More Types for Nested Data Parallel Programming

Manuel M. T. Chakravarty
School of Computer Science & Engineering
University of New South Wales
Sydney, Australia
chak@cse.unsw.edu.au

Gabriele Keller
Faculty of Information Technology
University of Technology, Sydney
Sydney, Australia
keller@socs.uts.edu.au

ABSTRACT

This paper generalises the flattening transformation—a technique for the efficient implementation of nested data parallelism—and reconciles it with main stream functional programming. Nested data parallelism is significantly more expressive and convenient to use than the flat data parallelism typically used in conventional parallel languages like High Performance Fortran and C*. The flattening transformation of Blelloch and Sabot is a key technique for the efficient implementation of nested parallelism via flat parallelism, but originally it was severely restricted, as it did not permit general sum types, recursive types, higher-order functions, and separate compilation. Subsequent work, including some of our own, generalised the transformation and allowed higher-order functions and recursive types. In this paper, we take the final step of generalising flattening to cover the full range of types available in modern languages like Haskell and ML; furthermore, we enable the use of separate compilation. In addition, we present a completely new formulation of the transformation, which is based on the standard lambda calculus notation, and replace a previously ad-hoc transformation step by a systematic generic programming technique. First experiments demonstrate the efficiency of our approach.

1. INTRODUCTION

Sisal successfully demonstrated that a thoroughly optimised implementation of a functional language can challenge conventional Fortran code on high-performance architectures [6]. It did so, however, by providing essentially the same sort of parallel programming model that its imperative rival prescribed. It is our aim to take the next step and to realise a drastically more convenient programming model whose efficient implementation requires transformation techniques that are difficult to automate in the presence of side-effecting computations. In concrete terms, we are concerned with the efficient implementation of *nested data parallelism*—a generalisation of flat data parallelism, which in turn is the model that is typically used in languages like High Performance Fortran and C* [10]. Nested data parallelism allows highly irregular parallel control and data structures, such as nested parallel loops, where the ranges of the instances of the inner loop dependent on the index value of the outer loop. Nested parallelism constitutes an attractive programming model and simplifies the implementation of important parallel algorithms, such as computations over sparse matrices as well as hierarchical n-body algorithms. Nevertheless, for fine-grained algorithms, it is usually easier to understand and implement than control parallelism; in addition, it comes with a language-based parallel cost model [4].

Todays most portable implementation technique for nested data parallelism is the *flattening transformation,* a program transformation that maps nested to flat parallelism and that was originally introduced by Blelloch and Sabot [5]. It leads to efficient implementations on a range of high-performance architectures in both the purely functional [7] and the imperative case [9]. Nevertheless, imperative languages face a serious problem here: Flattening is hard to automate in the presence of pointers and side effects [9]. Even special purpose extensions of imperative languages require serious restrictions [22]. This may ultimately confine imperative code to thread-based implementations of nested parallelism [2, 20], which currently seem to require shared-memory architectures—and certainly are not suitable for vector processors. Moreover, modern processors tend to include vector instructions—usually to speed up multimedia applications—that could be exploited by a flattening-based compiler. In summary, flattening is important, because nested data-parallelism is more expressive, whereas flat data-parallelism is easier to implement.

Originally, flattening suffered from a lack of support for advanced data types, such as general sum types, recursive types, and higher-order functions, and it did not allow separate compilation. In other words, programs were confined to operate over arrays containing basic types like integer numbers and tuples thereof; furthermore, the source code of the complete program—including all library modules—had to be available during compilation [5]. Subsequent research extended flattening to cover higher-order functions [23] and recursive types [16]. In this paper, we close the remaining gap to modern functional languages like Haskell and ML. We rephrase the transformation in the standard lambda calculus setting, extend it to the full range of algebraic types (in particular, user-defined sum types), and modify it such that it allows for separate compilation of program modules.

ICFP '00, Montreal, Canada.

The latter is a consequence of replacing a previously ad-hoc transformation step by a systematic generic programming technique, which was recently introduced by Hinze [12, 11]. First experiments with our implementation techniques on a Cray T3E led to absolute speedups when compared to a sequential implementation of the same algorithm in C.

In summary, the main contributions of this paper are the following:

- We rephrase the flattening transformation in a standard lambda calculus setting with free use of higher-order functions, currying, and lazy evaluation.
- We extend flattening to cover the full range of types supported in languages like Haskell and ML.
- We enable the use of separate compilation by virtue of expressing part of the transformation in a systematic generic programming framework.

From the perspective of sequential programming and denotational semantics, these extensions might not seem very spectacular. However, under the constraint of achieving an *efficient* data parallel implementation, they become formidable challenges. In particular, the last two have been recognised as being open problems for the last five years.[1] Overall, the results presented in this paper allow us to realise a flattening-based implementation of nested data parallelism as a conservative extension of a standardised language like Haskell or SML, instead of relying on special purpose languages, as it was necessary until now.

The reminder of the paper is structured as follows: Section 2 briefly discusses a data-parallel extension of Haskell and outlines the transformation techniques detailed in the rest of the paper. Section 3 introduces a lambda calculus supporting parallel arrays. Section 4 discusses a flattened representation of data types. Sections 5 and 6 present the core transformations of our extended flattening approach. Section 7 briefly discusses the handling of polymorphic functions under a separate compilation scheme and the problems induced by reference types in parallel functions. Finally, Section 8 reports some benchmark results, reviews related work, and concludes.

2. NESTED DATA PARALLELISM

The following brief introduction to nested data parallelism is based on an extension of Haskell by *parallel arrays*.[2] A parallel array is an ordered sequence of values that comes with a set of data parallel operations; moreover, any parallel array is distributed across the available processing nodes in the case where we execute the program on a distributed memory machine.

2.1 Haskell with Parallel Arrays

We denote the type of parallel arrays containing elements of type τ by $[|\tau|]$. We provide similar syntactic support for parallel arrays as for lists—in particular, $[|a_1, \ldots, a_n|]$ constructs a parallel array. List functions, such as *map* and *replicate*, have parallel counterparts distinguished by the suffix *"P"*, i.e., we have

$$\begin{array}{l} mapP \quad :: (\alpha \to \beta) \to [|\alpha|] \to [|\beta|] \\ \quad \text{— map a function over a parallel array} \\ replicateP :: Int \to \alpha \to [|\alpha|] \\ \quad \text{— create an array containing } n \text{ copies of a value} \end{array}$$

Furthermore, we use the infix operators $(!|)$ and $(+\!\!+\!\!+)$ to denote indexing and appending of parallel arrays.[3]

The essential *operational* difference between lists and parallel arrays is that collective operations, such as $mapP$, $filterP$, $replicateP$, and so on, execute in *parallel*. This holds even for nested uses of these operations—so,

$$mapP\ (mapP\ (+1))\ [|[|1,\ 2|],\ [|3,\ 4,\ 5|],\ [||],\ [|6|]|]$$

increments all numbers in the nested parallel array in a *single parallel step*. We can alternatively denote this computation by using an array comprehension:

$$[|x + 1 \mid x \leftarrow xs, xs \leftarrow [|[|1, 2|], [|3, 4, 5|], [||], [|6|]|]|]$$

The parallel semantics entails a couple of significant differences between lists and parallel arrays: (1) Parallel arrays are always finite; (2) the construction of a parallel array is strict in all elements; and (3) there are no constructors (such as [] and *Cons*) for construction and pattern matching. The third point is not absolutely necessary, but it encourages array manipulation by collective operations, which are required for expressing data parallelism. More background on this style of parallel programming is, for example, provided by Blelloch [4], who also discusses the two examples presented next.

2.2 Using Nested Data Parallelism

An important irregular data structure in high-performance computing are sparse matrices. If there is no extra information regarding the structure of a sparse matrix, it can be stored efficiently in the so-called compressed row format, which represents all non-zero elements of a matrix row in a list of column-index/value pairs—a list of such sparse rows comprises a sparse matrix:

$$\begin{array}{l} \textbf{type}\ SparseRow\ \ = [|(Int,\ Float)|] \quad \text{— index, value} \\ \textbf{type}\ SparseMatrix = [|SparseRow|] \end{array}$$

Now consider the multiplication of a sparse matrix with a dense vector, resulting in another dense vector. It can be expressed by nesting three levels of parallel operations:

$$\begin{array}{l} smvm \quad :: SparseMatrix \to [|Float|] \to [|Float|] \\ smvm\ sm\ vec = \\ \quad [|sumP\ \underbrace{[|x * (vec\,!|\,col) \mid (col,\ x) \leftarrow row|]}_{\text{products of one row}} \mid row \leftarrow sm|] \end{array}$$

The inner array comprehension computes all products of a single row of the matrix, $sumP$ adds the products in a parallel reduction, and the outer comprehension specifies that the products and sums for all rows should be computed in parallel. Overall, the complete matrix-vector multiplication has a parallel depth complexity proportional to the logarithm of the longest row (cf., [4] for details). It is possible to achieve the same behaviour in a language supporting only

[1] There was indeed a Usenet posting in which Guy Blelloch encouraged to look into these problems.

[2] A corresponding extension as well as the implementation outlined in the remaining sections applies also to other typed functional languages like Standard ML or Clean.

[3] We cannot use the simpler symbols (!) and (+), as they are already used in Haskell.

flat data-parallelism, but the code is significantly more involved.

