Functional–Based Synthesis of Systolic Online Multipliers

Tudor Jebelean
RISC-Linz
Email: tjebelea@risc.uni-linz.ac.at

Laura Szakacs
Babes-Bolyai University Cluj
Email: laura@cs.ubbcluj.ro

Abstract— **Systolic online algorithms for the multiplication of univariate polynomials and of multiple precision integers are synthesised using a novel method based on the following functional (or inductive) view: a systolic array is a head processor followed by an identical tail array. The synthesis method consists in first unfolding the functional expression of the target function until the first four elements are separated, and then by projecting the remaining list expression into the scalar space in order to obtain the transition function of the individual processors. The method is implemented as a set of rewrite rules in the *Theorema* system, and it generates the description of the systolic arrays in a completely automatic manner, starting from the functional definitions of the arithmetic operations.**

I. Introduction

The problem of integer multiplication on an online systolic array was once a challenge problem in cellular automata [1], [8], and its solution required a good deal of ingenuity. With the advent of systolic computations and their applications in digital circuit design (especially digital signal processing), intensive research lead to various methods for the systematic (and even automatic) design and synthesis methods of online systolic arrays.

However, most of these methods (see a short survey below) follow an *iterative view* of systolic arrays (and systolic computations): the arrays (and the computations) are represented as [multidimensional] matrices of a certain size (in fact many methods only work for a fixed size). This leads to complex operations over the multidimensional index space, and in fact to many repetitions in the synthesis process.

In this paper we use a *functional view* (or inductive view): an infinite systolic array is composed of a *head processor* and an identical *tail array*. Similarly, functional programs for list operations describe how to compute the head and the tail of the result in function of the head and the tail of the argument. By exploiting this similarity, we demonstrate on two case studies (multiplication of univariate polynomials and multiplication of multiple precision integers), that the synthesis problem can be solved by [essentially] rewriting of the functional programs.

We use a simplified version of online arrays, in which the output from the head processor towards the tail array consists in a copy of the main input, except for the first 2 elements: thus the tail array receives the second tail of the input. The synthesis method is based on two main properties, which are detected by equational reasoning based on the functional view of the array and of the computed functions:

- The list function computed by the array has the property that the 4-th tail of the result can be expressed recursively using the same function applied to the 2-nd tail of the input (and some other head and tail components which are easy to synthesise).
- The scalar expressions (involving individual numeric variables and functions upon them) describing the transition function of the head processor generate the list expressions (involving also list variables and list functions) describing the function realised by the array according to certain simple rewrite rules, which are also "reversible".

These two facts lead to the following synthesis method:

- Unfold the list expression of the target list function until it has the required property stated above. Unfolding consists in extracting repetitively the scalar expression of the head and the list expression of the tail, by using the functional definitions of the list functions and a few simple unfolding rules.
- Project the final list expression into the scalar space, by using the "reversed" rules mentioned above, and thus obtain the expression of the transition function.

This paper presents the principles used for synthesis in the two case studies, however the development of a complete method needs more experiments and also a continuation of the theoretical investigation in order to possibly identify necessary conditions upon the list functions which can be synthesised, as well as other aspects – notably the termination problem, which is not treated in this paper. However, we believe that the case studies presented here already demonstrate the power of the method and the interesting mathematical aspects of the interplay between functional programming and systolic computing.

The method works completely automatically and it is implemented in the *Theorema* system (www.theorema.org) [2], basically as a set of rewrite rules, together with a simulator which allows the visualisation of concrete computations. The *Theorema* system is a mathematical assistant developed at RISC–Linz under the supervision of B. Buchberger, and combines facilities for proving, solving, and computing for the purpose of exploring mathematical theories and algorithms. The system is currently implemented on top of the computer algebra system *Mathematica* [17].

Related Work: Most of the design methods use the concept

usually referred to as *space-time transformation methodology*. The work of many researchers like Quinton, Robert, Van Dongen [12], [14], [13], Delosme and Ipsen [3], Nelis and Deprettere [11] rely on the idea of this unifying approach. A review of the main ideas involved in these systolic algorithm synthesis methods is presented by Song in [15].
Many algorithms were already parallelised using the efficient technique of time-space transformations; however, this methodology also has some drawbacks.

The problem to solve should usually be formulated in the form of *uniform recurrence equations*, and it is not always easy to find such a recurrence equation. The *uniformization problem*, that is to say the problem of transforming linear recurrence equations into uniform recurrence equations, tackled by Quinton and Dongen in [13], Fortes and Moldovan [5] and others is not completely solved.

On the other hand, the time-space transformation method depends heavily on finding an affine timing function. The problem with finding affine timing function is that one needs to solve a system of linear recurrence equations, which is generally difficult and also possible only for systems having certain properties.

Although our method is less general in the sense that it generates a systolic array with a certain property, however making use of this property we get a more powerful design method, that does not have to investigate the whole index space of the problem, but considers only the relation between some computational steps.

