Vectorizing a non-strict functional language for a data-parallel "Spineless (not so) Tagless G-machine": DRAFT

Jonathan M.D. Hill* Department of Computer Science Queen Mary & Westfield College University of London

Abstract

The essence of data-parallelism is a $\mathcal{O}(1)$ map function. A data-parallel interpretation of map is the application of a function to every element of a parallel data structure at the same time. This model is at odds with a version of map over lists. Although list map can be interpreted as applying a function to every element of a list, in a non-strict functional language the function applications only occur to those elements of the list required by a subsequent computation.

We reconcile these opposing views of map using a three tiered model: (1) a non-strict data-parallel evaluation mechanism based upon 'aims' [5] is used that combines the "only evaluate what is required" philosophy of non-strict evaluation, with the "evaluate everything synchronously, and in parallel" mechanism of a data-parallel paradigm; (2) program transformations inspired by the map distributivity law are used to vectorize functional programs that contain map; (3) the resulting vectorized programs are compiled into machine code that mimics an abstract machine based upon the Spineless Tagless G-machine [9]. The novel features of this machine are that it incorporates the 'aim' mechanism of data-parallel evaluation, and case analysis of algebraic data-types has been vectorized by performing tag-checking in parallel. These extensions have been incorporated into the Glasgow Haskell compiler [10], and code is being generated for the CPP Distributed Array Processor (DAP), a massively parallel SIMD machine.

1 Introduction

Why should data-parallelism be incorporated into a non-strict programming language? Without going over well-trod ground [6], we identify higher order functions and lazy evaluation as being desirable in a data-parallel language. In [5, 4] a non-strict data-parallel model was presented in which the programmer expresses parallel computations in terms of PODs and POD comprehensions. PODs are parallel data-structures that are an abstraction of the processing elements of a parallel machine. They share many of the characteristics of monolithic arrays [8], however their distinguishing feature is they are unbounded and potentially infinite. Using a notation analogous to list comprehensions [3], POD comprehensions provide a framework by which communication and parallel operations on PODs can be expressed. Using these extensions, higher order functions such as map, fold and scan, each having a better complexity than their sequential counterparts, can be developed to encapsulate general patterns of parallel computation. However, unlike existing implementations of these functions, non-strictness plays an important role in that it enables potential computations on infinite PODs to be expressed. As described in Hughes [6], lazy data-structures provide a powerful mechanism of modularising programs because functions that compute over infinite data-structures can be simply composed, or 'glued' together. In practical terms this means we

^{*}This work has been supported through a SERC case award in association with Cambridge Parallel Processing. Author's address: Department of Computer Science, Queen Mary & Westfield College, University of London, Mile End Road, E1 4NS. Email: Jon.Hill@dcs.qmw.ac.uk

```
> map f [] = [] > exListA xs = sum (map (+1) xs)
> map f (x:xs) = f x : map f xs > exListB xs = sum (take 10 (map (1/) xs))
Figure 1: An example using list map
```

can ignore problems that arise from composing functions that perform computations over differing sized parallel data structures, we can use infinite PODs and perform finite computations on the resulting 'glued' functions.

2 The *aim* of non-strict data-parallelism

Effective use of a data-parallel machine relies upon the synchronous evaluation of a sufficient number of elements of a parallel object—this seems to be at odds with a non-strict evaluation strategy. To highlight this dichotomy we investigate the potential for data-parallelism using lazy lists. The mapping of the increment function (+1) in example exListA of figure 1 could be applied in a data-parallel manner to each element of the list xs, because the surrounding sum consumes all of the resulting list. As a general rule, if a map expression is enclosed by a function that is head and tail strict [14], then data-parallel evaluation of the map is *probably* feasible. Things are not quite so simple in example exListB. Because of the non-strictness associated with take, the mapping of the reciprocal function (1/) is only applied to the first ten elements of the list xs. We can see from the definition of exListB that we could apply the inner map in a data-parallel manner to the first ten elements of xs, but does this form a general rule? It seems that the delaying mechanism of non-strict evaluation throttles any possibilities of data-parallelism inherent in functional programs that use map.

A simple solution to these problems is to ensure that map is both head and tail strict [7]. However, in **exListB**, if any elements of the portion of the list **xs** after element ten contained zero, then the result of evaluation will always be \perp , regardless of whether the list element that contained zero was required. We find this unnecessary strictness uncomfortable in a non-strict language.

We propose an evaluation mechanism that combines the desirable features of the lazy and strict evaluation of *map*. Whenever a *map*-like computation is forced, multiple elements of the parallel object being mapped evaluate their results in synchrony. However, the mechanism retains a non-strict semantics. We observe from figure 1 that a portion of this list in **exListB** is forced by the evaluation of **sum**. We parallelise this process by providing the programmer with a series of IO routines¹ that evaluate a collection of the elements of a data-parallel object all at the same time. We overcome the problems faced in **exListB** by using a data-parallel non-strict evaluation mechanism that maintains a record (the *aim*) of the elements of the parallel object which need to be evaluated. Whenever a map-like computation is forced, as with the strict version of *map*, evaluation of the function applications of the *map* occur synchronously and in parallel. However, only those elements defined by the aim are evaluated. At the cost of introducing a new non-strict evaluation mechanism, *map* can be implemented with a constant time complexity, whilst retaining all the benefits of non-strict evaluation.

