Program Transformations in the POLCA Project

Jan Kuper[1], Lutz Schubert[2], Kilian Kempf[2], Colin Glass[3], Daniel Rubio Bonilla[3], and Manuel Carro[4]

[1]University of Twente, Enschede, The Netherlands
[2]Ulm University, Ulm, Germany
[3]High Performance Computing Center, Stuttgart, Germany
[4]Imdea Software Institute, Madrid, Spain

***Abstract*—The POLCA project develops annotations on fragments of imperative code to guide program transformations for better utilization of resources. These annotations express the computational essence of the code fragments without referring to memory usage or execution time. That makes the annotations mathematical in nature such that provably correct transformations can be applied to them and the corresponding code fragment can be transformed accordingly for more optimal resource usage, for example on a multi-core platform or on an FPGA.**

Keywords: Program transformations, code partitioning, formal methods, optimization.

I. Introduction

It is widely recognized that parallelism of applications plays an increasingly important role for application developers. However, writing an efficient parallel program is difficult and converting an exitsing iterative program into a parallelised version is is not as straight-forward as in particular the processor manufacturers would have us believe. In fact, it is easy to write code which performance declines once parallelised.

The key factor that hampers performance is thereby the data dependency between parallel threads: concurrent access to memory, in particular read-write dependencies, not only effectively serialises the code again, but also easily leads to lower performance because of increasing data access times.

Much effort is being invested into automating the parallelisation of code and into providing libraries and language extensions to support that. Yet, hardly any of them take the pattern of an algorithm into consideration, a pattern is rather seen as a software engineering recommendation than as a characterization of the code itself. The POLCA project considers the repetitive pattern of an algorithm as a key concept for parallelisation and for improving performance. This is of particular importance in the areas of High Performance Computing (HPC) and Embedded Systems, where most applications effectively perform strongly repetitive tasks on huge data sets or on massive streaming data.

In this paper we use the term *repetition structure* for the regularity with which (a part of) an algorithm is executed multiple times over different sets of data. Such repetition structures may thereby take different forms, from iterating over an array, to complex mathematical equations per (sub)set of data.

Corresponding author: `j.kuper@utwente.nl`

Repetition structures act over time *and* space, specifically (a) the *data space* over which the algorithm acts and (b) its iterations and processing over (simulated and actual) *time*. Typical examples are eScience applications such as fluid dynamics, where the algorithm acts over the 3-dimensional space of the fluids and objects (expressed as data structures in memory) and the time for processing each iteration on the one hand and for simulating the flow over time (fourth dimension) on the other hand. When the repetition structure is known, expert HPC software developers can try parallelising the algorithm by breaking up the dependencies and typically distributing the *space-time* dimension of the algorithm over the different processors in a fashion that the dependencies are minimised. Within this paper we will show that the repetition structure can not only explicitly expressed to make it visible for developers, but also that it can be exploited for applying transformation patterns that convert repetition structures into different parallel versions with different performance.

We exploit this fact in the POLCA project to automate not only parallelisation, but also to exploit infrastructure heterogeneity, e.g. by adapting the code to vector platforms, exploiting intrinsics of GPUs or even FPGAs. In the project we develop a method that uses such information to partition an algorithm and express that in a dataflow graph such that a good mapping can be found on a given hardware platform. This paper will explain the underlying mathematical basis, i.e. the repetition structures and how appropriate transformations can be identified and applied.

Related work.: In the POLCA project the mathematical characterization and their transformations of repetition structures in algorithms is investigated to act as annotations on programming fragments in existing code, in order to analyse possibilities for partitioning the code and to distribute it over various platforms. The mathematics will thereby serve as a basis for the analysis of the costs of the different distributions.

This approach is rooted in a long tradition of mathematical approaches towards program specification and program development, and goes back to languages such as Iverson's *APL* (see [Ive62]). Notations used in this classic were later also used in *Squiggol*, as developed by Bird and Meertens (see [Mee86], [BdM97]), with the bi-annual conference series *Mathematics of Program Construction*, started in 1989, as a major platform.

Closely related to a mathematical approach are design methods based on dataflow graphs such as the Teraflux project

([Gio14, GS15]). The annotations exploited in the POLCA project can indeed be seen as formal characterizations of data dependencies, but the focus of the POLCA project is on code transformations based on the (provable) equivalence of different variants of an annotation.