Other attempts have been also made to overcome the drawbacks of the space-time transformation method. A remarkable alternate approach represents the methods based on viewing systolic design as program design. The work of Gribomont and Dongen [6] is based on the concept of *generic systolic array*, which means that it also makes use of the idea that the architecture of the systolic array is already chosen before the design process. However this method represents a program-oriented methodology, and for the same reason it has a low degree of automation. Our method is completely automatic.

Other methods, like that proposed by Kazerouni, Rajan and Shyamasundar in [9], [10] also try to avoid the tedious work with solving linear equation systems in order to find an adequate timing function. This method also generates solutions suitable for existing target architectures rather than designing new ones. The method essentially consist of mapping *normalised linear recurrence equations*, a subclass of linear recurrence equations, - which properly includes the class of uniform recurrence equations - onto a generic architecture called *basic systolic architecture* and then applying correctness preserving transformations to adopt this intermediate solution onto specific target architectures.

The drawback of this method is that it cannot work with parametrised problems: the size of the target architecture has to be fixed in advance.

For additional details one may also see the short survey about the systolic array design methods available in the literature [16].

II. Formal Background

This section introduces the notations used in this paper for the well known notions of lists, programs, and systolic arrays.

A. Scalars and lists

Both the systolic arrays and the functional programs which we consider in this paper act upon lists (finite or infinite) of fixed-size objects.

A fixed-size object is an object of a *scalar type*: A scalar type is an elementary type or a fixed-size tuple of scalar types. An elementary type (such as a finite set of symbols or a fixed-precision number type) can have only a finite number of instantiations. A fixed-size object will be called a *scalar*.

A *list type* over a certain scalar type characterises all the tuples (finite and infinite) of objects of that scalar type. A *list* is an object of a certain list type.

For any list $X = \langle x_0, x_1, \ldots, x_n, \ldots \rangle$, we denote by $H[X] = x_0$ the *head* of it, and by $T[X] = \langle x_1, \ldots, x_n, \ldots \rangle$ the *tail* of it. The k^{th} tail of X: $T_k[X] = \langle x_k, x_{k+1}, \ldots, x_n, \ldots \rangle$ is obtained by iterating T k times and removes the first k elements of X. By convention, $T_0[X] = X$, and note that $T_1 = T$. The k^{th} head of X is $H_k[X] = H[T_k[X]]$ and gives the $(k+1)^{\text{th}}$ element of X (thus $H_0 = H$).

The *concatenation* of two lists is denoted by "$\smile$":

$$\langle a_0, a_1, \ldots, a_k \rangle \smile X = \langle a_0, a_1, \ldots, a_k, x_0, x_1, \ldots \rangle.$$

The first operand must be finite, but the second may also be infinite. We also use "$\dot{\smile}$" for *prepending* a scalar to a (finite or infinite) list:

$$a \,\dot{\smile}\, X = \langle a \rangle \smile X.$$

Since in practice one actually uses only finite lists, we consider here only lists having a finite number of "interesting" values. Namely, we use (as in the theory of cellular automata) a special *quiescent symbol* "\$" (which belongs to all scalar types) in order to encode the "blank" values. Thus, an infinite list will start to have only blank values after a certain finite number of elements. Furthermore we will not allow "\$" to be interspersed among other elements, however we allow a list to start with a certain number of blanks.

B. Functional programs

Functions from scalar types to scalar types will be called *scalar functions*. Informally, scalar functions can be computed in constant time.

Functions from list types to list types will be called *list functions*. We will consider *only list functions acting upon infinite lists and producing infinite lists.*

We will also assume that our scalar functions produce blanks when applied to blanks, and that our list functions are producing lists of blanks when applied to lists of blanks.

A program describing a scalar function f is an expression involving elementary scalar functions (considered as "known"):

$$f[x] = E.$$

A program describing a list function F must indicate how to compute the head and the tail of the result, by starting from the tail and the head of the argument:

$$F[x \mathbin{\dot\smile} X] = E \mathbin{\dot\smile} \mathcal{E},$$

where E is a scalar expression and $\mathcal{E}$ is a list expression involving already known functions. Furthermore E may contain x, and $\mathcal{E}$ may contain x, X, and subexpressions of the form $F[T_i[X]]$.

Note that the syntactic restriction to one argument (and one value) is not essential. Indeed, a multiple scalar can be assigned a new scalar type, and a multiple list can be seen as single list by transposition:

$$\langle x \mathbin{\dot\smile} X,\ y \mathbin{\dot\smile} Y, \ldots\rangle^T = \langle x, y, \ldots\rangle^T \mathbin{\dot\smile} \langle X, Y, \ldots\rangle^T.$$

Therefore functions with mixed-type (scalar and list) argument and/or mixed-type value reduce to functions taking one scalar and one list and producing one scalar and one list. The most general case is:

$$F[x \mathbin{\dot\smile} X, y] = \langle E \mathbin{\dot\smile} \mathcal{E},\ E'\rangle,$$

where E, $\mathcal{E}$, and E' may also contain y.