This paper describes techniques that form a bridge between a data-parallel non-strict language and an implementation on a massively parallel SIMD machine. In the following sections we introduce a series of program identities that *vectorize* functional programs for evaluation on a data-parallel machine. The effectiveness of vectorizing functional languages is unquestionable—a realistic (weather prediction) SISAL program out-performs a Fortran version of the same program on the CRAY Y-MP [2]. We conclude the paper with an overview of a data-parallel Spineless Tagless G-Machine, and a selection of the machines state transition rules in the Appendix. For a detailed description of the machine see [].

¹ And programmer annotations in terms of forcing routines, see [5]

 $\begin{array}{l} \overline{\mathrm{MAP}}_{1} \ (\lambda \ x \to (2 \ast x) + 4) \ vec \\ \Rightarrow & \overline{\mathrm{MAP}}_{2} \ (+) \ (\overline{\mathrm{MAP}}_{1} \ (\lambda \ x \to 2 \ast x) \ vec) \ (\overline{\mathrm{MAP}}_{1} \ (\lambda \ x \to 4) \ vec) \\ \Rightarrow & \overline{\mathrm{MAP}}_{2} \ (+) \ (\overline{\mathrm{MAP}}_{2} \ (\ast) \ (\overline{\mathrm{MAP}}_{1} \ (\lambda \ x \to 2) \ vec) \ (\overline{\mathrm{MAP}}_{1} \ (\lambda \ x \to x) \ vec)) \\ & (\overline{\mathrm{MAP}}_{2} \ (+) \ (\overline{\mathrm{MAP}}_{2} \ (\ast) \ (\overline{\mathrm{MAP}}_{1} \ (\lambda \ x \to 4) \ vec) \\ \Rightarrow & \overline{\mathrm{MAP}}_{2} \ (+) \ (\overline{\mathrm{MAP}}_{2} \ (\ast) \ \ll \ldots 2 \ldots \gg vec) \ \ll \ldots 4 \ldots \gg \\ & \overline{\mathrm{rename}} \ \begin{array}{c} \frac{\mathrm{vector}}{(+)} \ (\frac{\mathrm{vector}}{(\ast)} \ (\frac{\mathrm{vector}}{(\ast)} \ \ll \ldots 2 \ldots \gg vec) \ \ll \ldots 4 \ldots \gg \\ & \mathrm{Figure} \ 2: \ \mathrm{Example} \ \mathrm{vectorization} \end{array}$

3 Vectorization and the distributive law of map

Classically a vectorizing compiler for a language such as Fortran transforms serial programs that typically contain DO loops, into code that utilises available vector instructions of a target machine. Unfortunately vectorizing compilers for imperative languages can be a rather 'hit and miss' affair. If the body of the loop being vectorized contains any dependency cycles, then vectorisation may fail unless the compiler can spot that the body is an instance of set of predefined templates such as reduction or recurrence operations (i.e. it special cases *fold* and *scan* of trivial operations such as addition or the maximum of two integers!) [15]. Such dependency cycles arise from the excessive sequencing of statements inherent in a imperative language because of assignment. Assignment statements can cause more serious problems if a function call is present in the body of a loop being vectorized. Vectorization of the function call can only occur if the function is known to be referentially transparent [2]. Talpin [13] uses a Hindley-Milner based type system in which it is possible to delimit the scope of side effects into regions. In his FX compiler for the Connection Machine, he uses these regions to deduce if a function is referentially transparent and can therefore be vectorized.

Our goal is similar to classical vectorisation techniques, however the referential transparency of a non-strict languages makes things considerably easier for us — it will become apparent that without referential transparency, none of the algebraic identities we use to vectorize programs would hold. The inspiration for this work comes from two sources, Steele's [12] law that " α distributes over function calls"², and Bird's [1] map distributivity law (1).

$$map (f \circ g) \equiv (map \ f) \circ (map \ g) \tag{1}$$

Figure 2 shows an example of the kind of transformations we propose to perform. The purpose of the transformations is to transform programs written in a style promoted by the left hand side of (1), into a form shown on the right hand-side (the notation used in figure 2 will be described in detail in the following section).

4 Algebraic identities for vectorization

Vectorization of a functional language based upon Peyton-Jones [9] STG language is presented in the following sections. The formal operational semantics of this language expressed as a state transition system can be found in the appendix. The important characteristics of the STG language is that evaluation is performed by case expressions, whereas letrec expressions delay evaluation, resulting in a heap allocation of a closure in the abstract machine. Peyton-Jones's language is extended with the following data-parallel constructs:

- Pre-vectorization constructs that arise from the desugaring of Data Parallel Haskell programs containing POD comprehensions. These constructs are the same as the 'primitive' parallel operations proposed in [5].
- ♠ Post-vectorization constructs that are introduced by the vectorization process.

