ELSEVIER

Science of Computer Programming 24 (1995) 83–95

Science of Computer Programming

Transformation of polynomial evaluation to a pipeline via Horner's rule

Peter G. Harrison[a,*], R. Lyndon While[b]

[a] *Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK*
[b] *Department of Computer Science, University of Western Australia, Nedlands, Perth, Western Australia 6009, Australia*

Received May 1992; revised November 1993; communicated by R.S. Bird

Abstract

We apply algebraic transformation techniques to synthesise Horner's rule for polynomial evaluation. Horner's rule is then transformed into a pipeline by the application of further axioms. The syntheses demonstrate the power of the algebraic style, in which inductive proof is replaced by constructive unfolding and folding of standard higher-order functions defined on lists.

1. Introduction

The algebraic style of program derivation for functional languages involves defining operations over some domain of interest (e.g. sets of numbers, booleans, lists, etc.), together with properties they satisfy in the form of equations – i.e. an algebra. These properties may be regarded as axioms from which expressions in the operators and the objects on which they act can be equated – perhaps together with other properties of the operators, such as the existence of an inverse or uniqueness. This kind of algebra is common in mathematics – group theory is but one instance. Algebraic program transformation operates essentially by the repeated application of axioms to an expression [1,2]. To aid mechanisation (we argue below), variables are abstracted from expressions, giving function-valued expressions composed of combinators. The axioms are therefore equations in these combinators and take on a more concise form. Not necessarily all variables are abstracted, since this would generate unwieldy expressions which would be difficult to manipulate mechanically and well nigh impossible to comprehend. In particular, symbols with fixed values, such as functions defined in terms of primitives and data-constants (i.e. values "known at compiletime"),

Corresponding author. E-mail: pgh@doc.ic.ac.uk.

Elsevier Science B.V.
SSDI 0167-6423(94)00012-4

need not be abstracted, since this would discard potentially useful information. Indeed, the final, transformed expression would probably have to be applied to such known data. There is a balance to be struck to obtain the level of variable abstraction that gives concise axioms which can easily be applied mechanically and with maximum generality.

We apply algebraic methods to synthesise Horner's rule from a direct definition of polynomial evaluation. We use the word "synthesise" to convey the generation of a (functional) program from some specification – here a higher-level functional program written to reflect the naive definition of polynomial evaluation. Since the specification and final algorithm are expressed in the same language, we are therefore performing program transformation. Moreover, the synthesis is mechanical, consistent with the conventional interpretation of the term "program transformation". Such synthesis/transformation can be viewed as one type of program derivation of the type described by [1] – mechanised but unstructured and restricted by the non-availability of "Eureka steps". The lack of structure arises from the repeated application of axioms taken from a single set. There are no lemmas or theorems as in more conventional program derivations – the whole synthesis can be regarded as one big theorem in this sense. However, the benefit of this monolithic style is the ability to mechanise by simple term-rewriting. The rewrite engine repeatedly tries to apply an axiom from the given set until an acceptable target program is attained. Of course, it would be quite possible to organise such a synthesis into a structured "derivation" with lemmas and theorems which would be more conducive to human digestion. But since our emphasis is on mechanisation, we present a monolithic synthesis with words of motivation here and there to explain the objective of the next section of the synthesis. These words can be seen as imparting a similar structure to that normally obtained through lemmas in a conventional derivation.

The primitive combinators we use are categorical in nature (see e.g. [4]) and are augmented with the standard "map" and "fold" higher-order functions (also combinators) defined on lists. These combinators give a rich set of axioms, many of which were first used in [7]. An important advantage of this particular set is that it gives expressions which retain much of the original program structure after variable abstraction. This is in contrast to may other abstraction algorithms, for example using the classical combinators [8]. Moreover, the categorical foundations – for the non–list combinators, at least – often facilitate the simple derivation of axioms and provide an elegant, variable-free semantics. In other words, the combinators are natural programming constructs, concise and powerful. However, we will not be concerned with such issues here, but will concentrate on giving a non-trivial program development using the techniques described.

Previous analyses of Horner's rule have relied to some extent on induction. This is necessary in some guise since we are dealing with lists (of coefficients) of unknown lengths. In our synthesis, the use of induction is embedded in the higher-order functions "fold" and (dyadic) "map" defined on lists. In particular, unfolding these definitions, applying axioms to the result and then folding back achieve the same

effect: the final fold is the inductive step. Thus, to achieve list induction (on functions definable by "map" and "fold" together with primitive first-order functions), we add to our axiom set the recursive definitions of "map" and "fold" (Axioms (32) and (31) in Section 3).

In the next section, we define a function that evaluates polynomials in the direct way, using standard higher-order functions and a user-defined function that generates an infinite list of powers of a number. In Section 3 we list the basic axioms we shall use (there are, of course, others). We apply these axioms in Section 4.1 to derive a lemma on the partial application of "dyadic map" to a function and an infinite list. In Section 4.2 we synthesise Horner's rule, in particular making use of a distributivity axiom (Axiom (50)), here of multiplication over addition. We complete the synthesis in Section 5 where further transformation yields a pipeline implementation of polynomial evaluation. Section 6 concludes with a summary and a discussion of related work, including other results achieved by our algebraic style of program synthesis.