C. *Online systolic arrays*

Informally, we view an *online systolic array* as a device consisting from one *head-processor* connected to a *tail-array*, which is an identical systolic array (fig. 1).

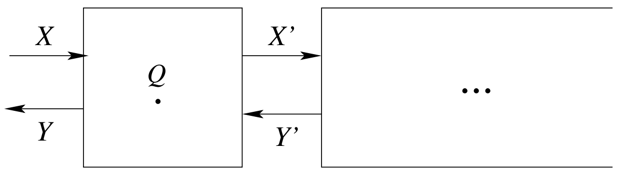

Fig. 1. Informal view of a systolic array

The array receives as input a list $X = \langle x_0, x_1, \ldots\rangle$ and outputs a list $Y = \langle y_0, y_1, \ldots\rangle$, while the list of the values of the internal state of the head-processor is $Q = \langle q_0, q_1, \ldots\rangle$. The communication with the tail-array is substantiated in the lists $X' = \langle x'_0, x'_1, \ldots\rangle$ and $Y' = \langle y'_0, y'_1, \ldots\rangle$. X and X', as well as Y and Y' are built upon the same scalar types.

At each discrete time step t (starting at 0), the array receives a value x_t and outputs a value y_t through the head-processor. Additionally the head processor outputs the value x'_t and receives the value y'_t (which are the input and the output, respectively, of the tail array), and updates its internal state q_t into q_{t+1}. The update is performed by a scalar function f (identical for all the processors), and uses the currently received values x_t and y'_t, as well as the current value of q_t. The outputs y_t and x'_t are parts of the same current value q_t, obtained by the projection functions p_x and p_y. More exactly:

$$q_{t+1} = f[x_t, q_t, y'_t], \tag{1}$$

$$y_t = p_y[q_t], \quad x'_t = p_x[q_t].$$

In more practical terms, the internal state is composed by a certain number of internal variables. We further convene that some of these variables represent the input X, some represent the output Y, and some represent the output X' towards the tail-array. The transition function f will be represented as a parallel assignment for all variables except the ones corresponding to X. Note that the array is completely specified by the list of these variables, their association to X, Y, X', and the assignment describing f.

The initial configuration of the array consists in blanks (as values of the internal states of all the processors), and we convene that f must have the property $f[\$,\$,\$] = \$$. Therefore $q_0 = y_0 = x'_0 = \$$, thus only $T[Y]$ and $T[Q]$ contain "interesting" values. We will say that the array computes the function $F[X] = T[Y]$. Usually, the infinite lists representing the input and the output will have "interesting" values only for a finite number of elements at the beginning, and the rest will be blanks.

The synthesis problem consists in finding f when F is known. When F is given as a recursive functional program, we will show in the sequel that f can be obtained by syntactic transformations of this program.

III. Arrays with delayed input pass-through

A. *Definition*

In this paper we consider a special type of online systolic arrays, whose behaviour is even more specific: the input X' of the tail-array is $T_k[X]$ for some fixed k, thus the computation of $F[X]$ will use $F[T_k[X]]$ computed by the tail-array. The data flow is illustrated in fig. 2 for the case $k = 2$.

This behaviour can be realized by including into the internal state a "state variable" s with values from {0 = \$, 1, 2, ..., k+2}, and the following assignments for s and x':

$$s \ := \begin{cases} s, \text{ if } x = \$ \text{ or } s = k+2 \\ s+1, \text{ if } x \neq \$ \text{ and } s < k+2 \end{cases}$$

$$x' \ := \begin{cases} \$, \text{ if } s < k \\ x, \text{ if } s = k \end{cases}$$

When s is $k+2$, then the first "interesting" value computed by the tail array becomes available.

B. *The direct problem*

We will derive now the general recursive equation for the function F computed by such an array. Let us denote by G the list function which gives $T[Q]$ from X: $GX = TQ$. (In the sequel we will sometimes omit the brackets denoting function application, when this does not lead to ambiguities.) Furthermore let us consider the list extensions of the scalar functions characterising the array:

$$\vec{f}[u \mathbin{\dot\smile} U, v \mathbin{\dot\smile} V, w \mathbin{\dot\smile} W] = f[u, v, w] \mathbin{\dot\smile} \vec{f}[U, V, W],$$

$$P_x[u \mathbin{\dot\smile} U] = p_x[u] \mathbin{\dot\smile} P_x[U],$$

$$P_y[u \mathbin{\dot\smile} U] = p_y[u] \mathbin{\dot\smile} P_y[U].$$

Note that they all commute with T:

$$\vec{f}[TU, TV, TW] = T\vec{f}[U, V, W],$$

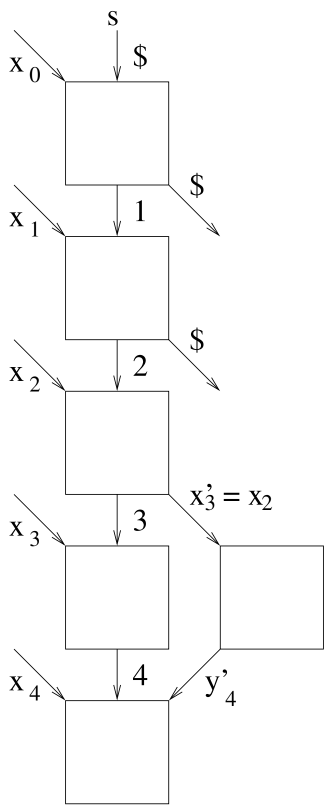

Fig. 2. Data flow in an array with input pass-through delayed by 2.

