Parallel Processing Letters, Vol. 12, No. 2 (2002) 249–266
© World Scientific Publishing Company

COSTING NESTED ARRAY CODES

ROMAN LECHTCHINSKY
ISTI, Technische Universtiät Berlin, Berlin, Germany
rl@cs.tu-berlin.de

MANUEL M. T. CHAKRAVARTY and GABRIELE KELLER
CSE, University of New South Wales, Sydney, Australia
{chak,keller}@cse.unsw.edu.au

Received April 2002
Revised September 2002
Accepted by S. Gorlatch & C. Lengauer

ABSTRACT

We discuss a language-based cost model for array programs built on the notions of work complexity and parallel depth. The programs operate over data structures comprising nested arrays and recursive product-sum types. In a purely functional setting, such programs can be implemented by way of the *flattening transformation* that converts codes over nested arrays into vectorised code over flat arrays. Flat arrays lend themselves to a particularly efficient implementation on standard hardware, but the overall efficiency of the approach depends on the flattening transformation preserving the asymptotic complexity of the nested array code. Blelloch has characterised a class of first-order array programs, called *contained* programs, for which flattening preserves the asymptotic depth complexity. However, his result is restricted to programs processing only arrays and tuples. In the present paper, we extend Blelloch's result to array programs processing data structures containing arrays as well as arbitrary recursive product-sum types. Moreover, we replace the notion of containment by the more general concept of *fold* programs.

Keywords: Nested data parallelism; cost-calculus; flattening; program transformation; functional programming.

1. Introduction

In this paper, we develop a language-based parallel cost model for array programs operating over data structures comprising nested arrays and recursive product-sum types. In a purely functional setting, nested array programs provide for a convenient high-level model to code algorithms from the domain of scientific and engineering computing [7]; however, their efficient implementation is a considerable challenge. The *flattening transformation*, which converts array codes over nested array structures into vectorised code over flat arrays is an attractive implementation technique as it facilitates the use of the same efficient arrays of unboxed, primitive types that are used in conventional array languages, such as Fortran or C [2,6]. The vectorised code, moreover, uses collection-oriented array operations that enable a variety of further optimisations [13,14].

For the functional array language Nesl, Blelloch [3] has demonstrated how a language-based cost model facilitates the development of parallel algorithms based on the notion of *depth* and *work* complexity. His framework, however, is restricted to data structures involving arrays and tuples. In this paper, we extend his approach to cover recursive product-sum types as well as higher-order functions.

In summary, this paper makes the following main contributions:

- We define a language-based cost calculus defining depth and work complexity for higher-order programs operating over structures constructed from product-sum types and arrays (Section 3).
- We show that at least a large class of primitive recursive functions can be expressed such that flattening does not alter their complexity. In particular, the depth complexity is preserved, which is known to not be the case for some programs outside of this class (Section 4).

1.1. Related work

Blelloch's cost calculus [3] for nested array programs covers first-order programs involving tuples and arrays. We extend it to cover higher-order features as well as recursive product-sum types. This extension is based on our previous work on extending the flattening transformation to include such data structures [6] and makes essential use of ideas from intensional type analysis [8] and generic programming [9].

Blelloch [2] introduces the notion of *containment* to characterise programs for which the costs derived by way of the language-based model coincides with the costs incurred by a program after the flattening transformation has been applied. This line of work has been extended by Riely & Prins [15] who introduced a type system that rejects programs for which flattening would worsen the depth complexity. We replace Blelloch's notion of containment by the more general concept of *fold* programs [10], which are straight-forward to define over product-sum types.

The issue concerning the complexity of flattening can be circumvented by relying on a thread-based implementation of array-based nested data parallelism [4]. However, for a thread-based implementation, the efficient realisation on distributed-memory machines is an open problem.

An entirely different approach has been investigated by Jay et al. [11]. They, however, do not consider the properties of flattening.

2. Purely Functional Array Programming

The functional programming paradigm is attractive for the development of array programs as demonstrated by the constant interest in functional languages with strong array support [2,3,5,17,12,1,7]. In particular, the resulting programs are on a fairly high level and close to the mathematical foundations on which many of the algorithms rest. Moreover, the absence of side effects as well as the collective operations used in some approaches promise a smooth route to parallel implementations.

Although the background to the work presented in this paper is our extension of Haskell by nested parallel arrays [7], the results of this paper generally applies to purely functional nested array programs, and in particular, their implementation by way of the *flattening transformation*. More precisely, our formalisation will be by way of a functional core language including array primitives; we introduce this language next. Afterwards, we briefly describe the idea behind the flattening

Definitions	D	$\to$	$\mathsf{C\ V_1 \cdots V_n = T}$	(type definition, $n \geq 0$)
		$\mid$	$\mathsf{V :: T}$	(type signature)
		$\mid$	$\mathsf{V\ V_1 \cdots V_n = E}$	(function definition, $n \geq 1$)
Types	T	$\to$	V	(variable)
		$\mid$	$\mathsf{T_1 \times T_2}$	(product)
		$\mid$	$\mathsf{T_1 + T_2}$	(sum)
		$\mid$	$\mathsf{T_1 \to T_2}$	(functional)
		$\mid$	$\mathsf{[:T:]}$	(parallel array)
		$\mid$	$\mathsf{C\ T_1 \cdots T_n}$	(type application, $n \geq 0$)
Expressions	E	$\to$	C	(constant)
		$\mid$	V	(variable)
		$\mid$	$\mathsf{E\ E_1 \cdots E_n}$	(n-ary application, $n \geq 1$)
		$\mid$	**let** $\mathsf{V = E_1}$ **in** $\mathsf{E_2}$	(local binding)
		$\mid$	**case** E **of** $\triangleleft\mathsf{V_1 \to E_1};\ \triangleright\mathsf{V_2 \to E_2}$	(selection)

Fig. 1. Grammar and primitives of the core language

transformation and specify the transformation rules. In the section following the current, we discuss a cost calculus for the depth and work complexity of nested programs and their flattened counterparts.

2.1. A Core Language

To formalise our results we use the typed core language displayed in Figure 1 and include a set of constants to manipulate data structures composed of recursive product-sum types and arrays. A program in this language is a set of definitions produced by the non-terminal D.

The language includes named type definitions of the form $T\,\alpha_1 \cdots \alpha_n = t$, where the type on the right-hand side is a standard product-sum type including functions and arrays. The latter are represented by types $[:t:]$, where t is the element type of the array. Function definition, function applications, and local bindings are standard.

Products are formed by the constructor $\langle \cdot, \cdot \rangle_{\langle \alpha,\beta \rangle} :: \alpha \to \beta \to \alpha \times \beta$ and are decomposed by $\mathsf{fst}_{\langle \alpha,\beta \rangle} :: \alpha \times \beta \to \alpha$ and $\mathsf{snd}_{\langle \alpha,\beta \rangle} :: \alpha \times \beta \to \beta$. Moreover, sums are formed by $\triangleleft_{\langle \alpha,\beta \rangle} :: \alpha \to \alpha + \beta$ and $\triangleright_{\langle \alpha,\beta \rangle} :: \beta \to \alpha + \beta$; they are decomposed by **case** expressions. We assume the nullary type constructors (), `Int`, and `Bool` as well as the usual arithmetic and logical operations to be available.