²In connection machine Lisp, α has the same meaning as map

(program)	prog	\mapsto	$\{bind\}^+$				
(binding)	bind	\mapsto	var = lf				
(lambda form)	lf	\mapsto	$\lambda \{var\}^* \to expr$				
(expression)	expr	→	<pre>letrec {bind}⁺ in expr case expr of alts default var {atom}* prim {atom}* constr {atom}* literal</pre>	local definition case analysis function application primitive operation constructor application			
			MAP _n (lf var) expr1 exprnSEND varvarFETCH varvarcase expr of palts default	POD Map & sending communication & fetching communication & parallel case analysis \$			
(atom)	atom	⊷ 	$var \mid literal \ll \dots atom \dots \gg$	constant infinite POD 🌲			
(case alternatives)	alts	→ 	{literal -> expr ;}* {constr {var}* -> expr ;}*	primitive alternative algebraic alternative			
(parallel alternative)	palts	⊷ 	{∀ literal -> expr ;}* {var}* {∀ constr -> expr ;}*	parallel primitive alternative 🌲 parallel algebraic alternative 🌲			
(case default)	default	→ 	var -> expr default -> expr	binding default wildcard default			
Figure 3: Syntax for a data-parallel extended STG language							

Data-parallelism is expressed in the STG language by $\overline{\text{MAP}}_n$ expressions. The semantics of this expression is to apply a function of arity n, in a curried manner to n primitive vectors — it is analogous to the family of Haskell map-like functions map, zipWith, zipWith3, ..., zipWith_n. The objective of vectorization is to apply successive program identities to an STG program, such that any $\overline{\text{MAP}}_n$ expressions are reduced to a form that can be directly implemented on a dataparallel machine. The primitive vectors used in the STG language are a low level abstraction of the parallel data structures of a data parallel machine (the PODs of [5] are built on-top of these vectors). Vectors are array like data structures that are unbounded and potentially infinite. The important difference between vectors and PODs is that every cell of a vector is known to be defined — i.e with an initialised array of size four, we know that there exists a cell at position two, even though that cells contents may be undefined.

4.1 Vector form

The starting point for the vectorization process is the assumption that for every primitive operation in the STG language, there exists an analogous primitive vector operation. For example given an unboxed [11] addition operation of type " $(+\#)::Int\# \rightarrow Int\# \rightarrow Int\#$ " we assume there is a vector addition primitive of type " $(+\#)::vector Int\# \rightarrow vector Int\# \rightarrow vector Int\#$ "; where the convention prim is used to represent the vector version of the primitive prim. Given the knowledge that such primitives exist, the basic program identity is the conversion of a primitive application into a vector form that is directly implementable on a data-parallel machine.

 $\begin{array}{cccc} \text{letrec} & f = \lambda \; x \; y \to (x+2) \ast y & \quad \text{letrec} & f &= \; \lambda \; x \; y \to (x+2) \ast y \\ \text{in} & \stackrel{vectorize}{\Rightarrow} & \quad \overline{\mathbf{f}} &= \; \lambda \; x \; y \to (x\overline{+} \ll \ldots 2 \ldots \gg) \overline{\ast} y \\ & \overline{\text{MAP}}_1(\lambda \; y \to f \; 1 \; y) \; vec & \quad \text{in} \\ & & \quad \overline{\mathbf{f}} \ll \ldots 1 \ldots \gg \; vec \\ & & \quad \text{Figure 4: Vectorising local bindings} \end{array}$

The goal of vectorization is to simplify all the $\overline{\text{MAP}}_n$ expressions in a STG program until they reach vector form.

4.2 Basic rules for expressions

Rule I for primitive applications is generalised in a manner similar to Steele's [12] α distribution law. By pushing the *map* inside the arguments of the application, and moving the function being mapped outwards, more opportunities for simplification to vector form are exposed.

Applications (II)

Rule III is a special case of the identity simplification " $(\lambda \ x \to x) \ y \equiv y$ ".

 $\overline{\mathrm{MAP}}_n \ (\lambda \ x_1 \dots x_i \dots x_n \to x_i) \ v_1 \dots \ v_i \dots v_n$ $\Rightarrow v_i$

Map simplification (III)

Rule IV is rather subtle in that it would seem that the side condition would be extremely hard to satisfy. The reason for the side-condition is that the expression " $\overline{\text{MAP}}_1$ ($\lambda x \rightarrow 42\#$) \perp " reduces to \perp , and not an infinite vector containing 42#. The rule is sound however because we can guarantee that each of the vectors represented by the atoms $v_1 \dots v_n$ will never be \perp , whenever such an expression is evaluated³. The rule follows from the fact that vectors are infinite data structures in which every cell (by cell we mean the location that identifies an element of a vector, and not its contents) of a vector is known to be defined. The mapping of a constant lambda form such as " $\lambda x \rightarrow 42\#$ " over such an infinite vector will always produce an infinite vector containing the number 42#. We write such an infinite vector as $\ll \ldots 42\# \ldots \gg$.