$$P_x[TU] = TP_x[U], \quad P_y[TU] = TP_y[U],$$

and also that (1) extends to the list equations:

$$TQ = \vec{f}[X, Q, Y'] = \vec{f}[X, Q, P_yQ'],$$

where Q' denotes the list of internal states of the first processor of the tail array.

In order to simplify the presentation, we will develop the expressions for the case $k = 2$ (but the generalisation is straightforward).

Clearly we need to express the function G, and then $F = P_yG$. We know that $GX = TQ$, whose first 4 values are: $q_i = f[x_{i-1}, q_{i-1}, \$]$, $i = 1, 4$, thus they do not use any result from the tail array. The behaviour after this moment allows the derivation of a recursive expression for T_4G:

$$T_4GX = T_5Q = \vec{f}[T_4X,\ T_4Q,\ T_4P_yQ'].$$

On the right-hand side, $T_4Q = T_3GX$, because $TQ = GX$. Also: $T_4P_yQ' = P_yT_4Q'$ (commutativity of P_y with T), and $T_4Q' = GT_3X'$ (the tail–array computes the same function G), and $T_3X' = T_2X$ (input pass–through). Thus one obtains the characteristic equation for G:

$$T_4GX = \vec{f}[T_4X,\ T_3GX,\ P_yGT_2X].$$

Additionally one may use $F = P_yG$ in order to transform this into:

$$T_4FX = P_y\vec{f}[T_4X,\ T_3GX,\ FT_2X], \tag{2}$$

which shows that a function F that can be unfolded until the expression of T_4F contains FT_2 is a good candidate for implementation on an online systolic array.

C. Expressions vs. functions

If E is a scalar expression with variables from the internal state, then the list of values of E depends on X. We will say that E realizes the function F_E, and also that it realizes the list $F_E[X]$. For instance, the expression s realizes the list $F_s[X] = \langle 0, 1, \ldots, k+1, k+2, k+2, k+2, \ldots \rangle$, and also $F_x[X] = X$.

Let f be a scalar function in two variables and $\vec{f}$ defined as $\vec{f}[u \smile U, v \smile V] = f[u, v] \smile \vec{f}[U, V]$. Then the expression $f[E1, E2]$ realizes the function $\vec{f}[F_{E1}, F_{E2}]$. This relation allows us to transform a scalar expression into its list function and also the other way around, by recursive projection of the list expressions into the scalar space.

Of particular interest are the head and tail functions H_i and T_i mentioned in the previous section. They can be realized by adding some suitable variables to the internal state.

The list having (almost) all elements equal to H_i is realized by a "static" variable h_i having the assignment:

$$h_i := \begin{cases} x, \text{ if } s = i \\ h_i, \text{ if } s \neq i \end{cases}.$$

Let us also consider the "transition" variables z_0, z_1, z_2, z_3 having the assignments:

$$z_0 = z_1,\ z_1 = z_2,\ z_2 = z_3,\ z_3 = x.$$

In the expression of T_4FX, the subexpression T_4X will be realized by the expression x, and each T_iX will be realized by the expression z_i (for $0 \leq i \leq 3$).

D. The inverse problem

The previous considerations allow us to construct the systolic array in a systematic manner by transformations and projections of the target function F.

First one unfolds the recursive expression of F until one obtains an expression $\mathcal{E}$ for T_4FX which contains FT_2X. (The unfolding principles are common knowledge in the program transformation theory and are illustrated in the next section.) By unfolding one also obtains the scalar expressions for the first 4 values of FX, which can be directly used in the expression of the transition function f.

Second one adds the necessary static and transition variables to the internal state, according to the occurrences of H_i and T_i in the expression $\mathcal{E}$.

Third one projects the expression $\mathcal{E}$ into the scalar space, in order to obtain the assignment for y:

- The subexpression FT_2X is projected into y', because it corresponds to the output of the tail-array.
- T_4X is projected to x, because it is the current input of the array.
- H_i and T_i are projected to the corresponding static and transit variables.
- Recursively, each subexpression of the form $\vec{f}[\mathcal{E}_1, \mathcal{E}_1]$ is projected into the corresponding $f[E_1, E_2]$.

The expressions corresponding to the first 4 values of the output contain only scalar functions and subexpressions of the form $H_i[X]$, which are projected into the variables h_i, with the exception that if $H_i[X]$ occurs in the expression of $H_i[F[X]]$, then it is projected into x.