OpenMP and FORTRAN compilers exploit similar information, in particular loop iterations over arrays to break up the code into parallel segments. However, typically these approaches rely on intrinsic *concurrency* of the inner loop segment, i.e. that the iterations show no data dependencies across iterations, or at best highly controllable one.

We deviate from these approaches in our strong focus on the *structure* that is expressed by various higher order functions (see Figure 1), and by varying on the *notation* of those structures such that the transformations and the proofs of their correctness become more algebraic.

Acknowledgment: The work presented here was partially funded by the European Commission under FP7-ICT-2013.3.4, grant agreement 610686.

II. Repetition Structures

In this section we will elaborate some core repetition structures (time and space iterations) that form the basis of many computationally intensive problems, i.e., that require performance through exploitation of the platform and parallelism. For the purpose of explanation, we will restrict ourselves to one-dimensional structures, several of which are given in Figure 1. In the first column the *name* of the structure is given, in the second column its graphical representation, and in the third column the notation in terms of the functional programming language Haskell[1]. Below we will also introduce a more mathematical notation for some repetition structures which gives the possibility to prove the correctness of transformations by algebraic reasoning.

As can be seen from the graphical representation of these repetition structures, some of them can be executed in parallel. This notably holds for *map* and *zipWith*, whose results consist of lists of values zs. Other structures, such as *foldl*, are accumulative in nature, and contain data dependencies, though depending on the platform, the individual functional applications still may be executed at the same time. The accumulative structures in Figure 1 all have a starting value a, and an end value w, and sometimes a sequence of intermediate values zs. Finally, the argument n of *itn* (for *iterate*) indicates how often the function f has to be repeated.

The iteration structures presented in Figure 1 are *higher order functions* in a functional language, most of them being standard predefined functions in, e.g., Haskell. A novel aspect of the POLCA project is that, unlike the standard view in a functional programming language, these higher order functions are not considered as recursively defined functions, but rather as denoting *computational structures*, as shown in Figure 1, and which can be manipulated and transformed by mathematical means.

Note also that the repetition structures in Figure 1 may be seen as *hardware architectures* which makes it possible to translate the structures directly into hardware, for example on an FPGA (see [Baa15]).

Though the graphical representation of the various structures should be clear in itself, we nevertheless give a small example of every structure to illustrate its usage. In the examples below we use the following definitions:

$$\begin{aligned} f\ x &= 2x{+}1 \\ g\ a\ x &= (a{+}x,\ a{+}x) \end{aligned}$$

The following examples should explain the meaning of the higher order functions above:

$$\begin{aligned}
&\mathit{itn}\ f\ 2\ 5 && = 95 \\
&\mathit{itnscanl}\ f\ 2\ 5 && = [2,5,11,23,47,95] \\
&\mathit{map}\ f\ [1,2,3,4,5] && = [3,5,7,9,11] \\
&\mathit{zipWith}\ (+)\ [1,2,3]\ [1,2,3] && = [2,4,6] \\
&\mathit{foldl}\ (+)\ 0\ [1,2,3,4,5] && = 15 \\
&\mathit{scanl}\ (+)\ 0\ [1,2,3,4,5] && = [0,1,3,6,10,15] \\
&\mathit{mapAccumL}\ g\ 0\ [1,2,3,4,5] && = (15,[1,3,6,10,15])
\end{aligned}$$

We illustrate the usage of some of the above structures by the dot product of two (equally long) vectors $\vec{u}$ and $\vec{v}$, defined as follows:

$$\vec{u} \bullet \vec{v} = \sum_{i=0}^{n-1} u_i{\cdot}v_i$$

That is, the elements in the vectors $\vec{u}$ and $\vec{v}$ are first multiplied element-wise, and then added. Figure 2 shows how a combination of *zipWith* and *foldl* may define the dot product, expressed as (using *us*, *vs* instead of $\vec{u}$, $\vec{v}$):

$$us \bullet vs \;=\; \mathit{foldl}\ (+)\ 0\ (\mathit{zipWith}\ *\ us\ vs)$$

We introduce the following mathematical notation for the two higher order functions involved here:

$$\begin{aligned}
\mathit{foldl}\ (*)\ a\ xs &\;:\; a \circledast xs \\
\mathit{zipWith}\ (*)\ xs\ ys &\;:\; xs\ \widehat{*}\ ys
\end{aligned}$$