As a second example of a highly irregular data parallel algorithm, let us consider quicksort. The following definition of *qsort* holds little surprise for a functional programmer:

```
qsort    :: Ord α ⇒ [|α|] → [|α|]
qsort xs = if nullP xs then
             [||]
           else
             let
               m      = xs !| (lengthP xs `div` 2)
               ss     = [|s | s ← xs, s < m|]
               ms     = [|s | s ← xs, s == m|]
               gs     = [|s | s ← xs, s > m|]
               sorted = [|qsort xs' | xs' ← [|ss, gs|]|]
             in
             (sorted !| 0) +++ ms +++ (sorted !| 1)
```

There is, however, one peculiarity. The recursive calls to *qsort* are performed in an array comprehension ranging over a nested array structure. Given the parallel semantics of array comprehensions, the two recursive calls are executed in parallel—this would *not* be the case if we were to write *qsort ss* +++ *ms* +++ *qsort gs*!

The parallelism in *qsort* is obviously highly irregular and depends on the initial ordering of the array elements; nevertheless, the flattening transformation can rewrite the above definition of *qsort* into a flat data parallel program. Thus, in principle, it would be possible to achieve the same parallel behaviour in Fortran—it is, however, astonishingly tedious. Hierarchical n-body codes, such as the Barnes-Hut algorithm [1], exhibit a similar parallel structure as *qsort* and there is considerable interest in their high-performance implementation. We showed that rose trees of the form

```
data Tree = Node Particle [|Tree|]
```

lend themselves to a particularly elegant nested data parallel implementation of the Barnes-Hut algorithm, which our program transformations can compile into efficient code [16].

In addition to the obvious uses of sum types, the extension of flattening to the full range of Haskell's types, as presented here, allows a declarative type-directed control of data distribution. Consider the operational implications for an array of arrays $[|[|Int|]|]$ versus an array of (sequential) lists $[|[Int]|]$. On a distributed memory machine, the integer elements of the former will be evenly distributed over the available processing elements; in particular, if the subarrays vary substantially in size, they may be split up across processor boundaries. Consequently, the potential parallelism over the nested array is maximised. In contrast, arrays of lists are optimised for sequential operations over the sublist; nevertheless, the sequential processing of all the sublists is expected to proceed in parallel. In other words, we have a parallel operation consisting of a number of sequential steps inside. Overall, the choice between different mixes of parallel and sequential structures allows the programmer to influence the granularity of parallel operations, which is known to help implementing efficient parallel programs.

2.3 Compilation by Transformation

Figure 1 outlines the structure of our compilation system. The present paper is concerned with the first two transformation steps in the grey box: vectorisation and specialisation of array primitives. Together, these two steps realise the flattening transformation, i.e., they transform parallel computations given in the nested data-parallel style into equivalent computations expressed in the less expressive, but easier to implement flat style.

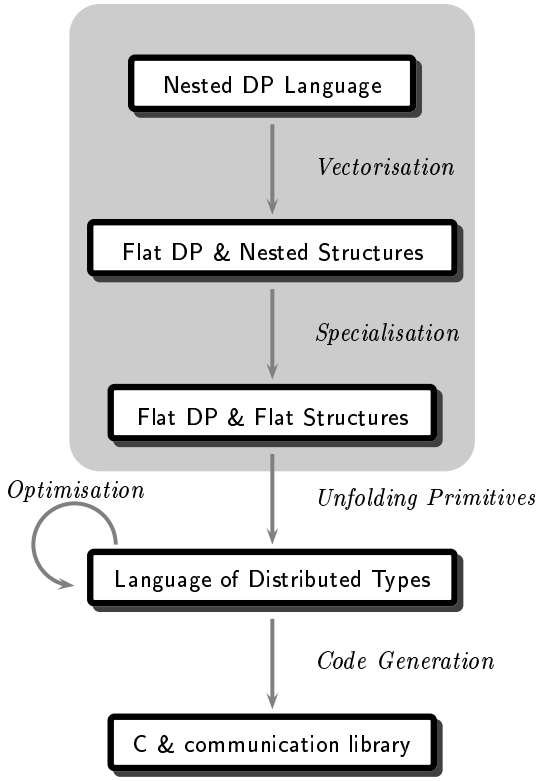

Figure 1: Outline of the compilation system

To gain an intuitive understanding of the purpose of the first two transformation steps, let us reconsider the initial example of this section:

```
mapP (mapP (+ 1)) [|[|1, 2|], [|3, 4, 5|], [||], [|6|]|]
```

The purpose of the first step of flattening—i.e., of vectorisation—is to provide an alternative, parallel version of each scalar function occurring in the program; for example, any scalar function over integers is vectorised to a function that executes the same operation elementwise over parallel arrays of integers. In other words, a function f is pointwise *lifted* into vector space, which we denote by $f^{\uparrow}$. Primitive functions, like +, are already provided by the system in a vectorised form. Overall, we can represent our example by

```
let
  xss = [|[|1, 2|], [|3, 4, 5|], [||], [|6|]|]
in
xss +↑↑ subdivideP (replicateP 6 1) [|2, 3, 0, 1|]
```

The expression *replicateP* 6 1 gives us $[|1, 1, 1, 1, 1, 1|]$. The array $[|2, 3, 0, 1|]$ captures the nesting structure of *xss*, i.e., it equals $[|lengthP\ xs \mid xs \leftarrow xss|]$—we call it the *segment descriptor* of *xss* as it specifies the segmentation of *xss* into subarrays. The auxiliary function *subdivideP* imposes the given segmentation structure on the array passed as the first argument. Finally, $+^{\uparrow\uparrow}$ adds the two arrays elementwise in parallel.

At this point, it might seem as if we could simplify the resulting code by introducing an auxiliary function $inc^{\uparrow}$ instead of using *replicateP* in combination with $+^{\uparrow}$. However, we then also have to lift the definition $inc = (+1)$, which will contain *replicateP* and $+^{\uparrow}$ in exactly the same way as we used these two functions above. So, nothing is gained. We will discuss the details of the vectorisation transformation in Section 5. We do this for the first time in a framework that uses a conventional lambda calculus notation and allows currying and lazy evaluation.

So far, it might appear as if we would need all primitive and user-defined functions lifted to parallel arrays of arbitrary nesting depth. A central insight of [5] is that this is not necessary and that indeed elementwise and once lifted versions of all function suffice. Moreover, it is advantageous for parallel execution to separate the structure of each parallel data structure from its contents. In the above example, it is useful to separate the segment descriptor of *xss* from the integer values of the nested array. We realise this by a transformation that implements all parallel arrays of complex types by parallel arrays of integers and floating point numbers together with sequential structures (like products) that combine a set of these primitive arrays into a compound structure. Operations on arrays of complex types are specialised correspondingly. Our example, can be represented as

let
$\quad xss\ =\ ((4,\ [\![2,\ 3,\ 0,\ 1]\!]),\ (6,\ [\![1,\ 2,\ 3,\ 4,\ 5,\ 6]\!]))$
in
$(fst\ xss,\ (snd\ (fst\ xss),\ snd\ (snd\ xss)\ +^{\uparrow}\ replicateP\ 6\ 1))$

All structure manipulations, such as *subdivideP* are replaced by simple projections and $+^{\uparrow}$ is needed only in its once lifted form. The details of the concrete representation of arrays of complex types shall be given in Section 4 and the specialisation of operations on these arrays in Section 5. Compared to previous work, we extended the concrete representation to cover sum types; moreover, we define the specialisations of arrays and array operations for the first time as type-indexed definitions in a generic programming framework. As a result, specialisations is now also possible in the presence of separate compilation.

Whilst the fully flattened program exhibits parallelism in a form suitable for a wide range of parallel architectures, the code is usually rather fine-grained and completely lacks any consideration for exploiting the memory hierarchies of modern processor architectures. We use a combination of an intermediate language that makes distribution explicit in the type system and aggressive deforestation techniques to produce code that performs well on networked, cache-based processors [17, 15]—this is handled by the transformation steps following the grey box in Figure 1 and not further discussed in the present paper.

Currently, the only alternative to using the flattening transformation for the implementation of general nested parallelism are thread-based compilation techniques [2, 20]. Due to the very fine granularity of the outlined style of nested parallel programming, it is however unclear how a thread-based approach can be efficiently implemented on distributed-memory architectures. Moreover, flattened code is suitable for multimedia vector architectures and can exploit the vector instructions that are usually included into modern high-performance processors.

2.4 Restrictions

Throughout this paper, we impose one restriction on the use of parallel arrays by the user: The elements of a parallel array may not contain functionals. We do this for two reasons:

- Functions in parallel arrays imply control parallelism. So for example, the expression $[\![f\ a \mid f \leftarrow [\![foo,\ bar]\!]]\!]$ would obviously require us to execute the evaluation of *foo a* and *bar a* in parallel and breaks the paradigm of data parallelism.
- If we execute any seemingly control parallel code sequentially, the addition of functionals in parallel arrays would significantly complicate the technical part of this paper without bringing any obvious gain.

Nevertheless, the presented program transformations will introduce arrays of functions, but they will always contain repetitions of the same function, so that we can guarantee data parallel execution.

Flattening in its current form is geared towards handling irregular parallelism, at the expense of not optimising regular parallelism. It would be preferable to recognise regular structures and treat them specially. Particularly interesting seem shape-based approaches, e.g., those of SAC [25] and GoldFISh [14], as flattening already distinguishes between shape and data.