All these transformations are readily specified as rewrite rules and are used in order to generate the online array in a completely automatic manner, as illustrated in the examples below.

Of course the automatic generation succeeds only if the unfolded version of F has the appropriate shape and contains only functions which are also "projectable", but this is absolutely natural, since we cannot expect that every recursive function is realizable by an online array.

E. Unfolding

The purpose of unfolding a list expression $\mathcal{E}$ is to transform it into $E \dot{\smile} \mathcal{E}'$, where E is a scalar expression representing the first element of the list.

We implemented an unfolding function $\mathcal{U}$ as a set of rewrite rules:

- Scalar expressions, as well as already unfolded ones remain unchanged:

$$\mathcal{U}[E] = E, \tag{3}$$

$$\mathcal{U}[E \dot{\smile} \mathcal{E}] = E \dot{\smile} \mathcal{E}. \tag{4}$$

- A list variable or the tail of it is further decomposed into head and tail:

$$\mathcal{U}[X] = H_0[X] \dot{\smile} T_1[X]. \tag{5}$$

$$\mathcal{U}[T_i[X]] = H_i[X] \dot{\smile} T_{i+1}[X]. \tag{6}$$

- If $\vec{g}$ is a list function, then one first unfolds the [list] argument, and then applies the recursive definition of $\vec{g}$:

$$\mathcal{U}[\vec{g}[\mathcal{E}]] = \mathcal{U}[\vec{g}[\mathcal{U}[\mathcal{E}]]], \tag{7}$$

$$\mathcal{U}[\vec{g}[E \dot{\smile} \mathcal{E}]] = E' \dot{\smile} \mathcal{E}', \tag{8}$$

  where $E, \mathcal{E}, E'$ and $\mathcal{E}'$ are the corresponding instances of the expressions occurring in the definition of $\vec{g}$. (Similar rules apply to mixed scalar–list functions.)

IV. Examples

A. Polynomial multiplication

We demonstrate now the method by synthesising the online multiplier of univariate polynomials.

An univariate polynomial (like e. g. $a_0 + a_1 x + a_2 x^2 + \ldots$) is represented by the list of its coefficients (lowest degree first): $A = \langle a_i \rangle_{i=0}^{\infty}$, with an infinite number of redundant zeroes at the end. The type of the coefficients is not important, we just assume it is some scalar type having ring properties.

We also assume as known the scalar operations "$\ddot{+}$" and "$\ddot{*}$" in the ring of the coefficients, as well as the following functional definitions of the operations on polynomials:

- addition of a scalar with a polynomial:

$$a \dot{+} (b \dot{\smile} B) = (a \ddot{+} b) \dot{\smile} B$$

- addition of polynomials:

$$(a \dot{\smile} A) + (b \dot{\smile} B) = (a \ddot{+} b) \dot{\smile} (A + B)$$

- multiplication of a scalar with a polynomial:

$$a \dot{*} (b \dot{\smile} B) = (a \ddot{*} b) \dot{\smile} (a \dot{*} B)$$

- multiplication of polynomials: $(a \dot{\smile} A) * (b \dot{\smile} B) = (a \ddot{*} b) \dot{\smile} ((a \dot{*} B) + (b \dot{*} A) + (0 \dot{\smile} (A * B)))$

Using these definitions and the unfolding rules presented previously, we unfold the expression "$A * B$", extracting repetitively the scalar expression representing the first element of the result, until the list expression representing the tail of the result contains $T_2[A] * T_2[B]$. (For brevity we denote $H_i[A]$ by a_i, $T_i[A]$ by A_i, and similarly for B.)

$$\begin{aligned}
&A * B = \\
&= (a_0 \dot{\smile} A_1) * (b_0 \dot{\smile} B_1) \\
&= \langle a_0 \ddot{*} b_0 \rangle \smile + \begin{cases} a_0 \dot{*} \; B_1 \\ b_0 \dot{*} \; A_1 \\ 0 \dot{\smile} (A_1 * B_1) \end{cases} \\
&= \langle a_0 \ddot{*} b_0 \rangle \smile + \begin{cases} a_0 \dot{*} \; (b_1 \dot{\smile} B_2) \\ b_0 \dot{*} \; (a_1 \dot{\smile} A_2) \\ 0 \dot{\smile} A_1 * B_1 \end{cases} \\
&= \langle a_0 \ddot{*} b_0, \; a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1 \rangle \smile + \begin{cases} a_0 \dot{*} \; B_2 \\ b_0 \dot{*} \; A_2 \\ A_1 * B_1 \end{cases} \\
&= \ldots \\
&\;= \langle \quad a_0 \ddot{*} b_0, \\
&\qquad a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1, \\
&\qquad a_2 \ddot{*} b_0 \ddot{+} a_1 \ddot{*} b_1 \ddot{+} a_0 \ddot{*} b_2), \\
&\qquad a_3 \ddot{*} b_0 \ddot{+} a_2 \ddot{*} b_1 \ddot{+} a_1 \ddot{*} b_2 \ddot{+} a_0 \ddot{*} b_3 \; \rangle \smile \\
&\smile ((a_0 \dot{*} B_4) + (b_0 \dot{*} A_4) + \\
&\qquad +(a_1 \dot{*} B_3) + (b_1 \dot{*} A_3) + (A_2 * B_2))
\end{aligned}$$

