, and the derivative should be an expression in $Term_\Sigma(\sigma)$. We define the stream behaviour of instantiated expressions inductively. For the constants this is clearly:

initial value	derivative
$X^n(0) = 0$	$(X^n)' = X^{n-1}$, $n \geq 1$
$X^0(0) = [1](0) = 1$	$[1]' = [0]$
$[0](0) = 0$	$[0]' = [0]$

For the variable σ, we are not able to determine $\sigma(0)$ and σ' due to σ itself being indeterminate. But this is not a problem, since we only need to consider instantiated expressions, and we observe that in any instantiated expression σ always occurs as part of an expression $X \cdot \sigma$, which represents the stream $0\!:\!\sigma$. We therefore define

$$(X \cdot \sigma)(0) = 0 \quad \text{and} \quad (X \cdot \sigma)' = \sigma.$$

The stream behaviour of non-atomic instantiated expressions different from $X \cdot \sigma$ is obtained by taking the stream differential equations in Figures 1 and 2 as inductive definitions over 2-adic, respectively mod-2, expressions. This means, however, that (instantiated) expressions of the form α/β in which $\beta(0) = 0$ have no stream behaviour, and hence that not all expressions have Mealy behaviour. This is, for instance, the case with the 2-adic expression $\theta = \frac{[1]}{[1]+\sigma}$, since $\theta(1:\sigma) = \frac{[1]}{[1]+([1]+X\times\sigma)}$ which has a denominator with initial value 0, and so $\theta[1]$ and θ_1 are undefined.

To sum up, the initial output and derivative of an expression θ in $\mathit{Term}_\Sigma(\sigma)$ on input $a \in 2$, if well-defined, are given by:

$$\theta[a] := \theta(a\!:\!\sigma)(0) \quad \text{and} \quad \theta_a(\sigma) := \theta(a\!:\!\sigma)'. \tag{4}$$

Note that if σ does not occur in θ, then $\theta(a\!:\!\sigma) = \theta$ for $a \in 2$. So, as is usual, constant expressions can be interpreted as 0-ary functions, and hence they can have both stream behaviour and Mealy behaviour, whereas expressions that contain σ can only have Mealy behaviour. Definition (4) can be extended from bits to bitwords $w \in 2^*$ in the same manner as in Definition 2.2, and when we speak of *derivative expressions* then we generally mean all expressions θ_w, $w \in 2^*$, for some given specification θ.

Example 4.1 Consider the 2-adic expression $\theta = \frac{X^2\times\sigma}{[1]+X}$. The instantiation of θ with the bit 1, is

$$\theta(1\!:\!\sigma) = \frac{X^2 \times ([1] + X \times \sigma)}{[1] + X}.$$

The intial output on input 1, is $\theta[1] = X^2(0) \wedge ([1] + X \times \sigma)(0) = 0 \wedge 1 = 0$, and the derivative θ_1 is

$$\begin{aligned}\theta(1\!:\!\sigma)' &= \left(\frac{X^2\times([1]+X\times\sigma)}{[1]+X}\right)' = \frac{(X^2\times([1]+X\times\sigma))'-[0]\times([1]+X)'}{[1]+X}\\ &= \frac{((X^2)'\times([1]+X\times\sigma)+[0]\times([1]+X\times\sigma)')-[0]\times([1]+X)'}{[1]+X}\\ &= \frac{(X\times([1]+X\times\sigma)+[0]\times([0]+\sigma+[0]))-[0]\times([0]+[1]+[0])}{[1]+X}\end{aligned}$$

It is clear that this expression can be simplified using the identities of the 2-adic bitstream algebra to yield the equivalent expression $\frac{X+X^2\times\sigma}{[1]+X}$. The computation of this reduced form for arbitrary expressions from $\mathit{Term}_\Sigma(\sigma)$ is explained in the next section.