These operations are defined as follows:

$$\begin{aligned}
a \ \textcircled{$\star$}\ [\,] &= a \\
a \ \textcircled{$\star$}\ (x{:}xs) &= (a{\star}x)\ \textcircled{$\star$}\ xs \\[1ex]
[\,]\ \widehat{\star}\ [\,] &= [\,] \\
(x{:}xs)\ \widehat{\star}\ (y{:}ys) &= (x{\star}y)\ :\ (xs\ \widehat{\star}\ ys)
\end{aligned}$$

In this notation, the definition of the dot product is:

$$us \bullet vs \;=\; 0 \oplus (us\ \widehat{*}\ vs)$$

[1]We follow the Haskell convention and separate the arguments of a function by spaces rather than by commas, and we (usually) will not surround arguments by brackets.

Some further remarks on notations standard in Haskell parlance: we use *xs*, *ys*, etc., rather than $\vec{x}$, $\vec{y}$, for vectors and lists, where "*xs*" indicates the plural of "x". Likewise, "xss" denotes a sequence of sequences of values.

The operation ":" denotes the *cons* operation, i.e., the extension of a given list with an element in front, $+\!\!+$ denotes *concatenation* of two lists, $[\,]$ stands for the empty list.

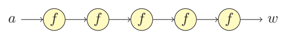
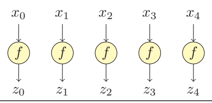
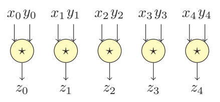
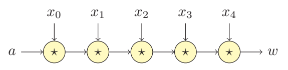
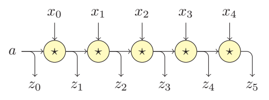
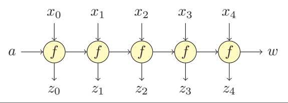
Name	*Structure*	*Haskell*	*Imperative Code*
itn		$w = itn\ f\ a\ n$	`w = a;` `for (i = 0; i < n; i++)` `w = f(w);`
map		$zs = map\ f\ xs$	`for (i = 0; i < n; i++)` `zs[i] = f(xs[i]);`
zipWith		$zs = zipWith\ (\star)\ xs\ ys$	`for (i = 0; i < n; i++)` `zs[i] = xs[i] ⋆ ys[i];`
foldl		$w = foldl\ (\star)\ a\ xs$	`w = a;` `for (i = 0; i < n; i++)` `w = w ⋆ xs[i];`
scanl		$zs = scanl\ (\star)\ a\ xs$	`zs[0] = a;` `for (i = 0; i < n; i++)` `zs[i+1] = zs[i] ⋆ xs[i];`
mapAccumL		$(w, zs) = mapAccumL\ f\ a\ xs$	`w = a;` `for (i = 0; i < n; i++)` `(w,zs[i]) = f(w,xs[i]);`

Figure 1. Some repetition structures

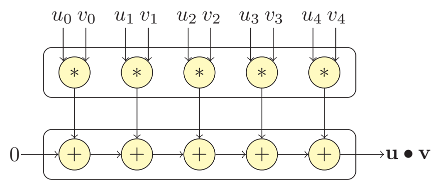


Figure 2. Composition of structures: dot product

III. Transformation rules

In this Section we discuss some transformation rules to manipulate the structure of an algorithm, as indicated by higher order functions which are given as *annotations* to fragments of imperative code of which they give the "computational structure". The transformations on the annotations are mathematical in nature, hence provably correct, and are mirrored by corresponding transformations on the imperative code which by themselves would be very hard to find. The intention of such transformations is to find a better distribution over the available hardware and exploit the specific characteristics of the hardware better, such that the performance of an algorithm may be improved, as supported by cost models. Here we mention just two types of transformations:

Structural transformations focus on rules for higher order functions and change the global structure of an algorithm without changing the algebraic formula that is computed. These rules aim at redistributing an algorithm over space and/or time.

Algebraic transformations change the algebraic operations that are present in an algorithm and may influence the number of algebraic operations, such as additions and multiplications, that are needed.

In this paper we limit ourselves to the first group of transformations and their effect on an imperative algorithm, and illustrate how a correctness proof of a transformation rule looks like. We choose a simple example to discuss the transformation rules: the summation of a sequence of numbers of which we give a few equivalent formulations exploiting transformation laws.