2. Polynomial evaluation

We begin with the most direct definition of polynomial evaluation, defining the function E by

$$\mathsf{E}\,z\langle a_0, \ldots, a_n\rangle = \sum_{i=0}^{n} a_i z^i,$$

where $\langle\cdots\rangle$ denotes a list, here of finite length $n+1$. The partial application of E to a number z can be expressed as

$$\mathsf{E}\,z = \Sigma \circ \times *^2(\mathsf{powers}\ z),$$

where powers z returns the infinite list of integer powers of z from 0, $*^2$ denotes the "dyadic map" function and Σ sums the elements of a list of integers. These combinators are all defined formally in Section 3.

A more efficient method of evaluating polynomials is using Horner's rule:

$$\mathsf{H}\,z\langle a_0, \ldots, a_n\rangle = a_0 + z(a_1 + \cdots + z(a_{n-1} + za_n)\cdots)$$

We can define H by the expression

$$\mathsf{H}\,z = ((+ \bullet \times)z)^{\sim} \not\leftarrow_0$$

where $\bullet$ and $^{\sim}$ denote a modified compose and a permutator, combinator, respectively, and $\not\leftarrow_0$ denotes the "fold from the right" function with the base value 0. These are also defined in Section 3.

The main result of the paper is an axiomatic transformation from E to H (Section 4) and then on to a pipeline implementation (Section 5). A pipeline is the simplest form of

linear process parallelism, defined in our formalism by an application of the expression $\circ \not\leftarrow_{id}$ to a list of composable functions – for which the domain of each function (except the last) contains the codomain of the next. In practical terms, a datum is passed to the last function in the list and the result fed successively to the preceding function until the final result emerges from the head function. If a list of data is fed to the last function, it is possible for all functions to be processing (different data) concurrently, given a sufficiently long data-list. Such a pipeline can be implemented efficiently on a wide range of computer architectures, but its effective utilisation requires the constituent functions to process their data in approximately the same time, otherwise a bottleneck will occur – the pipeline can operate no faster than its slowest stage. In this paper we denote a pipeline by the symbol π (Axiom (43)).

3. Definitions

The synthesis uses various combinators, both standard and non-standard. This section lists the definitions of all the combinators used and the axioms which we shall use for rewriting terms in our transformations – hence the use of the name "axiom". The definitions and axioms are organised into groups describing properties of particular combinators. Each axiom may be justified extensionally, by applying each side to sufficient arguments to yield a non-function object. Justification of the axioms is needed with respect to some accepted semantic model and the extensional "proofs" provide one such. In fact, many of the axioms are identities in a Cartesian closed category model of functional programming [4]. This is one reason for choosing categorical combinators: a rich source of axioms and proof techniques that operate at the function-level (in addition to their conciseness and natural interpretation as programming constructs).

We assume that the combinators have non-strict semantics (except where otherwise stated) and that the source functions are well-typed. This allows exceptional cases, due to type violation, to be omitted. For example, we assume an uncurried function will always be applied to a pair and need not account for other cases (producing the undefined result). Our choice of a non-strict semantics is not an important issue in itself. Any other established semantics would be acceptable, but the collection of axioms that would hold (extensionally) under it may be different. For example, Axiom (11) does not hold under strict semantics (where $\bar{x}\perp \neq x$ unless $x = \perp$). However, the differences are minimal.

Basic program-forming combinators

The first group of combinators are the standard "program-forming operations" of FP, augmented with • (extended ∘) and ˜ (permutator).

$$\mathsf{id}\, x = x \tag{1}$$

$$\bar{x}y = x \tag{2}$$

$$[f_1, \ldots, f_n]x = \langle f_1 x, \ldots, f_n x \rangle \tag{3}$$

$$i_j \langle x_1, \ldots, x_j \rangle = x_i \quad \text{if } 1 \leqslant i \leqslant j \tag{4}$$

$$(p \to q;r)x = \begin{cases} qx & \text{if } px \\ rx & \text{if not}(px) \end{cases} \tag{5}$$

$$(f \circ g)x = f(g\,x) \tag{6}$$

$$(f \bullet g)x = f \circ g\,x \tag{7}$$

$$f\tilde{}\,xy = f\,y\,x \tag{8}$$

The first eight axioms describe properties of $\circ$.

$$\mathrm{id} \circ f = f \tag{9}$$

$$f \circ \mathrm{id} = f \tag{10}$$

$$\bar{x} \circ f = \bar{x} \tag{11}$$

$$(f \circ g) \circ h = f \circ (g \circ h) \tag{12}$$

$$[f_1, \ldots, f_n] \circ h = [f_1 \circ h, \ldots, f_n \circ h] \tag{13}$$

$$i_j \circ [f_1, \ldots, f_j] = f_i \quad \text{if } 1 \leqslant i \leqslant j \tag{14}$$

$$(p \to q;r) \circ f = p \circ f \to q \circ f\,;\, r \circ f \tag{15}$$

$$f \circ (p \to q;r) = p \to f \circ q\,;\, f \circ r \quad \text{for strict } f \tag{16}$$

Note that Axiom (16)' does not require f to be strict when p is boolean-valued everywhere.