As an example for a type definition, consider $List\,\alpha = () + \alpha \times List\,\alpha$, which defines parametric lists (products take precedence over sums). Mapping over these lists can be defined as follows:

$$\begin{aligned} map\quad &:: (\alpha \to \beta) \times List\,\alpha \to List\,\beta \\ map\,f\,l &= \textbf{case}\ l\ \textbf{of}\ \triangleleft\_ \to \triangleleft();\ \triangleright xxs \to \triangleright\langle f(\mathsf{fst}\,xxs), map\,f\ (\mathsf{snd}\,xxs)\rangle \end{aligned}$$

To improve the readability of expressions, we take the freedom to use multiple bindings in a single **let-in** expression and to use pairs as patterns on the left-hand side of function definitions and let-bindings (instead of using fst and snd on the right-hand side).

Remark 1. We assume that type definitions are *not* mutually recursive. The presented framework could be extended to include mutually recursive type definitions at the expense of more tedious notation without any new insights.

Remark 2. The core language has been defined such that it is *not* possible to build closures. This is a significant restriction and we regard extending the formalism to include closures as future work. Although many closure creating programs can already be handled with the techniques described in the following, there are cases that require significant extensions.

Definition 1 (Pretypes) *Given a type definition $T\,\alpha_1 \cdots \alpha_n = t$ we call the type $T'\,\alpha_1 \cdots \alpha_n\,\beta = t'$, where t' is derived from t by replacing every occurrence of $T\alpha_1 \cdots \alpha_n$ by β, the* pretype *of T.*

A type T and its pretype T' are related by the equation

$$T\,\alpha_1 \cdots \alpha_n = \mu(T'\,\alpha_1 \cdots \alpha_n)$$

where μ is the fixed point operator on types. We will need this definition of pretypes in Section 4.1.

Array values are constructed by the function replicateP $::$ $Int \times \alpha \rightarrow [:\alpha:]$, which creates an array containing as many copies of its second argument as the first dictates. Moreover, the function lengthP $::$ $[:\alpha:] \rightarrow Int$ determines the length of an array, $(!:)$ $::$ $[:\alpha:] \times Int \rightarrow \alpha$ extracts the given element from an array, and $(+\!\!+)$ $::$ $[:\alpha:] \times [:\alpha:] \rightarrow [:\alpha:]$ concatenates two arrays. Finally, we assume that we have standard list functions, such as *map*, *filter*, and so on, also available for arrays; in particular, mapP $::$ $(\alpha \rightarrow \beta) \times [:\alpha:] \rightarrow [:\beta:]$ maps a function over an array (more details on the definition of these primitives follows in Section 3).

For notational convenience, we sometimes use *array comprehensions* (oriented by Haskell list comprehensions) instead of explicitly using the array primitives. Comprehensions can be compiled into explicit array primitives using rules such as the ones given in [7]. For example, given a matrix in the compressed row format

$$\begin{aligned} &\mathit{SparseRow} &&= [:\langle \texttt{Int}, \texttt{Float}\rangle:] \\ &\mathit{SparseMatrix} &&= [:\mathit{SparseRow}:] \end{aligned}$$

we can define sparse matrix/vector multiplication as

$$\begin{aligned} &\mathit{smvm} &&:: \mathit{SparseMatrix} \times [:\texttt{Float}:] \rightarrow [:\texttt{Float}:] \\ &\mathit{smvm}\ \mathit{sm}\ \mathit{vec} &&= [:\mathit{sumP}\ [:\mathit{vec}\,!:\,i * v \mid \langle i, v\rangle \leftarrow \mathit{row}:] \mid \mathit{row} \leftarrow \mathit{sm}:] \end{aligned}$$

which lends itself to a direct parallel implementation.

Remark 3. We restrict the elements of arrays to types that do not contain functions. This is essentially to avoid control parallel expressions, such as

$$[:f\ x \mid f \leftarrow [:\mathit{foo}, \mathit{bar}:] \mid x \leftarrow [:a, b:]:]$$

The flattening transformation, discussed below, will introduce arrays of functions, but they will always correspond to dataparallel computations.

2.2. The Flattening Transformation

The flattening transformation converts array codes over nested array structures into code over flat arrays. It is an attractive implementation technique as it facilitates the use of the same efficient arrays of unboxed, primitive types that are used in conventional array languages, such as Fortran or C [2,6]; in particular, flat array codes are more amenable to a parallel implementation.

Lifting Functions. Essential to the data parallel nature of array codes is the function mapP, as mapP f *arr* can be regarded as the data parallel execution of f over the elements of an array *arr*. The partial application mapP f can be regarded as *lifting* a function over a type τ into a function over $[:\tau:]$ (arrays of τ). As in a language supporting higher-order functions it is statically unknown whether a given function will be needed in its original or lifted version, flattening generates for each function definition $f\ a_1\ \cdots\ a_n\ =\ e$ a *vectorised* variant $\overline{f}$ by pairing the original function with its lifted counterpart, where the latter is denoted by $f^{\uparrow}$:

$$\begin{array}{ll}
\overline{f}\,() & = (f^0, f^{\uparrow}) \\
f^0 & :: t_1 \times \cdots \times t_n \rightarrow t \\
f^0\ a_1\ \cdots\ a_n & = \mathcal{V}[\![e]\!] \\
f^{\uparrow} & :: [:t_1:] \times \cdots \times [:t_n:] \rightarrow [:t:] \\
f^{\uparrow}\ a_1\ \cdots\ a_n & = e^{\uparrow(len\ a_1)}
\end{array}$$

The transformation $\mathcal{V}[\![\cdot]\!]$ adapts the scalar code to take the pair representation into account. If it encounters a non-lifted application, it applies the first component of the pair:

$$\begin{array}{lll}
\mathcal{V}[\![v]\!] & \mid v \text{ is a toplevel function} & = \overline{v}\,() \\
 & \mid \textit{otherwise} & = v \\
\mathcal{V}[\![e\ e_1 \cdots e_n]\!] & & = (\mathsf{fst}\ \mathcal{V}[\![e]\!])\ \mathcal{V}[\![e_1]\!]\ \cdots\ \mathcal{V}[\![e_n]\!] \\
\mathcal{V}[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!] & & = \textbf{let}\ v\ =\ \mathcal{V}[\![e_1]\!]\ \textbf{in}\ \mathcal{V}[\![e_2]\!] \\
\mathcal{V}[\![\textbf{case}\ e\ \textbf{of}\ \triangleleft a \rightarrow e_1;\ \triangleright b \rightarrow e_2]\!] & & = \textbf{case}\ \mathcal{V}[\![e]\!]\ \textbf{of}\ \triangleleft l \rightarrow \mathcal{V}[\![e_1]\!];\ \triangleright r \rightarrow \mathcal{V}[\![e2]\!]
\end{array}$$

Note that lifting a variable that is not a function, the variable does not change. However, since the expression that is bound to the variable is also lifted, the variable refers now to an array value. More interesting is the lifting operation $(\cdot)^{\uparrow l}$, which is parametrised with the length of the array over which an expression is lifted; we call this the *lifting context.*

$$\begin{array}{lll}
c^{\uparrow l} & & = \mathsf{replicateP}\ l\ c \\
v^{\uparrow l} & \mid x \text{ is a toplevel function} & = \overline{v}\,() \\
 & \mid \textit{otherwise} & = v \\
(e\ e_1\ \cdots\ e_n)^{\uparrow l} & & = (\mathsf{snd}\ e^{\uparrow l})\ e_1^{\uparrow l}\ \cdots\ e_n^{\uparrow l} \\
(\textbf{let}\ v\ =\ e_1\ \textbf{in}\ e2)^{\uparrow l} & & = \textbf{let}\ v\ =\ e_1^{\uparrow l}\ \textbf{in}\ e_2^{\uparrow l}
\end{array}$$

In contrast to $\mathcal{V}[\![\cdot]\!]$, the rule for application in $(\cdot)^{\uparrow l}$ uses the second component (i.e., the lifted version) of the function pair.