Thus we obtain an expression of the form required by (2):

$$T_4[A * B] = + \begin{cases} H_0[A] \dot{*} T_4[B] \\ H_0[B] \dot{*} T_4[A] \\ H_1[A] \dot{*} T_3[B] \\ H_1[B] \dot{*} T_3[A] \\ T_2[A] * T_2[B] \end{cases}$$

as well as the first 4 values of the output.

Let us denote the input by xa and xb and the corresponding static and transition variables by ha_i, hb_i, za_i, zb_i. According to the rules presented in the previous section, the expression on the right–hand side is projected into:

$$
\begin{aligned}
&(ha_0 \ddot{*} xb) \ddot{+} (hb_0 \ddot{*} xa) \ddot{+} \\
&\ddot{+} (ha_1 \ddot{*} zb_3) \ddot{+} (hb_1 \ddot{*} za_3) \ddot{+} y'
\end{aligned}
$$

The first 4 elements are projected into:

$$
\begin{aligned}
\langle\ & xa \ddot{*} xb, \\
& hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb, \\
& ha_1 \ddot{*} hb_1 \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb, \\
& hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb\ \rangle
\end{aligned}
$$

Thus each processor of the array has the input variables xa, xb, and the output variable y and communicates with the next processor through the variables xa', xb', and y'. The internal variables are $s, ha_0, hb_0, ha_1, hb_1, za_3, zb_3$, and y, from which y is the output variable. The transition function is described by the assignments presented in the previous section for the variables $s, ha_i, hb_i, za_3, zb_3, xa', xb'$, and by the following assignment for y:

$$
\begin{cases}
\$ & \text{,if } s = \$ = xa \\
xa \ddot{*} xb & \text{,if } s = \$ \neq xa \\
hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb & \text{,if } s = 1 \\
ha_1 \ddot{*} hb_1 \ddot{+} & \\
\quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb & \text{,if } s = 2 \\
hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} & \\
\quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb & \text{,if } s = 3 \\
hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} & \\
\quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb \ddot{+} y' & \text{,if } s = 4
\end{cases}
$$

B. Integer multiplication

The multiplication of two numbers is similar to the polynomial multiplication problem, the only significant difference is the carry propagation.

Let β be the radix for integer representation ($\beta > 1$). We consider that integer numbers are represented by the list of their digits, least significant digits first:

$$\langle a_0, a_1, a_2, \ldots \rangle \text{ represents } a_0 + a_1\beta + a_2\beta^2 + \ldots$$

We will use a scalar type for *digits* (positive integers less than β) and the corresponding list type for arbitrary precision integers, but also a scalar type for *large integers* – but not arbitrary large. The later is determined by the fixed number of operations performed by the transition function, and for sufficiently large β will be equivalent to three digits. Unless otherwise specified, in the sequel we will use "scalar" for large integer.

On large integers we define the operation "*d*" of *carry decomposition*

$$d[a] \;=\; \langle a \mod \beta, \lfloor \frac{a}{\beta} \rfloor \rangle,$$

which generates a pair consisting in the least significant digit of a and the "carry". Note that the carry is not necessarily a digit.

We consider as known the scalar operations " $\ddot{+}$ " and " $\ddot{*}$ " acting on large integers (thus also on digits).

The operation " $\dot{+}$ " of addition between a scalar and a list is defined as:

$$a \dot{+} (b \dot{\smile} B) = \text{Let}\{\langle y, r \rangle = d[a];\ y \dot{\smile} ((r \ddot{+} b) \dot{+} B)\}$$

Here the construct "Let" contains a local assignment and a final expression which is the result of the construct. This construct is necessary in order to avoid repeated computations of the same expression, in particular when the result occurs both in the head and in the tail of the resulting list (as above). The projection of this construct (when translating from list expressions into scalar expressions) is performed by inserting (if necessary) the local variables into the internal state and by adding the respective assignment to the transition function, at the time step corresponding to the stage of decomposition of the main function (see the example below).

