Partial Vectorisation of Haskell Programs

Manuel M. T. Chakravarty[1], Roman Leshchinskiy[1], Simon Peyton Jones[2], and Gabriele Keller[1]

[1] Programming Languages and Systems, School of Computer Science and Engineering, University of New South Wales, {chak,rl,keller}@cse.unsw.edu.au
[2] Microsoft Research Ltd, Cambridge, England, simonpj@microsoft.com

Abstract. Vectorisation for functional programs, also called the flattening transformation, relies on drastically reordering computations and restructuring the representation of data types. As a result, it only applies to the purely functional core of a fully-fledged functional language, such as Haskell or ML. A concrete implementation needs to apply vectorisation selectively and integrate vectorised with unvectorised code. This is challenging, as vectorisation alters the data representation, which must be suitably converted between vectorised and unvectorised code. In this paper, we present an approach to partial vectorisation that selectively vectorises sub-expressions and data types, and also, enables linking vectorised with unvectorised modules.

Keywords: Vectorisation; flattening; program transformation; Haskell

1 Introduction

The idea of implementing *nested data parallelism* [1] in functional programs by a vectorising program transformation is at least as old as Blelloch & Sabot's seminal work [2, 3] on the *flattening transformation.* We have since generalised the basic idea to cover the central features of modern functional languages, such as algebraic data types, parametric polymorphism, and higher-order functions [4–6]. However, apart from a prototype that compiled a subset of Paralation Lisp, the only complete implementation of vectorisation by flattening was, to the best of our knowledge, the experimental NESL system [7]. Due to its experimental nature, NESL was a rather limited functional language; for example, it did not admit user-defined algebraic data types and higher-order functions, had only rudimentary I/O, and did not have a module system or support separate compilation. In fact, NESL was implemented as a whole program compiler that vectorised the entire program.

In our implementation of vectorisation in the Glasgow Haskell Compiler (GHC) as part of the *Data Parallel Haskell project* [8], we cannot follow NESL's approach. We expect that a real application will consist of a computationally-intensive core that must be vectorised, embedded in a larger program that parses its command line, reads configuration files, drives a GUI, outputs Postscript, and so on. None of this surrounding code can or should be vectorised.

Consequently, we need a form of *selective* vectorisation that vectorises as much as possible, but leaves sub-expressions that depend on impure features, or unvectorised external code as is. Moreover, we must integrate vectorised with unvectorised code, which is challenging because vectorisation alters the representation of data structures and functional values; hence, values need to be suitably converted when passed between vectorised and unvectorised code.

This paper makes the following technical contributions:

- We give the first presentation of a *selective vectorisation transformation* that only vectorises sub-expressions that do not rely on impure or otherwise non-vectorisable features and in particular on external, unvectorised data structures and code (Section 3).
- We describe the integration of vectorised and unvectorised modules and the additional information that vectorisation requires to be maintained across separately compiled modules (Section 4).

Before we dive into these technical details, Section 2 summarises the main ideas of vectorisation and introduces our approach by example. We cover related work in Section 5.

2 Vectorisation in a nutshell

The various aspects of vectorising purely functional programs including algebraic data types, parametric polymorphism, and higher-order functions were described in detail in previous work [4–6, 8]. In the following, we summarise the core ideas with a concrete example. Afterwards, we motivate and informally illustrate the main ideas of the present paper by extending our example to include I/O.

2.1 Data Parallel Haskell

Data Parallel Haskell (DPH) introduces a type of *parallel arrays*, denoted `[:e:]` for arrays of type `e`, together with a large number of parallel collective operations. As far as possible, these operations have the same names as Haskell's standard list functions, but with a `P` suffix added—i.e., `mapP`, `filterP`, `unzipP`, and so forth. The language also includes parallel array comprehensions which are similar to list comprehensions but operate on parallel arrays. Details of the language extension and examples are in [9].