The most involved case is that of handling **case** expressions. The key point here is that in the alternatives of the **case** expression (i.e., e_1 and e_2 below) the lifting

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

(a)

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

(b)

Fig. 2. (a) Flattened representation of parallel types and (b) pairing of normal with vectorised functions

context gets restricted. The original lifting context l is partitioned into contexts ll and lr where $l = ll + lr$, as the type constructor $(+)$ represents disjoint sums. (The function $\mathrm{FV}(e)$ yields the set of variables that are free in e).

$$
\begin{array}{l}
(\textbf{case}\ e\ \textbf{of}\ \triangleleft a \rightarrow e_1;\ \triangleright b \rightarrow e_2)^{\uparrow l}\ = \\
\quad \textbf{let} \\
\quad\quad \begin{array}{lcl}
v & = & e^{\uparrow l} \\
\mathit{flags} & = & \mathsf{fst}\ v \\
ll & = & \mathsf{lengthP}\ (\mathsf{packP}\ \mathit{flags}\ \mathit{flags}) \\
rl & = & \mathsf{lengthP}\ \mathit{flags}\ -\ ll \\
a & = & \mathsf{fst}\ (\mathsf{snd}\ v) \\
b & = & \mathsf{snd}\ (\mathsf{snd}\ v) \\
\mathit{lres} & = & \textbf{let}\ \forall v \in\ \mathrm{FV}(e_1) \setminus \{a\}.\ v\ =\ \mathsf{packP}\ \mathit{flags}\ v\ \textbf{in}\ e_1^{\uparrow ll} \\
\mathit{rres} & = & \textbf{let}\ \forall v \in\ \mathrm{FV}(e_2) \setminus \{b\}.\ v\ =\ \mathsf{packP}\ \mathit{flags}\ v\ \textbf{in}\ e_2^{\uparrow rl}
\end{array} \\
\quad \textbf{in} \\
\quad \mathsf{combineP}\ \mathit{flags}\ \mathit{lres}\ \mathit{rres}
\end{array}
$$

All arrays associated with variables occurring free in one of the two branches need to be *packed* so that only those elements corresponding to a particular alternative are processed in that alternative. Conversely, the results of the two alternative contexts have to be *combined* into an overall result for the outer context l.

Note how $e_1^{\uparrow ll}$ and $e_2^{\uparrow rl}$ are executed sequentially. This is necessary to preserve the data parallel nature of the implementation, but is also the source of the discussion about the parallel depth of flattened code in Section 4.

Flattening Data Structures. As mentioned before, collection oriented array operations, in particular lifted operations, can be executed most efficiently on flat array structures. Therefore, the flattening transformation decomposes complex data structures such that the structure information is separated from the primitive data values that are stored in the structure. The rules for the type transformation are listed in Figure 2: An array of unit type values is simply represented by its length, arrays of primitive types are represented as a pair, their length stored explicitly; arrays of pairs as pairs of arrays of equal length. Finally, arrays of sumtypes are stored as a nested pair: A flag array indicating to which side of the sum each array element belongs and a pair of arrays containing the actual values. The overall length of both of these arrays together is the same as the length of the original array and the flag array. Nested arrays are represented by a flat array containing all the data,

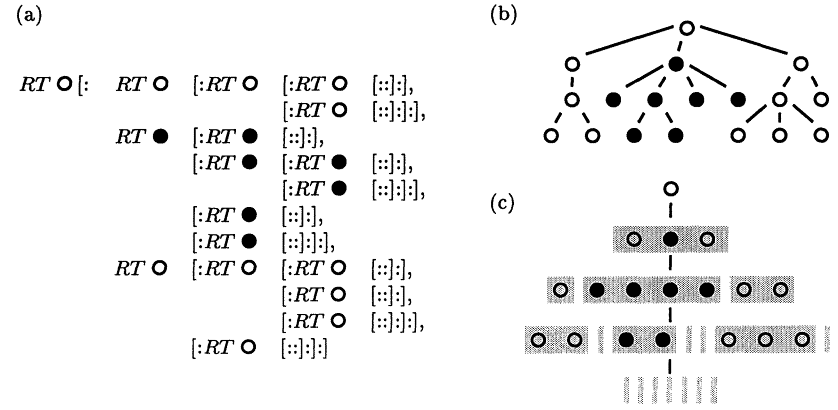

Fig. 3. Rose tree example: (a) nested term; (b) schematic representation of the source level structure; and (c) schematic representation of the flattened data structure.

and an array of containing the sizes of all subarrays. The latter is called the *segment descriptor.*

To see how the transformation affects a complex structure, consider a rose tree, defined as $RoseTree\ =\ RT\ t\ \ [:RoseTree:]$*for some primitive type t. Figure 3 shows an example of such a rose tree: Part (a) is the term representation of the tree where its node values are represented by bullets (the black bullets are used to identify one of the subtrees) and Part (b) a schematic representation of the same tree. In the flattened representation, the type changes to $RoseTree\ = RT\ t\ RoseTreeArr$, where $RoseTreeArr\ = RTA\,[:\ t\ :]\ \langle Segd, RoseTreeArr\rangle$. The type $RoseTreeArr$ is effectively a list of segmented arrays. Part (c), finally, is a schematic representation of the flattened tree: The first level is just the root value, the second level an array of all the node values of the first level of the original tree. The second level is a segmented array, where the values that are children of the same node grouped together in one subarray. Empty subarrays correspond to nodes with no children, so the last level of such a tree is always an array of empty subarrays.

3. A Cost Calculus

Next, we discuss a cost calculus for the core language based on work and depth, two complexity measures known to be useful for the analysis of data parallel algorithms [3]. We do so by defining the complexity of both the core language and a representative set of primitive operations. We assume a strict call-by-value semantics. This simplifies reasoning about the complexity of algorithms while still providing useful, albeit sometimes overly pessimistic, estimates for lazy languages.