 $\overline{\text{MAP}}_n \ (\lambda \ x_1 \dots x_n \to lit) \ v_1 \dots \ v_n$ $\Rightarrow \ll \dots lit \dots \gg \qquad \text{If } v_i \not\equiv \bot; \text{ where } 1 \le i \le n$ Constants (IV)

4.3 Mapping through the 'fire break' of a local binding

The rules presented so far enable expressions such as the one shown in figure 2 to be reduced to a vector form. However, vectorization can be easily interrupted by the local bindings in a program that cause a 'fire break' through which the algebraic identities cannot be applied. For example the body of the letrec in the left of figure 4 can be reduced to " $\overline{\text{MAP}_2} f \ll \ldots 1 \ldots \gg vec$ ", in which none of the identities I-IV can be used further.

Each primitive prim in the STG language has an analogous vector primitive with semantics " $\overline{MAP}_n \ prim$ "—such expressions can be reduced to vector form. The 'fire break' caused by local bindings can be overcome in a similar way by generating 'new primitives' for each binding in the **letrec**. As with the vector form simplification, the semantics of these new bindings is equivalent to " $\overline{MAP}_n f$ " (where n is the arity of f).

³This invariant is guaranteed by the compiler, see [?] for a justification

 $\begin{array}{l} \texttt{letrec} \ f_1 = \lambda \ x_{1,1} \dots x_{1,n} \to expr_1 \\ \vdots \quad \vdots \quad & \vdots \\ f_k = \lambda \ x_{k,1} \dots x_{k,m} \to expr_k \\ \texttt{in } expr \end{array} \\ \Rightarrow \texttt{letrec} \ \frac{f_1 = \lambda \ x_{1,1} \dots x_{1,n} \to expr_1}{f_1 = \overline{\mathsf{MAP}}_n \ (\lambda \ x_{1,1} \dots x_{1,n} \to expr_1) \ x_{1,1} \dots x_{1,n} \\ \vdots \quad & \vdots \\ f_k = \lambda \ x_{k,1} \dots x_{k,m} \to expr_k \\ f_k = \overline{\mathsf{MAP}}_m \ (\lambda \ x_{k,1} \dots x_{k,m} \to expr_1) \ x_{k,1} \dots x_{k,m} \\ \texttt{in } expr \end{array}$

We adopt the convention that \overline{f} is the mapped version of the binding f. Rule VI follows in the same way as the vector form identity. If a binding \overline{f} exists, then a map expression such as the one shown in the left of figure 4 can be reduced to the expressions shown in the right—rules V and VI provide a bridge across the fire break.

$\overline{\mathrm{MAP}}_n f v_1 \dots v_n$		$\mathbf{D}'_{\mathbf{T}}$ is a simplify that $(\mathbf{V}\mathbf{I})$
$\Rightarrow \overline{\mathbf{f}} \ v_1 \dots v_n$	Only if \overline{f} exists	Binding simplification (VI)

Rule VII is a generalisation of the program identity for literals (rule III). If the x in the body of the lambda expression in rule VII is free, then the effect is to convert x into an infinite vector in which every element contains x.

$\overline{\mathrm{MAP}}_n (\lambda \ x_1 \dots x_n -$	$(x) v_1 \dots v_n $	
$\Rightarrow \ll \ldots x \ldots \gg$	If $x \notin \{x_1 \dots x_n\}$	Variable vectorization (VII)
	and $\overline{\mathbf{x}}$ does not exist	

In the current implementation of the compiler for the DAP, this seemingly trivial rule is usually the sole culprit for programs failing to vectorize! The problem is there is no general method of transforming an object of type α into an infinite vector of α s—the conversion is ad-hoc for each type. One solution to this problem is to overload the conversion process using a similar technique to Haskell overloading. As the vectorizing compiler is based upon the Glasgow Haskell compiler [10], it would be a natural choice to use Haskell's overloading mechanism. Unfortunately we cannot do this (vectorization occurs as a compiler pass after overloading has been resolved), so the compiler has to re-implement the overloading mechanism to implement rule VII.

```
> update::(Eq a, Pid a) => <<a;b>> -> a -> b -> <<a;b>>
> update vec i x = << (| a; if a == i then x else b |) | (|a;b|) <<- vec >>
```

The function update is an example in which rule VII is used during vectorization. update provides a $\mathcal{O}(1)$ updating mechanism of PODs using a map-like transformation expressed as a POD comprehension. The result of vectorizing this function is an infinite vector containing **x** at each element (rule VII), is merged with the vector to be updated (see section 5.1). This updating mechanism provides yet another solution to the constant time array updating problem in purely functional languages. By using the rather large hammer of data-parallelism, a $\mathcal{O}(1)$ complexity update is achieved, albeit at the cost of $\mathcal{O}(N)$ space complexity.