The next four axioms describe properties of $\to$.

$$f \to \overline{\mathsf{true}};\overline{\mathsf{false}} = f \tag{17}$$

$$f \to f;g = f \to \overline{\mathsf{true}};g \tag{18}$$

$$f \to (f \to g;h);j = f \to g;j \tag{19}$$

$$f \to g;(f \to h;j) = f \to g;j \tag{20}$$

The next three axioms describe properties of $\tilde{}$.

$$(\tilde{}\,) \circ (\tilde{}\,) = \mathrm{id} \tag{21}$$

$$(f \circ g)\tilde{}\,x = f\tilde{}\,x \circ g \tag{22}$$

$$(f \bullet g)\tilde{}\,x = f \circ g\tilde{}\,x \tag{23}$$

List operations

The basic list-building and decomposition functions are now defined, together with three axioms describing properties of cons.

$$\mathsf{cons}\,\langle x_0, \langle x_1, \ldots, x_n \rangle\rangle = \langle x_0, x_1, \ldots, x_n \rangle \tag{24}$$

$$\mathsf{isnil}\,\langle x_1, \ldots, x_n \rangle = (n = 0) \tag{25}$$

$$\mathsf{hd}\,\langle x_1, \ldots, x_n \rangle = x_1 \quad \text{if } n > 0 \tag{26}$$

$$\mathsf{tl}\,\langle x_1, \ldots, x_n \rangle = \langle x_2, \ldots, x_n \rangle \quad \text{if } n > 0 \tag{27}$$

$$\mathsf{isnil} \circ \mathsf{cons} = \overline{\mathsf{false}} \tag{28}$$

$$\mathsf{hd} \circ \mathsf{cons} = 1_2 \tag{29}$$

$$\mathsf{tl} \circ \mathsf{cons} = 2_2 \tag{30}$$

We next introduce the list-manipulating operators, $\leftthreetimes$ (fold), *(map), zip and $*^2$ (dyadic map), and two axioms describing properties of $*^2$.

$$f \leftthreetimes_b = \mathsf{isnil} \to \bar{b};\ \Lambda^\bullet f \circ [\mathsf{hd}, f \leftthreetimes_b \circ \mathsf{tl}] \tag{31}$$

$$f^* = \mathsf{insil} \to \mathsf{id};\ \mathsf{cons} \circ [f \circ \mathsf{hd}, f^* \circ \mathsf{tl}] \tag{32}$$

$$\mathsf{zip}\ xs = \mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [[\overline{\mathsf{hd}\ xs}, \mathsf{hd}], (\mathsf{zip} \circ \mathsf{tl})\, xs \circ \mathsf{tl}] \quad \text{for infinite xs} \tag{33}$$

$$f^{*2} = (\Lambda^\bullet f)^* \bullet \mathsf{zip} \tag{34}$$

$$(f \circ g)^{*2} = f^{*2} \circ g^* \tag{35}$$

$$(f \bullet g)^{*2} = f^* \bullet g^{*2} \tag{36}$$

Currying and uncurrying operators

"Curry" and "uncurry", written Λ_i and $\Lambda^\bullet$, are defined as follows, along with four axioms describing properties of $\Lambda^\bullet$.

$$\Lambda_i f\, x_1, \ldots, x_i = f\langle x_1, \ldots, x_i \rangle \tag{37}$$

$$\Lambda^\bullet f\, \langle x_1, \ldots, x_n \rangle = f\, x_1, \ldots, x_n \tag{38}$$

$$\Lambda^\bullet f^\sim \circ [g, h, j_1, \ldots, j_n] = \Lambda^\bullet f \circ [h, g, j_1, \ldots, j_n] \tag{39}$$

$$\Lambda^\bullet f \circ [g] = f \circ g \tag{40}$$

$$\Lambda^\bullet f \circ [\bar{x}, g_1, \ldots, g_n] = \Lambda^\bullet (f x) \circ [g_1, \ldots, g_n] \tag{41}$$

$$\Lambda^\bullet f \circ [g \circ h, j_1, \ldots, j_n] = \Lambda^\bullet (f \circ g) \circ [h, j_1, \ldots, j_n] \tag{42}$$

Pipeline axioms

The two pipeline axioms, given in equivalent form in [7], are given below. They will be used in the synthesis of the pipeline in Section 5. First we define two more combinators – the pipeline combinator π and pair.