Figure 4 defines the depth complexity of the constructs of the core language. Here, $e \twoheadrightarrow e'$ means that the e evaluates to the normal form e' and $D_{\texttt{prim}}(p\ e_1\ \cdots\ e_n)$

*Strictly speaking, our language has no data constructors, so the definition should read $RoseTree = t \times [:RoseTree:]$. However, to avoid excessive use of brackets, we write it using a data constructor RT.

$$\begin{array}{ll}
D(x) & = 0 \textbf{ if } x \textbf{ is a constant or a variable} \\
D(e\; e_1\; \cdots\; e_n) & = D(e) \;+\; \sum_{i=1}^{n}\; D(e_i) \\
& \quad + \left\{ \begin{array}{ll}
1 \;+\; D(e'/\{e'_i/v_i\}) & \textbf{if } e \rightarrow f,\; e_i \rightarrow e'_i \\
& \textbf{and } f\; v_1\; \cdots\; v_n \;=\; e' \\
& \textbf{is a top} - \textbf{level definition} \\
D_{prim}(p\; e'_1\; \cdots\; e'_i) & \textbf{if } e \;\rightarrow\; p,\; e_i \rightarrow e'_i \\
& \textbf{and } p \textbf{ is a primitive}
\end{array} \right. \\
D(\textbf{let } v \;=\; e_1 \textbf{ in } e_2) & = D(e_1) \;+\; D(e_2\{e'_1/v\}) \\
D\left(\begin{array}{l} \textbf{case } e_1 \textbf{ of } \;\; \triangleleft\, l \;\rightarrow\; e_2; \\ \qquad\qquad\quad\; \triangleright\, r \;\rightarrow\; e_3 \end{array} \right) & = D(e_1) \;+ \left\{ \begin{array}{ll} D(e_2\{e'_1/l\}) & \mathit{if}\; e_1 \;\rightarrow\; \triangleleft\, e'_1 \\ D(e_3\{e'_1/1\}) & \mathit{if}\; e_2 \;\rightarrow\; \triangleright\, e'_1 \end{array} \right.
\end{array}$$

Fig. 4. Rules for deriving the depth complexity for core language expressions.

denotes the depth complexity of applying a primitive p to the fully evaluated expressions e_1 to e_n.

As the definition of work complexity has the same structure it is omitted here for brevity. In fact, the only difference is in the complexity of parallel primitives. While surprising at first, this merely highlights the fact that, in the discussed language, parallelism is expressed solely through operations on parallel arrays. The above definition is very similar to complexity rules used for sequential languages [16]. This suggests that the emphasis should be put on the analysis of primitive operations which, as we will see next, is significantly more involved.

3.1. The Complexity of Structure Traversals

With respect to the the complexity of primitive array operations note that, after the type transformation from Figure 2, elements in parallel arrays are never shared. Structured elements are split into their primitive constituent parts and collected into arrays, which then are combined with the original structure. Overall, the resulting structure consists of *structure nodes* and flat arrays (i.e., arrays of primitive types). A function that operates on a flattened structure processes structure nodes sequentially, but each flat array in one parallel step. Accordingly, the work complexity of most primitive array operations is proportional to the overall number of elements in the flat arrays encountered during the traversal, whereas the depth depends on the number of structure nodes involved in the computation.

For instance, operator $(\mathbin{+\!\!\!+\!\!\!+})$ concatenates two arrays by traversing their flattened representations and concatenating corresponding flat arrays; it can be defined inductively over the type structure of the element type of the concatenated arrays. In other words, we can formulate $(\mathbin{+\!\!\!+\!\!\!+})$ using intensional type analysis [8] or generic programming techniques [9], as we have argued in more detail in [6].

$$\begin{array}{llll}
\multicolumn{4}{l}{(\mathbin{+\!\!\!+\!\!\!+}_{\langle\alpha\rangle}) :: \mathcal{F}[\![[{:}\alpha{:}]]\!] \times \mathcal{F}[\![[{:}\alpha{:}]]\!] \rightarrow \mathcal{F}[\![[{:}\alpha{:}]]\!]} \\
\mathit{xsl} & \mathbin{+\!\!\!+\!\!\!+}_{\langle()\rangle} & \mathit{ysl} & = \mathit{xsl} + \mathit{ysl} \\
\langle \mathit{xsl}, \mathit{xse}\rangle & \mathbin{+\!\!\!+\!\!\!+}_{\langle\mathtt{Int}\rangle} & \langle \mathit{ysl}, \mathit{yse}\rangle & = \langle \mathit{xsl} + \mathit{ysl}, \mathit{xse} \mathbin{+\!\!\!+\!\!\!+}_{\mathtt{Int}}\; \mathit{yse}\rangle \\
\langle \mathit{xsA}, \mathit{xsB}\rangle & \mathbin{+\!\!\!+\!\!\!+}_{\langle\alpha\times\beta\rangle} & \langle \mathit{xsA}, \mathit{xsB}\rangle & = \langle \mathit{xsA} \mathbin{+\!\!\!+\!\!\!+}_{\langle\alpha\rangle}\; \mathit{ysA},\; \mathit{xsB} \mathbin{+\!\!\!+\!\!\!+}_{\langle\beta\rangle}\; \mathit{ysB}\rangle \\
\mathit{xs} & \mathbin{+\!\!\!+\!\!\!+}_{\langle[:\,\alpha\,:]\rangle} & \mathit{ys} & = \mathit{xs} \mathbin{+\!\!\!+\!\!\!+}^{C}_{\langle\alpha\rangle}\; \mathit{ys} \\
\multicolumn{4}{l}{\langle \mathit{xsSel}, \langle \mathit{xsA}, \mathit{xsB}\rangle\rangle \mathbin{+\!\!\!+\!\!\!+}_{\langle\alpha+\beta\rangle} \langle \mathit{ysSel}, \langle \mathit{ysA}, \mathit{ysB}\rangle\rangle =} \\
\multicolumn{4}{l}{\qquad \langle \mathit{xsSel} \mathbin{+\!\!\!+\!\!\!+}_{\langle\mathtt{Bool}\rangle}\; \mathit{ysSel},\; \langle \mathit{xsA} \mathbin{+\!\!\!+\!\!\!+}_{\langle\alpha\rangle}\; \mathit{ysA},\;\; \mathit{xsB} \mathbin{+\!\!\!+\!\!\!+}_{\langle\beta\rangle}\; \mathit{ysB}\rangle\rangle}
\end{array}$$

The first equation says that arrays with elements of type () are concatenated by adding the integer values represented length of the arrays; arrays of primitive element type have their length components added and the flat arrays concatenated; arrays of tuples concatenate the component arrays; and so on. This definition clearly follows the equations presented in Figure 2.

Since the concatenation of two primitive arrays xs and ys has a depth of $\mathcal{O}(1)$ and a work of $\mathcal{O}(\mathsf{lengthP}\ xs + \mathsf{lengthP}\ ys)$, the parallel depth of $(+\!\!+)$ is proportional to the number of primitive parallel arrays that have to be concatenated, while the work depends both on the number and the lengths of these arrays—i.e. on the overall number of primitive values involved. In fact, this is true for most primitive operations. We define the parallel complexity of such operations in terms of two functions, `size` and `nodes`, which count primitive values and primitive nodes, respectively.

To see how the transformation of recursive data structures and work complexity interact, reconsider the rose tree example from Figure 3. Extracting a subtree from the array of children of the root node is a simple index operation in the source language and has constant cost. To extract the corresponding subtree in the flattened representation, the operation has to sequentially traverse the list and extract the corresponding chunk of values from each array in one parallel step, so the overall depth is linear with respect to the depth of the longest path in the rose tree. From an efficiency point of view, indexing represents the worst type of operations: The transformation is optimised to support collective operations; this comes at the expense of element-oriented operations.

