A Monadic Calculus for Parallel Costing of a Functional Language of Arrays

C.B. Jay[1], M.I. Cole[2], M. Sekanina[1], and P. Steckler[1]

[1] School of Computing Sciences, University of Technology, Sydney, P.O. Box 123, Broadway NSW 2007, Australia; email: {cbj,milan,steck}@socs.uts.edu.au

[2] Department of Computer Science, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Edinburgh, EH9 3JZ, Scotland; email: mic@dcs.ed.ac.uk

Abstract. Vec is a higher-order functional language of nested arrays, which includes a general folding operation. Static computation of the shape of its programs is used to support a compositional cost calculus based on a *cost monad*. This, in turn, is based on a *cost algebra*, whose operations may be customized to handle different cost regimes, especially for parallel programming. We present examples based on sequential costing and on the PRAM model of parallel computation. The latter has been implemented in Haskell, and applied to some linear algebra examples.

1 Introduction

Second-order combinators such as map, fold and zip provide programmers with a concise, abstract language for writing skeletons for implicitly parallel programs, as in [Ski94], but there is a hitch. These combinators are defined for list programs (see [BW88]), but efficient implementations (which is the point of parallelism, after all) are based on arrays. This disparity becomes acute when working with nested arrays, which are rarely supported in parallel practice. Nesl [BCH+94] is a notable exception, but still does not support folding over them. Our approach is to redevelop the theory of arrays to support the combinators while retaining constant-time access. We illustrate the efficacy of this approach by implementing *static* estimates of shapes and parallel work, as compared to, say, the dynamically obtained estimates for Nesl [BG96].

Efficient array access is based on direct access to the storage of array entries, typically in contiguous blocks of memory. List storage, however, is by allocating cons cells during evaluation, which introduces significant access costs. Our approach to arrays uses the *syntax* of lists, so that combinators can be introduced in the functional style, but constrained so that a compiler can still determine the length, or more generally the shape, of any array expression. This process of *shape analysis* can be thought of as compile-time/run-time separation in a two-level operational semantics [NN92]. Success requires that the program be *shapely*, that is, the shape of the result is determined by those of its inputs, as is typical in, say, linear algebra. Shapeliness, and successful shape analysis, require a new approach, which we have illustrated in a small functional language, Vec.

While already of interest in sequential programming, knowledge of shapes is central to efficient distribution of arrays across a network of processors. In this paper we show how to estimate the cost of parallel work in Vec. Each potential distribution of a regular array results in a cost estimate for the program, whose minimisation determines the choice. These results illustrate the feasibility of the technique. We will soon cost communications, too, using the shape to determine the length of each message, and a few hardware parameters (as in BSP [Val90] or LogP [Ca93]) to compute its transmission cost.

Costs are computed compositionally, so that we are able to associate to each program fragment a cost function, from the number of processors to time. These are combined using a *cost algebra* from which we construct a cost monad (a specialisation of computational monads [Mog89, Wad92, Gur91] which determines the cost of a program from that of its components. The specific choice of algebra varies according to the computational model. We give examples of cost algebras for sequential computation, and for a PRAM "shared-memory" model [FW78]. A Haskell implementation of our cost model is available on request to the authors.

The remaining sections of the paper are: Controlling shape; The Vec language; The Size language (in which shape analysis is performed); Cost algebras (which describe how to combine costs); Sequential costs; PRAM costs; Relationship to other work; and Conclusions and future work.

2 Controlling shape

We wish to design a functional language in which all operations are *shapely*, that is, the shapes of their results are determined by the shapes of their inputs. For example, matrix multiplication is shapely, but filtering of a list is not (since the length of the result depends on the entry values). This is a natural restriction for array programming, where storage is allocated before entry computation. The design of a language suitable for shape analysis is motivated by consideration of the following operations, which act on a vector of integers v:

$$\mathsf{if}\ (\mathsf{hd}\ v) = 1\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \tag{1}$$
$$\mathsf{if}\ (\mathsf{length}\ v) = 1\ \mathsf{then}\ v\ \mathsf{else}\ \mathsf{append}\ v\ w \tag{2}$$
$$\mathsf{if}\ (\mathsf{hd}\ v) = 1\ \mathsf{then}\ v\ \mathsf{else}\ \mathsf{append}\ v\ w \tag{3}$$
$$\mathsf{rec}\ f.\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * f(n-1) \tag{4}$$
$$\mathsf{rec}\ f.\lambda v.\mathsf{if}\ \mathsf{length}\ v > 100\ \mathsf{then}\ f\ (\mathsf{append}\ v\ w)\ \mathsf{else}\ v \tag{5}$$

1. This operation is shapely: the result is an integer, a datum whose shape is trivial and therefore unaffected by the condition.
2. Shapely: the shape of the result (the length of the resulting vector) depends on the choice of the branch, and that choice is determined by the length of v, that is, by a shape.
3. Non-shapely: the choice of the branch and thus the length of the resulting vector depends on an integer (a datum).