4.4 Mapping through a primitive *case* expression

The STG language provides a primitive form of case analysis similar to a C-style switch statement. The important characteristic of *case* in the STG machine is it provides the only mechanism by which evaluation is forced—the discriminant of the *case* is reduced to weak head normal form. In

$eqInt = \lambda \ x \ y \to$	case $(x \ -\# \ y)$ of		$\overline{\operatorname{eqInt}} = \lambda \ x \ y \to$	$\overline{\operatorname{Int}} = \lambda \; x \; y \to \; \overline{\operatorname{case}} \; (x \; \overline{-\#} \; y) \; of$		
	$egin{array}{ccc} 0 \# & o & 1 \#; & {}^v \ ext{default} & o & 0 \# \end{array}$	$\stackrel{vectorize}{\Rightarrow}$		∀ 0# default	$ \stackrel{\rightarrow}{\rightarrow} \ll \dots 1 \# \dots \gg; \\ \stackrel{\rightarrow}{\rightarrow} \ll \dots 0 \# \dots \gg $	
	Figure 5: Vectorisir	ig a bin	a primitive (case		

a similar vein to the rules presented so far, when vectorization reduces an expression to a *case* enclosed by a map, the solution to vectorizing the *case* is to push the map inside the alternatives of the *case*.

$$\begin{split} & \overline{\mathrm{MAP}}_{n} \begin{pmatrix} \operatorname{case} expr \text{ of } \\ lit_{1} & \to expr_{1} \text{ ;} \\ \lambda x_{1} \dots x_{n} \to \vdots & \vdots \vdots \\ lit_{k} & \to expr_{k} \text{ ;} \\ default \to expr_{d} \end{pmatrix} v_{1} \dots v_{n} \\ & \Rightarrow \overline{\mathrm{case}} \left(\overline{\mathrm{MAP}}_{n} (\lambda x_{1} \dots x_{n} \to expr) v_{1} \dots v_{n} \right) \operatorname{of} \\ & \forall lit_{1} & \to \overline{\mathrm{MAP}}_{n} (\lambda x_{1} \dots x_{n} \to expr_{1}) v_{1} \dots v_{n} \text{ ;} \\ & \vdots & \vdots \\ & \forall lit_{k} & \to \overline{\mathrm{MAP}}_{n} (\lambda x_{1} \dots x_{n} \to expr_{k}) v_{1} \dots v_{n} \text{ ;} \\ & \mathrm{default} \to \overline{\mathrm{MAP}}_{n} (\lambda x_{1} \dots x_{n} \to expr_{k}) v_{1} \dots v_{n} \text{ ;} \\ & \mathrm{default} \to \overline{\mathrm{MAP}}_{n} (\lambda x_{1} \dots x_{n} \to expr_{d}) v_{1} \dots v_{n} \end{split}$$

The operational semantics for \overline{case} is described in Appendix B.2. The salient features of an evaluation of a vectorized *case* expression as shown in the right of figure 5 are:

- The vector represented by the discriminant " $x \perp \# y$ " is evaluated in parallel;
- Every vector element of the discriminant that matches against the literal 0# evaluates the first alternative of the case expression synchronously and in parallel;
- Those vector elements failing to match against the first alternative evaluate the default alternative;
- The vector that results from the case expression is created by merging the vector resulting from the first alternative, with the vector from the default, using a priority specified by the matched literal of the discriminant;

If a variable default binding is used in a primitive case expression, then the translation of the default in rule VIII is changed to:

 $x \to expr_d \ \Rightarrow \overline{\mathbf{x}} \to \overline{\mathrm{MAP}}_{n+1}(\lambda \ x \ x_1 \dots x_n \to expr_d) \ x \ v_1 \dots v_n$

5 Vectorizing algebraic data types

Vectorization is guided by the goal of reducing expressions into a form that can be directly implemented on a data-parallel machine. Similarly, the construction of vectors containing algebraic data types (ADT's), and the scrutiny of such forms by case analysis is guided by following machine constraints:

- Constraint 1 Some data-parallel machines (i.e the CPP DAP) require that vectors only contain unboxed primitive data-types—the hardware is not suited to the parallel evaluation of vectors of pointers.
- Constraint 2 When a parallel *case* expression such as the one shown in figure 5 is evaluated, potentially all the alternative of the *case* need evaluating. The result of each of these alternatives may be a head normal form that contains as yet unevaluated closures (e.g an ADT). These resulting



head normal forms need to be "merged" into a single vector that encapsulates the meaning of the entire *case*.

Wish 1 If different elements of the discriminant of a vector contain the same data, and a successful match of a *case* alternative occurs, then each of the vector elements that matched evaluate the body of the matched alternative at the same time.

As a running example the vector shown in figure 6 is transformed into a series of representations with varying degrees of suitability for a data-parallel machine.

5.1 A naive implementation

A trivial way of fulfilling the constraints imposed on the parallel representation of constructors is to represent a vector of ADT's as a specialised, but purely sequential closure that we term a 'case closure'. A case closure, written as $\triangleleft (\alpha_1, adt_1), \ldots, (\alpha_n, adt_n) \triangleright$ is a representation for a vector of ADT's in which those processors that contain a \checkmark in the boolean vector α_i all contain the same algebraic data type represented by adt_i ; where $1 \leq i \leq n$. Figure 7 shows the case closure representation of the exemplar vector.