$$\pi = \circ /_{\mathrm{id}} \tag{43}$$

$$\mathsf{pair}\ \mathsf{x}\mathsf{y} = \langle \mathsf{x}, \mathsf{y} \rangle \tag{44}$$

$$\mathsf{f}/ = (\pi \circ \mathsf{f}^*)^{\sim} \tag{45}$$

$$\pi \circ (*)^{\sim}\ \mathsf{xs} \circ \mathsf{f} = (2_2 \circ \pi(\mathsf{g}^{\sim} * \mathsf{xs})) \bullet \mathsf{pair} \tag{46}$$

$$\text{where } \mathsf{g} = (\Lambda^{\bullet}{}_{\circ}) \circ [\mathsf{pair} \circ 1_2, \Lambda^{\bullet}((^{\sim}) \circ \mathsf{f})]$$

Axiom (45) essentially says that folding a function f over a list (from the right) is equivalent to first *mapping* f over the list, producing a list of partial applications, and then composing the elements of this list in a pipeline. In this way, an accumulator which is intially the base value of the fold is successively updated as required. To see this, applying the right-hand side of the axiom to the base value e and the list $[\mathsf{x}_1, \ldots, \mathsf{x}_n]$yields

$$(\pi \circ \mathsf{f}^*)^{\sim}\ \mathsf{e}[\mathsf{x}_1, \ldots, \mathsf{x}_n] = \pi[\mathsf{f}\mathsf{x}_1, \ldots, \mathsf{f}\mathsf{x}_n]\mathsf{e} = \mathsf{f}\mathsf{x}_1(\mathsf{f}\mathsf{x}_2(\cdots \mathsf{f}\mathsf{x}_{n-1}(\mathsf{f}\mathsf{x}_n\mathsf{e})\cdots))$$

Similarly, applying both sides of Axiom (46) to the objects y and z yields

$$\begin{aligned}(\pi \circ (*)^{\sim}\ [\mathsf{x}_1, \ldots, \mathsf{x}_n] \circ \mathsf{f})\mathsf{y}\mathsf{z} &= \pi[\mathsf{f}\,\mathsf{y}\,\mathsf{x}_1, \ldots, \mathsf{f}\,\mathsf{y}\,\mathsf{x}_n]\mathsf{z} \\ &= \mathsf{f}\,\mathsf{y}\,\mathsf{x}_1(\mathsf{f}\,\mathsf{y}\,\mathsf{x}_2(\cdots \mathsf{f}\,\mathsf{y}\,\mathsf{x}_{n-1}(\mathsf{f}\,\mathsf{y}\,\mathsf{x}_n\mathsf{z})\cdots))\end{aligned}$$

However, the left-hand side, when so applied, represents a pipeline in which the value y is *broadcast* to each stage, whereas the right-hand side represents a pipeline in which the value of y is passed from stage to stage.

Miscellaneous combinators

The combinator Σ introduced in the previous section to sum a list of numbers is defined by

$$\Sigma = +/_0 \tag{47}$$

The next three axioms describe consequences of commutativity, associativity and distributivity.

$$\mathsf{f}\ \text{commutative} \Rightarrow \quad \mathsf{f}^{\sim} = \mathsf{f} \tag{48}$$

$$\mathsf{f}\ \text{associative} \Rightarrow \quad \mathsf{f} \circ \mathsf{f}\mathsf{x} = \mathsf{f} \times \bullet \mathsf{f} \tag{49}$$

$$\mathsf{g}\ \text{distributes over}\ \mathsf{f} \Rightarrow \mathsf{f}/_{\mathsf{b}} \circ \mathsf{g}^* = \mathsf{g} \circ \mathsf{f}/_{\mathsf{b}} \tag{50}$$

The final three axioms describe properties of multiplication, written $\times$, and powers.

$$\times 1 = \mathsf{id} \tag{51}$$

$$\mathsf{hd}(\mathsf{powers}\ z) = 1 \tag{52}$$

$$\mathsf{tl}(\mathsf{powers}\ z) = (\times z)^*(\mathsf{powers}\ z) \tag{53}$$

4. Transformation to Horner's rule

Throughout the transformations we shall use the associativity of $\circ$ (Axiom (12)) freely without accreditation. First, we transform the partial application of f^{*2} to an infinite list into a convenient recursive form (analogous to Definitions 31 and 32) that can be used in a folding step.