In order to treat carry propagation in a functional way, we use the operation " $\dot{\rightarrow}$ " (prepend with carry addition):

$$a \dot{\rightarrow} A \;=\; \mathit{Let}\{\langle y, r \rangle = d[a];\ y \dot{\smile} (r \dot{+} A)\}$$

Obviously

$$a \dot{\smile} A = a \dot{\rightarrow} A$$

when a is a digit, and also:

$$
\begin{aligned}
a \dot{+} (b \dot{\smile} B) &= (a \dot{+} b) \dot{\rightarrow} B \\
a \dot{+} (b \dot{\rightarrow} B) &= (a \dot{+} b) \dot{\rightarrow} B
\end{aligned}
$$

The operations with lists are defined by:

$$
\begin{aligned}
&(a \dot{\smile} A) + (b \dot{\smile} B) = (a \ddot{+} b) \dot{\rightarrow} (A + B) \\
&a \dot{*} (b \dot{\smile} B) = (a \ddot{*} b) \dot{\rightarrow} (a \dot{*} B) \\
&(a \dot{\smile} A) * (b \dot{\smile} B) = \\
&\ = (a \ddot{*} b) \dot{\rightarrow} ((a \dot{*} B) + (b \dot{*} A) + (0 \dot{\smile} A * B))
\end{aligned}
$$

and note that:

$$(a \dot{\rightarrow} A) + (b \dot{\rightarrow} B) = (a \ddot{+} b) \dot{\rightarrow} (A + B).$$

Similarly to what happened by polynomial multiplication, we unfold the expression of the multiplication:

$$
\begin{aligned}
&A * B = \\
&= (a_0 \dot{\smile} A_1) * (b_0 \dot{\smile} B_1) \\
&= \ (a_0 \ddot{*} b_0) \dot{\rightarrow} + \begin{cases} a_0 \dot{*} B_1 \\ b_0 \dot{*} A_1 \\ 0 \dot{\smile} (A_1 * B_1) \end{cases} \\
&= \text{Let}\{\langle y, r \rangle = d[a_0 \ddot{*} b_0]; \\
&\qquad\quad y \dot{\smile} (r \dot{+} + \begin{cases} a_0 \dot{*} B_1 \\ b_0 \dot{*} A_1 \\ 0 \dot{\smile} (A_1 * B_1) \end{cases})\},
\end{aligned}
$$

which gives the expression for the first value of the output as y, while the tail of the output is further unfolded and transformed using the properties of "$\dot{\rightarrow}$":

$$
\begin{aligned}
T_1[A*B] &= r \dot{+}\ + \begin{cases} a_0 \dot{*}\ (b_1 \dot{\smile}\ B_2) \\ b_0 \dot{*}\ (a_1 \dot{\smile}\ A_2) \\ 0 \dot{\smile}\ A_1 * B_1 \end{cases} \\
&= r \dot{+}\ + \begin{cases} (a_0 \ddot{*} b_1) \dot{\rightarrow} (a_0 \dot{*}\ B_2) \\ (b_0 \ddot{*} a_1) \dot{\rightarrow} (b_0 \dot{*}\ A_2) \\ 0 \dot{\rightarrow} A_1 * B_1 \end{cases} \\
&= r \dot{+} ((a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1 \ddot{+} 0) \dot{\rightarrow} + \begin{cases} a_0 \dot{*}\ B_2 \\ b_0 \dot{*}\ A_2 \\ A_1 * B_1 \end{cases}) \\
&= (r \ddot{+} a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1) \dot{\rightarrow} + \begin{cases} a_0 \dot{*}\ B_2 \\ b_0 \dot{*}\ A_2 \\ A_1 * B_1 \end{cases} \\
&= \mathrm{Let}\{\langle y, r\rangle = d[r \ddot{+} a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1]; \\
&\qquad y \dot{\smile}\ (r \dot{+}\ + \begin{cases} a_0 \dot{*}\ B_2 \\ b_0 \dot{*}\ A_2 \\ A_1 * B_1 \end{cases})\},
\end{aligned}
$$

which gives the expression of the second value of the input as y and the second tail of the output. The later is further transformed in a similar manner into:

$$
\begin{aligned}
&T_2[A*B] = \\
&\mathrm{Let}\{\langle y, r\rangle = d[r \ddot{+} a_2 \ddot{*} b_0 \ddot{+} a_1 \ddot{*} b_1 \ddot{+} a_0 \ddot{*} b_2]; \\
&\qquad y \dot{\smile}\ (r \dot{+}\ + \begin{cases} a_0 \dot{*}\ B_3 \\ b_0 \dot{*}\ A_3 \\ a_1 \dot{*}\ B_2 \\ b_1 \dot{*}\ A_2 \\ 0 \dot{\smile}\ (A_2 * B_2) \end{cases})\},
\end{aligned}
$$

and finally:

$$
\begin{aligned}
&T_3[A*B] = \\
&\mathrm{Let}\{\langle y, r\rangle = \\
&\qquad d[r \ddot{+} a_3 \ddot{*} b_0 \ddot{+} a_2 \ddot{*} b_1 \ddot{+} a_1 \ddot{*} b_2 \ddot{+} a_0 \ddot{*} b_3]; \\
&\qquad y \dot{\smile}\ (r \dot{+}\ + \begin{cases} a_0 \dot{*}\ B_4 \\ b_0 \dot{*}\ A_4 \\ a_1 \dot{*}\ B_3 \\ b_1 \dot{*}\ A_3 \\ A_2 * B_2 \end{cases})\}
\end{aligned}
$$

gives the expression of the fourth value of the output and the list expression for $T_4[A*B]$ becomes of the same form as in (2).