The basic algorithm is the sequential addition of all the values in vs, as shown as the first structure in Figure 3, and which is expressed by the function *foldl* in Figure 1, where also a corresponding imperative program is given. The variants A, B, C can all be derived by formal transformations, which we will give below. In order to formulate these transformations, we will assume a function $\mathbf{s}_m$ which splits a list vs of length n into k consecutive sublists of length m (for simplicity we assume that n is divisible by m). For example:

$$\mathbf{s}_3\ [1,2,3,4,5,6] = [[1,2,3],[4,5,6]]$$

Hence, if vss is defined as $\mathbf{s}_m\ vs$, we have that

$$concat\ vss = vs.$$

a) Variant A: This variant splits the total list in subsequences of length m by the function $\mathbf{s}_m$. Then all values in the first subsequence are added, after which the result of that is used as the starting value for the addition of the second

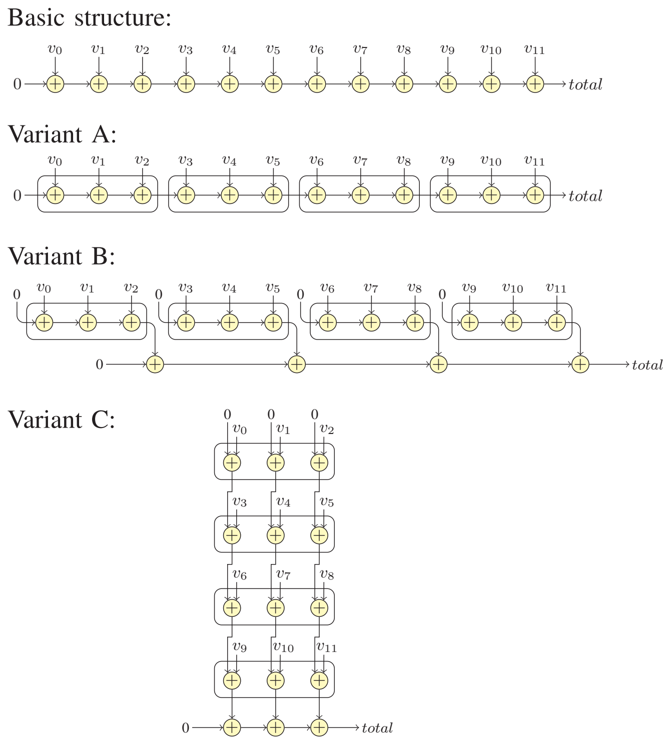

Basic structure:
Variant A:
Variant B:
Variant C:
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11}
0
total

Figure 3. Structures for the summation of a row of numbers

subsequence, etcetera, see Figure 3. This variant may be useful when the total sequence is too long to fit on, e.g., an FPGA. With this transformation it may be cut in pieces where the same combinational circuit is re-used. Here we will ignore the fact that in such a case some memory elements should be put in between.

In order to derive the specification of this structure, consider one box as a single operation with a starting value on the left, and a sequence *xs* from above, leading to a result y at the right. This value y then is the starting value of the next subsequence. According to Figure 1, the operation on a subsequence is the operation $\oplus$. Thus, the operation $\oplus$ has to be repeatedly applied to the sequence of subsequences, starting with 0. In other words, this again is the *foldl* structure, but now not over a sequence of numbers, but over a sequence of sequences of numbers. Hence, Haskell-like formulation the specification of this partitioning is

$$total \;=\; foldl\ (foldl\ (+))\ 0\ vss$$

whereas in mathematical notation it is

$$total \;=\; 0 \;\textcircled{$\oplus$}\; vss$$

The advantage of the mathematical formulation is that the proof of the correctness of this transformation is a straightforward algebraic reasoning. We have to prove:

$$0 \;\textcircled{$\oplus$}\; (\mathbf{s}_m\ vs) \;=\; 0 \oplus vs \tag{1}$$

We need one lemma:

Lemma 1. $a \circledast (xs +\!\!+\, ys) = (a \circledast xs) \circledast ys.$

Proof. By induction on the list xs:

$$\begin{aligned}
& a \circledast ([\,] +\!\!+\, ys)\\
&\quad = (a \circledast [\,]) \circledast ys\\[1ex]
& a \circledast ((x{:}xs) +\!\!+\, ys)\\
&\quad = a \circledast (x : (xs +\!\!+\, ys))\\
&\quad \overset{def\circledast}{=} (a \star x) \circledast (xs +\!\!+\, ys)\\
&\quad \overset{IH}{=} ((a \star x) \circledast xs) \circledast ys\\
&\quad \overset{def\circledast}{=} (a \circledast (x{:}xs)) \circledast ys \quad \square
\end{aligned}$$