The crucial difference between Haskell lists and parallel arrays is that the latter have a parallel evaluation semantics. More precisely, demand for any element of a parallel array results in the evaluation of all elements—in particular, on a parallel machine, we expect the evaluation of these elements to happen in parallel. Figure 1 gives an excerpt from a two-dimensional Barnes-Hut n-body simulator, an example that we chose because it is both computationally intensive and very hard to express using flat data parallelism. The parallelism happens inside `oneStep`, where all particles are processed in a single parallel step and where the main workhorses `buildTree` and `accelerate` (which, in turn, contain

```
type Vector   = (Float, Float)
type Area     = (Vector, Vector)

data MassPnt  = MassPnt { mass :: Float, location :: Vector }
data Particle = Particle { center :: MassPnt, velocity :: Vector }

data Tree = Node MassPnt [:Tree:]    — Rose tree for spatial decomposition

— Perform spatial decomposition and build the quadtree
buildTree :: [:MassPnt:] -> Tree

— Change velocity of each particle in ps according to force affected by masses in tree
accelerate :: Tree -> [:Particle:] -> [:Particle:]

— Move a mass center according to the given velocity
movePnt :: MassPnt -> Vector -> MassPnt

— Move a particle according to its velocity
moveParticle :: Particle -> Particle
moveParticle p = let v = velocity p
                 in Particle (movePnt (center p) v) v

— Compute one step of the n-body simulation
oneStep :: Float -> [:Particle:] -> [:Particle:]
oneStep ps
  = [: moveParticle p | p <- accelerate tree ps :]
  where
    mps   = [:mp | Particle mp v <- ps:]
    tree  = buildTree mps
```

Fig. 1. Excerpt of 2-D Barnes-Hut n-body code

parallel computations) are invoked. The function `buildTree` constructs a quadtree and `accelerate` uses it to compute the acceleration of a set of particles in $O(n \log n)$ work complexity. All this needs to be vectorised for parallel execution. More precisely, we need to ensure that we call the fully vectorised variants of the functions $\texttt{buildTree}_\text{V}$, $\texttt{accelerate}_\text{V}$, and $\texttt{moveParticle}_\text{V}$, to achieve $O(\log^2 n)$ parallel step complexity; details are in [10, 4, 9].

2.2 Full vectorisation and why it may fail

Consider a top-level function definition $f :: t \texttt{ = } e$, where t is the (monomorphic) type of f. The *full vectorisation* transformation generates a new variant of f, thus:

$$f_\text{V} \texttt{ :: } \mathcal{T}[\![t]\!] \texttt{ = } \mathcal{V}[\![e]\!] \quad \text{— If } e \text{ is vectorisable}$$

```
oneStepIO :: [:Particle:] -> IO [:Particle:]
oneStepIO ps
  = do { print qs; return qs }          — Side effecting I/O computation
  where
    mps  = [:mp | Particle mp v <- ps:]  — purely functional code...
    tree = buildTree mps                 — ...that must be...
    qs   = [: moveParticle p             — ... vectorised and...
           | p <- accelerate tree ps:]   — ... run in parallel
```

Fig. 2. Parallel code mixed with I/O

Here, f_{V} is the *fully vectorised* variant of f, whose right-hand side is generated by the *full vectorisation transform* $\mathcal{V}[\![\cdot]\!]$. Full vectorisation returns an expression of a different type to the input, so the type of f_{V} is obtained by vectorising the type t, thus $\mathcal{T}[\![t]\!]$. In general, if $e :: t$ then $\mathcal{V}[\![e]\!] :: \mathcal{T}[\![t]\!]$.

Full vectorisation is the flattening transformation of Blelloch and Sabot, subsequently elaborated by ourselves to handle polymorphism, user-defined algebraic data types, and higher-order functions [4–6]. It is, however, not the subject of this paper, so we will keep details of full vectorisation to a minimum.

In real programs, however, full vectorisation of the entire program may be neither possible, nor even desirable. Haskell[1] supports a significant number of impure features, including monadic I/O and mutable variables, exceptions, thread-based concurrency, and calls to external C code. Code using these impure features resists vectorisation due to such code's dependence on a particular evaluation order.

As an example, consider the code in Figure 2 which extends the original `oneStep` with a call to the I/O function

```
print :: Show a => a -> IO ()
```

which prints values to the standard output. Its purpose here is to output the state of simulated particles after each time step, for example, to drive an animation. We cannot vectorise the entire body of `oneStepIO` *because we do not have a vectorised version of* `print`. In all likelihood, the module `System.IO`, which exports `print`, will not have been compiled with vectorisation in the first place, since vectorising it would be pointless. But even if we tried to vectorise `System.IO`, we would still not get $\texttt{print}_{\mathrm{V}}$ because of this function's dependence on sequential C procedures. In short, the full vectorisation transform $\mathcal{V}[\![e]\!]$ may *fail*. It may fail because it encounters some impure feature in e that prevents vectorisation; or because e mentions some imported function f that was compiled without vectorisation, or for which vectorisation failed. In the latter case, no binding for f_{V} would have been created.