3. A LAMBDA CALCULUS WITH PARALLEL ARRAYS

We formalise our extended flattening transformation based on a simply-typed lambda calculus supporting parallel arrays; we call the calculus λ_{PA}. We exclude parametrised types and *mutual* type recursion from the calculus to reduce notational noise, but briefly discuss these features in Section 7.1. The calculus λ_{PA} has the syntactic structure formalised in Figure 2(a).

The construction of type terms is standard and includes boolean values (Bool) and integers (Int) as an example for primitive types;[4] the symbol () denotes the unit type. Our only extension to the type structure are parallel arrays, denoted $[\![\tau]\!]$, where τ is the element type. We will see later that, for the purpose of the concrete implementation, we can regard $[\![\tau]\!]$ as a *typed-indexed type,* i.e., a type assuming different concrete representations in dependence on the type parameter τ [11]. It will be notationally convenient to distinguish parallel arrays indexed by Bool and Int, i.e., $[\![\mathsf{Bool}]\!]$ and $[\![\mathsf{Int}]\!]$ from the concrete representation of an array consisting of Bool and Int values. Therefore, we denote the latter by BoolArr and IntArr, respectively.

Expressions (or lambda terms) are typed and make value recursion explicit. Variables are explicitly typed—for example, the expression $\lambda v^{\mathsf{Int}}.e$ denotes a function over integers—but we often omit type superscripts if they are clear from the context. We assume a set C of constants and primitive operations. The symbol () is overloaded to denote a constant representing the single element contained in the unit type (); furthermore, C contains the usual arithmetic

[4] In the presence of sum types, defining Bool as a basic type may seem redundant. However, we need arrays of Bool in the flattened target representation of sum types and, thus, require a basic type Bool to avoid a cyclic definition.

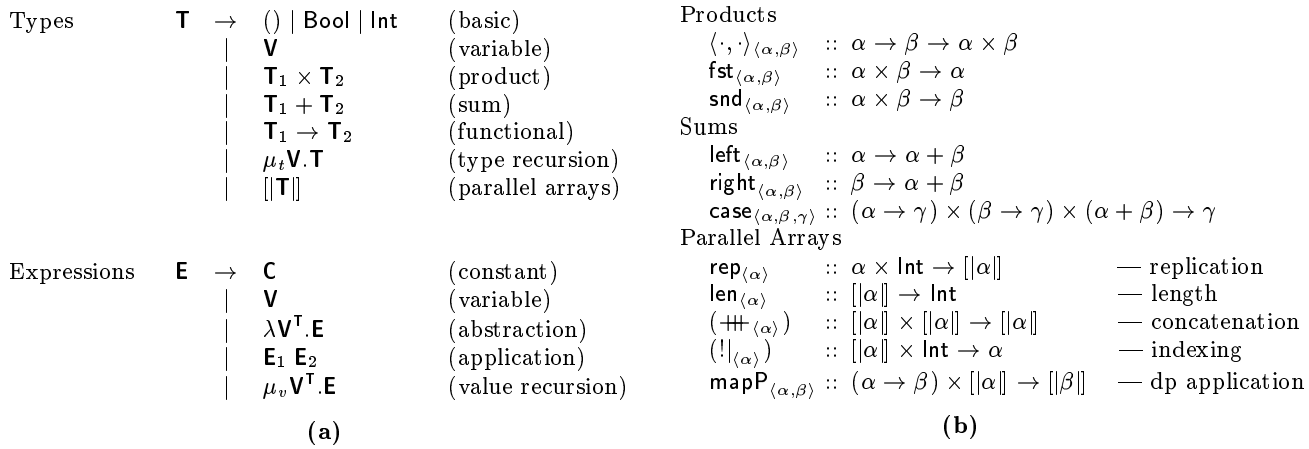

Figure 2: Grammar and primitives of λ_{PA}

and logic operations on boolean and integer values. Some of the primitive operations are type indexed, i.e., parametrised with a type—we will later see that they are not simply polymorphic. Figure 2(b) displays a list of λ_{PA}'s primitive operations. Note that we abbreviate *replicateP* to rep and *lengthP* to len. Moreover, we distinguish concrete functions, such as $(+\!\!+\!\!+_{\mathsf{Int}}) :: \mathsf{IntArr} \to \mathsf{IntArr} \to \mathsf{IntArr}$, from their type-indexed counterparts, such as $(+\!\!+\!\!+_{\langle\mathsf{Int}\rangle}) :: [|\mathsf{Int}|] \to [|\mathsf{Int}|] \to [|\mathsf{Int}|]$. We often drop the type index when it is clear from the context and we usually write $(+\!\!+\!\!+)$ and $(!|)$ infix and use Haskell's sectioning rules. We also use $(\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2) = (\lambda v.e_2)\ e_1$. Although, for an efficient implementation of a wide range of data parallel algorithms, we need some more functions (like permutations, reductions, and scans), we omit them here as their inclusion would not lead to any additional technical insights. See, for example, [15] for a comprehensive discussion of the required functions and their parallel implementation. Regarding mapP, we assume that the restrictions outlined in Subsection 2.4 are met.

We will neither formalise the type system nor the semantics of λ_{PA} due to the limited space and as both follow the standard definitions discussed in textbooks, such as [19].

4. FLATTENED PARALLEL STRUCTURES

Parallel operations over flat arrays of primitive data types (such as integer and floating-point numbers) can be performed particularly easily and efficiently, such arrays are the preferred data structure in Fortran programs for exactly this reason. Hence, parallel processing encourages the decomposition of complex data structures such that the structure information is separated from the primitive data values that are stored in the structure [5, 23, 14]. We call this a *flattened* data structure representation. In previous work [16], we demonstrated that the program transformation that replaces the source-level representation by the flattened representation is best understood as a specialisation procedure of the primitive functions operating on these data structures. This insight allowed us to find an efficient flattened representation for recursive data structures, and thus, to apply the flattening transformation to important algorithms, such as hierarchical n-body codes.

In Section 6, we shall elaborate on this specialisation procedure. However, first we will discuss, in the present section, a flattened representation of parallel arrays; and then, in Section 5 the vectorisation transformation, which generates for each function in the program a vectorised variant operating on a whole array of operands.

4.1 Separation of Structure and Data

Our flattened representation of parallel structures obeys two principles:

1. Parallel arrays of complex types are represented by a complex type that has parallel arrays of primitive type at its leafs.
2. Functions occur in two variants: in a sequential and in a vectorised form.

The first principle was used by Blelloch & Sabot [5] for a restricted set of data structures: An array of pairs is represented by a pair of arrays and an array of arrays is represented by a segment descriptor and a flat array. The segment descriptor encodes how the elements of the flat array are to be partitioned, for example,

$$[[|[1,2],[3,4,5]|],[|[|],[|6|]|]] \implies \begin{cases} [|2,3,0,1|] & \text{segd} \\ [|1,2,3,4,5,6|] & \text{data} \end{cases}$$

We previously pointed out that we can use the same idea to represent recursive structures, as long as we use a segment descriptor at each level of the recursion [16]. The flattened representation of a rose tree (cf. Subsection 2.2), then, becomes a list whose length equals the depth of the represented tree, where a segment descriptor in each list node specifies the branching structure of the tree. However, the most difficult point here is not so much the flattened representation, but specialisation of the primitive operations over parallel arrays to the flattened structure. We will return to this topic in Section 6.

Interestingly, we can apply the principle also to sum types. Given an array of a sum, $[|\tau_1 + \tau_2|]$, we can represent it by one array for each of its components τ_1 and τ_2 plus an additional array—called the *selector*—that determines in which order the component elements are placed in the compound array. We encode the latter by an array of boolean values, where false corresponds to elements from type τ_1 and true to

elements from type τ_2. As an example consider the following parallel array of sequential lists:

$$[\![[], [1,2], [], [], [3]]\!]$$
$$\Longrightarrow \langle [\![\mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}]\!], \langle [\![[], [], []]\!], [\![[1,2], [3]]\!]\rangle\rangle$$

Sequential lists are formed from a sum type, which distinguishes the two alternative constructors [] (nil) and *Cons*. Hence, all empty lists are collected into one array and all non-empty lists in the other array. This process is recursively repeated for the array holding the non-empty lists. The process terminates after as many steps as the length of the longest list. The last node of the flattened representation has the form $\langle [\![]\!], \langle \bot, \bot\rangle\rangle$, where $\bot$ represents an undefined value (see also Section 6.1). The empty selector indicates the termination of the recursive structure, which means that the contents of the two substructures is irrelevant—none of the array processing operations will touch these components when they encounter an empty selector. Figure 3(a) defines the representation of type-indexed array types by primitive arrays.

Finally, recognising the second of the previously stated two principles, we replace any function by a pair consisting of the original function and a vectorised variant of that function. Roughly speaking, whenever a function is applied, the original function or its vectorised version is used in dependence on whether the application is within a sequential or a data parallel computation. Within the vectorised version of a function, all other functions are just represented by their vectorised form—the sequential form is never needed.

The definitions in Figure 3 combine all these rules. Part (a) represents arrays of compound types by compound types of simple integer and boolean arrays. Furthermore, for a given type τ, $\mathcal{F}[\![\tau]\!]$ from Part (b) yields a representation of that type, where all functions are represented by pairs comprised out of their normal and vectorised variant. Outside of parallel arrays, we merely replace functions by pairs of functions, where the vectorised variant forms the second component. The type transformation $\mathcal{F}[\![\cdot]\!]$ does not explicitly cover the case of parallel arrays as Part (a) eliminates all $[\![\cdot]\!]$ constructors and there cannot be any enclosed functions due to the restriction discussed in Section 2.4.

