Transformational derivation of (parallel) programs using skeletons*

Eerke A. Boiten† — Eindhoven University of Technology
A. Max Geerling — University of Nijmegen
Helmut A. Partsch† — University of Ulm

Abstract

We describe a framework for the derivation of programs for arbitrary (in particular, parallel) architectures, motivated by a generalization of the derivation process for sequential algorithms. The central concept in this approach is that of a *skeleton*: on the one hand, a higher-order function for targeting transformational derivations at, on the other hand representing an elementary computation on the architecture aimed at. Skeletons thus form a basis for intermediate languages, that can be implemented once and for all, as a process separate from individual program developments. The available knowledge on the derivation of (higher-order) functional programs can be used for deriving parallel ones.

This paper presents an overview of the method, illustrated with an example (trapezoidal rule on SIMD processor array), and ideas for future research.

1 Introduction and overview

The introduction of various computer networks and parallel computers in recent years has led to a large increase in available computing power and diversity of architectures. A large amount of parallel software, however, still needs to be developed in order to fully exploit these. This may be due to the fact that judging (or preferably: formally determining) correctness and complexity constitute even more difficult tasks in the case of a parallel algorithm. Probably for similar reasons, parallelizing compilers have not been particularly successful yet. Another problem is that each parallel architecture has its particular kinds of primitives and algorithms, and thus software has to be developed separately for each architecture. We seek to solve these problems by applying formal methods.

This paper presents a modest approach to deriving parallel programs. Techniques already known from the area of transformational (or: calculational) program development [11, 16] are used for and extended to the derivation of parallel programs. The method of using so-called skeletons [9] that is described here has been introduced by Darlington and others [10] and used in several case studies [17, 14, 12, 13].

Recent work by Pepper [19] emphasizes the problem of data distribution in parallel-program development in this approach. The formalization is done by means of so-called *covers* [23], which partition a data structure in such a way that subobjects may overlap. The overlapping parts in a cover entail communication patterns.

The crucial step in generalizing the transformational approach to also apply to parallel programming lies in a reconsideration and generalization of the goals of transformational programming for sequential architectures (we admit that this generalization has been made in hindsight). We observe that the process of transformation of functional programs to imperative ones aims at one particular recursion scheme, or *higher-order function*, viz. tail recursion. This aim is

*This paper is based on a presentation by the first author at the Research Institute for Declarative Systems colloquium on parallelism, University of Nijmegen, November 19, 1992.

†Part of this research has been carried out while the authors were at the University of Nijmegen, sponsored by the Netherlands Organization for Scientific Research (N.W.O.) under grant 63/62-518 (the NFI project Specification and Transformation Of Programs).

generalized for the case of an arbitrary architecture to a specific set of higher-order functions, the so-called *skeletons*. Covers form a basis for defining skeletons in Pepper's approach.

Section 2 motivates the use of skeletons by reconsidering the development model for sequential algorithms. In section 3, the skeleton method of program derivation is laid out in full. In section 4, a processor array is characterized in terms of skeletons. An example derivation for such a machine is given in section 5. The final section draws a few conclusions and mentions a number of interesting research subjects.

2 A new viewpoint on the derivation of sequential programs

A lot of experience has been gained with the transformational derivation of efficient sequential programs from functional or even non-operational formal specifications [6]. Comprehensive textbooks, based on the Munich CIP project, are [7, 16]. Feather [11] gives an extensive summary of approaches to program transformation and the most important techniques employed in those. Various sequential algorithms known from the literature have been explained, reinvented, or even corrected by transformational derivation. Current emphasis is on extension and generalization: from particular derivations to derivations and heuristics for larger classes of problems, from particular programs to general patterns of programs and types of recursion and onwards to general data types with corresponding recursion patterns, from (mostly) functional formalisms to relational, logic, and parallel formalisms.

So much for the sales talk — the body of knowledge developed and to be developed in the future will undoubtedly prove very useful for the derivation of parallel algorithms as well, in particular where it concerns calculi of higher-order functions.

It has been recognized that in a complete program derivation from a descriptive formal specification to an efficient program several language levels can be distinguished; employing a so-called "wide spectrum language" that encompasses all language levels may smoothen transitions between the levels. A notable example of such a language is the one developed in the CIP project, called CIP-L [2] – we will make use of a variant of CIP-L in this paper, basically a (usually) explicitly typed, in most places functional notation. Program developments according to the CIP method can be depicted as in Figure 1.