Below, the notation "*xss*" stands for a sequence of (sub)sequences and *concat xss* is the concantenation of all subsequences in *xss*. We prove the following theorem:

Theorem 2. $a \;\textcircled{$\circledast$}\; xss \;=\; a \circledast (concat\ xss).$

Proof. By induction on the list xss:

$$\begin{aligned}
& a \;\textcircled{$\circledast$}\; [\,]\\
&\quad \overset{def\bigcirc}{=} a\\
&\quad \overset{def\circledast}{=} a \circledast [\,]\\
&\quad = a \circledast (concat\ [\,])\\[1ex]
& a \;\textcircled{$\circledast$}\; (xs : xss)\\
&\quad = (a \circledast xs) \;\textcircled{$\circledast$}\; xss\\
&\quad \overset{IH}{=} (a \circledast xs) \circledast (concat\ xss)\\
&\quad \overset{L1}{=} a \circledast (xs +\!\!+\, concat\ xss)\\
&\quad = a \circledast (concat\ (xs : xss)) \quad \square
\end{aligned}$$

Now equation (1) follows immediately, since $concat\ (\mathbf{s}_m\ xs) \;=\; xs.$

b) Variant B: In variant B all values within subsequences are added independently, and then the results are added. Note that for this variant the facts that the operation + is associative and that 0 is the unit element of + are used. In Haskell format the specification of variant B is ("*pts*" stands for "partial totals"):

$$\begin{aligned}
& total \;=\; foldl\ (+)\ 0\ pts\\
& \quad \textbf{where}\\
& \quad pts \;=\; map\ (foldl\ (+)\ 0)\ vss
\end{aligned}$$

In mathematical format this is expressed as:

$$total \;=\; 0 \oplus (\,\widehat{0 \oplus}\;\; vss)$$

Now equation (2) can be proven by the same algebraic reasoning as with Variant A:

$$0 \oplus (\,\widehat{0 \oplus}\;\; (\mathbf{s}_m\ vs)) \;=\; 0 \oplus vs \tag{2}$$

c) Variant C: The elements in the subsequences are added element-wise in parallel, starting with a sequence of zeroes of adequate length, and add the results. Here, + is assumed to be associative and commutative, and 0 is the unit element of +.
Let *zs* be the equence of m zeroes needed for the vector of start values for the additions, then this partitioning can be defined in a Haskell-style formalism as follows:

$$\begin{aligned} \mathit{total} &= \mathit{foldl}\ (+)\ 0\ \mathit{pts} \\ &\quad \textbf{where} \\ &\quad \mathit{pts} = \mathit{foldl}\ (\mathit{zipWith}\ (+))\ \mathit{zs}\ \mathit{vss} \end{aligned}$$

In a mathematical formalism the definition now is be:

$$\mathit{total} \ = \ 0 \ \textcircled{+} \ (\mathit{zs} \ \widehat{\textcircled{+}} \ \mathit{vss})$$

As before the proof of the relevant equation is by algebraic reasoning:

$$0 \ \textcircled{+} \ (\mathit{zs} \ \widehat{\textcircled{+}} \ (\mathbf{s}_m \ \mathit{vs})) \ = \ 0 \ \textcircled{+} \ \mathit{vs} \qquad (3)$$

d) Imperative code.: Figure 4 shows the imperative code that corresponds to the variants A, B, C mentioned above. We remark that the imperative code is (apart from some cosmetic renamings of variables) generated automatically based on the transformations on the annotations.

The automatic code generation is not yet fully developed, as for example can be seen from the redundant assignment `pts=zs` in the code for Variant C. However, the power of being guided by the annotations in this code generation is that the annotations are formulated purely in terms of *data dependencies only*, and thus *memory dependencies*, which make traditional code transformations next to impossible, are not a relevant notion.