[1] Here we mean the extension of Haskell 98 implemented by GHC, which has many additional features that will be in the next standard. Nevertheless, already Haskell 98 supports a range of I/O operations.

2.3 Selective vectorisation

We cannot vectorise the whole of `oneStep`, but we still want to selectively vectorise *as much of it as possible*, so that the code computing the new particles is evaluated in parallel (the `where` clause in Figure 2). In general, for each top-level binding f`::`t `=` e we apply the *selective vectorisation transform* $\mathcal{S}[\![\cdot]\!]$ to e, thus:

$$f \texttt{ :: } t \texttt{ = } \mathcal{S}[\![e]\!]$$

In contrast to full vectorisation, selective vectorisation keeps the result type the same—if necessary by introducing suitable conversions. This is exactly what happens with `oneStepIO`, for example.

In fact, even if we *are* able to fully vectorise f, we must still retain a binding of name and type f`::`t. After all, f might be exported and used by a module not compiled with vectorisation or used in a context that we cannot vectorise. To keep matters simple and predictable, we therefore generate the binding f`::`t `=` $\mathcal{S}[\![e]\!]$ regardless of whether or not full vectorisation succeeds.

Conversions. The selective vectorisation transform should vectorise the pure, performance-critical part of `oneStepIO` and get $\texttt{qs}_\text{V}$ `::` $\mathcal{T}[\![\texttt{[:Particle:]}]\!]$. This means that although we cannot vectorise `print qs`, we still want to use $\texttt{qs}_\text{V}$ as the argument to `print`. But the types do not match! So we must convert from $\mathcal{T}[\![\texttt{[:Particle:]}]\!]$ to `[:Particle:]` using the (overloaded) function `fromV`. Thus, selective vectorisation of `oneStepIO` should turn `print qs` into `print (fromV` $\texttt{qs}_\text{V}$`)`.

The functions `fromV` and `toV` marshal arguments and results "across the border" between un-vectorised and vectorised code. The selective vectorisation transform generates suitable `fromV` and `toV` functions, based on the types to be marshaled. However, this is neither possible nor desirable for all types—for complicated types it is simply too expensive. Hence, selective vectorisation decides which sub-expressions to vectorise, using both

- the presence or absence of vectorised versions $\texttt{f}_\text{V}$ of the free variables `f` of the expression, and
- the presence or absence of conversion functions `fromV` and `toV` at the required types (i.e., the types of the free variables and result).

We formalise this idea in Section 3.2.

Optimality. In general, there is more than one way to selectively vectorise a given expression. This raise the question of which of multiple translations to choose, and especially, whether one translation is "better" than the others. Unfortunately, these questions are not easy to answer as we have two potentially opposing requirements. On one hand, we want to vectorise as much code as possible—after all, only vectorised code will make good use of parallelism. On the other hand, the use of the conversion functions `fromV` and `toV` can be expensive if large data structures are converted, especially if that happens repeatedly

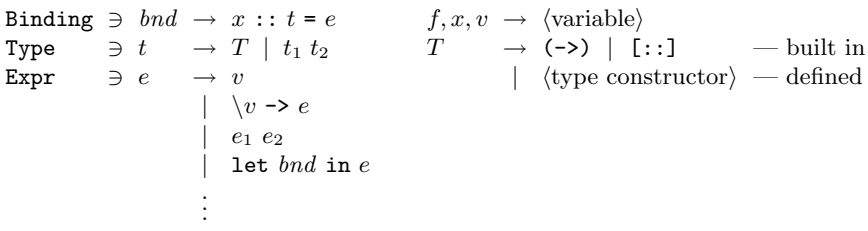

Fig. 3. Fragment of GHC's Core intermediate language

in a recursive function. We leave a detailed analysis of this trade off and the development of a cost model or a heuristic approach to decide on which sub-expression to vectorise for future work. The transformation formalised in the next section simply attempts to vectorise as many sub-expression as possible. However, some guidance by the programmer is possible as the transformation does not assume that conversions are available for all types; i.e., by not having conversions for types inhabited by values that may be costly to convert (e.g., complex tree structures), programmers can indirectly guide selective vectorisation.