By virtue of the mapping $\mathcal{F}[\![\cdot]\!]$ it is easy to see that the type $[\![\mu_t tree.\mathsf{Int} \times [\![tree]\!]]\!]$—a parallel forest of rose trees storing integers—is represented by

$$\mu_t tree.\underbrace{(\mathsf{Int} \times [\![\mathsf{Int}]\!])}_{\langle \text{nodes}\rangle} \times (\underbrace{\mathsf{Int} \times [\![\mathsf{Int}]\!]}_{\langle \text{segd}\rangle} \times tree)$$

where $\langle$nodes$\rangle$ represents the node values of one level of the forest and $\langle$segd$\rangle$ the segment descriptor partitioning the next level of the forest into subtrees. This structure is, in effect, well suited for the parallel implementation of the spatial decomposition trees that are, for example, frequently used in hierarchical n-body algorithms [16].

In λ_{PA}, the previously mentioned type of parallel arrays containing sequential lists, $[\![[\alpha]]\!]$, can be formalised straight forward as $[\![\mu_t list.() + (\alpha \times list)]\!]$. According to Figure 3, it has the following flattened representation:

$$\mu_t list.(\underbrace{(\mathsf{Int} \times [\![\mathsf{Bool}]\!])}_{\langle \text{selector}\rangle} \times (\mathsf{Int} \times ([\![\alpha]\!] \times list)))$$

Generally, given a function $f :: \tau \to \sigma$, the correspondence between the flattened representations $\mathcal{F}[\![\tau]\!]$ and $\mathcal{F}[\![\sigma]\!]$ of the types and the vectorisation and specialisation transformations presented in the next two sections is as depicted in the following commuting diagram:

$$\begin{array}{ccc} \tau & \xrightarrow{\quad f \quad} & \sigma \\ {\scriptstyle \mathcal{F}[\![\cdot]\!]}\downarrow & & \downarrow{\scriptstyle \mathcal{F}[\![\cdot]\!]} \\ \mathcal{F}[\![\tau]\!] & \xrightarrow[\mathsf{fst}(\mathcal{V}[\![f]\!])]{} & \mathcal{F}[\![\sigma]\!] \end{array}$$

We will discuss the vectorisation transformation $\mathcal{V}[\![\cdot]\!]$ in the following section.

5. VECTORISATION OF FUNCTIONS

We already noted that our formulation of the flattening transformation consists of two steps: (1) Vectorisation of functions and (2) specialisation of primitive operations to the concrete representation of parallel data structures. These two steps are discussed in the current and the following section, respectively.

Vectorisation requires us to replace every function f of the original program by a pair $\langle f', f^\uparrow\rangle$, where all local functions in f' did undergo the same transformation. Furthermore, $f^\uparrow$ is f *lifted* pointwise into vector space, i.e., it is a variant of f that can perform f on all elements of a parallel array in a single parallel step—see also Subsection 2.3. Let us consider, for example, the source expression $[\![x + y \mid x \leftarrow e]\!]$ (note that y is free), which in λ_{PA} becomes

$$\mathbf{let}\, f = \lambda x^{\mathsf{Int}}.x + y\ \mathbf{in}\ \mathsf{mapP}\ f\ e$$

Let us assume that in the target language, the primitive operation $(+)$ has type $\langle \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}, [\![\mathsf{Int}]\!] \times [\![\mathsf{Int}]\!] \to [\![\mathsf{Int}]\!]\rangle$, i.e., it is a pair of scalar addition and elementwise vector addition. Then, the lifted variant $f^\uparrow$ of the abstraction bound to f would be $\lambda x^{[\![\mathsf{Int}]\!]}.(\mathsf{snd}\ (+))\ \langle x, \mathsf{rep}\ (\mathsf{len}\ x)\ y\rangle$—i.e., vectorised addition (denoted by $(\mathsf{snd}\ (+))$) is applied to the array argument x and a sufficient number of copies of y (the value of y is copied, because it is free, i.e., a constant scalar in this example).

Previous formalisations of the flattening transformation did only handle named functions, and thus, distinguished between scalar and lifted variants of functions using names, such as f and $f^\uparrow$. In the present setting based on the lambda calculus, we cannot follow this approach; instead, we literally replace every lambda abstraction by a pair of functions. In our example, we thus get

$$\begin{array}{l} \mathbf{let} \\ \quad f = \langle \lambda x^{\mathsf{Int}}.(\mathsf{fst}\ (+))\ \langle x, y\rangle, \\ \qquad\quad \lambda x^{[\![\mathsf{Int}]\!]}.(\mathsf{snd}\ (+))\ \langle x, \mathsf{rep}\ (\mathsf{len}\ x)\ y\rangle\rangle \\ \mathbf{in} \\ (\mathsf{snd}\ f)\ e \end{array}$$

Note how the mapping of the scalar function is replaced by a direct application of the lifted variant. Unfortunately, we cannot simply replace mapP by snd in the case of nested occurrences of mapP—we will return to this point in the next section, for the moment let us just consider the lifting of functions. In this example, the scalar version of f is not needed anymore, but in general, a function may occur both in sequential and parallel contexts.

There is one more point that should be noted about the previous example. The variables x and y were treated in

$$
\begin{array}{lcl}
[\![()]\!] & = & \mathsf{Int} \\
[\![\mathsf{Bool}]\!] & = & \mathsf{Int} \times \mathsf{BoolArr} \\
[\![\mathsf{Int}]\!] & = & \mathsf{Int} \times \mathsf{IntArr} \\
[\![\tau_1 \times \tau_2]\!] & = & [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![\tau_1 + \tau_2]\!] & = & \mathsf{BoolArr} \times [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![\tau_1 \to \tau_2]\!] & = & [\![\tau_1]\!] \to [\![\tau_2]\!] \\
[\![[\![\tau]\!]]\!] & = & [\![\mathsf{Int}]\!] \times [\![\tau]\!]
\end{array}
$$

(a)

$$
\begin{array}{lcl}
\mathcal{F}[\![\cdot]\!] & :: & \mathsf{T} \to \mathsf{T} \\
\mathcal{F}[\![x]\!] \quad | \; x \in \mathsf{C} \cup \mathsf{V} & = & x \\
\mathcal{F}[\![\tau_1 \times \tau_2]\!] & = & \mathcal{F}[\![\tau_1]\!] \times \mathcal{F}[\![\tau_2]\!] \\
\mathcal{F}[\![\tau_1 + \tau_2]\!] & = & \mathcal{F}[\![\tau_1]\!] + \mathcal{F}[\![\tau_2]\!] \\
\mathcal{F}[\![\tau_1 \to \tau_2]\!] & = & (\mathcal{F}[\![\tau_1]\!] \to \mathcal{F}[\![\tau_2]\!]) \\
 & & \times ([\![\tau_1]\!] \to [\![\tau_2]\!]) \\
\mathcal{F}[\![\mu_t v.\tau]\!] & = & \mu_t v.\mathcal{F}[\![\tau]\!]
\end{array}
$$

(b)

Figure 3: (a) Flattened representation of parallel types and (b) pairing of normal with vectorised functions

different ways in the body of the lifted function. By lifting the function, the argument variable x becomes vector-valued (i.e., it is raised from type Int to $[\![\mathsf{Int}]\!]$). We call such variables *parallel variables*. In contrast, y is a *scalar variables*—it is not lifted into vector space.[5] Moreover, we call the number of values over which a parallel variable ranges, the *size of the parallel context*. In the previous example, the size of the parallel context is $\mathsf{len}e$, i.e., the number of times that the body of $[\![x + y \mid x \leftarrow e]\!]$ is evaluated.

Now, we are ready to formalise vectorisation. The function $\mathcal{V}[\![\cdot]\!]$ takes an expression and yields its vectorised form (x may either be a constant or a variable):

$$
\begin{array}{lcll}
\mathcal{V}[\![\cdot]\!] & :: & (\mathsf{E} : \sigma) \to (\mathsf{E} : \mathcal{F}[\![\sigma]\!]) & \\
\mathcal{V}[\![x]\!] & = & x & \\
\mathcal{V}[\![\lambda v^{\tau}.e]\!] & = & \langle \lambda v^{\mathcal{F}[\![\tau]\!]}.\mathcal{V}[\![e]\!], & \text{— sequential} \\
 & & \quad \lambda v^{[\![\tau]\!]}.e^{\uparrow[v]} \rangle & \text{— parallel} \\
\mathcal{V}[\![e_1\, e_2]\!] & = & (\mathsf{fst}\, \mathcal{V}[\![e_1]\!])\, (\mathcal{V}[\![e_2]\!]) & \\
\mathcal{V}[\![\mu_v v^{\tau}.e]\!] & = & \mu_v v^{\mathcal{F}[\![\tau]\!]}.\mathcal{V}[\![e]\!] &
\end{array}
$$

Note how types are transformed by the function $\mathcal{F}[\![\cdot]\!]$ flattening types, abstractions are represented by pairs of functions, and applications consistently choose the first—i.e., the original variant—of each abstraction. Usage of the lifted variants depends on the concrete definition of mapP, which we shall consider in the following section. First, we formalise the lifting process, where $e^{\uparrow vs}$ means lifting an expression e in a context of parallel variables vs.

$$
\begin{array}{lcl}
(\cdot)^{\uparrow vs} & :: & (\mathsf{E} : \sigma) \to (\mathsf{E} : [\![\sigma]\!]) \\
x^{\uparrow v:vs} \quad | \; x \in (v : vs) & = & x \\
\quad | \; otherwise & = & \mathsf{rep}\, (\mathsf{len}\, v)\, x \\
(\lambda v^{\tau}.e)^{\uparrow vs} & = & \lambda v^{[\![\tau]\!]}.e^{\uparrow (v:vs)} \\
(e_1\; e_2)^{\uparrow vs} & = & (e_1^{\uparrow vs})\, (e_2^{\uparrow vs}) \\
(\mu_v v^{\tau}.e)^{\uparrow vs} & = & \mu_v v^{[\![\tau]\!]}.e^{\uparrow vs +\!\!+ [v]}
\end{array}
$$

The first equation distinguishes between parallel and scalar variables. The former are treated as being a parallel array already, whereas the latter are replicated according to the size of the parallel context—which, as we see in the third equation, is the head of the list of parallel variables. The rule for recursion $\mu_v v.e$ puts the names of recursive function at the end of vs to avoid interfering with determining the size of the parallel context.