We need to use the same variables as in the case of polynomial multiplication, but also the additional variable r for the carry. The assignments are again all the same, with the exception of the assignment for y, which is replaced by a new assignment for $\langle y, r\rangle$:

$$
\begin{cases}
\langle \$, \$ \rangle & ,\text{if } s = \$ = xa \\
d[xa \ddot{*} xb] & ,\text{if } s = \$ \neq xa \\
d[r \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb] & ,\text{if } s = 1 \\
d[r \ddot{+} ha_1 \ddot{*} hb_1 \ddot{+} & \\
\quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb] & ,\text{if } s = 2 \\
d[r \ddot{+} hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} & \\
\quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb] & ,\text{if } s = 3 \\
d[r \ddot{+} hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} & \\
\quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb \ddot{+} y'] & ,\text{if } s = 4
\end{cases}
$$

The last expression (having 6 terms of 2 digits) indicates that 3 digits suffice as the size of a large integer, if $\beta > 6$. Thus we have also determined this scalar type.

V. Conclusions and Further Work

The similarity between the functional (or inductive) view of systolic arrays and the shape of functional programs, combined with the power of rewriting logic, leads to an effective, efficient and elegant method for the synthesis of systolic online multipliers.

The method is implemented in the *Theorema* system and generates both a polynomial multiplier and an integer multiplier in a completely automatic manner.

Although these two case studies do not reveal all the practical and theoretical aspects of the method (in particular: the termination problem, and the characterisation of the class of functions which are realizable in this manner), the results of the present investigation offer a good basis and motivation for continuing the functional–based study of the systolic algorithms.

Acknowledgements

The *Theorema* system is supported by FWF (Austrian National Science Foundation) – SFB project F1302. The research presented here is also part of the project on program verification and synthesis, which is supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research) in the frame of the Institute e-Austria Timisoara. The second author was partially supported by a CEEPUS scholarship.

References

[1] A. J. Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Trans. on Electronic Computers*, EC-14(3):394–399, Oct 1965.

[2] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, T. Kutsia, N. Popov, J. Robu, W. Windsteiger, F. Piroi, M. Rosenkranz. *Theorema*: Towards systematic mathematical theory exploration. *Journal of Applied Logic*, 2005 (to appear). `http://www.risc.uni-linz.ac.at/publications`.

[3] J.-M. Delosme and I. C. F. Ipsen. Systolic array synthesis: computability and time cones. In *Parallel algorithms & architectures (Luminy, 1986)*, pages 295–312, Amsterdam; New York, 1986, North-Holland.

[4] G. Even. *Design of VLSI Circuits Using Retiming*, Israel Institute of Technology, PhD Thesis, 1994

[5] J. A. B. Fortes, D. I. Moldovan, *Data broadcasting in linearly scheduled array processors*, In *11th Annual Symp. on Computer Architecture*, pages 224–231, 1984.

[6] P. Gribomont and V. Van Dongen. *Generic Systolic Arrays: A Methodology for Systolic Design*, *Lecture Notes in Computer Science*,668: 746-761, 1992.

[7] T. Jebelean. *Systolic multiprecision arithmetic*; PhD Thesis, RISC-Linz Report 94-37, April 1994.

[8] D. E. Knuth. *The Art of Computer Programming:Seminumerical Algorithms*, volume 2., p.297, Addison-Wesley, Reading, Massachusetts, 1981

[9] L. Kazerouni, B. Rajan, R. K. Shyamasundar. *Mapping Linear Recurrences onto Systolic Arrays* , In *IPPS: 10th International Parallel Processing Symposium*, IEEE Computer Society Press, 1995.

[10] L. Kazerouni, B. Rajan, R. K. Shyamasundar. *Mapping Linear Recurrence Equations onto Systolic Architectures*, *International Journal of High Speed Computing (IJHSC)*, 8(3):229–270, 1996

[11] H. W. Nelis and E. F. Deprettere. *Automatic Design and Partitioning of Systolic/Wavefront Arrays for VLSI*, *Circuits systems signal process*, 7(2), 1988.

[12] P. Quinton. *The systematic design of systolic arrays*, In Francoise Fogelman Soulie and Yves Robert and Maurice Tchuente editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987.

[13] P. Quinton, V. Van Dongen. *The Mapping of Linear Recurrence Equations on Regular Arrays*; Journal of VLSI Signal Processing 2, vol. 1, pp.95-113, 1989.

[14] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*, Prentice-Hall, 1990.

[15] S. W. Song. *Systolic algorithms: concepts, synthesis, and evolution*, Temuco, CIMPA School of Parallel Computing, Chile, 1994

[16] L. Szakacs. *Automatic Design of Systolic Arrays: A Short Survey.* Technical report no. 02-27 in RISC Report Series, University of Linz, Austria. December 2002.

[17] S. Wolfram. *The Mathematica Book*, 5th edition, Wolfram Media 2003.