3 Selective vectorisation precisely

Our description of selective vectorisation has been entirely informal thus far. In the rest of the paper we give a more precise description. The presentation is based on our earlier work [11, 6], where we introduced vectorisation as a *total* transformation on a source language that included only vectorisable constructs, types, and primitive functions. In what follows we show how to extend this work to a source language that does not have this convenient property. We do this by specifying a *partial* transformation that may fail for some expressions, and by precisely characterising which parts of an expression are vectorised, and which are not. Our implementation uses a particular source language — namely, GHC's Core language [12] – but our method will work for *any* language.

In the following, we use double square brackets $[\![\cdot]\!]$ not only to denote source code fragments that are transformed by one of our transformation functions (i.e., to denote source code "arguments"), but also for the source code "results" of these transformation functions. In other words, we use $[\![\cdot]\!]$ much like quasi-quotes in meta-programming systems, such as Template Haskell.

We will consider only *monomorphic* programs. Our system is quite capable of handling polymorphism (and must do so for Haskell), but polymorphism adds complications that distract from the main point of this paper, which is partial vectorisation.

Figure 3 displays the fragment of Core that is relevant for the present paper. The left-hand side column gives the names for the syntactic categories that we use in the following to give type signatures to translation schemes.

3.1 Vectorising types

In general, if $e :: t$ then $\mathcal{V}[\![e]\!] :: \mathcal{T}[\![t]\!]$. In such situations it is usually illuminating to look at the type transform first. There is one case for each form of type in Figure 3[2]:

```
T[[·]] :: Type -> Type
T[[(->)]]              = (:->)
T[[[::]]]              = PA
T[[T]] | ⟨T_V exists⟩  = T_V
T[[t1 t2]]             = T[[t1]] T[[t2]]
```

The type transform simply replaces functions (`->`) with vectorised functions (`:->`), and arrays (`[::]`) with vectorised arrays (`PA`), and other data types `T` with a vectorised version of that type, $\texttt{T}_\text{V}$.

In general, like the term transform, the type transform is *partial*: given a type constructor `T`, $\mathcal{T}[\![\texttt{T}]\!]$ fails if $\texttt{T}_\text{V}$ does not exist. There are two cases to consider: either `T` is a *primitve* type, or it is an *algebraic data type*, which we consider next in turn.

Primitive types. For some primitive types, such as `Int`, the vectorised version is the same as the ordinary version; that is, $\texttt{Int}_\text{V} = \texttt{Int}$. But for other primitive types, there might *be* no vectorised version; for example $\mathcal{T}[\![\texttt{IO}]\!]$ fails, because there is no vectorised version $\texttt{IO}_\text{V}$ of Haskell's IO monad.

User-defined algebraic data types. Suppose `T` is a user-defined algebraic data type `T`. In order to vectorise code involving `T` we need its vectorised version $\texttt{T}_\text{V}$. We can generate $\texttt{T}_\text{V}$ from `T` by by vectorising its component types in the obvious way. Thus, for example,

```
data T  = C t1 t2 | D
```

generates the new data type declaration:

```
data T_V = C_V T[[t1]] T[[t2]] | D_V
```

If any of the argument types cannot be vectorised, then neither can `T`. In the special (but very common) case where $\mathcal{T}[\![\texttt{t1}]\!] = \texttt{t1}$ and $\mathcal{T}[\![\texttt{t2}]\!] = \texttt{t2}$, we can avoid creating a fresh data type, instead simply setting $\texttt{T}_\text{V} = \texttt{T}$, just as we do for `Int`. So, returning to Figure 1, we have $\texttt{MassPnt}_\text{V}$ = `MassPnt` and $\texttt{Particle}_\text{V}$ = `Particle`. In contrast, for `Tree` we get

```
data Tree_V = Node_V MassPnt (PA Tree_V)
```

[2] We use Haskell's guard notation here. The guard "| $\langle\texttt{T}_\text{V}$ exists$\rangle$" means "this equation applies only if $\texttt{T}_\text{V}$ exists".