4.1. Partial application of dyadic map

Applying both sides of Axiom (34) to (infinite) zs gives us

$$\begin{aligned}
&f^{*2}zs = ((\Lambda^{\bullet}f)^* \bullet \mathsf{zip})zs \\
&= \quad \{\text{Unfold } \bullet \,\&(^*);\ \textit{Axioms 7 \& 32}\} \\
&\quad (\mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [\Lambda^{\bullet}f \circ \mathsf{hd},\ (\Lambda^{\bullet}f)^* \circ \mathsf{tl}]) \circ \mathsf{zip}\ zs \\
&= \quad \{\text{Distributive zip zs};\ \textit{Axioms 15 \& 9}\} \\
&\quad \mathsf{isnil} \circ \mathsf{zip}\ zs \to \mathsf{zip}\ zs;\ \mathsf{cons} \circ [\Lambda^{\bullet}f \circ \mathsf{hd}, (\Lambda^{\bullet}f)^* \circ \mathsf{tl}] \circ \mathsf{zip}\ zs \\
&= \quad \{\text{Simplify condition};\ \textit{Axioms 33, 16, 10, 28, 11, 18 \& 17}\} \\
&\quad \mathsf{isnil} \to \mathsf{zip}\ zs;\ \mathsf{cons} \circ [\Lambda^{\bullet}f \circ \mathsf{hd}, (\Lambda^{\bullet}f)^* \circ \mathsf{tl}] \circ \mathsf{zip}\ zs \\
&= \quad \{\text{Simplify consequent};\ \textit{Axioms 33 \& 19}\} \\
&\quad \mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [\Lambda^{\bullet}f \circ \mathsf{hd}, (\Lambda^{\bullet}f)^* \circ \mathsf{tl}] \circ \mathsf{zip}\ zs \\
&= \quad \{\text{Unfold zip};\ \textit{Axiom 33}\} \\
&\quad \mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [\Lambda^{\bullet}f \circ \mathsf{hd}, (\Lambda^{\bullet}f)^* \circ \mathsf{tl}] \circ \\
&\qquad (\mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [[\overline{\mathsf{hd}\ zs}, \mathsf{hd}], (\mathsf{zip} \circ \mathsf{tl})zs \circ \mathsf{tl}]) \\
&= \quad \{\text{Distribute over conditional};\ \textit{Axioms 16 \& 20}\} \\
&\quad \mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [\Lambda^{\bullet}f \circ \mathsf{hd}, (\Lambda^{\bullet}f)^* \circ \mathsf{tl}] \\
&\quad \circ \mathsf{cons} \circ [[\overline{\mathsf{hd}\ zs}, \mathsf{hd}], (\mathsf{zip} \circ \mathsf{tl})zs \circ \mathsf{tl}])
\end{aligned}$$

$$= \quad \{\text{Select head and tail; } \textit{Axioms 13, 29, 14, 30 \& 14}\}$$

$$(\mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [\Lambda^{\bullet} f_{\circ}\, [\overline{\mathsf{hd}\,\mathsf{zs}},\ \mathsf{hd}], (\Lambda^{\bullet}\mathsf{f})^{*} \circ (\mathsf{zip} \circ \mathsf{tl})\,\mathsf{zs} \circ \mathsf{tl}]$$

$$= \quad \{\text{Simplify first element; } \textit{Axioms 41, 40 \& 6}\}$$

$$\mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [(\mathsf{f} \circ \mathsf{hd})\mathsf{zs} \bullet \mathsf{hd},\ (\Lambda^{\bullet}\mathsf{f})^{*} \circ (\mathsf{zip} \circ \mathsf{tl})\,\mathsf{zs} \circ \mathsf{tl}]$$

$$= \quad \{\text{Simplify second element; } \textit{Axioms 6, 7 \& 6}\}$$

$$\mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [(\mathsf{f} \circ \mathsf{hd})\mathsf{zs} \circ \mathsf{hd}, (((\Lambda^{\bullet}\mathsf{f})^{*} \bullet \mathsf{zip}) \circ \mathsf{tl})\,\mathsf{zs} \circ \mathsf{tl}]$$

Applying Axiom (34) now yields

$$\mathsf{f}^{*2}\mathsf{zs} = \mathsf{isnil} \to \mathsf{id};\ \mathsf{cons} \circ [(\mathsf{f} \circ \mathsf{hd})\,\mathsf{zs} \circ \mathsf{hd},\ (\mathsf{f}^{*2} \circ \mathsf{tl})\mathsf{zs} \circ \mathsf{tl}] \tag{54}$$

Eq. (54) will be used as an axiom in the following section, the synthesis of Horner's rule.

4.2. *Transformation of* E *to* H

We begin with the definition of E partially applied to the variable of the polynomial.

$$\mathsf{E}\,\mathsf{z} = \Sigma \circ \mathsf{x}^{*2}(\mathsf{powers}\ \mathsf{z})$$

$$= \quad \{\text{Unfold } \Sigma \text{ and } \not{+}\text{; } \textit{Axioms 47, 31 \& 47}\}$$

$$(\mathsf{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\mathsf{hd},\ \Sigma \circ \mathsf{tl}]) \circ \mathsf{x}^{*2}(\mathsf{powers}\ \mathsf{z})$$

$$= \quad \{\text{Distribute over conditional; } \textit{Axioms 154 \& 11}\}$$

$$\mathsf{isnil} \circ \mathsf{x}^{*2}(\mathsf{powers}\ \mathsf{z}) \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\mathsf{hd},\ \Sigma \circ \mathsf{tl}] \circ \mathsf{x}^{*2}(\mathsf{powers}\ \mathsf{z})$$

$$= \quad \{\text{Simplify condition; } \textit{Axioms 54, 16, 10, 28, 11, 18 \& 17}\}$$

$$\mathsf{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\mathsf{hd},\ \Sigma \circ \mathsf{tl}] \circ \mathsf{x}^{*2}(\mathsf{powers}\ \mathsf{z})$$