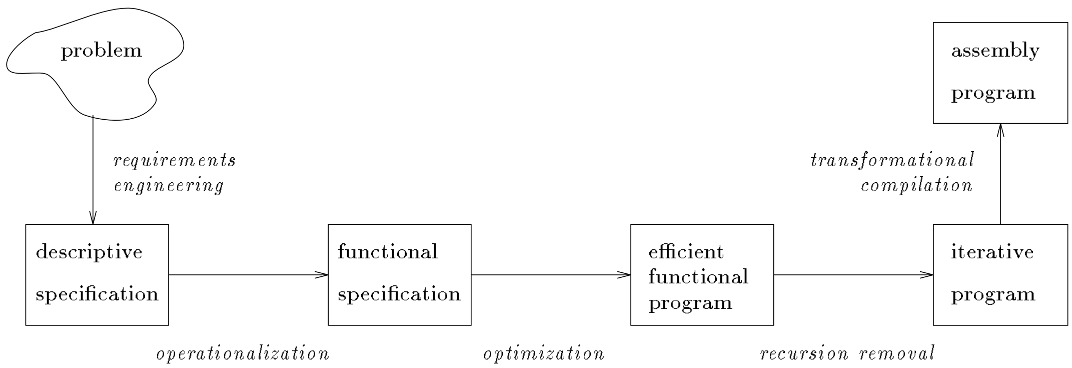

Figure 1: Transformational sequential-program development

Requirements engineering is a less formal activity that produces a first formal specification from an informal problem statement. All the other arrows in the diagram represent formal steps, viz. sequences of applications of *transformation rules* – formal semantics-preserving transitions between program schemes, resulting in equivalent or more deterministic programs, selected not automatically but by a program developer. In the second stage constructs that are not operational are replaced by operational ones. The third phase optimizes the resulting functional program. These two phases may very well be combined or interleaved, for instance when a transformation is known from a non-operational program scheme satisfying special properties directly into an efficient functional program. The recursion removal, or *coding* phase is basically a change of notation. Relatively few optimizations, most having to do with variable usage, data type implementation, and sequentialization of assignments, are left at the imperative level. (It is assumed that optimizations that can be done at the functional level are done there.) Thus, the final step may indeed be called transformational compilation — in fact, transformation steps at this level are mostly fully automatable.

As it is shown here, there is something particular about the level of "efficient functional programs". The picture says the next step will be recursion removal – but direct recursion removal is only applied to programs of a very specific shape:

$$f(x) = \textbf{if } T(x) \textbf{ then } H(x) \textbf{ else } f(K(x)) \textbf{ fi}$$

(tail recursion) which can be immediately translated to the iterative program shape

$vx := x$;
while $\neg T(vx)$ **do** $vx := K(vx)$ **od**;
return $H(vx)$.

This gives a *funnel* model of program derivation for sequential architectures: during the operationalization and optimization phases, there is a large freedom with respect to program constructs to be used, but ultimately all programs must pass through tail-recursion to descend to the imperative level (where, again, a wide variety of program shapes may eventually result).

Let us abbreviate the schema above to $TR(T, H, K)$. This way we have captured the essence of iterative programs in one higher-order function TR: given functions $T : \mathbf{m} \to \mathbf{bool}$, $H : \mathbf{m} \to \mathbf{n}$ and $K : \mathbf{m} \to \mathbf{m}$ for particular types $\mathbf{m}$ and $\mathbf{n}$, we have the function $TR(T, H, K) : \mathbf{m} \to \mathbf{n}$ defined by either of the two programs above. Note that the loop above indeed does "everything" a loop-program can do: initialization, one loop (with possibly inner loops and conditionals in K), and delivering of a final expression (again, possibly using conditionals and loops).[1]

Here, in the sequential case, we found a higher-order function sufficient to characterize basically all programs on our target machine. Let us call this higher-order function a *skeleton* for sequential architectures: a schematic program, requiring some functions as arguments, that has a straightforward interpretation as a typical elementary computation on the target architecture.

To be honest, conditionals and non-recursive abstraction are also needed through the entire development process, but they can be taken through to the imperative level with no change. Those constructs are so basic, that they will not be mentioned explicitly as "skeletons" anymore in the sequel – actually, with function application and composition, they can be seen as the basic combining forms for skeletons.

Note that the concept of a "variable" needs no independent introduction here; it just comes up in the architecture-specific interpretation of the skeleton. This follows immediately from the perspective chosen to describe the TR-skeleton. One could call this skeleton *control-oriented*, i.e., variables are abstracted away and the computation structure is given by means of a control structure, in this case a **while**-loop. Another perspective would be one in which the architecture is described by valid operations on variables, we call this a *data-oriented* view.

[1] There is no point in discussing non-terminating loop programs, as are often used to model parallel systems, here, since we now discuss fully sequential programs. Termination of the loop above is equivalent to termination of the recursion in the tail-recursive scheme, which must be proved for the functional program to be meaningful anyway.

3 The skeleton method

This section defines skeletons and outlines the corresponding development method. Above we found the following definition for skeletons.

Definition 1 *A* skeleton *for a particular architecture is a higher-order function that has a straightforward interpretation as a typical elementary computation on that architecture.*

This definition suggests that we have a look at some architectures and designate certain computations to be "typical" and "elementary" for a particular architecture. Figure 2 shows some architectures. The architectures are characterized (using Flynn's taxonomy) by their instruction (I) and data (D) streams, which both can be multiple (M) or single (S).