Functions. The vectorised version of function arrow `(->)` is the type of vectorised functions `(:->)`, but how is `(:->)` defined? Consider the following (contrived) example:

```
app :: (Int -> Int) -> (Int, [:Int:])
app f = (f 1, [:f x | x <- [:1, 2, 3:]:])
```

Here we apply `f` outside and inside an array comprehension; in the former case we must run `f` sequentially, but in the latter it should be evaluated in parallel. To support parallel application of `f`, vectorisation generates a data-parallel, or *lifted* version of `f`, denoted by $\texttt{f}^{\uparrow}$, such that if `f :: t -> u`, then $\texttt{f}^{\uparrow}$ `:: PA` $\mathcal{T}[\![\texttt{t}]\!]$ `->` `PA` $\mathcal{T}[\![\texttt{u}]\!]$ (for full details of lifting, see [11]). In the fully-vectorised version of `app`, we therefore need `f`'s regular as well as its lifted variant, so we must pass *both* versions of `f` to $\texttt{app}_\texttt{V}$. To a first approximation, therefore, the type `(:->)` is defined thus:

```
data a :-> b = MkFun (a -> b) (PA a -> PA b)
```

That is, vectorisation replaces a function of type `t -> u` by a *pair* of functions, of type (`PA` $\mathcal{T}[\![\texttt{t}]\!]$ `->` $\mathcal{T}[\![\texttt{u}]\!]$`, PA` $\mathcal{T}[\![\texttt{t}]\!]$ `->` `PA` $\mathcal{T}[\![\texttt{u}]\!]$`)`. This definition is not quite right, because of nested functions and partial applications [6], but the details are not important for this paper. All that we need is the existence of the vectorised function constructor `(:->)`, and its apply operator

```
($:) :: (a :-> b) -> a -> b
```

Vectorised arrays. Under selective vectorisation, the non-vectorised (and hence sequential) part of the program may still manipulate "parallel" arrays. For example, we might read a file to create a parallel array of type `[:Int:]`, that is then passed to a vectorised computation. Conversely, in the function `oneStepIO` in Figure 2, we consume a parallel array produced by a vectorised computation in sequential I/O code.

While the type of parallel arrays `[:a:]` in the source language is parametric, the representation of arrays and array operations after vectorisation depends on the element type. For example, an array of pairs is represented as a pair of arrays. The reasons for this requirement, and a sketch of how we realise this in our implementation using *type families*, are provided in previous work [4, 8].

Concretely, `PA` is a type-indexed data type family representing vectorised arrays. In GHC's type-family notation [13] we write

```
data family PA (a::*)
```

Then we give a `data instance` declaration for each type that we want to store in a vectorised array. For example:

```
data instance PA (a,b) = PAPair (PA a) (PA b)
```

Hence, for each user-defined algebraic data type, we must generate a `data instance` declaration that describes how a vectorised array of such values is represented. For example, the `Tree` type in Figure 1 generates the following declaration:

```
data instance PA Tree = Node_PA (PA MassPnt) (Segd, PA Tree)
```

`Segd` is a *segment descriptor* encoding the structure of nested arrays; c.f., for example [8] for more details of our use of type-indexed data types.

3.2 Vectorising expressions

Now we are ready to consider the selective vectorisation of expressions. Our approach relies on three mutually recursive transformation schemes defined in Figure 4:

```
𝒱⟦·⟧  :: Expr t -> Env -> Maybe (Expr 𝒯⟦t⟧)
𝒮⟦·⟧  :: Expr t -> Env -> Expr t
𝒮𝒱⟦·⟧ :: Expr t -> Env -> (Maybe (Expr 𝒯⟦t⟧), Expr t)
```

Full vectorisation $\mathcal{V}[\![\cdot]\!]$, and selective vectorisation $\mathcal{S}[\![\cdot]\!]$, have already been introduced, although here we give them types that (a) express partiality by returning a `Maybe`, and (b) express the type transformation by parameterising `Expr`.

The definitions of $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{S}[\![\cdot]\!]$ do not directly depend on each other. Instead, the recursive knot is tied by $\mathcal{SV}[\![\cdot]\!]$ which uses both transformations to transform sub-expressions. $\mathcal{SV}[\![\cdot]\!]$ acts as a mediator between selective and partial vectorisation. Its main task is to intertwine vectorised and unvectorised code by introducing appropriate conversions.

The transformations are parametrised with an environment which maps variables to their vectorised versions if available. This information is required by partial vectorisation to transform variables, as apparent in the corresponding rule taken from Figure 4:

```
𝒱⟦x⟧ env
  | (x ↦ x_V) ∈ env = Just ⟦x_V⟧
  | otherwise       = Nothing
```

In the following, we look at the transformations in more detail and explain what happens at the interfaces between vectorised and unvectorised code.