4. Shapely: the result is an integer and therefore its shape is known (it is the trivial shape) regardless of the number of iterations of the recursion.
5. Problematic: although shapely, the evaluation of the resulting shape may not terminate, which makes the operation unsuitable for shape analysis.

The VEC language will introduce type-based restrictions to discriminate among these examples.

3 The VEC language

The VEC language is a simply typed lambda calculus with products, a unit type, and a vector type constructor. The language as presented here is a variant of that presented in [JS97]. The types are given by

$$\begin{array}{l} D ::= \mathsf{nat} \mid \mathsf{bool} \mid \ \ldots \\ \tau \ ::= D \mid \mathsf{sz} \mid \mathsf{un} \mid \tau{\times}\tau \mid \mathsf{vec}\ \tau \\ \theta \ ::= \tau \mid \theta{\times}\theta \mid \theta \to \theta \ . \end{array}$$

D ranges over *datum* types whose shapes are trivial. τ ranges over *data* types. They include the datum types, the type sz of *sizes*, and are closed under finite products and vectors. The set of sz values is isomorphic to the set of natural numbers, but these values are shapes rather than data, and they will represent lengths of vectors and index vector entries. The vector construction can be iterated to produce arbitrarily nested arrays. For example, we define $\mathsf{mat}\ \tau = \mathsf{vec}\ (\mathsf{vec}\ \tau)$ to represent matrices with entries of type τ. The stratification of types between τ and θ disallows types for vectors of functions.

We will also need the *discrete* types, indicated by δ, which are constructed without using the vector construction or sz:

$$\delta ::= D \mid \mathsf{un} \mid \delta{\times}\delta \mid \delta \to \delta \ .$$

The VEC terms are

$$t ::= d \mid c \mid x \mid \lambda x^{\theta}.\ t \mid t\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{ifs}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{rec}\ f^{\delta}.\ t$$

d ranges over basic datum constants and operations. These are assumed to contain the nat numerals 0,1,2,... and ordinary booleans. c ranges over the combinators with non-trivial shapes. These include the *size numerals* ~0,~1,... . Basic arithmetic operations, such as $+$ and $*$, are overloaded to act on both numbers and sizes. We use sz to represent "shape booleans" with ~0 indicating truth and all other sizes indicating falsehood. The remaining combinators, and the general typing rules, are given in Figure 1. Type superscripts on terms will be elided where appropriate.

The combinators for finite products and vectors are all standard. The use of list constructors for vectors was done deliberately to emphasize that the distinction between vectors and lists is not in the typing, but in the shaping. Other combinators can be added, as long as they are shapely.

$$
\begin{array}{llll}
\mathsf{unit} & : \mathsf{un} & \mathsf{tl}^{\tau} & : \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau \\
\mathsf{pair}^{\theta,\theta'} & : \theta \to \theta' \to \theta \times \theta' & \mathsf{entry}^{\tau} & : \mathsf{vec}\ \tau \to \mathsf{sz} \to \tau \\
\mathsf{fst}^{\theta,\theta'} & : \theta \times \theta' \to \theta & \mathsf{length}^{\tau} & : \mathsf{vec}\ \tau \to \mathsf{sz} \\
\mathsf{snd}^{\theta,\theta'} & : \theta \times \theta' \to \theta' & \mathsf{map}^{\tau,\tau'} & : (\tau \to \tau') \to (\mathsf{vec}\ \tau \to \mathsf{vec}\ \tau') \\
\mathsf{sing}^{\tau} & : \tau \to \mathsf{vec}\ \tau & \mathsf{fold}^{\tau,\tau'} & : (\tau \to \tau' \to \tau') \to \tau' \to \mathsf{vec}\ \tau \to \tau' \\
\mathsf{cons}^{\tau} & : \tau \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau & \mathsf{zip}^{\tau,\tau'} & : \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau' \to \mathsf{vec}\ (\tau \times \tau') \\
\mathsf{hd}^{\tau} & : \mathsf{vec}\ \tau \to \tau & \mathsf{iter}^{\tau} & : (\tau \to \tau) \to \tau \to \mathsf{sz} \to \tau
\end{array}
$$