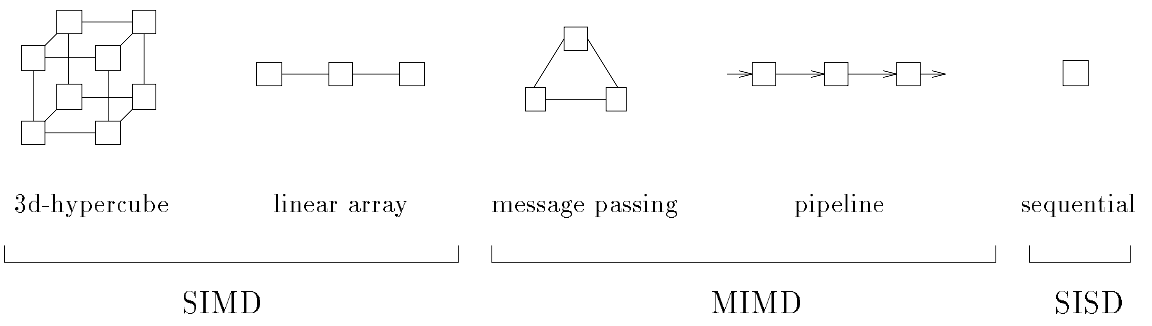

Figure 2: Computer architectures

Just as with the TR-skeleton, skeletons for these architectures could be given in a control- or data-oriented way. This distinction corresponds with the *process-parallel* or *data-parallel* type of programming. Process-parallel programming is based on the ability of performing independent computations, whereas data-parallel programming implies that the same (sequence of) operation(s) operates on every element of a data structure (not necessarily synchronously). SIMD architectures are largely programmed in this latter way. On the other hand, programming MIMD architectures one has a choice between either a data- or a process-parallel manner. We propose skeletons for the architectures of Figure 2 that are categorized as being data- or process-parallel.

The TR-skeleton for sequential architectures given above is an example of a control-oriented (i.e., process-"parallel") skeleton. Another example of this kind is a skeleton for pipelines [10]:

$$\mathit{pipe} : [\mathit{stream}(\_) \rightarrow \mathit{stream}(\_)] \times \mathit{stream}(\mathbf{m}) \rightarrow \mathit{stream}(\mathbf{n})$$
$$\mathit{pipe}(\mathit{fs}, xs) = \mathit{apply}(\mathit{reduce}(\circ, \mathit{fs}), xs)$$

where the *reduce* operation accumulates all values in a list with a binary operation, $\circ$ is function composition, and $[\mathbf{m}]$ is the type of lists of $\mathbf{m}$. This skeleton is process-parallel, because it is expressed in terms of the functions that represent the processors in the pipeline. Communication is left implicit in this skeleton, it is abstracted into the function composition.

Skeletons for the remaining three architectures (message passing, linear array and hypercube) are described in a data-parallel way. Data-parallel programming on message-passing architectures is reflected by the SCMD programming style. SCMD is an abbreviation for same-code-multiple-data. This means that all processors run the same program, but in an asynchronous manner, i.e., not in lockstep. Communication protocols are an example of this kind of programs.

SIMD architectures lend themselves perfectly to the derivation of data-parallel programs when we abstract the processors into a data structure. In that case, variables of processors are lifted to arrays of memory elements. Communication occurs only along fixed patterns which will be reflected by skeletons that manipulate the elements of the data structure in a proper way, for example following the hypercube structure.

The "core"-skeleton for these architectures – the so-called *map* skeleton – applies a given function to every element in the data structure:

$$map : ((\mathbf{m} \to \mathbf{n}) \times S(\mathbf{m})) \to S(\mathbf{n})$$
$$map(f, a) = (\mathbf{index}\ i)S(\mathbf{n}) : f(a(i))$$

(where S is a data structure corresponding to the processor structure, and the type **index** indexes S-structures). See section 4 for a concretization of SIMD skeletons in the case of a linear array.

In the data-parallel case communication is made explicit in terms of skeletons, and the control structure is abstracted away, e.g., into the *map* skeleton.

How then can skeletons be profitably used? In our point of view, skeletons have a dual role: on the one hand they are abstractions of computational features of particular architectures, on the other hand they are formal objects for targeting formal derivations at. Or, from another viewpoint, particular architectures are characterized by the sets of skeletons that represent the efficient computation structures on those architectures.

In the sequential case we noted that the development traject from tail recursion on is trivial and mostly fully automatable. In general, we can make use of the fact that skeletons actually form an *intermediate* language, that can be implemented independently of particular programs — moreover, since the skeletons are assumed to denote elementary computations, this implementation should require little new knowledge.