3.3 Embedding unvectorised sub-expressions

To see how the transformations defined in Figure 4 allow for mixing vectorised and unvectorised code, let us consider an example that demonstrates how vectorised code may depend on unvectorised code. Assume a variable `M.constTable :: [:Int:]` defined in a module `M` that was not compiled with vectorisation; i.e., $\texttt{M.constTable}_\text{V}$ does not exist. In a naive implementation, we might abandon the vectorisation of an expression such as `sumP M.constTable` altogether and evaluate it sequentially. However, this is clearly suboptimal; instead, we would like to convert `M.constTable` to a vectorised representation (this is easily possible for arrays of primitive types) and pass it to the vectorised, i.e., parallel implementation of `sumP`. Ultimately, we would like to have

```
𝒮⟦·⟧ :: Expr t -> Env -> Expr t
𝒮⟦x⟧ env     = ⟦x⟧
𝒮⟦e₁ e₂⟧ env = ⟦e₁S e₂S⟧
  where
    (_, e₁S) = 𝒮𝒱⟦e₁⟧ env
    (_, e₂S) = 𝒮𝒱⟦e₂⟧ env
```

. . . ⟨similar for other cases of $\mathcal{S}[\![\cdot]\!]$⟩ . . .

```
𝒮⟦let x :: t = e₁ in e₂⟧ env
  = ⟦let bs in e₂S⟧
  where
    (bs, env') = 𝒮𝒱_B⟦x :: t = e₁⟧ env
    e₂S = 𝒮⟦e₂⟧ env'

𝒱⟦·⟧ :: Expr t -> Env -> Maybe (Expr 𝒯⟦t⟧)
𝒱⟦x⟧ env
  | (x ↦ x_V) ∈ env = Just ⟦x_V⟧
  | otherwise       = Nothing
𝒱⟦e₁ e₂⟧ env
  | (Just e₁V, _) <- 𝒮𝒱⟦e₁⟧ env
  , (Just e₂V, _) <- 𝒮𝒱⟦e₂⟧ env = Just ⟦e₁V $: e₂V⟧
  | otherwise                  = Nothing
𝒱⟦let x :: t = e₁ in e₂⟧ env
  | (Just e₂V, _) <- 𝒮𝒱⟦e₂⟧ env' = Just ⟦let bs in e₂⟧
  | otherwise                    = Nothing
  where
    (bs, env') = 𝒮𝒱_B⟦x :: t = e₁⟧ env
```

. . . ⟨similar for other cases of $\mathcal{V}[\![\cdot]\!]$⟩ . . .

```
𝒮𝒱⟦·⟧ :: Expr t -> Env -> (Maybe (Expr 𝒯⟦t⟧), Expr t)
𝒮𝒱⟦e⟧ env = case 𝒱⟦e⟧ env of
              Just e_V
                | ⟨fromV e_V exists⟩ -> (Just e_V, ⟦fromV e_V⟧)
                | otherwise          -> (Just e_V, e_S)
              Nothing
                | ⟨toV e_S exists⟩   -> (Just ⟦toV e_S⟧, e_S)
                | otherwise          -> (Nothing, e_S)
  where
    e_S = 𝒮⟦e⟧ env

𝒮𝒱_B⟦·⟧ :: Binding -> Env -> ([Binding], Env)
𝒮𝒱_B⟦x :: t = e⟧ env
  | ⟨t is vectorisable⟩
  , (Just e_V, e_S) <- 𝒮𝒱⟦e⟧ env' = ([x::t = e_S, x_V::t_V = e_V], env')
  | otherwise                    = ([x::t = 𝒮⟦e⟧ env], env)
  where
    env'  = env ∪ {x ↦ x_V}
```

Fig. 4. Selective vectorisation

$$\mathcal{V}[\![\texttt{sumP M.constTable}]\!]\ \texttt{env} = \texttt{Just}\ [\![\texttt{sumP}_\text{V}\ \texttt{\$:}\ (\texttt{toV M.constTable})]\!]$$