The down-side to this representation is that the matching of such case closures has a complexity linear to the size of the case closure. More seriously, different alternatives of the case expression may be evaluated N times; where N is the size of the case closure, and N can be much larger than the number of alternatives in the case expression. For example if the case closure contained the lists "Cons (MkInt 1#) Nil)" and "Cons (MkInt 2#) Nil", then if a case expression had a Cons alternative, this would be evaluated twice! We need a better representation for ADT's.

5.2 The "inside out" transformation

The solution we adopt to the adverse effects of case closures, is to merge case closures with the representation of algebraic data types.

5.3 The "Inside out" transformation applied to trees data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a) insideoutdate Tree a = <<Empty# | Leaf# | | Branch# | NotHere#>> <<a>> (Tree a) (Tree a)

5.4 The program identity for constructors

 $\overline{\mathrm{MAP}}_{n}(\lambda \ x_{1} \dots x_{n} \to constr \ a_{1} \dots a_{n}$ $\Rightarrow \ll \dots constr \dots \gg \begin{array}{c} (\overline{\mathrm{MAP}}_{n} \ (\lambda \ x_{1} \dots x_{n} \to a_{1}) \ v_{1} \dots \ v_{n}) \\ \vdots & \vdots & \vdots \\ (\overline{\mathrm{MAP}}_{n} \ (\lambda \ x_{1} \dots x_{n} \to a_{k}) \ v_{1} \dots \ v_{n}) \end{array}$ Constructors (IX)

5.5 The program identity for case analysis



References

- R. S. Bird. Algebraic identities for program calculation. The Computer Journal, 32(2):122-126, 1989.
- [2] D. Cann. Retire fortran? a debate rekindled. Communications of the ACM, Aug. 1992.
- [3] J. Darlington. Program transformation and synthesis: present capabilities. Technical Report 77/43, Dept of Computing and Control, Imperial College, London, Sept. 1977.
- [4] J. M. D. Hill. Data Parallel Haskell: Mixing old and new glue. Technical Report 611, Department of computer Science, QMW, Dec. 1992. Available by FTP from ftp.dcs.qmw.ac.uk in /pub/cpc/jon_hill/dpGlue.ps.
- [5] J. M. D. Hill. The aim is laziness in a data-parallel language. In K. Hammond and J. T. O'Donnell, editors, *Functional programming Glasgow*, 1993. Available by FTP from ftp.dcs.qmw.ac.uk in /pub/cpc/jon_hill/aimDpLaziness.ps.
- [6] J. Hughes. Why functional programming matters. The Computer Journal, 32(2):98-107, 1989.
- [7] G. K. Jouret. Compiling functional languages for SIMD architectures. In *Third IEEE Symposium on parallel and distributed processing*, pages 79-86, 1993.

- [8] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. Communications of the ACM, 8(2):89-101, Feb. 1965. Part 2 in CACM Vol 8(2) 1965, pages 158-165.
- [9] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 1992.
- [10] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: a technical overview. In *Joint Framework for Information Technology* (JFIT) Conference, Keele, March 1993.
- [11] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In Functional Programming Languages and Computer Architecture, Sept. 1991.
- [12] G. L. Steele Jr. and W. D. Hillis. Connection machine Lisp : Fine-grained parallel symbolic processing. In ACM Conference on Lisp and Functional Programming, pages 279-297, 1986.
- [13] J.-P. Talpin. Aspects théoriques et praticques de l'inférence de type et d'effets. PhD thesis, L'Ecole nationale supérieure des mines de Paris, May 1993. (Thesis in English).
- [14] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In Functional Programming Languages and Computer Architecture, number 274 in LNCS, 1987.
- [15] M. Wolfe. Optimizing supercompilers for supercomputers. Pitman, 1989.

Appendices: Extremely drafty

A The representation of parallel closures

Unevaluated computations are represented by closures of the form $(vs \land xs \rightarrow e) ws$. When evaluation of such a closure is completed, an update is performed that overwrites the closure with its computed head normal form. One of the distinguishing features of our data parallel implementation is its updating mechanism.

In DPHaskell we term the aim of evaluation to be the set of processors which we wish to evaluate to head normal form. Evaluation of a closure that represents an unevaluated parallel computation is performed under a particular aim—the resulting head normal form is *only* defined at those processors defined by the aim. This raises a problem; we cannot simply update a closure with its head normal form without losing the representation of those processors not defined by the aim. A solution to this problem is to introduce a hybrid closure that can represent both head normal forms *and* thunks at the same time:

$$(vs \setminus \alpha a \setminus \neg e)ws$$

The desired reading of such a 'Parallel Closure' is those processors defined by α , have a value represented by the head normal form 'a'; those processors not defined by α have a value represented by the thunk $(vs \setminus u \{\} \rightarrow e)ws$. As will become apparent latter, we distinguish between two kinds of parallel closure; those which contain constructor or primitive value head normal forms, and those which contain function values⁴.