Generalizing the "funnel" model given for sequential programs, transformational derivation of skeleton programs could proceed in several ways. For now, the most obvious generalization seems the most feasible way: instead of transforming towards tail recursion the process is targeted at functional programs that can be expressed in terms of the available skeletons. By folding then the skeleton instances can be made explicit. Note, however, that it might also be possible to transform particular non-operational specifications into skeletons directly. Figure 3 contrasts the traditional development of parallel algorithms (the upper-right track) to the skeleton method (the lower-left track).

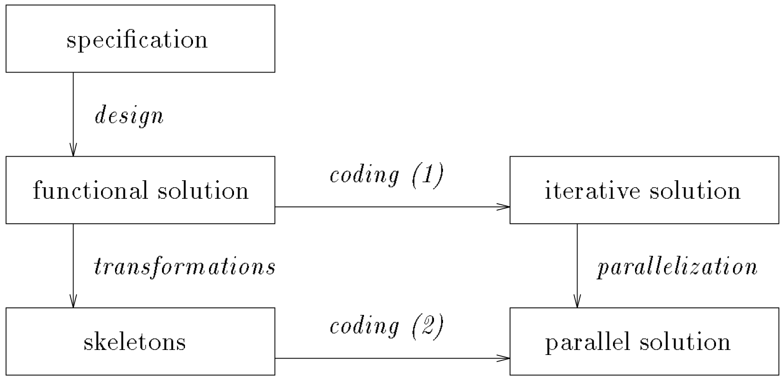

Figure 3: Parallel-program development

This model clearly distinguishes a design phase, whereas the model in Figure 1 focuses on the optimization aspect (which does not stand out in this model). In the *design* step, most of the optimizations performed will change the structure of the program. Other optimizations should take place in the *coding (1)* step and in the *transformations* step, of course. Thus, the *coding (1)* step is still a difficult one. *Parallelization* of iterative solutions also does not seem to be easy, in particular since functional programs contain much more inherent parallelism. In the bottom-left track of the figure, the *coding (2)* step can be factored out of program developments, since it can be performed once and for all for a particular skeleton on a particular machine. The *transformations* step may still be a hard one, although techniques known from sequential program derivation form a solid basis for carrying out this step.

The idea of distinguishing separate design and transformation phases suggests the concept of a *program skeleton*, as opposed to the *architecture skeleton* of definition 1. This definition of a skeleton is closer to the one used in [9, 10, 21]:

Definition 2 *A* program skeleton *is a higher-order function that can be expressed in terms of the available skeletons for some architecture.*

A program skeleton represents a whole class of algorithms with similar program structures, for which efficient implementations can be derived for various architectures. The advantage is obvious: the transformations step – a hard one – only has to made once for such a class of programs. So, program skeletons are used to increase expressiveness. Examples of such skeletons are divide&conquer and parallel prefix. Operators like *map* and *reduce* (cf. [4], see section 4) are also useful for specifying programs in, even though their implementations may not always be obvious (cf. *reduce* below) – this is also extensively argued by Skillicorn [21].

The notion of program skeleton is relevant for solving another problem we identified for parallel programming, namely the diversity of available architectures, that results in the situation that software has to be developed separately for each machine. The skeleton method may help in overcoming this *portability* problem by providing *inter-skeleton transformations*. Of course, some skeletons may be valid for several architectures — it does not matter that their *implementations* differ. The *map* skeleton, for instance, is valid for all SIMD architectures. Furthermore, transformational programming has moved to argument-free, higher-order reasoning as it is – calculi like BMF [4, 15] and FP [1] can already be viewed as calculi for inter-skeleton transformations.

4 Skeletons for linear arrays

We consider as an example a processor array machine of the SIMD class, based on the treatment in [17, 12, 13]. The data processors (DP_i) can send information to their left and right neighbors, and they all have a (small) local memory DM_i. In the example below, we assume that all processors know their indices in the array. There is a coordinating instruction processor (IP) that starts up the computations (from IM) and that can send values to all processors or to a particular one (see Figure 4).

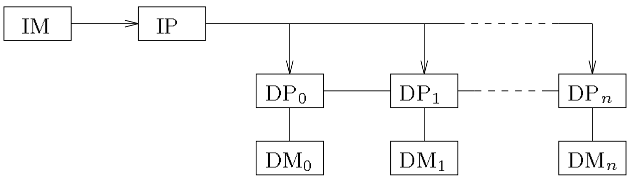

Figure 4: Linear processor array

We model the (information in the memory of) the processors by arrays, and those in turn by functions of some type $\mathbf{inat} \to \mathbf{m}$ with $\mathbf{inat} = (\mathbf{nat}\ i : 0 \le i \le n)$ for some n (i.e., there are $n + 1$ processors). Some "typical elementary computations" on such arrays are given below.