Apart from the "failure" which can arise when an expression has no Mealy behaviour, expressions can give rise to infinite Mealy behaviour, i.e., causal bitstream functions which have no finite-state implementation. Such an example is given by the 2-adic expression $\sigma \times \sigma$. It is easy to show by induction on $n \in \mathbb{N}$ that $(\sigma \times \sigma)_{0^n}$ is equivalent to $X^n \times (\sigma \times \sigma)$. For different n, these expressions are clearly not equivalent, and hence $\langle \mathit{Beh}(\theta)\rangle$ has infinitely many states. Similarly, one can show that $\frac{[1]}{[1]+X\times\sigma}$ has only infinite Mealy implementations. However, it is easy to show that specifications of the form

$\frac{p+q\cdot\sigma}{r}$, for constant polynomial expressions p, q and r, have finite-state Mealy behaviour, and the above observations suggest that this is the most general form of realisable specifications.

4.2 Deciding equivalence of expressions

A crucial property of the two bitstream algebras is that we can effectively decide whether two derivative expressions are equivalent. This decision method relies on the fact that the two bitstream algebras are integral domains. Given two arbitrary expressions θ and η, we first compute what we call their normal forms. The *normal form of an expression* θ is a fraction N_θ/D_θ of two polynomial expressions in distributive normal form, i.e., N_θ and D_θ are sums of monomials of the form $C_n\cdot\sigma^n$, where C_n is a constant (polynomial) coefficient. We will denote the distributive normal form of a polynomial expression p by $\mathrm{PNF}(p)$. In the case θ is already a fraction $\theta = \rho/\delta$ of polynomial expressions, then the normal form of θ is $\mathrm{PNF}(\rho)/\mathrm{PNF}(\delta)$. Two expressions θ and η are then equivalent if $\mathrm{PNF}(N_\theta \cdot D_\eta) \equiv \mathrm{PNF}(N_\eta \cdot D_\theta)$, where $\equiv$ denotes syntactic equality.

The normal form N_θ/D_θ is essentially a normal form with respect to certain identities of the two bitstream algebras. A part of these identities are common to all integral domains, but the reduction of constant polynomial expressions (i.e, the coefficients C_n in the polynomial normal form) is specific to each bitstream algebra:

Reducing 2-adic coefficients: Due to the numeric interpretation of the 2-adic constants and operations, we can interpret a constant polynomial 2-adic expression z (i.e. z is built without variables or divison /) as an integer $\mathrm{Val}(z)$. For example, $\mathrm{Val}(-X^2 + ([1] + X) \times X^2) = -4 + (1 + 2)4 = 8$. Any x in $\mathbb{N}$ can then be written as its symbolic binary expansion $\mathrm{BinExp}(x)$, where the constant X represents the base 2. E.g., $\mathrm{BinExp}(5)$ is the expression $[1] + X^2$. The normal form of a constant polynomial 2-adic expression z is defined as $\mathrm{BinExp}(\mathrm{Val}(z))$ if $\mathrm{Val}(z) \geq 0$, and $-\mathrm{BinExp}(-\mathrm{Val}(z))$ otherwise. For example, the normal form of $[1] - X^3$ is $-([1] + X + X^2)$.

Reducing mod-2 coefficients: Any constant polynomial mod-2 expression can be rewritten to a sum of signed powers of the variable X (by applying distrbutivity and other ring laws). Due to the nilpotency of $\oplus$ it is relatively easy to see that such a sum can be reduced to a normal form by applying the identities $\alpha \oplus \alpha = [0]$ and $\ominus\alpha = \alpha$. This normal form consists of a sum of unique powers of X^n ordered ascendingly on n. For example, the sum $X^2 \oplus X^1 \oplus X^0 \oplus \ominus X^3 \oplus X^2$, has the normal form $[1] \oplus X^1 \oplus X^3$.