[5]The use of the word *scalar* might be a bit misleading. In fact, y could be an array-valued variable and we would still call it scalar. We are interested in whether or not a variable has different values in the parallel computation space of a lifted operation, not in whether it refers to a scalar value or an array.

At first sight, it might seem as if the rules do not work for functions f, when we have $f^{\uparrow vs}$ with $f \notin vs$. However, consider that $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]$, according to Figure 3, and that we yet have to define the specialisation of $\mathsf{rep}_{\langle \alpha \to \beta \rangle}$. As we will see in Subsection 6.1, we have

$$\mathsf{rep}_{\langle \alpha \to \beta \rangle}\, \langle n, f \rangle \;=\; \mathsf{snd}\, f$$

Due to the transformation performed by $\mathcal{V}[\![\cdot]\!]$, the value of f will be a pair whose second component contains the lifted variant of the function. In other words, we implement a vector of copies of one function by the vectorised version of that function—instead of applying a function multiple times in parallel to the different elements of a parallel array, we apply its vectorised counterpart once to that array as a whole.

6. SPECIALISATION OF PRIMITIVE OPERATIONS

As already mentioned, one of the central ideas of our approach to handling recursive data types in flattening is the use of a type-based specialisation procedure for the primitive operations of λ_{PA} [16]. In the following, we will improve our previous approach by formalising the definition of the primitive operations in the style of generic programming; consequently, we can, then, replace our previous ad hoc specialisation procedure by the systematic specialisation of Hinze's generic programming framework [12, 11]. This not only provides our approach with a formal foundation, but also allows us to flatten the full range of data types supported in languages like Haskell and ML. Moreover, the new specialisation technique is applicable in the presence of separately compiled modules. This was neither the case with our previous specialisation routine nor with other formulations of the flattening transformation, as in both cases all parametric polymorphism had to be removed from the program by generating appropriate instances. This is no longer the case as Hinze's transformation generates generic definitions that use function-valued arguments to enable their use on all instances of a parametrised type.

6.1 Operations on Parallel Structures

The most interesting operations are those on parallel arrays—in our definition of λ_{PA}, these are (rep), $(+\!\!+)$, $(!\,|)$, len, and mapP. In order to reach an efficient parallel implementation of these operations, we select the flattened form outlined in Section 4.1 as the concrete representation for parallel arrays. For an parallel array $[\![\tau]\!]$, this concrete representation depends on the concrete type τ; thus, we can regard $[\![\cdot]\!]$ as a type-indexed type [11]. Correspondingly, we understand

a primitive $p_{\langle\tau\rangle}$ operating on arrays $[|\tau|]$ as a type-indexed function. In other words, in dependence on the type index, we can choose an appropriate flattened representations for each parallel array and array primitive. This facilitates high-performance parallel execution, as the flattened representation consists of nothing but scalars, products, functions, and arrays of integers and booleans. In particular, the only concrete operations needed on parallel arrays are those operating over arrays of integers and booleans. All others are generated by a specialisation procedure from the generic definitions that we discuss in this section. As mentioned in Section 3, to separate the concrete operations over parallel arrays of integers and booleans notationally from the generic operations, we write them without the angle brackets—for example, we write $\mathsf{rep}_{\mathsf{Int}}$ instead of $\mathsf{rep}_{\langle\mathsf{Int}\rangle}$.

6.1.1 Data Parallel Application

We saw in Section 5 that—after vectorisation—the primitive mapP merely has to apply the second component of the pair that represents the mapped function. The situation is a little more involved in the case where applications of mapP are nested (as in the example from Subsection 2.3). In this case, the inner use of mapP itself is lifted and we have to ensure that we do not need any of the innermost functions in a form that is lifted more than once. In fact, already Blelloch & Sabot [5] pointed out that in the nested case, we can use once lifted functions if we ignore the additional structure information (i.e., segment descriptors) of the array-valued argument. Moreover, the result will require the same structure information as the argument. See [23, 18, 15] for a more detailed explanation.

Overall, we can define mapP by the following pair of functions, which represent the conventional and lifted variant, respectively:

$$
\begin{array}{lll}
\mathsf{mapP} & :: \mathcal{F}[\![(\alpha \to \beta) \times [|\alpha|] \to [|\beta|]\!] & \\
\mathsf{mapP}\ \langle f, a\rangle = & \langle \mathsf{snd}\ f\ a, & \text{— sequential} \\
& \ \langle \mathsf{fst}\ a, f\ (\mathsf{snd}\ a)\rangle\rangle & \text{— lifted}
\end{array}
$$

To understand the lifted variant, we have to take the representation of arrays of arrays as a pair of segment descriptor and data array into account and have to recognise that an array of functions is realised as a simple function over arrays (not as a pair of functions). Both facts are apparent from the definitions in Figure 3.

6.1.2 Replication

Generally, there are three class of operations over arrays: (1) those that increase the nesting depths, (2) those that maintain the nesting depths, and (3) those that reduce the nesting depths. The generic definitions of the members of each class are similar, and thus, we chose a member of each for inclusion into λ_{PA}.

Replication belongs to the first class—it takes a value of type τ to $[|\tau|]$. Like in the case of mapP, we need both a conventional and a lifted variant. To keep the definitions readable, we will define them independently, as rep^0 and rep^1, respectively, and imply $\mathsf{rep} = \langle \mathsf{rep}^0, \mathsf{rep}^1\rangle$. Furthermore, we will use an equational notation, instead of explicit lambda abstractions and recursion (the translation to plain λ_{PA} is straight forward).

$$
\begin{array}{lll}
\mathsf{rep}^0_{\langle\alpha\rangle} & :: \mathcal{F}[\![\alpha]\!] \times \mathsf{Int} \to \mathcal{F}[\![[|\alpha|]\!] & \\
\mathsf{rep}^0_{\langle()\rangle} & \langle n, x\rangle = n &
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{rep}^0_{\langle\mathsf{Int}\rangle} & \langle n, x\rangle = \langle n, \mathsf{rep}^0_{\mathsf{Int}}\ \langle n, x\rangle\rangle & \\
\mathsf{rep}^0_{\langle\mathsf{Bool}\rangle} & \langle n, x\rangle = \langle n, \mathsf{rep}^0_{\mathsf{Bool}}\ \langle n, x\rangle\rangle & \\
\mathsf{rep}^0_{\langle\alpha\times\beta\rangle} & \langle n, x\rangle = \langle \mathsf{rep}^0_{\langle\alpha\rangle}\ \langle n, \mathsf{fst}\ x\rangle,\ \mathsf{rep}^0_{\langle\beta\rangle}\ \langle n, \mathsf{snd}\ x\rangle\rangle & \\
\mathsf{rep}^0_{\langle\alpha+\beta\rangle} & \langle n, x\rangle = & \\
\quad \textbf{if}\ n == 0\ \textbf{then} & & \\
\quad\quad \langle \mathsf{rep}^0_{\langle\mathsf{Int}\rangle}\ \langle 0, 0\rangle, \langle \bot, \bot\rangle\rangle & & \\
\quad \textbf{else} & & \\
\quad\quad \mathsf{case} & & \\
\quad\quad\quad (\lambda x'.\langle \mathsf{rep}_{\mathsf{Bool}}\ \langle n, \mathsf{false}\rangle,\ \langle \mathsf{rep}^0_{\langle\alpha\rangle}\ \langle n, x'\rangle,\ \mathsf{rep}^0_{\langle\beta\rangle}\ \langle 0, \bot\rangle\rangle\rangle) & & \\
\quad\quad\quad (\lambda x'.\langle \mathsf{rep}_{\mathsf{Bool}}\ \langle n, \mathsf{true}\rangle,\ \langle \mathsf{rep}^0_{\langle\alpha\rangle}\ \langle 0, \bot\rangle,\ \mathsf{rep}^0_{\langle\beta\rangle}\ \langle n, x'\rangle\rangle\rangle) & & \\
\quad\quad\quad x & & \\
\mathsf{rep}^0_{\langle\alpha\to\beta\rangle} & \langle n, x\rangle = \mathsf{snd}\ x & \\
\mathsf{rep}^0_{\langle[|\alpha|]\rangle} & \langle n, x\rangle = \textbf{if}\ n == 0\ \textbf{then} & \\
& \quad \langle \mathsf{rep}^0_{\langle\mathsf{Int}\rangle}\ \langle 0, 0\rangle, \bot\rangle & \\
& \textbf{else} & \\
& \quad \mathsf{rep}^{C0}_{\langle\alpha\rangle}\ \langle n, x\rangle &
\end{array}
$$

Here, we use the symbol $\bot$ to denote subexpressions whose value is irrelevant—in a lazy language, this might indeed be a diverging value. In any case, the definition of the primitives ensures that these values are never touched. Furthermore,

$$\mathsf{rep}^C_{\langle\alpha\rangle} :: [|\alpha|] \times \mathsf{Int} \to [|[|\alpha|]|]$$

represents, so called, *chunkwise* distribution. It creates a parallel array from repeated occurrences of a shorter array. As its definition is tedious, but not very surprising, we omit it—see [16, 15] for details on chunkwise operations and why it is worthwhile to treat them specially on distributed memory machines.