$$= \quad \{\text{Select head and tail; } \textit{Axioms 54, 16, 20, 13, 29, 14, 30 \& 14}\}$$

$$\mathsf{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [(\mathsf{x} \circ \mathsf{hd})\ (\mathsf{powers}\ \mathsf{z}) \circ \mathsf{hd},\ \Sigma \circ (\mathsf{x}^{*2} \circ \mathsf{tl})\ (\mathsf{powers}\ \mathsf{z}) \circ \mathsf{tl}]$$

$$= \quad \{\text{Simplify first element; } \textit{Axioms 6, 52, 51 \& 9}\}$$

$$\mathsf{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\mathsf{hd},\ \Sigma \circ (\mathsf{x}^{*2} \circ \mathsf{tl})\ (\mathsf{powers}\ \mathsf{z}) \circ \mathsf{tl}]$$

$$= \quad \{\text{Simplify second element; } \textit{Axioms 6, 53 \& 6}\}$$

$$\mathsf{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\mathsf{hd},\ \Sigma \circ (\mathsf{x}^{*2} \circ (\mathsf{x}\,\mathsf{z})^{*})\ (\mathsf{powers}\ \mathsf{z}) \circ \mathsf{tl}]$$

$$= \quad \{\text{Distribute } (^{*2}) \text{ over } \circ\text{; } \textit{Axiom 35}\}$$

$$\mathsf{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\mathsf{hd},\ \Sigma \circ (\mathsf{x} \circ (\mathsf{x}\,\mathsf{z}))^{*2}\ (\mathsf{powers}\ \mathsf{z}) \circ \mathsf{tl}]$$

$$= \quad \{\text{Associativity of x; } \textit{Axiom 49}\}$$

$$\text{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\text{hd},\ \Sigma \circ ((\text{x z}) \bullet \text{x})^{*2}\ (\text{powers z}) \circ \text{tl}]$$

$$= \quad \{\text{Distribute } (*^2) \text{ over } \bullet\text{; } \textit{Axiom 36}\}$$

$$\text{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\text{hd},\ \Sigma \circ ((\text{x z})^{*} \bullet \text{x}^{*2})\ (\text{powers z}) \circ \text{tl}]$$

$$= \quad \{\text{Unfold } \bullet \text{ and } \Sigma\text{; } \textit{Axioms 7 \& 47}\}$$

$$\text{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\text{hd},\ (+ \not{\leftarrow}_0) \circ (\text{x z})^{*} \circ \text{x}^{*2}\ (\text{powers z}) \circ \text{tl}]$$

$$= \quad \{\text{Distribute x over } +\text{; } \textit{Axiom 50}\}$$

$$\text{isnil} \to \bar{0};\ (\Lambda^{\bullet} +) \circ [\text{hd},\ (\text{x z}) \circ (+ \not{\leftarrow}_0) \circ \text{x}^{*2}\ (\text{powers z}) \circ \text{tl}]$$

$$= \quad \{\text{Commutativity of } +\text{, fold } \Sigma\text{; } \textit{Axioms 39, 48 \& 47}\}$$

$$\text{isnil} \to \bar{0};\ (\Lambda^{\bullet} +^{\sim}) \circ [(\text{x z}) \circ \Sigma \circ \text{x}^{*2}\ (\text{powers z}) \circ \text{tl},\ \text{hd}]$$

$$= \quad \{\text{Promote x z; } \textit{Axiom 42}\}$$

$$\text{isnil} \to \bar{0};\ \Lambda^{\bullet}\ (+ \circ (\text{x z})) \circ [\Sigma \circ \text{x}^{*2}\ (\text{powers z}) \circ \text{tl},\ \text{hd}]$$

$$= \quad \{\text{Commute construction; } \textit{Axioms 39 \& 7}\}$$

$$\text{isnil} \to \bar{0};\ \Lambda^{\bullet}\ ((+ \bullet \text{x})\,\text{z})^{\sim} \circ [\text{hd},\ \Sigma \circ \text{x}^{*2}\ (\text{powers z}) \circ \text{tl}]$$

Thus, we have

$$\text{E z} \equiv \Sigma \circ \text{x}^{*2}\ (\text{powers z})$$

$$= \text{isnil} \to \bar{0};\ \Lambda^{\bullet}\ ((+ \bullet \text{x})\,\text{z})^{\sim} \circ [\text{hd},\ \Sigma \circ \text{x}^{*2}\ (\text{powers z}) \circ \text{tl}]$$