$$
\textbf{id}\ \frac{\Gamma(x) = \theta}{\Gamma \vdash x : \theta} \qquad \textbf{rec}\ \frac{\Gamma, f : \delta \vdash t : \delta}{\Gamma \vdash \mathsf{rec}\ f^{\delta}.\ t : \delta}
$$

$$
\textbf{abs}\ \frac{\Gamma, x : \theta \vdash t : \theta'}{\Gamma \vdash \lambda x^{\theta}.\ t : \theta \to \theta'} \qquad \textbf{app}\ \frac{\Gamma \vdash t : \theta \to \theta' \quad \Gamma \vdash t' : \theta}{\Gamma \vdash t\ t' : \theta'}
$$

$$
\textbf{if}\ \frac{\Gamma \vdash t : \mathsf{bool} \quad \Gamma \vdash t' : \delta \quad \Gamma \vdash t'' : \delta}{\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' : \delta} \qquad \textbf{ifs}\ \frac{\Gamma \vdash t : \mathsf{sz} \quad \Gamma \vdash t' : \theta \quad \Gamma \vdash t'' : \theta}{\Gamma \vdash \mathsf{ifs}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' : \theta}
$$

Fig. 1. VEC type inference rules

There are two forms of conditional: a *data conditional* (keyword if), whose condition is given by a datum (of type bool), and a *shape conditional* (keyword ifs), whose condition is a shape (of type sz). The data conditional allows the condition to be data-dependent, but ensures shapeliness by requiring the branches to be of discrete type, and hence trivial shape, as in Equation 1. By contrast, the branch taken by the shape conditional is known by shape analysis, so the branches may have arbitrary types and shapes. In VEC, Equation 2 takes the form

$$\mathsf{ifs}\ (\mathsf{length}\ v) = \tilde{}1\ \mathsf{then}\ v\ \mathsf{else}\ \mathsf{append}\ v\ w\ .$$

Note that Equation 3 cannot be expressed in VEC, as desired.

General recursion of the form rec $f^{\delta}.t$ is only supported for functions f of discrete type. This suffices for functional completeness, since the computable functions on natural numbers are all available, but does not generate non-terminating shape analyses. Finite iteration of arbitrary operations is expressed by iteration, using the iter combinator.

Let us introduce some notation to improve readability:

$$
\begin{array}{ll}
\langle t_1, t_2 \rangle & = \mathsf{pair}\ t_1\ t_2 \\
h :: t & = \mathsf{cons}\ h\ t \\
[t_1, t_2, \cdots, t_n] & = t_1 :: (t_2 :: (\cdots :: (\mathsf{sing}\ t_n)))\ .
\end{array}
$$

Also, we may write VEC functions using *patterns*, defined by the grammar

$$\rho ::= x \mid \langle \rho, \rho \rangle$$

with the restriction that any variable may appear at most once in a pattern. The notation $f\rho = t$ defines a VEC function $f = \lambda w.t'$, where w is fresh and t' is t

with the variables in ρ suitably substituted. For example, $f\ \langle\langle x,y\rangle,z\rangle = x+y+z$ is syntactic sugar for $f = \lambda w.(\mathsf{fst}\ (\mathsf{fst}\ w)) + (\mathsf{snd}\ (\mathsf{fst}\ w)) + (\mathsf{snd}\ w)$. We omit formal specification of these substitutions.

The evaluation relation is given by a standard eager operational semantics, which we omit for lack of space.

To ensure termination of analysis, in this version of VEC we allow the recursion combinator rec (and the data conditional if) to act on discrete types only. Thus, Equation 5 is not well-formed. This restriction is ameliorated by introducing additional combinators, such as map, fold, and iter.

4 The SIZE language

VEC shapes can be isolated in a sub-language called SIZE, whose types are

$$\theta ::= \mathsf{un} \mid \mathsf{sz} \mid \theta\times\theta \mid \theta \to \theta\ .$$

The discrete types in SIZE do not include datum types. Hence there is no need for data conditionals or recursion. Its terms are given by

$$t ::= c \mid x \mid \lambda x^{\theta}.\ t \mid t\ t \mid \mathsf{ifs}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$$

where c denotes those VEC combinators whose types are in SIZE. Its type inference rules are the applicable VEC rules. We define $\mathsf{bang} : \delta$ to be the canonical term of discrete size type δ. For example, for type $\theta = \mathsf{un} \to \mathsf{un}\times\mathsf{un}$, we have $\mathsf{bang}^{\theta} = \lambda x^{\mathsf{un}}.\ \langle \mathsf{unit}, \mathsf{unit}\rangle$. For any SIZE type θ, we may define a function $\mathsf{szmax}^{\theta\to\theta\to\theta}$ that gives the maximum of two SIZE terms, using pointwise maximum where θ is a function type.