As well as providing parallel closures, the other fundamental closure type is the 'CaseClosure'. This comes in one of two flavours: $\triangleleft \alpha_1 a_1 \dots \alpha_n a_n \triangleright$; such that each processor defined by $alpha_i$ has a constructor defined in the corresponding processor of a_1 ; $\triangleleft \alpha_1 a_1 \dots \alpha_n a_n \triangleright$ in which each of the a_1 is the address of function valued closure. The use of 'Case Closure's will be addressed in the subsequent sections.

B An overview of the data-parallel STG machine

B.1 Basic Rules

(1)

$$\overline{(Eval} (f xs) \rho \alpha) \qquad as \{\} ps rs ms us h \sigma$$
such that $val \rho \sigma f = Addr a$

$$\Rightarrow (\overline{Enter} a \alpha) \quad (val \rho \sigma xs) + as \{\} ps rs ms us h \sigma$$
(1)

$$\overline{(Enter} a \alpha) \qquad as \{\} ps rs ms us h[a \mapsto (vs \setminus n xs \rightarrow e)ws_f] \sigma$$
such that $length(as) \ge length(xs)$
(2)

$$\Rightarrow (\overline{Eval} e \rho \alpha) as' \{\} ps rs ms us h \sigma$$
where $ws_a + as' = as$

$$length(ws_a) = length(xs)$$

$$\rho = \rho[vs \mapsto ws_f, xs \mapsto ws_a]$$

⁴ These are the result of a partial application update—more on these beasts latter

B.2 Parallel Case expressions

(3)

$$(\overline{Eval} \ (\overline{case} \ e \ of \ alts) \ \rho \ \alpha) \ as \ \{\} \ ps \qquad rs \ ms \ us \ h \ \sigma$$

$$\Rightarrow (\overline{Eval} \ e \ \rho \ \alpha) \qquad as \ \{\} \ (alts, \rho, \alpha, ps) : rs \ ms \ us \ h \ \sigma$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(4)$$

$$(5)$$

$$(5)$$

$$(6)$$

$$(6)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

$$(7)$$

Unlike vanilla constructors, evaluation of a parallel constructor causes a heap allocation—the extent of the parallel object is not known until runtime. Notice how the $\overline{MergeCon}$ instruction uses the address of the constructor as its first argument.

$$(\overline{MergeCon} \ con \ \alpha^{1}) \ as \qquad \{\} \ \{\} \ r@(alt, \rho, \alpha^{2}, ps): rs \ ms \ us \ h \ \sigma$$
sanity check $\alpha^{1} \equiv \alpha^{2}$ if all is not a send
$$\Rightarrow \overline{FlatCon} \quad as \ \{(con, \alpha^{1})\} \ \{\} \qquad r: rs \ ms \ us \ h \ \sigma$$

$$(6) \qquad \overline{FlatCon} \quad as \ (con, \alpha): fs \ ps \ rs \ ms \ us \ h[con \mapsto 4\alpha_{1} \ c_{1} \cdots \alpha_{n} \ c_{n} \ \nu] \ \sigma$$

$$\Rightarrow \overline{FlatCon} \quad as \ (con, \alpha): fs \ ps \ rs \ ms \ us \ h[con \mapsto 4\alpha_{1} \ c_{1} \cdots \alpha_{n} \ c_{n} \ \nu] \ \sigma$$

$$(6) \qquad \overline{FlatCon} \quad as \ (con, \alpha): fs \ ps \ rs \ ms \ us \ h \ \sigma$$

$$(6) \qquad \overline{FlatCon} \quad as \ (con, \alpha): fs \ ps \ rs \ ms \ us \ h \ \sigma$$

$$(6) \qquad \overline{FlatCon} \quad as \ (con, \alpha^{1}): fs \ ps \ rs \ ms \ us \ h \ \sigma$$

$$(7) \qquad \overline{FlatCon} \quad as \ (con, \alpha^{1}): fs \ ps \ r@(u) \ (v_{1} \cdots v_{n} \ v_{n}) \ (v_{1} \ -> e_{n}; \ (v_{1} \ -> e_{n}; \ (v_{2} \ ->$$

The $\overline{FlatCon}$ state flattens the tree like structure of a case closure. Whenever the top most item on the flatten stack is an ordinary parallel constructor, we apply it to the return continuation, building up a list of what RHS need to be evaluated on the path stack.

When the flatten stack is empty, the path stack will contain a number of expressions that are to to be evaluated, then "merged" together. Because the resulting head normal forms may contain thunks, we allocate a case closure to represent the merged constructor. The number of slots in this case closure can be found from the depth of the path stack. We empty the return stack to force a stall on the continuation passing style of the STG machine, the address of the merged case closure is pushed onto the merge stack, ms, with the other saved stacks.