Example 4.2 Consider the expression $\theta = \frac{[1]}{[1]+X} + \sigma + [1]$. Using the laws of integral domains, θ is rewritten to $\frac{([1]+[1]+X)+([1]+X)\cdot\sigma}{[1]+X}$. After reducing the coefficients, we obtain:

$$\text{normal form in 2-adic algebra: } \frac{X^2+([1]+X)\times\sigma}{[1]+X}$$
$$\text{normal form in mod-2 algebra: } \frac{X\oplus([1]\oplus X)\otimes\sigma}{[1]+X}$$

For an expression θ, we define the *size (or length) of* θ, denoted by $len(\theta)$, as the number of symbol occurrences in θ. The time complexity of computing PNF(p) of a polynomial expression p is in the worst case exponential in $len(p)$ due to the duplication of subexpressions when applying the distributive law (e.g. $a \cdot (b + c) = a \cdot b + a \cdot c$). That is, $T(\text{PNF}(p)) = 2^{O(len(\theta))}$. Consequently, the time complexity of computing the normal form of an arbitrary expression θ is also $2^{O(len(\theta))}$, and checking equivalence of two normalised expressions also carries an exponential cost: Checking equivalence of $\theta = N_\theta/D_\theta$ and $\eta = N_\eta/D_\eta$ has a worst case time complexity of $2^{O(len(\theta)+len(\eta))}$.

Although efficiency has not been the main concern in our implementation, we note that the equivalence check can be optimised when the initial specification θ has a normal form in which the denominator D_θ is constant. This applies, in particular, to rational functions. Recall the definitions of the derivative of a fraction in the two bitstream algebras:

$$\text{2-adic:} \quad (\alpha/\beta)' = (\alpha' - [\alpha(0)] \times \beta')/\beta$$
$$\text{mod-2:} \quad (\alpha \oslash \beta)' = (\alpha' \oplus [\alpha(0)] \otimes \beta') \oslash \beta$$

From these definitions, and the fact that for constant D_θ, we have $D_\theta(a : \sigma) = D_\theta$ for any $a \in 2$, we see that all derivatives of $\theta = N_\theta/D_\theta$ will also have denominator D_θ. Hence in order to decide whether two derivatives of θ, say δ and η, are equivalent, it suffices to check syntactic equality of their normal forms: $N_\delta \equiv N_\eta$ and $D_\delta \equiv D_\eta$ This can be done in linear time: $O(len(\delta) + len(\eta))$. In fact, in the special case of rational functions it suffices to check syntactic equality of the normal form numerators, but this equivalence test would not be sound, in general, since then, e.g., $[1]/([1] + X)$ and $[1]/[1]$ would be considered equivalent. The complexity of this numerator-only check is still linear in the input, and we have therefore decided to use the (sound) equality of the entire normal form.

5 The Construction

5.1 Algorithm

Our method for constructing a Mealy machine from a given bitstream specification θ can be seen as a generalisation of Brzozowski's [2] method for constructing deterministic finite automata from regular expressions. Starting from the specification θ, we compute for each bit $a \in 2$, the transitions corresponding with input a, and iterate this for the derivatives of θ until no new transitions are found, i.e., a fixpoint has been reached. We represent a (partially constructed) Mealy machine as a list of labelled transitions, and the fixpoint computation is initiated with the list containing the transitions from the initial specification θ to its immediate derivatives θ_0 and θ_1. We also keep track of the current states/derivatives in a list S, which is initialised to contain (the normal form of) θ. In each iteration, we check for new states, that is, destinations of the new transitions which are not already contained in the state list. If new states are found, we compute the transitions starting from each of these new states. These transitions are necessarily new to the transition list. We then add the new states to the state list, and add the new transitions to the transition list, and continue with the next iteration. The construction is perhaps best illustrated by means of an example.

Example 5.1 Consider the 2-adic specification $\sigma/(X^2 - [1])$. The numeric interpretation of this specification is $\sigma/3$, and hence its normal form is $\theta = \sigma/([1] + X)$. We now compute the normalised derivatives of θ:

$$\begin{aligned}
\theta_0 &= (\tfrac{X\times\sigma}{[1]+X})' &&= \tfrac{\sigma-[0]\times([1]+X)'}{[1]+X} &&= \tfrac{\sigma}{[1]+X} = \theta\\
\theta_1 &= (\tfrac{[1]+X\times\sigma}{[1]+X})' &&= \tfrac{\sigma-[1]\times([1]+X)'}{[1]+X} &&= \tfrac{-[1]+\sigma}{[1]+X}
\end{aligned}$$