As stated in section 3, the most characteristic operator for this architecture is *map*, the simultaneous application of one function to a value in each processor.

$$map(f,a): \begin{array}{|c|c|c|c|}\hline a_0 & a_1 & \cdots & a_n \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline f(a_0) & f(a_1) & \cdots & f(a_n) \\ \hline\end{array}$$

$map : ((\mathbf{m} \rightarrow \mathbf{n}) \times (\mathbf{inat} \rightarrow \mathbf{m})) \rightarrow (\mathbf{inat} \rightarrow \mathbf{n})$
$map(f,a) = (\mathbf{inat}\ i)\mathbf{n} : f(a(i))$

Communication to right neighbors corresponds to shifting the array one position to the right; a general right shifting skeleton is given below.

$$shiftr(a,d,v): \begin{array}{|c|c|c|c|}\hline a_0 & a_1 & \cdots & a_n \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|c|c|c|}\hline v & \cdots & v & a_0 & a_1 & \cdots & a_{n-d} \\ \hline\end{array}$$

$shiftr : ((\mathbf{inat} \rightarrow \mathbf{m}) \times \mathbf{nat} \times \mathbf{m}) \rightarrow (\mathbf{inat} \rightarrow \mathbf{m})$
$shiftr(a,d,v) = (\mathbf{inat}\ i)\mathbf{m} : \mathbf{if}\ i < d\ \mathbf{then}\ v\ \mathbf{else}\ a(i-d)\mathbf{fi}$

Similarly one can define left shifts and (for different architectures) circular shifts. Note that $shiftr(a,d,v)$ is only primitive for $d = 1$ on the architecture we sketched; for a hypercube with 2^k processors an arbitrary shift takes k time steps. For a fully connected processor network even any d gives a primitive communication.

One particular processor receiving a value is modeled by the *update* skeleton. Note that this could be implemented by broadcasting (value, index) pairs, because the processors know their indices.

$$update(a,k,v): \begin{array}{|c|c|c|c|c|}\hline a_0 & \cdots & a_k & \cdots & a_n \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|c|}\hline a_0 & \cdots & v & \cdots & a_n \\ \hline\end{array}$$

$update : ((\mathbf{inat} \rightarrow \mathbf{m}) \times \mathbf{inat} \times \mathbf{m}) \rightarrow (\mathbf{inat} \rightarrow \mathbf{m})$
$update(a,k,v) = (\mathbf{inat}\ i)\mathbf{m} : \mathbf{if}\ i = k\ \mathbf{then}\ v\ \mathbf{else}\ a(i)\mathbf{fi}$

A broadcast of a value v is represented by a skeleton $new(v)$ that returns a vector of all v's.

$$new(v): \quad \longrightarrow \begin{array}{|c|c|c|c|}\hline v & v & \cdots & v \\ \hline\end{array}$$

$new : \mathbf{m} \rightarrow (\mathbf{inat} \rightarrow \mathbf{m})$
$new(v) = (\mathbf{inat}\ i)\mathbf{m} : v$

If all processors know their indices in the array, one can use a (nullary) skeleton *inew*:

$$inew(\ ): \quad \longrightarrow \begin{array}{|c|c|c|c|}\hline 0 & 1 & \cdots & n \\ \hline\end{array}$$

$inew : \rightarrow (\mathbf{inat} \rightarrow \mathbf{inat})$
$inew() = (\mathbf{inat}\ i)\mathbf{inat} : i$

The zip operation given below corresponds to bringing together two values from the processor memory.

$$zip(a,b): \begin{array}{c}\begin{array}{|c|c|c|c|}\hline a_0 & a_1 & \cdots & a_n \\ \hline\end{array} \\ \\ \begin{array}{|c|c|c|c|}\hline b_0 & b_1 & \cdots & b_n \\ \hline\end{array}\end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline (a_0,b_0) & (a_1,b_1) & \cdots & (a_n,b_n) \\ \hline\end{array}$$

$zip : ((\mathbf{inat} \rightarrow \mathbf{m}) \times (\mathbf{inat} \rightarrow \mathbf{n})) \rightarrow (\mathbf{inat} \rightarrow (\mathbf{m} \times \mathbf{n}))$
$zip(a,b) = (\mathbf{inat}\ i)(\mathbf{m} \times \mathbf{n}) : (a(i), b(i))$

Zip will often be the preparatory step for execution of a binary operator; as an abbreviation for the combination of *zip* and *map* we will use "zip with":

$$zipw(\oplus, a, b):\quad \begin{array}{|c|c|c|c|}\hline a_0 & a_1 & \cdots & a_n \\ \hline \end{array} \atop \begin{array}{|c|c|c|c|}\hline b_0 & b_1 & \cdots & b_n \\ \hline \end{array} \quad \longrightarrow \quad \begin{array}{|c|c|c|c|}\hline a_0 \oplus b_0 & a_1 \oplus b_1 & \cdots & a_n \oplus b_n \\ \hline \end{array}$$

$zipw : (((\mathbf{m} \times \mathbf{n}) \to \mathbf{p}) \times (\mathbf{inat} \to \mathbf{m}) \times (\mathbf{inat} \to \mathbf{n})) \to (\mathbf{inat} \to \mathbf{p})$
$zipw(f, a, b) = (\mathbf{inat}\ i)\mathbf{p} : f(a(i), b(i))$
$(= map(f, zip(a, b)))$

An important operation for specifying algorithms (cf. [21]) is the *reduce* operation – actually a program skeleton – that accumulates all values in an array with a binary associative operator. Remember we have $n+1$ processors available, numbered 0 to n.