We shall shortly give a translation from VEC to SIZE which generates execution costs, but first we must introduce the notion of a cost algebra.

5 Cost algebras

Cost algebras are used to model execution costs. A cost algebra has signature $(T, +, 0, \oplus, \otimes, \mathsf{max})$ with binary operations

$$\begin{aligned} +, \oplus &: T \to T \to T \\ \otimes &: \mathsf{sz} \to T \to T \\ \mathsf{max} &: T \to T \to T \end{aligned}$$

where $(T, +, 0)$ is a commutative monoid, and $\oplus$ is commutative with unit 0. We will write these operations in infix form.

The carrier T represents execution costs of some kind, for example, time. The operations $+$ and $\oplus$ are *sequential addition* and *parallel addition* of costs, corresponding to sequential and parallel composition of programs. The operation $\otimes$ is *parallel multiplication*, which determines the cost of computing in parallel many tasks that have the same cost.

It is tempting to introduce additional equations to the algebra, for example, to make $\oplus$ associative. However, associated tasks typically share resources in ways that affect the efficiency of the resulting algorithm. This lack of associativity motivates the introduction of $\otimes$ which could otherwise be treated as iterated parallel addition. In practice, static computation of $\otimes$ for n tasks takes constant time, whereas the corresponding iterated parallel addition is exponential.

For sequential program execution we take T to be clock ticks, with $+$ and $\oplus$ as addition on clock ticks, $\otimes$ as ordinary multiplication, and max as the ordinary maximum. For PRAM executions take T to be functions from numbers of processors to clock ticks, with $+$ and max as pointwise addition and maximum. Both these examples will be developed below.

We are going to track costs of our programs using a monad in the style of [Wad92], though our monad will carry some additional structure. From the programmer's perspective, a monad is a type constructor M coupled with three operations:

$$\begin{aligned}\mathsf{mmap} : (\theta \to \theta') \to M\theta \to M\theta' \\ \mathsf{unit} : \theta \to M\theta \\ \mathsf{join} : MM\theta \to M\theta\end{aligned}$$

for any types θ, θ'. (We write mmap to distinguish this operation from the VEC combinator map.) Now, any monoid $(T, +, 0)$ generates a monad by

$$\begin{aligned}M\theta &= \theta \times T \\ \mathsf{mmap}\ f\ \langle x, t\rangle &= \langle f\ x, t\rangle \\ \mathsf{unit}\ x &= \langle x, 0\rangle \\ \mathsf{join}\ \langle\langle x, t\rangle, t'\rangle &= \langle x, t + t'\rangle\ .\end{aligned}$$

$M\theta$ pairs the type θ, indicating the shape of a term, with T, the type of the cost of computing the term. To these we can add some other operations:

$$\begin{aligned}\mathsf{add}\ &:\ M\theta \to T \to M\theta \\ \mathsf{add}\ \langle x, t\rangle\ t' &= \langle x, t + t'\rangle \\ \mathsf{pre\_iter}\ &:\ (\theta \to M\theta') \to M\theta \to M\theta' \\ \mathsf{pre\_iter}\ f\ \langle x, t\rangle &= \mathsf{add}\ (f\ x)\ t\ .\end{aligned}$$

When the monoid is that of a cost algebra, then we use the parallel addition to define the operation

$$\begin{aligned}&\mathsf{capp} : M(\theta \to M\theta') \to M\theta \to M\theta' \\ &\mathsf{capp}\ \langle f, t\rangle\ \langle x, t'\rangle = \langle \mathsf{fst}\ (f\ x), (\mathsf{snd}\ (f\ x)) + (t \oplus t')\rangle\end{aligned}$$

which combines the costs associated with applications.