We can now apply Axiom (31) by observing that it is satisfied when the pattern $\text{f} \not{\leftarrow}_{\text{b}}$ is replaced by $\Sigma \circ \text{x}^{*2}$ (powers z) on both sides of the equation, binding f to $((+ \bullet \text{x})\text{z})^{\sim}$ and b to 0. Thus we have

$$\text{E z} \equiv \Sigma \circ \text{x}^{*2}\ (\text{powers z}) \sqsupseteq ((+ \bullet \text{x})\,\text{z})^{\sim} \not{\leftarrow}_0 \equiv \text{H z}$$

where $\text{f} \sqsupseteq \text{g}$ means that g approximates f in the function space. The inequality results from the fact that $((+ \bullet \text{x})\text{z})^{\sim} \not{\leftarrow}_0$ is the least fixed point of the instantiated axiom, by definition under a least fixed point semantics for recursive functions. However, we have only established that $\Sigma \circ \text{x}^{*2}$ (powers z) is *a* fixed point, not necessarily the least. But since our domain of lists of coefficients is flat and E z is defined wherever H z is, we have

$$\text{E z} = \text{H z}$$

This argument is the one usually used when considering the partial correctness of unfold/fold program transformations. Notice that it is orthogonal to the issue of strictness.

5. Transformation to a pipeline

Using the π-axioms, we can synthesise a pipeline for Horner's rule from our definition of H. This is an alternative synthesis from the one given in [7] in that it uses $^{\sim}$ -axioms as opposed to equivalent axioms using the application and constant combinators. First, we apply H z to a list of coefficients as and use Axiom (8):

$$\mathsf{H\,z\,as} = \mathsf{H}^{\sim}\ \mathsf{as\,z} = ((+\bullet\mathsf{x})\mathsf{z})^{\sim}\ \not{\iota}_0\ \mathsf{as}$$

$=$ {Introduce π; *Axiom 45*}

$$(\pi\circ(((+\bullet\mathsf{x})\mathsf{z})^{\sim})^{*})^{\sim}\ 0\ \mathsf{as}$$

$=$ {Unfold $\circ$; *Axioms 8 & 6*}

$$\pi((((+\bullet\mathsf{x})\mathsf{z})^{\sim})^{*}\ \mathsf{as})0 = \pi((*)((+\bullet\mathsf{x})\mathsf{z})^{\sim}\ \mathsf{as})0$$

$=$ {Switch arguments; *Axiom 8*}

$$\pi((*)^{\sim}\mathsf{as}((+\bullet\times)\mathsf{z})^{\sim})0 = \pi((*)^{\sim}\mathsf{as}((^{\sim})((+\bullet\times)\mathsf{z})))0$$

$=$ {Abstract z & switch arguments; *Axiom 6 three times & Axiom 8*}

$$(\pi\circ(*)^{\sim}\mathsf{as}\circ(^{\sim})\circ(+\bullet\times))^{\sim}0\ \mathsf{z}$$

$=$ {Introduce pipeline; *Axiom 46*}

$$((2_2\circ\pi(\mathsf{g}^{\sim}*\mathsf{as}))\bullet\mathsf{pair})^{\sim}0\ \mathsf{z}$$

where $\mathsf{g} = (\Lambda^{\bullet}\circ)\circ[\mathsf{pair}\circ 1_2, \Lambda^{\bullet}((^{\sim})\circ(^{\sim})\circ(+\bullet\times))]$

$=$ {Simplify g & unfold $\bullet$; *Axioms 21, 9 & 23*}

$$(2_2\circ\pi(\mathsf{g}^{\sim}*\mathsf{as})\circ\mathsf{pair}^{\sim}0)\ \mathsf{z}$$

where $\mathsf{g} = (\Lambda^{\bullet}\circ)\circ[\mathsf{pair}\circ 1_2, \Lambda^{\bullet}(+\bullet\times)]$

Abstracting z now yields

$$\mathsf{H}^{\sim}\ \mathsf{as} = (2_2\circ\pi(\mathsf{g}^{\sim}*\mathsf{as})\circ\mathsf{pair}^{\sim}\ 0$$

where $\mathsf{g} = (\Lambda^{\bullet}\circ)\circ[\mathsf{pair}\circ 1_2, \Lambda^{\bullet}(+\bullet\times)]$

This is clearly a pipeline, since its principal sub-expression is π applied to the list of functions $\mathsf{g}^{\sim *}\,\mathsf{as}$. This list can be determined statically as long as as is known. If $\mathsf{as} = \langle \mathsf{a}_0, \cdots, \mathsf{a}_n\rangle$, the ith stage of the pipeline implements the function $\mathsf{g}^{\sim}\ \mathsf{a}_i$ which maps an incoming pair $\langle \mathsf{z}, \mathsf{y}\rangle$ according to the rule

$$\mathsf{g}^{\sim}\ \mathsf{a}_i\langle \mathsf{z}, \mathsf{y}\rangle = \langle \mathsf{z}, \mathsf{a}_i + \mathsf{z}\times\mathsf{y}\rangle$$

The resulting pipeline can be visualised as shown in Fig. 1.