The initial outputs of θ are easily computed: $\theta[0] = 0$ and $\theta[1] = 1$. So the fixpoint computation is initiated with state list S_1, and transition list L_1:

$$\begin{aligned}
S_1 &= [\tfrac{\sigma}{[1]+X}],\\
L_1 &= [\langle\tfrac{\sigma}{[1]+X},\ 0|0,\ \tfrac{\sigma}{[1]+X}\rangle;\ \langle\tfrac{\sigma}{[1]+X},\ 1|1,\ \tfrac{-[1]+\sigma}{[1]+X}\rangle].
\end{aligned}$$

L_1 contains (a representation of) the paths of length 1 in the Mealy implementation of θ. In the first iteration, the paths of length 2 are computed by computing the derivatives of the new states. We find that θ_1 is the only new state. The initial outputs of θ_1 are $\theta_1[0] = 1$ and $\theta_1[1] = 0$, and the derivatives:

$$\begin{aligned}
\theta_{10} &= (\tfrac{-[1]+X\times\sigma}{[1]+X})' &&= \tfrac{((-[1])'+\sigma+[0])-[1]\times([1]+X)'}{[1]+X} &&= \tfrac{-X+\sigma}{[1]+X}\\
\theta_{11} &= (\tfrac{-[1]+([1]+X\times\sigma)}{[1]+X})' &&= \tfrac{((-[1])'+\sigma+[1])-[0]\times([1]+X)'}{[1]+X} &&= \tfrac{\sigma}{[1]+X} = \theta.
\end{aligned}$$

We now update our lists by adding θ_1 to S_1, and the new transitions to L_1, and we obtain:

$$\begin{aligned}
S_2 &= [\tfrac{\sigma}{[1]+X},\ \tfrac{-[1]+\sigma}{[1]+X}],\\
L_2 &= L_1 \mathbin{++} [\langle\tfrac{-[1]+\sigma}{[1]+X},\ 0|1,\ \tfrac{-X+\sigma}{[1]+X}\rangle;\ \langle\tfrac{-[1]+\sigma}{[1]+X},\ 1|0,\ \tfrac{\sigma}{[1]+X}\rangle]
\end{aligned}$$

In the next iteration, we compute the transitions from the new state $\theta_{10} =$

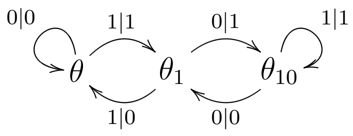
$\frac{-X+\sigma}{[1]+X}$. The initial outputs are $\theta_{10}[0] = 0$ and $\theta_{10}[1] = 1$, and the derivatives:

$$\begin{aligned}
\theta_{100} &= \left(\tfrac{-X+X\times\sigma}{[1]+X}\right)' &&= \tfrac{((-X)'+\sigma+[0])-[0]\times([1]+X)'}{[1]+X} &&= \tfrac{-[1]+\sigma}{[1]+X} = \theta_1\\
\theta_{101} &= \left(\tfrac{-X+([1]+X\times\sigma)}{[1]+X}\right)' &&= \tfrac{((-X)'+\sigma+[0])-[1]\times([1]+X)'}{[1]+X} &&= \tfrac{-X+\sigma}{[1]+X} = \theta_{10}.
\end{aligned}$$

We now add θ_{10} to S_2, and add the new transitions to L_2 to obtain the list containing paths of length 3:

$$\begin{aligned}
S_3 &= [\tfrac{\sigma}{[1]+X}, \tfrac{-[1]+\sigma}{[1]+X}, \tfrac{-X+\sigma}{[1]+X}],\\
L_3 &= L_2 \mathbin{++} [\langle \tfrac{-X+\sigma}{[1]+X}, 0|0, \tfrac{-[1]+\sigma}{[1]+X}\rangle; \langle \tfrac{-X+\sigma}{[1]+X}, 1|1, \tfrac{-X+\sigma}{[1]+X}\rangle]
\end{aligned}$$