A *cost monad* is a monad generated by a cost algebra. Note that one could abstract away from the underlying cost algebra, just as monads abstract from monoids. The cost monads appearing in the rest of the paper are all represented in the SIZE language.

$$
\begin{array}{ll}
tycost_M(D) & = \mathsf{un} \\
tycost_M(\mathsf{un}) & = \mathsf{un} \\
tycost_M(\theta \times \theta') & = tycost_M(\theta) \times tycost_M(\theta') \\
tycost_M(\mathsf{vec}\ \theta) & = \mathsf{sz} \times tycost_M(\theta) \\
tycost_M(\mathsf{sz}) & = \mathsf{sz} \\
tycost_M(\theta \to \theta') & = tycost_M(\theta) \to M(tycost_M(\theta'))
\end{array}
$$

$$
\begin{array}{lll}
cost(d) & = \{\mathsf{bang}\} & \text{where } d \text{ is a datum constant} \\
cost(d) & = \lambda' x, y.\langle \mathsf{bang}, \mathsf{binOpConst}\rangle & \text{where } d \text{ is a binary datum operation} \\
cost(x) & = \langle x, \mathsf{varConst}\rangle & \\
cost(\lambda x.t) & = \lambda' x.cost(t) & \\
cost(t\ t') & = \mathsf{capp}\ cost(t)\ cost(t') & \\
cost(\mathsf{rec}^n\ f.t) & = cost(t)[(\mathsf{fst}\ cost(t))/f]^n[\mathsf{bang}/f] & \\
cost(\mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'') & & \\
 & = \mathsf{add}\ (\mathsf{szmax}\ cost(t')\ cost(t''))\ (\mathsf{snd}\ cost(t)) & \\
cost(\mathsf{ifs}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'') & & \\
 & = \mathsf{add}\ (\mathsf{ifs}\ (\mathsf{fst}\ (t))\ \mathsf{then}\ cost(t')\ \mathsf{else}\ cost(t''))\ (\mathsf{snd}\ cost(t)) & \\
cost(\mathsf{length}) & = \lambda' x.\langle \mathsf{fst}\ x, \mathsf{lengthConst}\rangle & \\
cost(\mathsf{entry}) & = \lambda' x, y.\langle \mathsf{snd}\ x, \mathsf{entryConst}\rangle & \\
cost(\mathsf{pair}) & = \lambda' x, y.\langle\langle x, y\rangle, \mathsf{pairConst}\rangle & \\
cost(\mathsf{hd}) & = \lambda' x.\langle \mathsf{snd}\ x, \mathsf{hdConst}\rangle & \\
cost(\mathsf{tl}) & = \lambda' x.\langle\langle(\mathsf{fst}\ x) - \tilde{}1, \mathsf{snd}\ x\rangle, \mathsf{tlConst}\rangle & \\
cost(\mathsf{fst}) & = \lambda' x.\langle \mathsf{fst}\ x, \mathsf{fstConst}\rangle & \\
cost(\mathsf{snd}) & = \lambda' x.\langle \mathsf{snd}\ x, \mathsf{sndConst}\rangle & \\
cost(\mathsf{sing}) & = \lambda' x.\langle\langle \tilde{}1, x\rangle, \mathsf{singConst}\rangle & \\
cost(\mathsf{cons}) & = \lambda' x, y.\langle\langle(\mathsf{fst}\ y) + \tilde{}1, \mathsf{snd}\ y\rangle, \mathsf{consConst}\rangle & \\
cost(\mathsf{map}) & = \lambda' f, x.\langle\langle \mathsf{fst}\ x, \mathsf{fst}\ (f\ (\mathsf{snd}\ x))\rangle, (\mathsf{fst}\ x) \otimes (\mathsf{snd}\ (f\ (\mathsf{snd}\ x)))\rangle & \\
cost(\mathsf{fold}) & = \lambda' f, x, y.\mathsf{iter}\ (\mathsf{pre\_iter}\ (\mathsf{fst}\ (f\ (\mathsf{snd}\ y))))\ \langle x, \mathsf{foldConst}\rangle\ (\mathsf{fst}\ y) & \\
cost(\mathsf{iter}) & = \lambda' f, x, y.\mathsf{iter}\ (\mathsf{pre\_iter}\ f)\ \langle x, \mathsf{iterConst}\rangle\ y & \\
cost(\mathsf{zip}) & = \lambda' x, y.\langle\langle \mathsf{fst}\ x, \langle \mathsf{snd}\ x, \mathsf{snd}\ y\rangle\rangle, \mathsf{zipConst}\rangle &
\end{array}
$$

Fig. 2. Cost translations.

We use the following notation to describe pairs in SIZE whose second element is the 0 of the cost algebra:

$$
\begin{gathered}
\{t\} = \langle t, 0\rangle \\
\lambda' x.\ t = \{\lambda x.t\}\ .
\end{gathered}
$$

also, we may write $\lambda' x, y.t$ for $\lambda' x.\lambda' y.t$, and so on.