(9)

$$(\overline{MergeCon} \ con \ \alpha^{1}) \ as \ \{\} \ (\alpha^{2}, \rho, e) : ps \ \{\} \ (k, u, \alpha^{3}, ps', rs') : ms \ us \ h \ \sigma$$
such that $h \ u = \triangleleft \ a_{1} \ a'_{1} \cdots a_{k} \ a'_{k} \cdots a_{n} \ a'_{n} \triangleright$

$$\implies (\overline{Eval} \ e \ \rho \ \alpha^{2}) \ as \ \{\} \ ps \ \{\} \ (k+1, u, \alpha^{3}, ps', rs') : ms \ us \ h' \ \sigma$$
where $h' = h[u \mapsto \triangleleft \ a_{1} \ a'_{1} \cdots a_{k-1} \ a'_{k-1} \ \alpha^{1} \ con \cdots \alpha_{n} \ u_{n} \triangleright]$

The continuation passing style of the STG machine does'nt really fit in with the evaluation mechanism of the data parallel STG machine. When we evaluate a case expression, we need to evaluate *all* the RHS alternatives—hence we empty the return stack to force a stall of the STG continuation passing style, if the path stack is not empty, a merge will be performed and the next expression on the path stack evaluated.

$$(\overline{MergeCon} \ con \ \alpha^{1}) \quad as \ \{\} \ \{\} \ (n, u, \alpha^{2}, ps', rs') : ms \ us \ h \ \sigma$$
such that
$$h \ u = \triangleleft \ \alpha_{1} \ c_{1} \cdots \alpha_{n} \ c_{n} \triangleright$$

$$(10)$$

$$(10)$$

$$sanity check \ ps' = \{\} \Rightarrow rs' \neq \{\}$$

$$ps' \neq \{\} \Rightarrow rs' = \{\}$$

$$\Rightarrow (\overline{MergeCon} \ u \ \alpha^{2}) \ as \ \{\} \ ps' \ rs' \qquad ms \ us \ h' \ \sigma$$
where $h' = h[u \mapsto \triangleleft \alpha_{1} \ c_{1} \cdots \alpha_{n-1} \ c_{n-1} \ \alpha^{1} \ con \ \triangleright]$

B.3 A note on the representation of constructors

(11)

$$(\overline{Eval}\ con\ \rho\ \alpha) \qquad as \{\}\ ps\ rs\ ms\ us\ h\ \sigma$$
such that $h\ con = \triangleleft\ a_1\ a'_1 \cdots a_n a'_n \triangleright$
sanity check $a_1 \cup \cdots \cup a_n \subseteq \alpha$

$$\implies (\overline{MergeCon}\ con\ \alpha) \ as \{\}\ ps\ rs\ ms\ us\ h\ \sigma$$

—

In Peyton-Jone's STG machine, the heap only contains closures of the form " $(vs \mid xs \rightarrow e) ws$ "— for example a heap allocated constructor is wrapped in a non-updatable closure thus:

$$c \ xs \equiv (vs \setminus n \ \{\} \rightarrow c \ vs) \ xs$$

Because many of the rules 'look inside' the closure to perform updates or a merge, we adopt the opposite approach to make the presentation of the material a little clearer. Therefore a parallel constructor such as " $\ll c \gg xs$ " can be heap allocated without wrapping it within a closure. The downside to this approach is the extra rules needed for the evaluation of such closures.

B.4 Updating

$$(12) \overbrace{(\overline{Enter} \ a \ \alpha)}^{(\overline{Enter} \ a \ \alpha)} as \{\} ps rs ms us h[a \mapsto (vs \setminus u \{\} \rightarrow e)ws_f] \sigma$$

$$\implies (\overline{Eval} \ e \ \rho \ \alpha) \{\} \{\} \{\} \{\} \{\} \{\} (as, ps, rs, ms, a) : us h \sigma$$
where $\rho = [vs \mapsto ws_f]$

$$(MergeCon\ con\ \alpha) \qquad \{\} \ \{\} \ \{\} \ \{\} \ \{\} \ \{\} \ (as_u, ps_u, rs_u, ms_u, a_u) : us\ h \ \sigma$$

such that $h\ a_u = (vs \setminus u\ \{\} \rightarrow e)\ ws_f$
 $\Longrightarrow (\overline{MergeCon}\ con\ \alpha) \ as_u \ \{\} \ ps_u\ rs_u\ ms_u$
where $h_u = h[a_u \mapsto (vs \setminus \alpha\ con \setminus \neg e)\ ws_f]$

Applying the first update to a closure simply overwrites the closure with a parallel closure that contains the calculated head normal form.

$$(\overline{MergeCon}\ con\ \alpha) \qquad \{\} \ \{\} \ \{\} \ \{\} \ \{\} \ (as_u, ps_u, rs_u, ms_u, a_u) : us \ h \ \sigma$$
such that $h\ a_u = (vs \setminus \alpha'\ con' \setminus \neg e) ws_f$

$$\implies (\overline{MergeCon}\ con\ \alpha) \ as_u \ \{\} \ ps_u\ rs_u\ ms_u \qquad us \ h_u \ \sigma$$
where $h_u = h \begin{bmatrix} a_u \ \mapsto \ (vs \setminus (\alpha \oslash \alpha')\ u \setminus \neg e)\ ws_f, \\ u \ \mapsto \ d\ \alpha\ con\ \alpha'\ con' \triangleright \end{bmatrix}$

Applying an update to a re-entered parallel closure needs to merge the evaluated head normal form with its previously evaluated value.