In the following, we define the functions `size` and `nodes`, which capture the essential complexity parameters for traversals of flattened structures. In the following subsection, we then use `size` and `nodes` to formalise the complexity of the array primitives.

Definition 2 (*size*) *The function* `size`*, defined below, computes the asymptotic number of primitive values in the flat representation of an object.*

$$\begin{array}{lcl}
\mathtt{size}_{\langle \mathtt{a}\rangle}\ x & = & 1 \ \textit{if}\ \mathtt{a}\ \textit{is a primitive type} \\
\mathtt{size}_{\langle \alpha\times\beta\rangle}\ (x, y) & = & 1 + \mathtt{size}_{\langle\alpha\rangle}\ x + \mathtt{size}_{\langle\beta\rangle}\ y \\
\mathtt{size}_{\langle \alpha+\beta\rangle}\ (\mathrm{Inl}\ x) & = & 1 + \mathtt{size}_{\langle\alpha\rangle}\ x \\
\mathtt{size}_{\langle \alpha+\beta\rangle}\ (\mathrm{Inr}\ x) & = & 1 + \mathtt{size}_{\langle\beta\rangle}\ x \\
\mathtt{size}_{\langle [:\alpha:]\rangle}\ xs & = & 1 + \sum_{x\in\, xs} \mathtt{size}_{\langle\alpha\rangle}\ x
\end{array}$$

Note that `size` only yields a close upper bound on the number of primitive values encountered during the traversal of an object. The actual representation of the object in memory may require considerably less space if some parts of it are shared. For instance, a list of length n which contains the same value x in all positions can be represented in no more that $\mathcal{O}(n + \mathtt{size}\, x)$ space. However, sharing of representation does not lead to sharing of computations. During a traversal of the resulting data structure x will be encountered and processed n times, even if each time, the same computation is performed. Thus, the traversal will require $\mathcal{O}(n\ \mathtt{size}\, x)$ steps regardless of the object's representation which is precisely the complexity computed by `size`. Since we only consider finite objects, in the following discussion, we can safely assume that sharing does not occur at all.

While the size of a data structure can be determined quite easily, the number of nodes is more difficult to compute. This is due to the fact that, as described in Section 2, nodes of different array elements can be mapped to one array in the flat representation. Thus, simply counting the nodes of the elements is not sufficient as it will lead to an overly pessimistic estimate. We can, however, circumvent this problem by making use of the following observation: By neglecting sharing we can restrict ourselves to objects that are represented as (binary) trees such that each node is either a primitive values or a primitive parallel array. Obviously, there exists exactly one path from the root cell to each node of an object. Moreover, the flattening transformation will map two nodes of different parallel array elements to one parallel array if and only if the paths from the respective root cells to these nodes are equivalent. The number of nodes required to represent an array can, then, be computed from the *path sets* of its elements, where a path set of an object is the set of paths to all its nodes.

Definition 3 (*nodes*) *The function* `paths` *computes the path set of an object such that individual paths are represented by binary strings. It is defined as:*

$$
\begin{array}{lcl}
\mathtt{paths}_{\langle \mathtt{a} \rangle}\; x & = & \{[]\} \textit{ if } \mathtt{a} \textit{ is a primitive type} \\
\mathtt{paths}_{\langle \alpha \times \beta \rangle}\; (x, y) & = & \{[]\} \cup 0 \mathbin{\underline{:}} \mathtt{paths}_{\langle \alpha \rangle}\; x \cup 1 \mathbin{\underline{:}} \mathtt{paths}_{\langle \beta \rangle}\; y \\
\mathtt{paths}_{\langle \alpha + \beta \rangle}\; (\mathtt{Inl}\; x) & = & \{[]\} \cup 0 \mathbin{\underline{:}} \mathtt{paths}_{\langle \alpha \rangle}\; x \\
\mathtt{paths}_{\langle \alpha + \beta \rangle}\; (\mathtt{Inr}\; x) & = & \{[]\} \cup 1 \mathbin{\underline{:}} \mathtt{paths}_{\langle \beta \rangle}\; x \\
\mathtt{paths}_{\langle [:\alpha:] \rangle}\; xs & = & \{[]\} \cup \bigcup_{x \in xs} 0 \mathbin{\underline{:}} \mathtt{paths}_{\langle \alpha \rangle}\; x
\end{array}
$$

where $\underline{:}$ *is* $:$ *lifted to sets; i.e.,* $x \mathbin{\underline{:}} ps = \bigcup_{p \in ps} x : p$. *The function* `nodes`, *which is defined as* $\mathtt{nodes}\; x = |\mathtt{paths}\; x|$ *yields the overall number of nodes in an object.*

The relatively simple structure of types in our calculus leads to a natural encoding of paths by binary strings. Components of products and sums are distinguished by assigning different prefixes to their paths. For parallel arrays, the path set is computed by determining the path set of all elements and then taking their union, thus ensuring that each node of the array is accounted for exactly once.

In the next section, we rely on the above definitions for specifying the complexity of a number of primitive parallel operations. However, they are also useful in the broader context of arbitrary user-defined traversals. In particular, if values of primitive type are processed in constant time and primitive parallel arrays in a constant number of parallel steps, the work and depth of the computation are bounded by the size and number of nodes, respectively, of the traversed data structure. The analysis of such computations is further facilitated by an important property of `size` and `nodes`: Flattening does not change their asymptotic complexity. Section 4 shows how these insights can be applied to two well-known higher order traversals, *map* and *fold*.

3.2. Array primitives

We can now define the complexity of most primitive array operations in terms of `size` and `nodes`. For instance, with the generic definition of $(+\!\!+\!\!+)$ given in Section 3.1 it is straight forward to show that $W(xs +\!\!+\!\!+ ys) \in \mathcal{O}(\mathtt{size}\,(xs +\!\!+\!\!+ ys))$ and $D(xs +\!\!+\!\!+ ys) \in \mathcal{O}(\mathtt{nodes}\,(xs +\!\!+\!\!+ ys))$. Other primitive operations exhibit a

Table 1. Complexity of parallel primitives

Expression	Work	Depth
$xs \mathbin{+\!\!+} ys$	$\texttt{size}\ (xs \mathbin{+\!\!+} ys)$	$\texttt{nodes}\ (xs \mathbin{+\!\!+} ys)$
$xs\ !{:}\ i$	$\texttt{size}\ (xs\ !{:}\ i)$	$\texttt{nodes}\ (xs\ !{:}\ i)$
$\mathsf{replicateP}\ n\ x$	$\texttt{size}\ x$	$\texttt{nodes}\ x$
$\mathsf{lengthP}\ xs$	1	1
$\mathsf{concatP}\ xs$	1	1
$\mathsf{packP}\ fs\ xs$	$\texttt{size}\ (\mathsf{packP}\ fs\ xs) + \texttt{size}\ fs$	$\texttt{nodes}\ (\mathsf{packP}\ fs\ xs)$
$\mathsf{combineP}\ fs\ xs\ ys$	$\texttt{size}\ (\mathsf{combineP}\ fs\ xs\ ys)$	$\texttt{nodes}\ (\mathsf{combineP}\ fs\ xs\ ys)$

similar behaviour. Table 1 gives the work and depth complexities of some frequently used parallel array primitives. Here, we mainly use the result of the application of the primitive to define its complexity. Often, the same complexity class is obtained if the arguments are used instead. Thus, we obviously have $\mathcal{O}(\texttt{size}\ (xs \mathbin{+\!\!+} ys)) = \mathcal{O}(\texttt{size}\ xs + \texttt{size}\ ys)$ and $\mathcal{O}(\texttt{nodes}\ (xs \mathbin{+\!\!+} ys)) = \mathcal{O}(\texttt{nodes}\ xs + \texttt{nodes}\ ys)$. However, for example for packP, using the result actually leads to a better estimate since the operation does not have to traverse its entire argument. Note that the length of a parallel array can always be accessed from the array's root node in constant time as described in Section 2.2. Also, concatP which removes one nesting level can be executed in constant time since data and nesting information are stored separately in the flat representation.