IV. Further remarks, future work

In the previous sections we sketched the basic idea of the POLCA project: to annotate fragments of imperative code by its essential computational structure which then may guide code transformations for better performance on alternative platforms, supported by provably correct formal transformations of the annotations. In addition to the initial linear examples discussed in the previous sections, the POLCA project works on more complex structures, such as underlying vector products, matrix products, image filtering, signal processing, image processing, etc. Also transformations of combinations of various structures are being investigated. We conclude this paper with a short illustration of a somewhat more involved case: the already mentioned dot product of two vectors (see Figure 2). In Figure 5 the partitionings are shown graphically, whereas in Figure 6 the corresponding annotations with the automatically generated imperative code is given, where $\boxtimes$ is defined as follows:

$$a \boxtimes (u, v) \ = \ a + u * v$$

The function *zip* turns the two lists *us* and *vs* into one list of 2-tuples (u_i, v_i). We conclude with the mathematical form for this version of the dot product:

$$\mathit{dotpr} \ = \ 0 \ \textcircled{\boxtimes} \ (\mathit{zip}\ \mathit{us}\ \mathit{vs}).$$

Variant A

```
for (i = 0; i < k; i++) {
  for (j = 0; j < m; j++) {
    vss[i][j] = vs[i * m + j];
  };
};
total = 0;
for (i = 0; i < k; i++) {
  for (j = 0; j < m; j++) {
    total = total + vss[i][j];
  };
```

Variant B

```
for (i = 0; i < k; i++) {
  for (j = 0; j < m; j++) {
    vss[i][j] = vs[i * m + j];
  };
};
for (i = 0; i < k; i++) {
  pts[i] = 0;
  for (j = 0; j < m; j++) {
    pts[i] = pts[i] + vss[i][j];
  };
};
total = 0;
for (i = 0; i < k; i++) {
  total = total + pts[i];
};
```

Variant C

```
for (i = 0; i < m; i++) {
  zs[i] = 0;
};
for (i = 0; i < k; i++) {
  for (j = 0; j < m; j++) {
    vss[i][j] = vs[i * m + j];
  };
};
pts = zs;
for (i = 0; i < k; i++) {
  for (j = 0; j < m; j++) {
    pts[j] = pts[j] + vss[i][j];
  };
};
total = 0;
for (i = 0; i < k; i++) {
  total = total + pts[i];
};
```

Figure 4. Imperative code for variants A, B, C

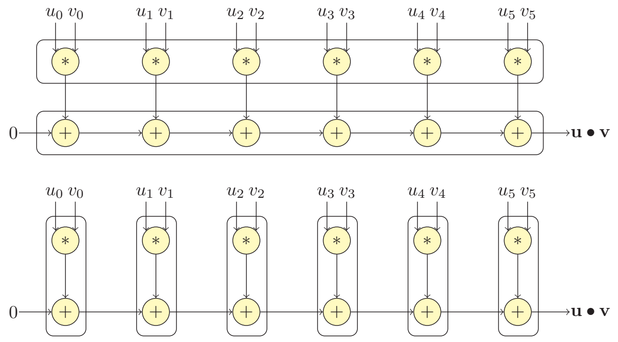


Figure 5. Two partitioning variants for the dot product

$dotpr \;=\; foldl\,(+)\,0\,(zipWith\,(*)\,us\,vs)$

```
for (i = 0; i < n; i++) {
  ws[i] = us[i] * vs[i];
};
dotpr = 0;
for (i = 0; i < n; i++) {
  dotpr = dotpr + ws[i];
};
```

$dotpr \;=\; foldl\,(\boxtimes)\,0\,(zip\;us\;vs)$

```
dotpr = 0;
for (i = 0; i < n; i++) {
  dotpr = dotpr + us[i] * vs[i];
};
```

Figure 6. Imperative code for the two dot products

As has been shown by this paper this form annotation and transformation allows to easily exploit the behavioural patterns of algorithms *without having to express them in a complicated fashion* and beyond simple information such as “loopindependent” or similar. With the kind of annotation that is close to the actual *intention* of the algorithm, we can easily generate a set of different types of provably identical code with different properties and degrees of parallelism.

References

[Baa15] Christiaan P.R. Baaij. *Digital Circuits in CλaSH — Functional Specifications and Type Directed Synthesis*. PhD thesis, University of Twente, The Netherlands, 2015.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[Gio14] Roberto Giorgi, *et al.* TERAFLUX: Harnessing dataflow in next generation teradevices. *Micronprocessors and Microsystems*, 38:976–990, 2014.

[GS15] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *Future Generation Computer Systems*, 53:100–108, 2015.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[Mee86] Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North-Holland Publishing Company, 1986.