We use the cost monad to support a syntax-directed translation from VEC to SIZE which describes the shape and cost of VEC terms. It is given in Figure 2. We may write just *cost*, suppressing the subscript, when M is understood. $tycost_M$ extends to type environments by applying it pointwise. One can show that if $\Gamma \vdash t : \theta$, then

$$tycost_M(\Gamma) \vdash cost_M(t) : M(tycost_M(\theta))\ .$$

The outermost M in $M(tycost_M(\theta))$ reflects the cost of evaluating t. Other copies of M occur in the result type of functions, to reflect the cost of applying them. The other interesting type translation is of vectors, which yields a shape, given by a length of type sz, and a cost, the uniform cost of entries.

The rationale for the costings are as follows. It costs nothing to evaluate a function, since it is a value. Costs of applications are obtained by combining costs of the function and argument using capp. Our cost model needs assistance with general recursions; the programmer must indicate the anticipated recursion depth n, using the following notation. Let $t[t'/x]^0 = t$ and $t[t'/x]^n = (t[t'/x]^{n-1})[t'/x]$. The *cost* translations of combinators typically refer to constants in T, such as $\mathsf{pairConst}$, that vary with the choice of monad and assumptions about the underlying machine, as we shall now see.

6 Sequential costs

Sequential executions are costed using the cost algebra $(T, +, \tilde{}0, +, *, \mathsf{max})$, where $T = \mathsf{sz}$. We (somewhat arbitrarily) assign times to the various cost constants by counting the number of variables appearing in the *cost* definition, for example, $\mathsf{varConst} = \tilde{}1$ and $\mathsf{foldConst} = \tilde{}4$.

Here are two examples of costing sequential programs.

$$cost(+\ 39\ 3) = \mathsf{capp}\ (\mathsf{capp}\ (\lambda' x, y.\langle \mathsf{bang}, \tilde{}1\rangle)\ \{\mathsf{bang}\})\ \{\mathsf{bang}\} \Rightarrow (\mathsf{bang}, \tilde{}1)\ .$$

The first of the pair is the shape of the integer result, and the second is the time to compute it, exactly the cost of doing the addition.

$$cost(\mathsf{fold}\ +\ 0\ [1,2,3,4]) \Rightarrow (\mathsf{bang}, \tilde{}15)\ .$$

It costs $\tilde{}7$ for building the vector: $\tilde{}1$ for a singleton, plus three conses at $\tilde{}2$ each. It costs $\tilde{}8$ for doing the fold: $\tilde{}4$ overhead (from $\mathsf{foldConst}$) plus four additions at $\tilde{}1$ each. The result $\tilde{}15$ is the sum of these two subtotals.

7 PRAM costs

For the parallel execution of VEC programs, the carrier T of an associated cost algebra is a set of functions from parallel machine descriptions to times. Here we choose the PRAM model, so a parallel machine is fully described by a number of available processors. Therefore, $T = \mathsf{sz} \to \mathsf{sz}$ represents *time functions*. As mentioned, sequential addition is pointwise addition on time functions, and max is pointwise maximum. Parallel addition is more complex. First, we define an operation that gives a time function for an optimal division of processors between two parallel tasks:

$$(f \oplus_0 g)\ p = \min_{0<q<p} \{\max \{f\ q, g\ (p-q)\}\}\ .$$

But sequential execution of the tasks may be faster, so define

$$(f \oplus g)\ p = \min\{(f+g)\ p, (f \oplus_0 g)\ p\}\ .$$

We use $\oplus$ when costing the evaluation of a pair, for instance, because the pair elements may be evaluated in parallel. Similarly, for an application, the operator and operand may be evaluated in parallel, so again $\oplus$ is used. Note that $\oplus$ is *not* associative: to compute $(f \oplus g) \oplus h$, we consider the possibility of sequentially adding f and g in parallel with h, a possibility not addressed in computing $f \oplus (g \oplus h)$.

Parallel multiplication, $\otimes$, is used to obtain a time function for running several tasks in parallel, as in mapping a function across a vector. Parallel multiplication $\otimes$ is defined by

$$
\begin{aligned}
(n \otimes f)\ p = {} & \mathsf{if}\ (n \,\mathsf{mod}\, p == \tilde{}0) \\
& \mathsf{then}\ (n \div p) * (f\ \tilde{}1) \\
& \mathsf{else}\ (n \div p) * (f\ \tilde{}1) + (f\ (p \div (n \,\mathsf{mod}\, p)))\ .
\end{aligned}
$$

The cost $(n \div p) * (f\ \tilde{}1)$ represents the cost of a sequential phase, while the rest of the cost, if any, comes from a parallel phase. In the sequential phase, we divide n tasks evenly among p processors, with perhaps some tasks left over. In the parallel phase, remaining tasks are divided among the processors. Phasing the computation into sequential and parallel parts is more efficient than simply rounding up the number of tasks n to a multiple of p.