$reduce : (((\mathbf{m} \times \mathbf{m}) \to \mathbf{m}) \times (\mathbf{inat} \to \mathbf{m})) \to \mathbf{m}$
$reduce(\oplus, a) = \oplus_{j=0}^{n} a(j)$

This operation is closely related to the "parallel prefix" operator, the implementation of which on various machines seems to be a main interest in the parallel programming community (cf. [5]).

$prefix : (((\mathbf{m} \times \mathbf{m}) \to \mathbf{m}) \times (\mathbf{inat} \to \mathbf{m})) \to (\mathbf{inat} \to \mathbf{m})$
$prefix(\oplus, a) = (\mathbf{inat}\ i)\mathbf{m} : \oplus_{j=0}^{i} a(j)$

An implementation of this program skeleton in terms of the skeletons of section 4 is derived in [13]. Here, only the result is given ($1_\oplus$ denotes the neutral element of $\oplus$):

$prefix(\oplus, a) = prefix'(shiftr(a, 1, 1_\oplus), 0, a)$ **where**
$\quad prefix'(a', i, b) =$ **if** $i = n$ **then** b **else** $prefix'(shiftr(a', 1, 1_\oplus), i+1, zipw(\oplus, b, a'))$ **fi**

5 Example: trapezoidal rule

This example was first treated in Paul Frederiks' master's thesis [12], and suggested by Norbert Völker. A well-known simple approximation of the definite integral $D = \int_a^b f(x)\mathrm{d}x$ is given by the so-called trapezoidal rule. The area between the curve for $f(x)$ and the x axis on the interval $[a, b]$ is divided into n equal intervals of size $h = \frac{b-a}{n}$. With $x_i = a + ih$, D is approximated by the sum of n trapezoids between the curve of $f(x)$ and the x axis, i.e.,

$$\frac{h}{2}(f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)).$$

Below we derive a skeleton program. The transformation rules used in this derivation are mostly of a simple, "calculational" kind, e.g. distributivity and commutativity of arithmetic operators. For a survey of more transformation rules and rules of a greater complexity, cf. [16]. According to the transformational tradition, we use the term "fold" for application of a definition in one direction, viz. introducing the defined notion.

We start from the initial specification:

$Trap : ((\mathbf{nat} \to \mathbf{nat}) \times \mathbf{nat} \times \mathbf{nat} \times \mathbf{nat}) \to \mathbf{nat}$
$Trap(f, x, y, n) =$ **let** $h = \frac{y-x}{n}$ **in**
$\quad \frac{h}{2}(f(x) + f(y)) + \frac{h}{2}\sum_{i=1}^{n-1} 2f(x + ih)$
$\quad\quad \equiv$ [calculus: $x = x + 0h$, $y = x + nh$; multiplication distributes over $\sum$]

$\frac{h}{2} f(x+0h) + \frac{h}{2} f(x+nh) + \frac{h}{2} \sum_{i=1}^{n-1} f(x+ih) + \frac{h}{2} \sum_{i=1}^{n-1} f(x+ih)$

$\equiv$ [+ commutative, $\sum$ index domain (2x)]

$\frac{h}{2} \sum_{i=0}^{n-1} f(x+ih) + \frac{h}{2} \sum_{i=1}^{n} f(x+ih)$

$\equiv$ [index translation; distributivity $\sum$ over +]

$\frac{h}{2} \sum_{i=1}^{n} (f(x+(i-1)h) + f(x+ih))$

$\equiv$ [index translation; 0 neutral for +]

$\frac{h}{2} \sum_{i=0}^{n}$ (**if** $i=0$ **then** 0 **else** $f(x+(i-1)h)$ **fi** +
if $i=0$ **then** 0 **else** $f(x+ih)$ **fi**)

$\equiv [i = 0 \equiv i < 1]$

$\frac{h}{2} \sum_{i=0}^{n}$ (**if** $i<1$ **then** 0 **else** $f(x+(i-1)h)$ **fi** +
if $i=0$ **then** 0 **else** $f(x+ih)$ **fi**)

Now, we can restrict the domain of *Trap* by changing the type of parameter n from **nat** to **inat** (a subtype of **nat**). After currying, we obtain a function having an array as result type.