In the next round of the fixpoint computation we find that there were no new states, hence no new transitions will be found, and the list L_3 is returned. The Mealy machine represented by L_3 has the following transition diagram:


$$\begin{tikzcd}
\theta \arrow[loop left, "0|0"] \arrow[r, bend left, "1|1"] & \theta_1 \arrow[l, bend left, "1|0"] \arrow[r, bend left, "0|1"] & \theta_{10} \arrow[l, bend left, "0|0"] \arrow[loop right, "1|1"]
\end{tikzcd}$$

We note that if θ has no Mealy behaviour then the computation will get stuck at some point, and if θ has only infinite-state implementations, then the process will not terminate. In order to deal with the latter problem, we provide the possibility of pre-specifying the maximum path length (automaton depth) in our program, see section 6.

5.2 Complexity

The time complexity of the construction can be expressed in the following quantities: M, the number of states in the constructed Mealy machine $\mathcal{M}$; R, the time cost of computing and reducing derivatives to normal form; and E, the time cost of determining equivalence of two derivative expressions.

During the fixpoint computation, for every state s in $\mathcal{M}$, we compute and reduce the two derivatives s_0 and s_1 exactly once. This yields a factor $M2R$. Furthermore, in each iteration round we remove duplicates (more precisely, equivalents) from a list of potentially new derivatives (the destinations of new transitions). The length of this list is bounded by $\log(M)$, each equivalence test between elements of this list can be done in time E, and the duplicates can be removed in time $E(\log(M))^2$. Finally, from the list of duplicate-free, potentially new derivatives, we remove the ones that are already in the list of current states. The list of current states has at most length M, hence this can be done in time $EM\log(M)$. Summing up, we obtain an overall complexiy of $O(M2R + EM(\log(M))^2 + EM^2\log(M)) = O(MR + EM^2\log(M))$.

For rational 2-adic functions, we can express M, R and E in terms of the size of the reduced input expression $\theta = \frac{p\times\sigma}{q}$ using the numeric interpretation, and we can therefore describe the time complexity of our synthesis algorithm for rational 2-adic functions in terms of the size of *theta*.

Proposition 5.2 *Let $\theta = \frac{p \times \sigma}{q}$ be a rational 2-adic function specification in normal form. A Mealy machine implementation of $Beh(\theta)$ can be constructed in time $2^{O(len(\theta))}$ using the algorithm described in this section.*

Proof. If we let $P = |Val(p)|$ and $Q = |Val(q)|$, then from Corollary 3.4, we know that $M \le P + Q$. Since p and q are in polynomial normal form, we also know that $len(p) = O(\log(P))$ and $len(q) = O(\log(Q))$ from which it follows that $len(\theta) = O(\log(P) + \log(Q))$, and hence $\log(M) = O(len(\theta)$ and $M = 2^{O(len(\theta))}$. Furthermore, one can show that the cost of computing a derivative expression, as well as the size of all derivative expressions, is linearly bounded by $len(\theta)$. The normal form computation is dominated by the complexity of computing PNF, which is exponential in the size of the input. It follows that $R = 2^{O(len(\theta))}$. The equivalence test for derivatives of rational functions can be carried out in linear time (cf. end of subsection 4.2). Hence the overall complexity of the construction for the rational 2-adic specification θ is $O(MR+EM^2\log(M)) = O((2^{O(len(\theta))})^2+O(len(\theta))(2^{O(len(\theta))})^2 O(len(\theta))) = 2^{O(len(\theta))}$. QED

6 Haskell Program

We have written a Haskell program which carries out the symbolic construction of Mealy machines from 2-adic and mod-2 specifications as described in section 5. The program produces as output a DOT source file (.dot) and a LaTeX-document (.tex). The DOT-file contains a graphical representation of the constructed automaton, and it can be rendered in various formats, e.g. postscript, using the Graphviz tool (`www.graphviz.org`). The LaTeX-file shows the input expression, its normal form, and a symbolic representation of the states and transitions in the constructed Mealy machine. The source code, documentation and an executable are available from

URL: `http://www.cwi.nl/~costa/diffcal`

We briefly explain the functionality of the program executable (called `diffcal`). More details can be found at the above URL.