Finally, here is the definition of the lifted variant of distribution:

$$
\begin{array}{lll}
\mathsf{rep}^1_{\langle\alpha\rangle} & :: \mathcal{F}[\![[|\alpha|]\!] \times \mathcal{F}[\![[|\mathsf{Int}|]\!] \to \mathcal{F}[\![[|[|\alpha|]|]\!] & \\
\mathsf{rep}^1_{\langle()\rangle} & \langle ns, xs\rangle = \langle ns, \mathsf{sum}\ ns\rangle & \\
\mathsf{rep}^1_{\langle\mathsf{Int}\rangle} & \langle ns, xs\rangle = \langle ns, \langle \mathsf{sum}\ ns, \mathsf{rep}^1_{\mathsf{Int}}\ \langle ns, xs\rangle\rangle\rangle & \\
\mathsf{rep}^1_{\langle\mathsf{Bool}\rangle} & \langle ns, xs\rangle = \langle ns, \langle \mathsf{sum}\ ns, \mathsf{rep}^1_{\mathsf{Bool}}\ \langle ns, xs\rangle\rangle\rangle & \\
\mathsf{rep}^1_{\langle\alpha\times\beta\rangle} & \langle ns, xs\rangle = & \\
\quad \langle ns, \langle \mathsf{rep}^1_{\langle\alpha\rangle}\ \langle ns, \mathsf{fst}\ xs\rangle,\ \mathsf{rep}^1_{\langle\beta\rangle}\ \langle ns, \mathsf{snd}\ xs\rangle\rangle\rangle & & \\
\mathsf{rep}^1_{\langle\alpha+\beta\rangle} & \langle ns, xs\rangle = & \\
\quad \textbf{let} & & \\
\quad\quad sel = \mathsf{fst}\ xs & & \\
\quad\quad left' = \mathsf{rep}^1_{\langle\alpha\rangle}\ \langle \mathsf{packP}\ \langle sel, ns\rangle,\ \mathsf{fst}\ (\mathsf{snd}\ xs)\rangle & & \\
\quad\quad right' = \mathsf{rep}^1_{\langle\beta\rangle}\ \langle \mathsf{packP}\ \langle \mathsf{not}^{\uparrow}\ sel, ns\rangle,\ \mathsf{snd}\ (\mathsf{snd}\ xs)\rangle & & \\
\quad \textbf{in} & & \\
\quad \langle ns, \langle \mathsf{rep}^1_{\mathsf{Bool}}\ \langle ns, selector\rangle,\ \langle left', right'\rangle\rangle\rangle & & \\
\mathsf{rep}^1_{\langle\alpha\to\beta\rangle} & \langle ns, xs\rangle = \mathsf{snd}\ xs & \\
\mathsf{rep}^1_{\langle[|\alpha|]\rangle} & \langle ns, xs\rangle = \mathsf{rep}^{C1}_{\langle\alpha\rangle}\ \langle ns, xs\rangle &
\end{array}
$$

The only difference in the structure of the result of the lifted from the conventional variant is that the former has an additional segment descriptor. In the case of distributions, this segment descriptor coincides with the value of the argument ns. The function sum is another primitive of the target language—it implements parallel reduction on arrays of integers. Furthermore, $\mathsf{packP} :: [|\mathsf{Bool}|] \times [|\alpha|] \to [|\alpha|]$ removes all elements from the second argument that correspond to values of false in the first argument—we do not formalise the definition of packP here, as its structure is not unlike that of concatenation (both operations preserve the nesting depth of the structures they operate over).

6.1.3 Concatenation

Next we define the append functions $(+\!\!+_{\langle\alpha\rangle})$ as an example of a primitive that maintains the nesting of its arguments.

$$
\begin{array}{ll}
(+\!\!+_{\langle\alpha\rangle}) & :: \mathcal{F}[\![[|\alpha|]]\!] \times \mathcal{F}[\![[|\alpha|]]\!] \to \mathcal{F}[\![[|\alpha|]]\!] \\
xs +\!\!+_{\langle()\rangle}\ ys & = xs + ys \\
xs +\!\!+_{\langle\mathsf{Int}\rangle}\ ys & = \langle \mathsf{fst}\ xs + \mathsf{fst}\ ys, \mathsf{snd}\ xs +\!\!+_{\mathsf{Int}} \mathsf{snd}\ ys\rangle \\
xs +\!\!+_{\langle\mathsf{Bool}\rangle}\ ys & = \langle \mathsf{fst}\ xs + \mathsf{fst}\ ys, \mathsf{snd}\ xs +\!\!+_{\mathsf{Bool}} \mathsf{snd}\ ys\rangle \\
xs +\!\!+_{\langle\alpha\times\beta\rangle} ys & = \langle \mathsf{fst}\ xs +\!\!+_{\langle\alpha\rangle} \mathsf{fst}\ ys,\ \mathsf{snd}\ xs +\!\!+_{\langle\beta\rangle} \mathsf{snd}\ ys\rangle \\
xs +\!\!+_{\langle\alpha+\beta\rangle} ys & = \\
\quad \langle \mathsf{fst}\ xs +\!\!+_{\langle\mathsf{Bool}\rangle} \mathsf{fst}\ ys, & \\
\quad\ \langle \mathsf{fst}\ (\mathsf{snd}\ xs) +\!\!+_{\langle\alpha\rangle} \mathsf{fst}\ (\mathsf{snd}\ ys), & \\
\quad\quad \mathsf{snd}\ (\mathsf{snd}\ xs) +\!\!+_{\langle\beta\rangle} \mathsf{snd}\ (\mathsf{snd}\ ys)\rangle\rangle & \\
xs +\!\!+_{\langle[\![\alpha]\!]\rangle} ys & = xs +\!\!+^{C}_{\langle\alpha\rangle} ys
\end{array}
$$

Note that the case $+\!\!+_{\langle\alpha\to\beta\rangle}$ is not covered. It is not needed, as the transformations do not generate such uses of $(+\!\!+)$ in the presence of the restrictions from Subsection 2.4. For brevity, we omit both the definition of chunkwise append $(+\!\!+^{C})$ and of the lifted variant.

6.1.4 Indexing and Length

Finally, we have the simplest case—namely that of an operation reducing the nesting depth of its structured argument. The prototypical example for this case is indexing:

$$
\begin{array}{ll}
(!|_{\langle\alpha\rangle}) & :: \mathcal{F}[\![[|\alpha|]]\!] \times \mathsf{Int} \to \mathcal{F}[\![\alpha]\!] \\
xs\ !|_{\langle()\rangle}\ i & = () \\
xs\ !|_{\langle\mathsf{Int}\rangle}\ i & = \mathsf{snd}\ xs\ !|_{\mathsf{Int}}\ i \\
xs\ !|_{\langle\mathsf{Bool}\rangle}\ i & = \mathsf{snd}\ xs\ !|_{\mathsf{Bool}}\ i \\
xs\ !|_{\langle\alpha\times\beta\rangle} i & = \langle \mathsf{fst}\ xs\ !|_{\langle\alpha\rangle}\ i,\ \mathsf{snd}\ xs\ !|_{\langle\beta\rangle}\ i\rangle \\
xs!|_{\langle\alpha+\beta\rangle}\ i & = \\
\quad \mathbf{if}\ (\mathsf{fst}\ xs\ !|_{\langle\mathsf{Bool}\rangle}\ i)\ \mathbf{then} & \\
\quad\quad \mathsf{snd}\ (\mathsf{snd}\ xs)\ !|_{\langle\beta\rangle}\ i & \\
\quad \mathbf{else} & \\
\quad\quad \mathsf{fst}\ (\mathsf{snd}\ xs)\ !|_{\langle\alpha\rangle}\ i & \\
xs!|_{\langle[\![\alpha]\!]\rangle}\ i & = x\ !|^{C}_{\langle\alpha\rangle}\ n
\end{array}
$$

Like in the definition of $(+\!\!+)$, we do not need a case for the type index $\alpha \to \beta$, as it is guaranteed not to occur. It is interesting to note that indexing of parallel arrays whose elements are of complex type, e.g., arrays of lists $[|[\alpha]|]$ is *not* a constant time operation. As the equations for product and sum types show, the cost is proportional to the depth of the extracted element. This might seem peculiar, but is in fact a natural consequence of a fully vectorised—and thus, perfectly load balanced—computation. The indexed subterm may be located on a single processor and must, thus, be communicated upon extraction, which in turn will generally entail touching all its sub-structures. In fact, even if we index a parallel array of integer values, the extracted value generally has to be broadcast to all processing nodes. This property of the execution model clearly leads to inefficiencies if followed stoically. More generally, this is an instance of the observation that overly aggressive load balancing actually can lead to less efficient code. We have discussed solutions to this problem in previous work [15, 17], which are employed in the stages following the grey box of Figure 1.