Trap : ((**nat** $\rightarrow$ **nat**) $\times$ **nat** $\times$ **nat**) $\rightarrow$ (**inat** $\rightarrow$ **nat**)
Trap(f,x,y) = **let** $h = \frac{y-x}{n}$ **in** (**inat** j)**nat**:
$\frac{h}{2} \sum_{i=0}^{j}$ (**if** $i<1$ **then** 0 **else** $f(x+(i-1)h)$ **fi** +
if $i=0$ **then** 0 **else** $f(x+ih)$ **fi**)

$\equiv$ [fold *zipw*]

zipw($\cdot$,(**inat** j)**nat**: $h/2$,(**inat** j)**nat** :
$\sum_{i=0}^{j}$ (**if** $i<1$ **then** 0 **else** $f(x+(i-1)h)$ **fi** +
if $i=0$ **then** 0 **else** $f(x+ih)$ **fi**))

$\equiv$ [fold *new*; fold *prefix*]

zipw($\cdot$, *new*($h/2$), *prefix*(+, (**inat** i)**nat**:
if $i<1$ **then** 0 **else** $f(x+(i-1)h)$ **fi** +
if $i=0$ **then** 0 **else** $f(x+ih)$ **fi**))

$\equiv$ [fold *zipw*]

zipw($\cdot$, *new*($h/2$), *prefix*(+, *zipw*(+,
(**inat** i)**nat**: **if** $i<1$ **then** 0 **else** $f(x+(i-1)h)$ **fi**,
(**inat** i)**nat**: **if** $i=0$ **then** 0 **else** $f(x+ih)$ **fi**)))

$\equiv$ [define $F' \equiv$ (**inat** k)**nat**: $f(x+kh)$]

zipw($\cdot$, *new*($h/2$), *prefix*(+, *zipw*(+,
(**inat** i)**nat**: **if** $i<1$ **then** 0 **else** $F'(i-1)$ **fi**,
(**inat** i)**nat**: **if** $i=0$ **then** 0 **else** $F'(i)$ **fi**)))

$\equiv$ [fold *shiftr*; fold *update*]

zipw($\cdot$, *new*($h/2$), *prefix*(+, *zipw*(+, *shiftr*(F',1,0), *update*(F',0,0))))

The expression F' can be easily transformed into skeleton form:

(**inat** k)**nat** : $f(x+kh)$

$\equiv$ [fold, 6x]

$map(f, zipw(+, new(x), zipw(\cdot, inew(), new(h))))$

Altogether, we have a parallel program in skeleton form

$Trap : ((\mathbf{nat} \to \mathbf{nat}) \times \mathbf{nat} \times \mathbf{nat}) \to (\mathbf{inat} \to \mathbf{nat})$
$Trap(f, x, y) = \mathbf{let}\ h = \frac{y-x}{n}\ \mathbf{in}$
$\quad zipw(\cdot, new(h/2), prefix(+, zipw(+, shiftr(F',1,0), update(F',0,0))))$
$\mathbf{where}\ F' : \mathbf{inat} \to \mathbf{nat}$
$\quad F' = map(f, zipw(+, new(x), zipw(\cdot, inew(), new(h))))$

As an illustration of the simplicity of implementing skeletons, take the processors DP_i to be simple stack processors, i.e., the memories DM_i are stacks. We assume the following operations are available on DP_i (one stack s, values received in buffers *bufl* and *bufr*).

Operation	informal description
START	empty s
PUSH x	s := push(x,s)
APPLY1 f	s := push(f(top(s)),pop(s))
APPLY2 $\oplus$	s := push(top(s) $\oplus$ top(pop(s)), pop(pop(s)))
COPY	s := push(top(s),s)
INDEX	s := push(i,s) where i is index of this processor in array
LEFTRCV	s := push(*bufl*,s)
RIGHTSND	top(s) sent to right neighbor; s := pop(s)
RCVLEFT	boolean, *bufl* contains fresh value?
TOPIS x	boolean, top(s)=x?

The program derived above can be straightforwardly transliterated to the following machine code program:

```
START
PUSH h
INDEX
APPLY2 *
PUSH x
APPLY2 +
APPLY1 f
COPY
RIGHTSND
INDEX
IF TOPIS 0
THEN POP
     POP
     PUSH 0
ELSE POP FI
IF RCVLEFT
THEN LEFTRCV
ELSE PUSH 0 FI
APPLY2 +
{code for PREFIX +}
PUSH 2
PUSH h
APPLY2 /
APPLY2 *
```

6 Conclusions and research perspectives

We described a framework for the derivation of programs for arbitrary (in particular, parallel) architectures, motivated by a generalization of the derivation of sequential programs. The example derivations here and in the previous case studies [17, 12] show that deriving programs for a processor array is "just as easy" as deriving sequential programs, due to the single flow of instruction [13]. From these case studies, it may be concluded that the "skeleton method" is a promising and, indeed, powerful one. One important reason for this is the separation of concerns between deriving programs and implementing elementary computations that is induced by employing an intermediate language. Essential is also that the large amount of expertise that is already available on the derivation of "sequential" algorithms can be used to a large extent.