The input to the executable must be supplied by the user by setting a number of options/flags. The input specification is supplied as a string `spec` together with a flag which indicates whether the string should be parsed as a 2-adic or a mod-2 expression: `--2adic=‘‘spec’’` or `--mod2=‘‘spec’’`. The string should be an expression `E` over the signature:

```
E ::= nat | X | X^n | varname | -E | E + E | E*E | E/E
```

where `nat` is a natural number, and `varname` is a string, which is a legal LaTeX expression when prefixed with a backslash. For example, `varname` could be the string `sigma`, which produces the LaTeX code `\sigma`. Although mod-2 expressions have no numeric interpretation, we still allow natural numbers in specifications by simply parsing them to the mod-2 version of BinExp(n). E.g., the strings `‘‘5’’` and `‘‘1 + X^2’’` read as mod-2 expressions parse to the same internal representation.

The second required input is a string `fname` which is used to name the output files, and it is supplied as `-o ‘‘fname’’`. The remaining flags are optional: For constant expressions, the program allows the construction of stream automata by setting the flag `-s`. In order to compute partial implementations of infinite or very large Mealy machines, the user can specify the maximum depth `DEP` of the constructed automaton by setting the option `-d DEP`. For 2-adic specifications, the program will produce the numeric interpretation in the LaTeX-output if the flag `-n` is set. To optimise the equivalence check for rational function specifications, the user can set the flag `-e` in which case equivalence is determined by syntactic comparison of the normal forms (see end of subsection 4.2).

Below are a few examples of how the executable may be used.

```
diffcal --2adic=‘‘(1 + X + -X^3)/(1+X)’’ -s -o ‘‘example1’’
diffcal --2adic=‘‘(7*sigma/(-5))’’ -o ‘‘example2’’
diffcal --mod2=‘‘sigma + -(3/9)’’ -e -o ‘‘example3’’
diffcal --2adic=‘‘sigma*sigma’’ -e -d 3 -o ‘‘example4’’
diffcal --2adic=‘‘1/(1 +(X*sigma))’’ -n -d 3 -o ‘‘example5’’
```

7 Discussion and Related Work

Our synthesis method is based on the, essentially coalgebraic, notion of stream function derivative, and we have already mentioned the similarity with Brzozowski's [2] method for constructing DFA's from regular expressions. Other related work includes Antimirov [1] in which partial derivatives are used in constructing nondeterministic finite automata from regular expressions, and Redziejowski [14] who constructs ω-automata using derivatives of ω-regular expressions, albeit in a much more complex setting.

Synthesis of Mealy (or Moore) type automata from logic specifications has a long and well-established history, see e.g. [3,10,18,7]. The main idea here is that a logic formula φ specifies a relation R_φ between input and output streams, and from φ one can construct an automaton $\mathcal{A}_\varphi$ which essentially accepts Mealy machines whose stream function behaviour f satisfies φ, meaning that for all input streams σ, $(\sigma, f(\sigma)) \in R_\varphi$. The actual synthesis step is realised through a constructive nonemptiness test of $\mathcal{A}_\varphi$.

Logic synthesis differs from our approach in the following ways: (i) A formula φ defines a relational requirement which may have several Mealy machine solutions, whereas bitstream expressions correspond with at most one solution. (ii) The automaton $\mathcal{A}_\varphi$ described above has the property that the finite-state requirement is built in: If $\mathcal{A}_\varphi$ accepts some Mealy coalgebra, then it accepts one with finitely many states, and such a solution is constructed during the nonemptiness test. In our approach, we need to know that a specification is realisable before we start our construction, since otherwise our algorithm may not terminate. We have shown that rational 2-adic functions are realisable. (iii) The automaton constructions and transformations carried out during logic synthesis are of a considerable (conceptual and computational) complexity, whereas the coalgebraic construction of Mealy machines using derivatives is direct and conceptually simple. Here the complexity arises from the need to decide equivalence of expressions, i.e., the normal form computation. We mention that the complexity of Mealy synthesis from linear temporal logic specifications is 2EXPTIME-complete in the size of the input specification (cf. [10]). We have shown that Mealy synthesis from rational 2-adic specifications is in EXPTIME (cf. Proposition 5.2).