In other words, we would like vectorisation to succeed for the entire expression even though it fails for one of the sub-expressions. That is why $\mathcal{V}[\![\cdot]\!]$ uses $\mathcal{SV}[\![\cdot]\!]$ to vectorise sub-expressions as it is the latter that can introduce the necessary conversions. For example, the rule for vectorising application given in Figure 4 passes the two sub-expressions on to $\mathcal{SV}[\![\cdot]\!]$ which tries to transform them such that they can be used in a vectorised context. In our example, vectorisation immediately succeeds for `sumP` which has a vectorised version:

$$\mathcal{SV}[\![\texttt{sumP}]\!]\ \texttt{env} = (\texttt{Just}\ [\![\texttt{sumP}_\text{V}]\!], [\![\texttt{sumP}_\text{S}]\!])$$

but fails for `M.constTable` which has no vectorised variant. Fortunately, $\mathcal{SV}[\![\cdot]\!]$ is able to rectify this by introducing a conversion:

$$\begin{aligned}&\mathcal{SV}[\![\texttt{M.constTable}]\!]\ \texttt{env}\\&\quad = (\texttt{Just}\ [\![\texttt{toV M.constTable}_\text{S}]\!]), [\![\texttt{M.constTable}_\text{S}]\!])\end{aligned}$$

This enables the application rule of $\mathcal{V}[\![\cdot]\!]$ to succeed, producing the desired result. Note that the definition of $\mathcal{V}[\![\cdot]\!]$ does not contain any interfacing logic—interfacing is delegated entirely to $\mathcal{SV}[\![\cdot]\!]$. This is also the reason why we only included three example rules in the definition of $\mathcal{V}[\![\cdot]\!]$—the complete definition can be obtained by using $\mathcal{SV}[\![\cdot]\!]$ in place of direct recursion in the definition of vectorisation given in [11] and by accounting for partiality in the exact same manner as we demonstrated for application.

3.4 Vectorising as much as possible

Using unvectorised in vectorised code is only half of the story, however. Arguably much more important is the ability to pass results of parallel computations to inherently unvectorisable tasks such as I/O. In fact, we have already seen an example where this is absolutely essential: the statement `print qs` in Figure 2 outputs the result of a computation which we expect to be executed in parallel and which, therefore, must be vectorised. Again, it is the task of $\mathcal{SV}[\![\cdot]\!]$ to introduce the necessary conversion before passing the computed value to the unvectorisable `print`.

The mechanism employed here is quite similar to the one discussed in the previous section. Since we cannot vectorise `print`, we have:

$$\mathcal{SV}[\![\texttt{print}]\!]\ \texttt{env} = (\texttt{Nothing}, [\![\texttt{print}_\text{S}]\!])$$

The situation is quite different for `qs`, however. Not only can it be fully vectorised to $\texttt{qs}_\text{V}$, the latter can also be converted to an unvectorised representation. In such a case, $\mathcal{SV}[\![\cdot]\!]$ will throw away the result of selectively vectorising the sub-expression and simply suitably convert the fully vectorised version. In our example, this amounts to:

$$\mathcal{SV}[\![\texttt{qs}]\!]\ \texttt{env} = (\texttt{Just}\ [\![\texttt{qs}_\text{V}]\!], [\![\texttt{fromV qs}_\text{V}]\!])$$

For the overall expression, the transformation rule for selectively vectorising applications then generates

$$\mathcal{S}[\![\texttt{print qs}]\!]\ \texttt{env} = [\![\texttt{print}_\text{S}\ (\texttt{fromV qs}_\text{V})]\!]$$

This is optimal in the sense that we evaluate as much as possible in parallel (namely all of `qs`) before passing the result to the inherently sequential code. The definitions of $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{SV}[\![\cdot]\!]$ ensure that the transformation always prefers fully vectorised expression to using selectively vectorised ones, thereby preserving the maximum degree of parallelism while still allowing for partiality of vectorisation.

3.5 Vectorising `let`

During selective vectorisation, `let` bindings are handled in just the same way as that described in Sections 2.2 and 2.3. Given a single[3] binding $x\texttt{::}t\ \texttt{=}\ e$, we always produce the selectively vectorised binding $x\texttt{::}t\ \texttt{=}\ \mathcal{S}[\![e]\!]$. Additionally, if e can be fully vectorised, we generate the fully vectorised binding $x_\text{V}\texttt{::}\mathcal{T}[\![t]\!]\ \texttt{=}\ \mathcal{V}[\![e]\!]$. In this case, we also extend the environment to account for the newly introduced vectorised version of x such that it is visible during the vectorisation of the body. This is described by the `let` cases of $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{S}[\![\cdot]\!]$, which invoke the transformation scheme $\mathcal{SV}_B[\![\cdot]\!]$ to deal with the binding. The same transformation is used to handle top-level bindings.