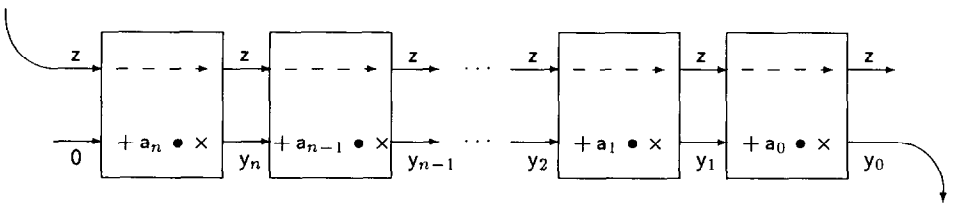

Fig. 1. A pipeline implementation of polynomial evaluation. $y_i = \sum_{j=1}^{n} a_j z^{j-i}$.

6. Conclusions

We have demonstrated the utility of the algebraic style of program transformation by using it to synthesise Horner's rule for polynomial evaluation and, from this, a pipeline implementation. This example is non-trivial in that it applies to polynomials of arbitrary degree and uses properties of the arithmetic operations, addition and multiplication. The synthesis is highly mechanised through the judicious choice of combinators at the appropriate level of variable-abstraction. Previous transformations known to the authors have relied on the use of induction in the synthesis of Horner's rule.

Moreover, a transformational tactic has been introduced which is able to make inductive inferences. We expand the recursive definition of the "fold" combinator, simplify the result algebraically, then match the resulting equation (between a previous version and the current version of the expression being transformed) against this recursion equation which defines "fold" as its least fixed point. This is a "folding step" which represents the use of the inductive hypothesis in a proof-by-induction. The result is actually an approximation in the function space, but for functions with flat co-domains, it becomes an equality if the original and transformed functions are defined on the same input. This is exactly the situation in any unfold – fold transformation methodology, of course [3].

Although described above in terms of the "fold" combinator, the same approach is also effective with recursive expansions of other combinators, such as "map". However, many of these, for example homomorphisms, can be defined in terms of "fold" and the required axiom can itself be derived using the approach with "fold" only. Indeed, we did derive just such an axiom for "dyadic map" applied to a function and an infinite list. The axioms used are very general in nature, have been applied to several other problems (see e.g. [7]), and appear capable of significant generalisation. Moreover, some of the combinators introduced can be used to write clear and concise programs at the source level.

The style of algebraic program derivation of which our transformations are an example have been followed by a number of authors. We have referred to [1] in various places as the seminal work and our own work [7] as a recent closely related application. Perhaps the best known of other work is the use of the language Ruby in

the formal design of VLSI chips [9]. Although this application is targeted on hardware development, logically it is no different from the derivation of efficient software. In either case the objective is an abstract network of processes represented by functions (or relations). It matters not whether these functions are implemented as hardware or software. In fact the Ruby work uses a relational algebra which has greater expressive power (any function is a relation, but not vice versa) but more restricted and unwieldy axioms. In particular, the composition rule for relations is more complex than the simple one that we have for functions and the extension to higher-order is considerably more complex. Nevertheless, primitive higher-order operators can be defined, giving a capability which, although not fully general, is quite adequate for practical purposes. For example, a right-fold operator analogous to ours is defined along with appropriate axioms. In fact, a form of Horner's rule itself is also used as an axiom. The Ruby system has been used successfully to derive various efficient VLSI designs from simple specifications for adders and multipliers, for example of arrays.

A number of other illustrative applications of our own algebra are reported in [7] and the inclusion of the approach into a system for deriving parallel algorithms for implementation on various architectures through "skeleton functions" is described in [5]. A skeleton function is one that abstracts the essential characteristics of an algorithm (or part thereof) whilst also having a simple implementation on some parallel machine – for example. a pipeline or a divide-and-conquer skeleton. The algebraic approach has been used to transform between skeletons [6] and in the derivation of a communication-intensive parallel quicksort algorithm; currently in preparation by Harrison and Sharp at Imperial College.

References

[1] R.S. Bird, *Lectures on Constructive Functional Programming*, Lecture notes, International Summer School on Constructive Methods in Computing Science (1988).

[2] R.S. Bird and P. Wadler, *Introduction to Functional Programming* (Prentice-Hall International, Hemel Hempstead, England, 1988).

[3] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *ACM* **24** No (1) (1977) 44–67.

[4] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, London, 1986)

[5] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, R.L. While and Q. Wu, Parallel programming using skeleton functions in: *Proceedings PARLE' 93*. Munich, Germany (1993).

[6] I.P. Guzman, P.G. Harrison and E. Medina, Pipelines for divide-and-conquer functions, *Comput. J.* **36** (1993) 254–268.

[7] P.G. Harrison, Towards the synthesis of static parallel algorithms: a categorical approach, in: *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, Pacific Grove, CA (1991) 50–68; also: B. Möller, ed., *Constructing Programs from Specifications* (North–Holland Amsterdam, 1991).

[8] J.R. Hindley and J.P. Seldin, *Introduction to Combinators and λ-calculus* (Cambridge University Press, Cambridge, England, 1986).

[9] G. Jones and M. Sheeran, Circuit design in Ruby, in: J. Staunstrup, ed., *Formal Methods for VLSI Design* (North–Holland Amsterdam, 1990) 13–70.