The principles of automaton synthesis using derivatives are clearly of a universal character, and it would be interesting to see if this technique can be generalised to other specification languages and automaton types than the ones already mentioned. Of particular interest, we mention PAR [17] which is a recently introduced declarative language for the specification of event-pattern reactive programs, a certain type of Mealy machines. The behaviour of PAR programs is defined corecursively, and their semantics is obtained via finality. Hence the main problem which must be solved in order to synthesise PAR programs is the need for an effective decision procedure to determine equivalence of PAR expressions. In general, this decision requirement seems to be the most challenging part of realising synthesis using derivatives. Coalgebraic methods may also be of interest here. See, for example, [6] which provides a coinductive proof system for the equivalence of regular expressions.

References

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

[2] Brzozowski, J., *Derivatives of regular expressions*, Journal of the ACM **11** (1964), pp. 481–494.

[3] Büchi, J. and L. Landweber, *Solving sequential conditions by finite-state strategies*, Transactions of the American Mathematical Society **138** (1969), pp. 295–311.

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

[5] Gouvêa, F., “p-adic Numbers,” Springer, 1993.

[6] Grabmayer, C., *Using proofs by coinduction to find “traditional” proofs*, in: J. L. Fiadero, N. Harman, M. Roggenbach and J. Rutten, editors, *Proceedings of*

the 1st Conference on Algebraic and Coalgebraic Methods in Computer Science (CALCO'05), LNCS **3629** (2005), pp. 175–193.

[7] Henzinger, T., S. Krishnan, O. Kupferman and F. Mang, *Synthesis of uninitialized systems*, in: *Proceedings of the 29th International Colloqium on Automata, Languages and Programming (ICALP'02)*, LNCS **2380** (2002), pp. 644–656.

[8] Jones, S. P., editor, "Haskell 98 Language and Libraries: the Revised Report." Cambridge University Press, 2003, Haskell homepage: www.haskell.org.

[9] Kohavi, Z., "Switching and Finite Automata Theory," McGraw-Hill, 1978.

[10] Pnueli, A. and R. Rosner, *On the synthesis of a reactive module*, in: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL'89)* (1989), pp. 179–190.

[11] Raney, G., *Sequential functions*, Journal of the ACM **5** (1958), pp. 177–180.

[12] Raney, G., *Functional composition patterns and power series reversion*, Transactions of the American Mathematical Society **94** (1960), pp. 441–451.

[13] Raney, G., *On continued fractions and finite automata*, Mathematische Annalen **206** (1973), pp. 265–283.

[14] Redziejowski, R., *Construction of a deterministic ω-automaton using derivatives*, Theoretical Informatics and Applications (RAIRO) **33** (1999), pp. 133–158.

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

[16] Rutten, J., *Algebraic specification and coalgebraic synthesis of Mealy machines*, Technical Report SEN-R0514, Centrum voor Wiskunde en Informatica (CWI) (2005), to appear in Proceedings FACS 2005.

[17] Sanchez, C., H. Sipma, M. Slanina and Z. Manna, *Final semantics for event-pattern reactive programs*, in: J. L. Fiadero, N. Harman, M. Roggenbach and J. Rutten, editors, *Proceedings of the 1st Conference on Algebraic and Coalgebraic Methods in Computer Science (CALCO'05)*, LNCS **3629** (2005), pp. 364–378.

[18] Vardi, M., *An automata-theoretic approach to fair realizability and synthesis*, in: P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided verification (CAV'95)*, LNCS **939** (1995), pp. 267–278.

[19] Vuillemin, J., *On circuits and numbers*, IEEE Transactions on Computers **43** (1994), pp. 868–879.