The length of an array is explicit in the flattened representation, so we will not give a formal definition of the primitive len here.

6.2 Operations on Sequential Structures

In contrast to the operations on parallel arrays, operations on sequential structures—such as products and sums that do not occur as elements of a parallel array—can be realised using their usual parametric polymorphic implementation. In our definition of λ_{PA}, these are the operations $\langle\cdot,\cdot\rangle$, $\mathsf{fst}_{\langle\alpha,\beta\rangle}$, $\mathsf{snd}_{\langle\alpha,\beta\rangle}$, $\mathsf{left}_{\langle\alpha,\beta\rangle}$, $\mathsf{right}_{\langle\alpha,\beta\rangle}$, and $\mathsf{case}_{\langle\alpha,\beta,\gamma\rangle}$. Nevertheless, we need these functions in both a scalar and a lifted variant. This is trivial for the former three: Their lifted variants are identical to the scalar ones, but require some thought for the latter three operations.

$$
\begin{array}{ll}
\mathsf{left}^{\uparrow}_{\langle\alpha,\beta\rangle} & :: [|\alpha \to \alpha + \beta|] \\
\mathsf{left}^{\uparrow}_{\langle\alpha,\beta\rangle}\ xs & = \langle \mathsf{rep}_{\langle\mathsf{Bool}\rangle}\ \langle \mathsf{len}\ xs, \mathsf{false}\rangle, \langle xs, \bot\rangle\rangle \\
 & \\
\mathsf{right}^{\uparrow}_{\langle\alpha,\beta\rangle} & :: [|\beta \to \alpha + \beta|] \\
\mathsf{right}^{\uparrow}_{\langle\alpha,\beta\rangle}\ xs & = \langle \mathsf{rep}_{\langle\mathsf{Bool}\rangle}\ \langle \mathsf{len}\ xs, \mathsf{true}\rangle, \langle \bot, xs\rangle\rangle \\
\mathsf{case}^{\uparrow}_{\langle\alpha,\beta,\gamma\rangle} & :: [|(\alpha \to \gamma) \times (\beta \to \gamma) \times (\alpha + \beta) \to \gamma|] \\
\mathsf{case}^{\uparrow}_{\langle\alpha,\beta,\gamma\rangle}\ f\ g\ xs & = \\
\quad \mathbf{let} & \\
\quad\quad selector & = \mathsf{fst}\ xs \\
\quad\quad rlen & = \mathsf{len}\ (\mathsf{packP}\ \langle selector, selector\rangle) \\
\quad\quad llen & = \mathsf{len}\ selector - rlen \\
\quad\quad left' & = \mathbf{if}\ llen \neq 0\ \mathbf{then}\ f\ (\mathsf{fst}\ (\mathsf{snd}\ xs))\ \mathbf{else}\ \bot \\
\quad\quad right' & = \mathbf{if}\ rlen \neq 0\ \mathbf{then}\ g\ (\mathsf{snd}\ (\mathsf{snd}\ xs))\ \mathbf{else}\ \bot \\
\quad \mathbf{in} & \\
\quad \langle selector, \langle left', right'\rangle\rangle &
\end{array}
$$

We do not define conditional expressions formally, but they behave entirely as expected.

The operations $\mathsf{left}^{\uparrow}$ and $\mathsf{right}^{\uparrow}$ encode a complete array of values as left or right components of a sum—they do so in a single parallel step. The lifted variant of case is more involved. It applies its arguments f and g to the left and right components of the lifted sum, respectively—but it is careful not to touch a component that represents an empty array. This is important as these components will not contain any meaningful value—see the definition of rep—and they may be part of a recursive data structure.

6.3 Specialisation of Generic Definitions

In the following, we briefly summarise the specialisation procedure of Hinze [12], as it is needed to instantiate the generic definitions of the primitives from Section 6.1, and thus, to obtain concrete definitions for the primitives at the required types. The general form for type-indexed values in λ_{PA} is

$$
\begin{array}{ll}
poly_{\langle\alpha::\star\rangle} & :: F\ \alpha \\
poly_{\langle()\rangle} & = poly_{()} \\
poly_{\langle\mathsf{Int}\rangle} & = poly_{\mathsf{Int}} \\
poly_{\langle\mathsf{Bool}\rangle} & = poly_{\mathsf{Bool}} \\
poly_{\langle\alpha\times\beta\rangle} & = poly_{\times}\ poly_{\langle\alpha\rangle}\ poly_{\langle\beta\rangle} \\
poly_{\langle\alpha+\beta\rangle} & = poly_{+}\ poly_{\langle\alpha\rangle}\ poly_{\langle\beta\rangle} \\
poly_{\langle[\![\alpha]\!]\rangle} & = poly_{[\![\cdot]\!]}\ poly_{\langle\alpha\rangle}
\end{array}
$$

Here, $\star$ denotes the kind of normal types, i.e., the set of types produced by $\mathbf{T}$. Type constructors map types to types—e.g., the kind of $[\![\cdot]\!]$ is $\star \to \star$. We can bring the definition of any $\star$-indexed function into the above form by supplying suitable values for F, $poly_{()}$, $poly_{\mathsf{Int}}$, $poly_{\mathsf{Bool}}$, $poly_{\times}$, $poly_{+}$, and $poly_{[\![\cdot]\!]}$, where we require that the following type constraints are satisfied:

$$
\begin{array}{ll}
poly_{()} & :: F\ () \\
poly_{\mathsf{Int}} & :: F\ \mathsf{Int} \\
poly_{\mathsf{Bool}} & :: F\ \mathsf{Bool} \\
poly_{\times} & :: \forall\alpha,\beta.F\ \alpha \to\ F\beta \to\ F\ (\alpha \times \beta) \\
poly_{+} & :: \forall\alpha,\beta.F\ \alpha \to\ F\beta \to\ F\ (\alpha + \beta) \\
poly_{[|\cdot|]} & :: \forall\alpha.F\ \alpha \to\ F\ [|\alpha|]
\end{array}
$$

The key idea behind Hinze's transformation is to promote $\star$-indexed functions to types of arbitrary kinds as follows: type abstraction is interpreted by value abstraction, type application by value application, and type recursion by value recursion. This allows us to define the promoted version of $poly_{\langle\cdot\rangle}$, which is denoted by $poly_{\langle\langle\cdot\rangle\rangle}$, as follows:

$$
\begin{array}{lll}
poly_{\langle\langle\tau::\kappa\rangle\rangle} & :: & F_{\langle\langle t::\kappa\rangle\rangle} \\
poly_{\langle\langle c\rangle\rangle}\ \rho & = & poly_c \\
poly_{\langle\langle x\rangle\rangle}\ \rho & = & \rho\ x \\
poly_{\langle\langle \tau_1\,\tau 2\rangle\rangle}\ \rho & = & (poly_{\langle\langle\tau_1\rangle\rangle}\ \rho)\ (poly_{\langle\langle\tau_2\rangle\rangle}\ \rho) \\
poly_{\langle\langle \Lambda\alpha.\tau\rangle\rangle}\ \rho & = & \lambda v.poly_{\langle\langle\tau\rangle\rangle}\ (\rho \cup \{\alpha \mapsto v\}) \\
poly_{\langle\langle \mu_t\,\alpha.t\rangle\rangle}\ \rho & = & \mu_v v.poly_{\langle\langle t\rangle\rangle}\ (\rho \cup \{\alpha \mapsto v\})
\end{array}
$$

The variable ρ denotes an environment, $x \mapsto \alpha$ extends the environment, and application of an environment to a type variable returns the associated type value. Moreover, $poly_{\langle\cdot\rangle}$ and $poly_{\langle\langle\cdot\rangle\rangle}$ are related by

$$poly_{\langle\tau::\star\rangle} \ = \ poly_{\langle\langle\tau::\star\rangle\rangle}\ \epsilon$$

where ϵ is the empty environment. As a result, we can specialise $poly_{\langle t\rangle}$ by specialising $poly_{\langle\langle t\rangle\rangle}$. Finally, $F_{\langle\langle t::\kappa\rangle\rangle}$ is given by

$$
\begin{array}{lll}
F_{\langle\langle u::\star\rangle\rangle} & = & F\ u \\
F_{\langle\langle u::\kappa_1\to\kappa_2\rangle\rangle} & = & \forall\ x.F_{\langle\langle x::\kappa_1\rangle\rangle}\ \to F_{\langle\langle u\ x::\kappa_2\rangle\rangle}
\end{array}
$$

A similar specialisation procedure for type-indexed types, which we will not detail here, is given by Hinze in [11].

7. ADDITIONAL FEATURES

7.1 Mutual Type Recursion and Parametrised Types

The calculus λ_{PA} did not include the treatment of mutual type recursion and parametrised types to reduce notational overhead. To handle mutual type recursion, we have to use type equations, such as,

$$List\ \alpha \ = \ () + (\alpha \times List\ \alpha)$$

instead of an explicit fixed-point combinator. The types of a program module are, then, given as a set of such equations, which may be mutually recursive. We can define the flattened representation of parallel types, as given in Figure 3, in exactly the same way for type equations.