3.3. Lifted array primitives

Each primitive array operation has a lifted version which operates on nested arrays. Recall that such arrays are represented by a pair of flat arrays. Thus, in addition to the data array a lifted primitive also has to compute the new segment descriptor. For instance, the lifted version of $\mathbin{+\!\!+}$ takes two nested arrays as arguments and joins them pairwise. Here, the new data array is generated by concatenating corresponding argument subarrays; the segment descriptor is obtained by adding the argument descriptors elementwise. Thus, the complexity of lifted $\mathbin{+\!\!+}$ is equivalent to that of its non-lifted version. This is also the case for other parallel array primitives discussed in this paper.

3.4. Operations on Sequential Structures

The three primitive tuple operations $\langle\cdot,\cdot\rangle$, fst and snd have the expected complexity of $\mathcal{O}(1)$ for both work and depth. This is also true for the lifted versions, which are, in fact, equivalent to the sequential ones due to the flattening transformation mapping arrays of tuples to tuples of arrays. The primitives $\triangleleft$ and $\triangleright$, which encode a value as a left and right component of sum, respectively, also have a constant work and depth. The lifted versions simply embed a parallel array in a tuple representing the array of sums. This is done in one parallel step. The embedding requires a flag array to be generated; thus, the work is proportional to the length of the argument array.

In Section 2.2, we have described how a **case** expression is lifted into a parallel context. Using the rules from Figure 4 and Table 1, it can be easily shown that the complexity of the lifted version is given by

$$\begin{aligned}
D((\textbf{case}\ e\ \textbf{of}\ \lhd a \rightarrow e_1;\ \rhd b \rightarrow e_2)^{\uparrow l}) &\in \mathcal{O}(D(e') + D(e_1') + D(e_2') \\
&\qquad + \textbf{nodes}\ e_1' + \textbf{nodes}\ e_2') \\
W((\textbf{case}\ e\ \textbf{of}\ \lhd a \rightarrow e_1;\ \rhd b \rightarrow e_2)^{\uparrow l}) &\in \mathcal{O}(W(e') + W(e_1') + W(e_2') \\
&\qquad + \texttt{size}\ e_1' + \texttt{size}\ e_2')
\end{aligned}$$

where $e^{\uparrow l} \rightarrow e'$, $e_1' = e_1^{\uparrow l}\{\mathsf{fst}\,(\mathsf{snd}\,e')/a\}$ and $e_2' = e_2^{\uparrow l}\{\mathsf{snd}\,(\mathsf{snd}\,e')/b\}$, provided that e_1 and e_2 do not contain free variables (if they do, we have to account for the applications of packP). Note that the complexity depends on the size and number of nodes of e_1' and e_2' which have to be combined to form a single parallel array. However, the combining step can be omitted if the two expressions evaluate to arrays containing only left and right components, respectively, of a binary sum. In this case, the arrays can be used directly to build the tuple representing the sum. The complexity is then given by

$$\begin{aligned}
D((\textbf{case}\ e\ \textbf{of}\ \lhd a \rightarrow \lhd e_1;\ \rhd b \rightarrow \rhd e_2)^{\uparrow l}) &\in \mathcal{O}(D(e') + D(e_1') + D(e_2')) \\
W((\textbf{case}\ e\ \textbf{of}\ \lhd a \rightarrow \lhd e_1;\ \rhd b \rightarrow \rhd e_2)^{\uparrow l}) &\in \mathcal{O}(W(e') + W(e_1') + W(e_2') \\
&\qquad + \mathsf{lengthP}\ e')
\end{aligned}$$

As we will see below, this situation occurs frequently in structure-preserving traversals. In the following, we assume that the optimised definition is used whenever possible.

4. Higher-order traversals

So far, we analysed the complexity of array codes including structure flattening; however, we have not yet explained the relationship between lifted functions and their original versions. In particular, the impact that flattening has on the depth complexity of programs deserves special attention. Blelloch [2] already pointed out that flattening can increase the depth complexity of recursive functions. He introduced the notion of *contained* functions to characterise functions on which flattening is well-behaved. Riely & Prins [15] took Blelloch's result one step further and, for a first-order language without product and sum types, provided a precise formalisation of which functions are negatively affected; they did so by way of a type system.

However, all this previous work did not consider data structures involving sum types and higher-order functions. In the following, we shall demonstrate that a large class of primitive recursive functions can be expressed by functions where flattening does not increase the asymptotic step complexity. We will arrive at this result taking an approach that is rather different to the mentioned previous work and make essential use of the higher-order nature of our core language. More precisely, we use a second-order combinator, namely *fold*, to formalise our statement. We first review a generic definition of *fold*, before using it to characterise the complexity of lifted functions.

4.1. Folding Product-Sum Types

We define the reduction operation *fold* by way of an algorithm introduced by Sheard & Fegaras [18]; from an algebraic data type T, it generates a definition fold_T that is sufficiently expressive to encode the recursive structure of all primitive recursive functions over T. In higher-order languages, the expressiveness of *fold* actually exceeds that of primitive recursion [10], but this is severely curtailed in our core language due to the lack of the ability to create closures.

4.1.1. The definition of *fold*

The following definitions are subject to two constraints: (a) functions appearing inside a data type may not be co-variant with respect to that data type and (b) the pretype of the folded data type may not be recursive in that data type (cf. Definition 1).

Moreover, please note that in the following two definitions, the locally defined function K is generating code in dependence of the type structure of the given data type. Hence, K is not part of the core language, but rather describes the algorithm that would be used by a compiler using the core language.

