Formal Derivation of SIMD Parallelism from Non-Linear Recursive Specifications

A. Max Geerling (max@cs.kun.nl)
Computing Science Institute, University of Nijmegen, Toernooiveld 1, NL-6525 ED Nijmegen, The Netherlands

Abstract. This paper presents a strategy for deriving SIMD-parallel (Single-Instruction-Multiple-Data) programs in a transformational style by parallelisation of non-linear recursive specifications. Parallelism is expressed by means of so-called skeletons.

Linear array and hypercube architectures and their skeletons are discussed in some detail. The strategy described is illustrated by derivations, such as the transformation of a generic specification for divide & conquer algorithms to an efficient implementation on hypercube architectures.

This paper links parallel programming and formal program development — which aren’t seen together very often, although both subjects are widely studied — by presenting a strategy for the development of parallel programs by means of transformational programming techniques [13, 8].

Using the formalism CIP-L [3, 1], we adopt the skeleton-approach advocated in [6], [7], [16] and [2]; a skeleton being a higher-order function that has a straightforward interpretation as a typical elementary computation on a particular architecture.

Section 1 defines the skeletons used in this paper. The strategy of deriving parallel programs as perceived now is discussed in Section 2. A new rule presented in Section 3 leads to a simpler strategy. A variant of the new rule is discussed in Section 4, together with an application: divide & conquer algorithms on hypercube architectures. Finally, Section 5 draws conclusions and gives questions for further research. Proofs for all the transformation rules presented here and further details on theory and examples can be found in [11].

1 Skeletons for SIMD Architectures

In this paper we restrict ourselves to the class of SIMD architectures. These architectures have the property that all processing elements (PE) execute the same instruction at a time in a synchronised manner, and this instruction operates on different data elements. Our restriction does not stem from a particular interest in synchronous/systolic algorithms, but is a starting point to develop transformation rules for a larger class of architectures. Moreover, our goal is to develop methodological knowledge for the formal derivation of parallel programs.

Regarding our definition of skeletons, the most “typical elementary computation” on SIMD architectures is the single instruction operating on multiple data, which is represented by the core skeleton of these architectures:
Application of an operation to all elements

\[
\text{MAP}: (\alpha \to \beta \times (\text{nat} \to \alpha)) \to (\text{nat} \to \beta) \\
\text{MAP}(f, a) = \lambda i. f(a(i))
\]

where \( \text{inat} = (\text{nat} : l \leq j \leq h) \) is a sub-domain of the naturals, \( \alpha \) and \( \beta \) are type variables and hence, \( \text{inat} \to \alpha \) is an abstraction of a processor array. \( \text{MAP} \) takes an operation \( f \) and an array \( a \) and returns an array in which each element is the result of operation \( f \) applied to the corresponding element of \( a \). Other useful skeletons for processor arrays are (from [14]):

**Lower bound**

\[
\text{L}: (\text{inat} \to \alpha) \to \text{inat} \\
\text{L}(a) = l
\]

**Constant array**

\[
\text{NEW}: \alpha \to (\text{inat} \to \alpha) \\
\text{NEW}(v) = \lambda i. v
\]

**'Identity' vector**

\[
\text{INEW}: \text{inat} \to \text{inat} \\
\text{INEW} = \lambda i. i
\]

**Higher bound**

\[
\text{H}: (\text{inat} \to \alpha) \to \text{inat} \\
\text{H}(a) = h
\]

**Length of an array**

\[
\text{LEN}: (\text{inat} \to \alpha) \to \text{nat} \\
\text{LEN}(a) = H(a) - L(a) + 1
\]

**Pairing**

\[
\text{ZIP}: ((\text{inat} \to \alpha) \times (\text{inat} \to \beta)) \to (\text{inat} \to (\alpha \times \beta)) \\
\text{ZIP}(a, b) = \lambda i. (a(i), b(i))
\]

All SIMD architectures share these skeletons. An implementation for stack machines is given as an illustration in [2]. Individual types of architectures differ in their topology and thus in the possible patterns of communication.

### 1.1 Linear Array

The linear array architecture consists of an array of data processors \( \text{PE}_i \) \((l \leq i \leq h)\) each of which can communicate with its neighbours, and has a (small) local memory. A coordinating instruction processor starts up the computation and can send values to all PEs.

PEs in a linear array can send values to their left and right neighbour. Since all PEs have to execute the same instruction at a time communication is expressed in terms of shifting all elements one position to the left or to the right. Dummy or neutral elements can be introduced in a situation where not all PEs should communicate. The next skeleton allows communication of more than one step at a time, although only one step a time (i.e. \( d = 1 \)) is an elementary computation on these architectures.

**Linear shift of elements**

\[
\text{SHIFTR}: ((\text{inat} \to \alpha) \times \text{nat} \times \alpha) \to (\text{inat} \to \alpha) \\
\text{SHIFTR}(a, d, v) = \lambda i. \text{if } i - d \geq L(a) \text{ then } a(i - d) \text{ else } v \\
\text{defined}(\text{SHIFTR}(a, d, v)) \Rightarrow d \leq \text{LEN}(a)
\]

\(^1\) The defined-predicate [13] - assertion - expresses partiality of an operation. From an expression \( \text{defined}(f(x)) \Rightarrow C(x) \) it follows that \( \neg C(x) \Rightarrow \neg \text{defined}(f(x)) \), i.e. the application of \( f \) to an \( x \) that does not satisfy \( C \) evaluates to undefined. Note, however, that \( f(x) \) may be undefined for values \( x \) that satisfy \( C(x) \). In derivations, \( C(x) \) provides information about the arguments of \( f \).