Apart from the ones mentioned above, case studies have been done by Harrison [14] on pipeline architectures, and by Pepper [18] on processor networks. Darlington et al. [10] give examples of skeletons for some more architectures. It is clear that more and larger case studies, both in describing more architectures and in deriving algorithms, should be done.

Central to a program development methodology is the formulation of development strategies (tactics, heuristics [11]). The available case studies of skeleton developments show large numbers of small steps, like in traditional unfold/fold developments [3], with little strategic information. Most of the steps are motivated by aspects of the envisaged solutions. Eventually, from these concrete motivations more abstract underlying principles of parallel-algorithm design should be extracted.

The case studies on SIMD architectures suggest a few candidates for strategic activities. In a SIMD architecture, all processors execute the same command simultaneously. Thus, it is important that function definitions are made uniform and total on their domain, such that the distribution of the domain over a number of processors leads to a low number of "borderline" cases with corresponding case introductions. Also, if the distribution of the domain is according to one of the several parameters of the function, a disentanglement of the parameters is required.

An important issue related to the question of strategies also seems to be the role of *data type transformations* [13]. Their application is twofold: on the one hand they provide a mechanism for inter-skeleton transformations and on the other hand they can map the problem data structure to the "data structure" of the architecture at hand. An example of the former can be found in [13], an example of the latter in [10]. In both cases covers can be profitably exploited. They provide the means to express problem data structures and architecture data structures in a unifying way. A related idea is that of *domain morphisms* [8].

Future research should also focus on the expansions of the model in two directions. First, the smallest piece of information should be larger than an **inat**, e.g. a segment, such that the number of PEs available is less a restriction. Second, the type of architectures should be broadened from SIMD to other data-parallel models, e.g. message-passing. A last observation is that skeletons, exactly because they represent basic computations, may be assigned computation costs, based on the costs of their components [22]. Based on this, formal complexity considerations may even be usable as a practical guideline in stepwise derivation of parallel programs.

Acknowledgements

For the main example in this paper we would like to thank Paul Frederiks. We would like to thank Klaus Achatz and Hans Meijer for their comments on this paper.

References

[1] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

[2] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, *Lecture Notes in Computer Science* **183**. Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design. NATO ASI Series Vol. F36*, pages 5–42. Springer-Verlag, Berlin, 1987.

[5] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[6] E.A. Boiten, H.A. Partsch, D. Tuijnman, and N. Völker. How to produce correct software – an introduction to formal specification and program development by transformations. *The Computer Journal*, 35(6):547–554, 1992.

[7] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, Berlin, 1982.

[8] M. Chen and Y. Choo. Domain morphisms: A new construct for parallel programming and formalizing program optimization. Technical Report YALEU/DCS/TR-817, Department of Computer Science, Yale University, 1990.

[9] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, London, 1989.

[10] J. Darlington, A.J. Field, P.G. Harrison, D. Harper, G.K. Jouret, P.J. Kelly, K.M. Sephton, and D.W. Sharp. Structured parallel functional programming. In H. Glaser and P. Hartel, editors, *Proc. Third International Workshop on the Implementation of Functional Languages on Parallel Architectures*, pages 31–51, 1991.

[11] M.S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Proc. IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation*, pages 165–196. North-Holland Publishing Company, Amsterdam, 1987.

[12] P.J.M. Frederiks. From applicative specifications to parallel programs. Master's thesis, Katholieke Universiteit Nijmegen, 1992.

[13] A.M. Geerling. Two examples of parallel-program derivation: Parallel-prefix and matrix multiplication. Research Report DoC 92/33, Imperial College London, 1992.

[14] P.G. Harrison. Towards the synthesis of static parallel algorithms: a categorical approach. In B. Möller, editor, *Proc. IFIP TC2 Working Conference on Constructing Programs from Specifications*, pages 49–69. North-Holland Publishing Company, Amsterdam, 1991.

[15] L.G.L.T. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, *CWI Monographs* **1**, pages 289–334, 1986.

[16] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer-Verlag, Berlin, 1990.

[17] H. Partsch. Some experiments in transforming towards parallel executability. In [20].

[18] P. Pepper. Deductive derivation of parallel programs. In [20].

[19] P. Pepper, J. Exner, and M. Südholt. Functional development of massively parallel programs. In *Formal Methods in Programming and Their Applications*, *Lecture Notes in Computer Science*, to appear 1993.

[20] R. Paige, J. Reif, and R. Wachter, editors. *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic Publishers, to appear 1993.

[21] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.

[22] D.B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 20:205–221, 1993.

[23] Y.V. Srinivas. A sheaf-theoretic approach to pattern matching and related problems. *Theoretical Computer Science*, 112:53–97, 1993.