Definition 4 (*map*) *Given a type $T\,\alpha_1\,\cdots\,\alpha_n = t$ we define the recursive function map_T as*

$$
\begin{array}{lll}
map & :: & (\alpha_1 \to \beta_1) \to \cdots \to (\alpha_n \to \beta_n) \\
 & & \to T\,\alpha_1\,\cdots\,\alpha_n \to T\,\beta_1\,\cdots\,\beta_n \\
map_T\,f_1\,\cdots\,f_n\,v & = & K[t]\,v \\
\quad \textbf{where} & & \\
\quad K[()] & v & = v \\
\quad K[\mathtt{Int}] & v & = v \\
\quad K[\alpha_i] & v & = f_i\,v \\
\quad K[T\,\alpha_1\,\cdots\,\alpha_n] & v & = map_T\,f_1\,\cdots\,f_n\,v \\
\quad K[S\,t_1\,\cdots\,t_m] & v & = map_S\,(K[t_1])\,\cdots\,(K[t_m])\,v \\
\quad K[t_1 \times t_2] & v & = \langle K[t_1]\,(\mathsf{fst}\,v), K[t_2]\,(\mathsf{snd}\,v)\rangle \\
\quad K[t_1 + t_2] & v & = \textbf{case}\ v\ \textbf{of}\ \triangleleft l \to \triangleleft(K[t_1]\,l);\ \triangleright r \to \triangleright(K[t_2]\,r) \\
\quad K[t_1 \to t_2] & v & = (K[t_2]) \circ v \circ (K[t_1]) \\
\quad K[[:t:]] & v & = \mathsf{mapP}\,(K[t])\,v
\end{array}
$$

Definition 5 (*fold*) *Given a type $T\,\alpha_1\,\cdots\,\alpha_n = t$ we define the recursive function $fold_T$ as*

$$
\begin{array}{lll}
fold_T & :: & (T'\,\alpha_1\,\cdots\,\alpha_n\,\beta \to \beta) \to T\,\alpha_1\,\cdots\,\alpha_n \to \beta \\
fold_T\,f\,v & = & f\,(K[t]\,v) \\
\quad \textbf{where} & & \\
\quad K[()] & v & = v \\
\quad K[\mathtt{Int}] & v & = v \\
\quad K[\alpha_i] & v & = v \\
\quad K[T\,\alpha_1\,\cdots\,\alpha_n] & v & = fold_T\,f\,v \\
\quad K[S\,t_1\,\cdots\,t_m] & v & = map_S\,(K[t_1])\,\cdots\,(K[t_m])\,v \\
\quad K[t_1 \times t_2] & v & = \langle K[t_1]\,(\mathsf{fst}\,v), K[t_2]\,(\mathsf{snd}\,v)\rangle \\
\quad K[t_1 + t_2] & v & = \textbf{case}\ v\ \textbf{of}\ \triangleleft l \to \triangleleft(K[t_1]\,l);\ \triangleright r \to \triangleright(K[t_2]\,r) \\
\quad K[t_1 \to t_2] & v & = (K[t_2]) \circ v \circ (K[t_1]) \\
\quad K[[:t:]] & v & = \mathsf{mapP}\,(K[t])\,v
\end{array}
$$

where $T'\,\alpha_1\,\cdots\,\alpha_n\,\beta = t'$ is the pretype of T (cf. Definition 1).

```
List α  =  () + (α × List α)

Tree α  =  α + (Tree α × Tree α)

add     :: Tree' Int Int → Int
add t   =
  case t of ◁l → l;
            ▷r → fst r + snd r

sum     :: Tree Int → Int
sum t   =  fold_Tree add t

ladd    :: List' (Tree Int) Int → Int
ladd s  =
  case t of ◁l → 0;
            ▷r → sum (fst r) + snd r

lsum    :: List (Tree Int) → Int
lsum s  =  fold_List ladd s
```

Fig. 5. Example program

The program depicted in Figure 5 provides two examples of functions that can be defined using *fold*. For instance, the function *sum* computes the sum of all elements for a tree of integers. Note how the accumulating function *add* is used to distinguish between leaf and non-leaf nodes and to perform the necessary computations. In fact, an important property of Definition 5 is that it does not alter the structure of the traversed term – it is the task of the accumulating function to combine the results computed so far into a single value depending on the structure of the folded term. Thus, the `case` expression used in the definition of *fold* can be optimised as described in Section 3.4 in the lifted version, thereby improving the complexity for a large class of functions.

4.1.2. Data parallel folding

In [7], we have demonstrated how a combination of parallel and sequential data structures can be used to control the degree of parallelism which is often crucial for achieving high performance. Such algorithms present a significant challenge with respect to complexity analysis; however, expressing them as *fold* programs considerably simplifies this task. In the following, we show how the approach described in Section 3 can be applied to such codes. In particular, we rely on the functions `size` and `nodes` to capture dependencies on the structure of the traversed terms. Since their asymptotic complexity is not affected by flattening, we can restrict ourselves to flattened programs; the results thus obtained are also applicable to core language programs which contain nested parallelism. We will concentrate on establishing upper bounds for work and depth of lifted *fold*. Once these are known, deriving the complexity of the sequential version becomes straight forward.

For every type T, the definition of $fold_T$ relies on *map* for traversing product-sum types other than T. Below, we show that for the class of programs analysed in this paper, the complexity of $map^\uparrow$ is not affected by flattening. In fact, $map^\uparrow$ traverses only the top-level nodes of its argument, touching each node exactly once. Thus, its complexity is entirely dependent on the supplied functions.

Lemma 1 *Let* $f_1, \cdots, f_n$ *be functions such that for each* f_i, $D(f_i\ xs) \in \mathcal{O}(\texttt{nodes}\ xs)$. *Then,* $D(map^\uparrow_T\ f_1 \cdots f_n\ xs) \in \mathcal{O}(\texttt{nodes}\ xs)$.

Lemma 2 *Let* $f_1, \cdots, f_n$ *be functions such that for each* f_i, $W(f_i\ xs) \in \mathcal{O}(\texttt{size}\ xs)$. *Then,* $W(map^\uparrow_T\ f_1 \cdots f_n\ xs) \in \mathcal{O}(\texttt{size}\ xs)$.

Proof. By induction over the structure of xs □.

In general, considering only the complexity of the accumulating function is not sufficient for obtaining an upper bound on the complexity of $\mathit{fold}^{\uparrow}$. This is due to the fact that the accumulating function can be applied to terms which have been computed by previous applications of the same function. Thus, the structure of partial results obtained by recursive invocations of $\mathit{fold}^{\uparrow}$ has to be taken into account. For this, it is necessary to describe the behaviour of $\mathit{fold}^{\uparrow}$ in a way that allows formal reasoning about its complexity.

Definition 6 (Recursive subterms) *Let x and y be two terms in normal form. We say that y is a* recursive subterm *of x if it is a subterm of x and has the same type. We assume that every term is a recursive subterm of itself. Furthermore, y is a* maximal recursive subterm *of x if it is a recursive subterm of x, $y \neq x$ and there exists no recursive subterm of x other than x and y of which y is a recursive subterm. The set of recursive subterms of x is denoted by $\mathit{Rec}(x)$ and the set of its maximal recursive subterms by $\mathit{Max}(x)$.*

An expression of the form $\mathit{fold}_T^{\uparrow}\, f\, xs$ is evaluated by recursively folding each maximal recursive subterm of xs and applying the accumulating function f to the resulting data structure. Essentially, a bottom up traversal of xs is performed such that f is applied to each recursive subterm which is subsequently replaced by the result of the application. Note that for arrays, the number of recursive subterms is bounded by the number of nodes in the flat representation. Thus, we can immediately identify an important property of data parallel folding.