For parallel execution, the cost constants referred to in the *cost* translation become constant time functions. We use constant time functions because the operations whose costs mention such constants cannot be parallelized. For example, $\mathsf{lengthConst} = \lambda p.\tilde{}1$ and $\mathsf{foldConst} = \lambda p.\tilde{}4$.

While folding cannot be parallelized in general, we can fold associative operations in parallel. We introduce a new combinator $\mathsf{pfold} : (\theta \to \theta \to \theta) \to \mathsf{vec}\ \theta \to \theta$ which satisfies

$$
\begin{aligned}
\mathsf{pfold}\ f\ [a] &= [a] \\
\mathsf{pfold}\ f\ (h :: t) &= f\ h\ (\mathsf{pfold}\ f\ t)\ .
\end{aligned}
$$

If an operation f has unit b, then for any vector v, $\mathsf{pfold}\ f\ v$ and $\mathsf{fold}\ f\ b\ v$ have the same value. The pfold algorithm has sequential and parallel phases, as for mapping. Knowledge of shapes allows us to redistribute work among the processors to increase efficiency, but an exhaustive search for the most efficient method would take too long. Instead, we adopt an algorithm, too complex to describe here, that constrains the number of shapes in play at any time. This approach allows us to replace iterated parallel additions by parallel multiplications.

Similarly, we introduce a term constructor that parallelises the construction of vectors. If $h : \tau$ and $t : \mathsf{vec}\ \tau$, then $\mathsf{pcons}(h,t) : \mathsf{vec}\ \tau$ is like $\mathsf{cons}\ h\ t$ but has cost given by

$$
\begin{aligned}
\mathit{cost}(\mathsf{pcons}(h,t)) &= \langle\langle n, s\rangle, n \otimes f\rangle \\
\text{where } n &= (\mathsf{length}\ t) + \tilde{}1 \\
\langle s, f\rangle &= \mathit{cost}(h)\ .
\end{aligned}
$$

That is, it represents the cost of loading the whole vector in a single, parallel operation. Note that pcons is a term constructor, rather than a combinator, since otherwise the cost of constructing the tail would be computed as part of the cost of application. A consequence is that analysing its cost is also significantly cheaper, since it avoids iterated parallel additions.

With these additional constructs we can make useful costings of linear algebra operations, such as matrix multiplication. In Figure 3, we give the costs of multiplying an $m \times 4$ matrix by a 4×1 vector on differing numbers of processors. The multiplication uses pfold for the inner product, which is mapped across the rows of the matrix. As expected of a PRAM model, the cost decreases uniformly as the number of processors increases, with little increase in the total number of cycles. More interesting is the stability of costs as the number of rows varies. With 8 processors, many costing algorithms would treat 33 rows the same way as 40 (the next multiple of 8), resulting in a significant increase in costs. Our algorithm avoids this approach, with the corresponding savings. These effects may not be notable where the number of rows is very large compared to the number of processors, but for programs consisting of many small subtasks, the cumulative savings may be significant.

		processors								
		1	2	3	4	5	6	7	8	
	32	1357	682	472	345	287	243	219	177	
	33	1399	708	472	364	303	245	219	186	execution times
m	40	1693	850	580	429	345	303	261	219	
	32	1357	1364	1416	1380	1435	1458	1533	1416	
	33	1399	1416	1416	1456	1515	1470	1533	1488	total cycles
	40	1693	1700	1740	1716	1725	1818	1827	1752	

Fig. 3. Statically-computed costs for multiplying an $m \times 4$ matrix by a 4×1 vector.

8 Relationship to other work

Any attempt to develop a parallel cost calculus is complicated by the absence of a generally accepted underlying cost model, with systems such as the PRAM [FW78] and BSP [Val90] offering competing tradeoffs between conceptual tractability and reflectivity of realistic machines. Previous work has reflected this diversity and has also largely been conducted in a relatively informal setting, particularly with respect to the central questions of composition and nesting of parallel operations.