The inclusion of polymorphism is less straight forward. The use of polymorphic functions prevents full specialisation of these functions (without knowing all type instances at which they occur), which is what has hindered previous formulations of flattening from including support for separate compilation in the presence of polymorphism. Fortunately, our novel use of Hinze's generic programming framework helps us here again. Hinze uses one higher-order argument in each specialised function for each type variable occurring in the type index of a generic value. These arguments contain functions that implement the specialised routine for the abstracted types, i.e., in the case of applying replication to a value of type $[\alpha]$, the specialised replication routine for $[\alpha]$ would get a replication routine for α as an argument. Type-indexed types are treated similarly. See [12, 11] for details.

The argument so far only covers the specialisation of primitives, but not the vectorisation transformation $\mathcal{V}[\![\cdot]\!]$. Interestingly, vectorisation is not affected by the presence of recursive types at all—as we already argued in [16]. Similarly, polymorphic definitions do not complicate $\mathcal{V}[\![\cdot]\!]$. The essential reason for this is that vectorisation is not a type-dependent transformation.

7.2 Why Pure Functions Matter

Hughes [13] cites higher order functions and lazy evaluation as the main advantages of functional programming. In the context of nested data parallelism, other factors come into play. One of them is, of course, the lack of side effects when executing multiple function invocations in parallel. In contrast to languages with side effects, NEPAL does not require artificial restrictions on the kind of computations that can be executed in parallel without altering the semantics of the program. In combination with the high abstraction level of functional programs, the compiler has a high degree of freedom in mapping source-level data structures and computations to concrete implementations. The high degree of expressiveness, usually allows to work without reference types, which are highly problematic in a parallel context: (1) They cannot be simply implemented by pointers on distributed-memory machines; (2) the presence of pointers generally exacerbates the difficulties of data dependency analysis; and (3) side effects usually reduce the set of admissible schedules, which reduces parallelism.

Although, it is possible to integrate the discussed approach to nested data parallelism into non-functional languages—as we did for C [8]—it is rather tedious, as non-trivial restrictions on the use of references have to be imposed within parallel computations. Moreover, the code generated by the flattening transformation is too fine grained to be directly executed efficiently on most architectures. This requires aggressive compiler optimisations, which are significantly easier to perform in a purely functional language, as the compiler can exploit its detailed knowledge about data dependencies.

Nepal allows user-defined dynamic data types within parallel arrays. This is possible due to the declarative way in which algebraic data types are specified in functional languages. The compiler is able to compute an efficient representation of those structures by mapping them onto isomorphic structures that contain only parallel arrays of basic type—which, as we have argued—is important for parallel performance. Such a transformation is only possible when we have sufficient control over how data structures are accessed and manipulated. In V, the nested parallel C dialect, we restricted the use of references in a parallel context to parameter passing to give the compiler more scope for optimisations. In the process of handling the remaining references, our prototype compiler for V transformed arrays of references into a single reference to one array. However, as [22] discusses, even with this strong restriction, it is possible to write programs where the flattened program does not accurately preserve the semantics of the source program without imposing further restrictions.

In summary, it appears to us that references types and fine-grained parallelism—especially when flattening is used—

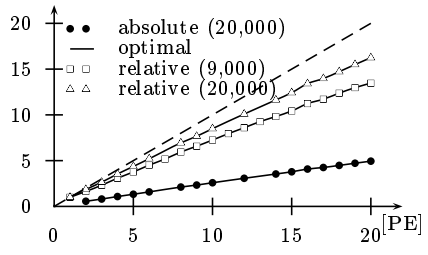

Figure 4: Absolute and relative speedup of Barnes-Hut

are conceptually incompatible. Consequently, we regard the lack of support for reference types in flattening-based implementations as a secondary issue.

8. CONCLUSION

Before the concluding remarks, let us briefly discuss our preliminary experimental results and related work.

8.1 Benchmarks

To ensure the practicability of our approach, we performed a number of experiments on various parallel machines including a Cray T3E. In particular, we measured the performance of a Barnes-Hut n-body code, which we manually derived using our transformation rules (we are currently working at a full compiler). We previously reported these experimental results in [16] for a more restricted form of the flattening transformation. Nevertheless, the new transformation produces essentially the same code for this particular code. Figure 4 displays the relative and absolute speedup, where the latter is in comparison to a sequential implementation of the same algorithm on a single processor. In other nested data parallel implementations [3], the spatial decomposition of the particles was, due to the restricted set of available data structures, encoded as nested parallel arrays. The disadvantage of such a representation is that it does not reflect the structure of the parallel algorithm, and thus, achieves poor locality. In fact, the inadequate representation implies up to about 20% runtime overhead already *on a single processor*—this factor increases with the number of processors, as the frequency of non-local memory access goes up. We are able to avoid this overhead in our implementation by storing the particles in a rose tree.

Using the type transformation techniques described in this paper in combination with various other optimising program transformations, we achieved an absolute speedup of about a factor 4 on 20 processors, compared with a sequential implementation of the same algorithm in C. Considering the highly irregular structure of the algorithm and the comparatively high communication overhead which it induces, this is a promising result. Furthermore, the curve is still almost linear for 20 processors (which was the maximal number of processors available to us).

Moreover, we achieve a relative speedup of 15 for 20 processors in the case of 20k particles, which demonstrates the good parallel behaviour of our transformations. We hope that we can further close the gap between relative and absolute speedup by applying more aggressive optimisations.

8.2 Related Work

Special purpose functional languages supporting flattening-based nested data parallelism include Nesl [4] and Proteus [23]. Although, Nesl takes some of its syntax from ML, it is far from covering the whole language and, in particular, ML's rich data structures. Proteus is a hybrid high-level language mixing functional and imperative language features; flattening was only applied to a sublanguage.

As we have already mentioned, flattening in its original, restricted form was introduced by Blelloch & Sabot [5]. Prins & Palmer [23] as well as Keller & Simons [18] formalised the previously rather informal description of the transformation. The former also extended it to cover higher-order functions. Palmer et al. [21] optimised the transformation and Riely et al. [24] showed that it preserves parallel complexity. Finally, we previously [16] extended it to user-defined recursive data structures.

8.3 Concluding Remarks

We presented a generalisation of flattening that allows us to apply this program transformation in the context of a standard functional languages like Haskell and ML. We feel that this is an important step towards the practical usability of nested data parallelism. Our transformation-based approach has had first positive practical results and we are currently in the process of implementing a full compiler.

In this paper, we chose parallel arrays as the only parallel data structure. However, a closer look at the program transformations reveals that other forms of collections of values are also suitable as long as we can separate structure from data values level by level—in the same way as we do that with segment descriptors for parallel arrays. In particular, we should be able to support parallel trees and parallel products.

Acknowledgements. We are indebted to the other two current members of the NEPAL project, Roman Lechtchinsky and Wolf Pfannenstiel, who helped to shape our ideas and provided valuable feedback. Furthermore, we thank Kai Engelhardt, Sven Panne, and the anonymous referees for their helpful comments and suggestions on a draft of this paper.

9. REFERENCES

[1] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986.

[2] G. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[3] G. Blelloch and G. Narlikar. A practical comparison of N-body algorithms. In *Dimacs Implementation Challenge Workshop*, 1994.

[4] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[5] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[6] D. Cann. Retire fortran? A debate rekindled. *Communications of the ACM*, 35(8):81, Aug. 1992.

[7] M. M. T. Chakravarty and G. Keller. How portable is nested data parallelism? In *Proc. of 6th Annual*

Australasian Conf. on Parallel And Real-Time Systems, pages 284–299. Springer-Verlag, 1999.

[8] M. M. T. Chakravarty, F.-W. Schröer, and M. Simons. V—nested parallelism in C. In *Proceedings of the Working Conference on Massively Parallel Programming Models (MPPM-95)*. IEEE Computer Society Press, 1995.

[9] S. Chatterjee, J. F. Prins, and M. Simons. Expressing irregular computations in modern Fortran dialects. In *Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[10] H. P. F. Forum. High Performance Fortran language specification. Technical report, Rice University, 1993. Version 1.0.

[11] R. Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 2000. To appear.

[12] R. Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*. ACM Press, 2000.

[13] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[14] C. B. Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, 2000. Forthcoming.

[15] G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1999.

[16] G. Keller and M. M. T. Chakravarty. Flattening trees. In D. Pritchard and J. Reeve, editors, *Euro-Par'98, Parallel Processing*, number 1470 in Lecture Notes in Computer Science, pages 709–719, Berlin, 1998. Springer-Verlag.

[17] G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In J. Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

[18] G. Keller and M. Simons. A calculational approach to flattening nested data parallelism in functional languages. In J. Jaffar, editor, *The 1996 Asian Computing Science Conference*, Lecture Notes in Computer Science. Springer Verlag, 1996.

[19] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[20] G. Narlikar and G. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, 1997.

[21] D. Palmer, J. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE, 1995.

[22] W. Pfannenstiel, M. Dahm, M. M. T. Chakravarty, S. Jähnichen, G. Keller, F.-W. Schröer, and M. Simons. Aspects of the compilation of nested parallel imperative languages. In J. Darlington, editor, *Proceedings of the Third International Conference on Programming Models for Massively Parallel Computers (MPPM '97)*, pages 102–109. IEEE Computer Society Press, 1998.

[23] J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM.

[24] J. W. Riely, J. Prins, and S. P. Iyer. Vectorization of nested-parallel programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models for Massively Parallel Computers*. IEEE Computer Society, 1995.

[25] S.-B. Scholz. On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *Proceedings of APL'98*, pages 40–45. ACM Press, 1998.