Lemma 3 *Let f be a function with the parallel depth complexity $D(f\ xs) \in \mathcal{O}(1)$. Then, $D(\mathit{fold}_T^{\uparrow}\, f\ xs) \in \mathcal{O}(\mathsf{nodes}\ xs)$.*

Lemma 3 establishes an upper bound on the depth complexity for a significant number of useful algorithms. For instance, given an array of integer trees ts, the expression $\mathit{sum}^{\uparrow}\ ts$ computes the sum of each tree such that each tree is traversed sequentially but all trees are processed in parallel. The accumulating function $\mathit{add}^{\uparrow}$ combines partial results in a constant number of parallel steps. Thus, the depth complexity of the entire computation is bounded by $\mathcal{O}(\mathsf{nodes}\ ts)$.

Unfortunately, not all algorithms can be conveniently expressed as a *fold* such that the accumulating function has a constant depth. An obvious example are nested uses of *fold* as used in the definition of *lsum*. Still, for a large class of such computations, the step complexity is not affected by flattening.

Lemma 4 *Let f be a function with $D(f\ xs) \in \mathcal{O}(\mathsf{nodes}\ xs)$ and $\mathsf{nodes}\ (f\ xs) \in \mathcal{O}(1)$. Then, $D(\mathit{fold}_T^{\uparrow}\, f\ xs) \in \mathcal{O}(\mathsf{nodes}\ xs)$.*

Proof. The expression $\mathit{fold}_T^{\uparrow}\, f\ xs$ is evaluated by performing a traversal of xs such that f is applied to $K^{\uparrow}\ rs$ for all $rs \in \mathit{Rec}(xs)$. The traversal itself requires $\mathcal{O}(\mathsf{nodes}\ xs)$ parallel steps. The complexity of the actual computations follows from the constraints on the depth of f. Thus,

$$D(\mathit{fold}_T^{\uparrow}\, f\ xs) \ \in\ \mathcal{O}(\mathsf{nodes}\ xs\ +\ \textstyle\sum_{rs\, \in\, \mathit{Rec}(xs)}\ \mathsf{nodes}\ (K^{\uparrow}\ rs))$$

We will now show that

$$\textstyle\sum_{rs\, \in\, \mathit{Rec}(xs)}\ \mathsf{nodes}\ (K^{\uparrow}\ rs) \in \mathcal{O}(\mathsf{nodes}\ xs)$$

The result of $K^\uparrow\ rs$ is a term obtained by replacing each maximal recursive subterm ms of rs by the result of $f\ (K^\uparrow\ ms)$. Thus,

$$\mathsf{nodes}\ (K^\uparrow\ rs)\ \leq\ \mathsf{nodes}\ rs\ -\ \textstyle\sum_{ms\ \in\ Max(rs)}\ \mathsf{nodes}\ ms\ +\ c$$

where c is the (constant) maximal number of nodes yielded by f. Accordingly,

$$\begin{aligned}\textstyle\sum_{rs\ \in\ Rec(xs)}\ \mathsf{nodes}\ (K^\uparrow\ rs)\ \leq\ & \textstyle\sum_{rs\ \in\ Rec(xs)}\ \mathsf{nodes}\ rs \\ & -\ \textstyle\sum_{rs\ \in\ Rec(xs)} \sum_{mss\ \in\ Max(rs)}\ \mathsf{nodes}\ ms \\ & +\ c|Rec(xs)|\end{aligned}$$

Note that every recursive subterm of xs other than xs itself is a maximal recursive subterm of exactly one recursive subterm of xs. Thus,

$$\textstyle\sum_{rs\ \in\ Rec(xs)} \sum_{ms\ \in\ Max(rs)}\ \mathsf{nodes}\ ms\ =\ \sum_{rs\ \in\ Rec(xs)}\ \mathsf{nodes}\ rs\ -\ \mathsf{nodes}\ xs$$

Furthermore, xs cannot have more recursive subterms than it has nodes, i.e.

$$|Rec(xs)|\ \leq\ \mathsf{nodes}\ xs$$

Thus,

$$\textstyle\sum_{rs\ \in\ Rec(xs)}\ \mathsf{nodes}\ (K^\uparrow\ rs)\ \leq\ (c+1)\,\mathsf{nodes}\ xs$$

□.

We have already shown that the step complexity of $sum^\uparrow$ is linear in the number of nodes of its argument. By Lemma 4, the depth of $lsum^\uparrow\ ss$ is also bounded by **nodes** ss. In fact, arbitrarily nested folds can be analysed in this way provided the accumulating functions satisfy the requirements stated above.

Lemma 5 *Let f be a function such that* $\mathtt{size}\ (f\ xs) \in \mathcal{O}(\mathsf{lengthP}\ xs)$ *and* $W(f\ xs) \in \mathcal{O}(\mathtt{size}\ xs)$. *Then,* $W(fold_T^\uparrow\ f\ xs) \in \mathcal{O}(\mathtt{size}\ xs)$.

Proof. Analogous to Lemma 4 □.

Lemma 5 defines a class of programs for which flattening does not change the work complexity. Obviously, most of these programs are also captured by Lemma 4. However, the two classes are not equivalent – in particular, accumulating functions which flatten parallel arrays often satisfy the requirements of the latter but not of the former.

5. Conclusions

We have discussed a formalism for establishing the depth and work complexity of array codes operating on data structures comprising product-sum types and nested arrays, including higher-order features. By way of a generic definition of *fold*, we demonstrated that a large class of primitive recursive functions can be formulated such that the flattening transformation does not adversely affect its asymptotic complexity.

It is known that there are programs where flattening worsens the depth complexity by sequentialising the branches of **case** expressions [15]. However, the known cases are not problematic in practice as the functions have a high work complexity, which means that there is a sufficient amount of excess parallelism on any realistic parallel machine.

There are a couple of areas into which the presented work can be extended. We are fairly certain that flattening is actually well behaved for a larger class of *fold* functions than those that we covered in this paper, but we have not found a proof for this conjecture yet. Moreover, as we can formalise flattening for functional languages that allow closure creation, it is interesting to extend the cost calculus to this case, too. In particular, this would drastically improve the expressiveness of *fold* programs.

References

[1] G. Blelloch, H. Burch, K. Crary, R. Harper, G. Miller, and N. Walkington. Persistent triangulations. *Journal of Functional Programming*, 11(5), 2001.

[2] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

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

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

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

[6] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In P. Wadler, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 94–105. ACM Press, 2000.

[7] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal—nested data parallelism in Haskell. In R. Sakellariou, J. Keane, J. R. Gurd, and L. Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference*, pages 524–534, Berlin, Germany, 2001. Springer-Verlag.

[8] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995.

[9] 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.

[10] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[11] C. Jay, M. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *LNCS*, pages 650–661. Springer-Verlag, 1997.

[12] C. B. Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, 37:207–224, 2000.

[13] 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.

[14] G. Keller and M. M. T. Chakravarty. Functional array fusion. In X. Leroy, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Pro-*

gramming (ICFP'01). ACM Press, 2001.

[15] J. Riely and J. Prins. Flattening is an improvement. In *Proceedings of the Seventh Static Analysis Symposium (SAS'2000)*, LNCS, pages 360–376. Springer-Verlag, 2000.

[16] D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[17] 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.

[18] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93*, pages 233–242, New York, 1993. ACM Press.