Skillicorn presents a cost calculus for a parallelized list BMF, using the notion of a "standard topology" for a datatype to describe the minimal connectivity required for efficient support of each primitive [Ski94]. Shape information is considered but the costing of, for example, folding is tackled informally according to the shape behaviour of the accompanying operation.

Nesl [Ble96] has precisely developed mechanisms for cost composition in a "work" and "depth" circuit-complexity model, but severely restricts nesting since only mapping is allowed at outer levels, with folded computations limited to a small number of special cases. Blelloch and Greiner's profiling semantics for Core-NESL [BG96] essentially records work and space usage while fully evaluating programs. By contrast, our cost translation produces a program which contains just enough summary information about the source in order to calculate its costs, as a kind of abstract interpretation. Further, the Core-NESL profiling semantics does not make any decisions about how programs are to be parallelized. We base our costs on optimal allocations of tasks to processors, which information can be used by a parallelizing compiler. While the DAG's generated by the NESL semantics suggest a task schedule, they do not indicate *where* the tasks are to execute.

A wide survey of other related work is presented by To [To95], most of it less formal than what we propose. The "oblivious" PRAM programs and compilation techniques of [ZL94] are related to our work on shapely programs by their use of static analysis. In the sequential case, [Weg75] and [Mét88] are examples of syntax-directed cost analyses, closely related to our own work. Both [HC88] and [FSZ91] describe average-case analyses. However, our work is distinguished by its exploitation of source program structure, concise information about bulk data structures and factoring out of the cost algebra. Note that our work on shape is unrelated to the *shape types* of Fradet and Le Métayer [FM97], which are used to represent graphs for modelling pointers.

9 Conclusions and future work

VEC supports a new account of arrays that combines the benefits of the list programming style with the efficiency of array programming, by means of static shape analysis. This paper represents the first attempt (known to us) to produce a formal cost calculus for a parallel programming language of nested arrays that automatically derives costs from program source. Its power derives from shape analysis, and is expressed using the now familiar technique of monadic programming. By designing cost algebras with varying notions of parallel addition and parallel multiplication, we can straightforwardly generate new cost monads to reflect different models of parallel execution.

Future work should proceed in four main directions. First, we will improve the existing cost calculus, both to make explicit the task schedules implicit in the costing, and to optimize the speed of the analysis. Currently, parallel addition takes $O(p)$ time, where p is a number of processors; we think we can reduce this to $O(log\ p)$ or less. Second, the accuracy of the existing cost model must be tested. We plan to compile VEC programs into FORK95 programs [KS95]. This language is designed to reflect the PRAM model, and an instrumented simulator is already available. Third, the computational model should be modified to recognize communication costs. Fourth, we are exploiting VEC features in an Algol-like language, called **Fish**, which also supports imperative features.

References

[BCH+94] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1), 1994.

[BG96] G.E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN '96 International Conference on Functional Programming*, pages 213–225, 1996.

[Ble96] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[Ca93] D.E. Culler and all. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[FM97] P. Fradet and D. Le Métayer. Shape types. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 27–39. ACM, New York, 1997.

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Average case analysis of algorithms. *Theoretical Computer Science*, 1991.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings 10th STOC.* ACM Press, 1978.

[Gur91] D.J. Gurr. *Semantic frameworks for complexity.* PhD thesis, University of Edinburgh, 1991. Available as ECS–LFCS–91–130.

[HC88] T. Hickey and J. Cohen. Automating program analysis. *Journal of the Association for Computing Machinery*, 35(1):185–220, 1988.

[JS97] C.B. Jay and M. Sekanina. Shape checking of array programs. In *Computing: the Australasian Theory Seminar, Proceedings, 1997*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.

[KS95] C.W. Kessler and H. Seidl. Fork95 Language and Compiler for the SB-PRAM. In *5th Intl. Workshop on Compilers for Parallel Computers*, 1995.

[Mét88] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *4th LICS Conf.*, pages 14–23. IEEE, 1989.

[NN92] F. Nielson and H.R. Nielson. *Two-level functional languages.* CUP, 1992.

[Ski94] D.B. Skillicorn. *Foundations of Parallel Programming.* Number 6 in Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[To95] H.W. To. *Optimizing the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Dept. of Computing, Imperial College, 1995.

[Val90] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Wad92] P. Wadler. Comprehending monads. *MSCS*, 2:461–493, 1992.

[Weg75] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[ZL94] W. Zimmermann and W. Löwe. An approach to machine-independent parallel programming. In *Parallel Processing: CONPAR 94 – VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 1994.

This article was processed using the LaTeX macro package with LLNCS style