As an example, the body of `moveParticle` from Figure 1 is vectorised as follows:

$$\begin{array}{l}\mathcal{V}[\![\texttt{let v = velocity p in Particle (movePnt (center p) v) v}]\!]\ \texttt{env}\\ \quad\texttt{= Just}\ [\![\texttt{let}\ \texttt{v}_\text{V}\ \texttt{=}\ \texttt{velocity}_\text{V}\ \texttt{\$:}\ \texttt{p}_\text{V}\\ \qquad\qquad\texttt{v}\ \ \texttt{= fromV (velocity}_\text{V}\ \texttt{\$:}\ \texttt{p}_\text{V}\texttt{)}\\ \qquad\texttt{in Particle}_\text{V}\ \texttt{\$:}\\ \qquad\qquad\texttt{(movePnt}_\text{V}\ \texttt{\$: (center}_\text{V}\ \texttt{\$:}\ \texttt{p}_\text{V}\texttt{)}\ \texttt{\$:}\ \texttt{v}_\text{V}\texttt{)}\ \texttt{\$:}\ \texttt{v}_\text{V}]\!]\end{array}$$

Obviously, there is ample room for optimisation here. For example, when $\mathcal{SV}_B[\![\cdot]\!]$ generates two bindings in a local `let`, only one of them may be used in the `let` body, but simple dead-code elimination will excise the unused binding.

What about recursion?[4] Whether the right-hand side of a recursive binding such as `r :: t = f r` can be vectorised depends, among other things, on whether a fully vectorised version of `r` is available. Of course, we cannot know if $\texttt{r}_\text{V}$ is available until we have tried to vectorise the right-hand side. Obviously, this is a classical case of circular dependency which our transformation must resolve.

Our solution to this problem is somewhat simple-minded, but effective. First, we attempt to vectorise the right-hand side assuming that $\texttt{r}_\text{V}$ exists. If it can be

[3] Both in the definition of the transformation and in the subsequent discussion, we restrict ourselves to a single, possibly recursive binding of the form `let` x `::` t `=` e_1 `in` e_2. Our method easily generalises to multiple mutually-recursive bindings.

[4] NB: Haskell does not distinguish between value and function bindings. Any `let` binding can be recusive.

fully vectorised, we generate two bindings as described above using the two expressions produced by $\mathcal{SV}[\![\cdot]\!]$. If partial vectorisation fails, however, we are faced with an additional complication. We cannot use the selectively vectorised expression that was produced by $\mathcal{SV}[\![\cdot]\!]$ under the assumption that $\texttt{x}_\text{V}$ exists; after all, it may contain references to the latter. The following example demonstrates this. Suppose we have the following functions:

```
f   :: Int -> T
g   :: T -> Int
gV  :: TV :-> Int
```

Here, `g` has a vectorised version but `f` does not. Moreover, we assume that the type `T` is vectorisable but does not support conversion to and from $\texttt{T}_\text{V}$. Then, for the binding `x :: T = f (g x)` partial vectorisation would fail but selective vectorisation, if performed under the assumption that $\texttt{x}_\text{V}$ exists, would yield `x :: T = f (toV (`$\texttt{g}_\text{V}$ `$: `$\texttt{x}_\text{V}$`))`. This definition is unusable, however, as we have no binding for $\texttt{x}_\text{V}$ after all. We have to selectively vectorise the binding once more, this time omitting $\texttt{x}_\text{V}$ from the environment. In our example, this would leave the original binding unchanged which is, indeed, the only possible solution.

4 Selective vectorisation of modules

Between all the modules forming a program, some modules will be compiled with vectorisation and some will not. In the following, we discuss how these two types of modules fit together and what extra information vectorisation requires to be communicated between separately compiled modules.

4.1 Modules compiled without vectorisation

The scheme for vectorising expressions, and in particular bindings, presented before is designed such that modules compiled without vectorisation

1. do not have to be aware of vectorisation at all and
2. are just a special case of those compiled with vectorisation.

Concerning Point (1), vectorised modules still contain all the original type definitions of the source and, although bindings of the form v `::` t `=` e are transformed into v `::` t `=` e_S, their interface remains the same. Concerning Point (2), even if a module is compiled with vectorisation, it is conceivable that none of its functions or type declarations is actually vectorisable under partial vectorisation. In this case, the interface of the module remains as if it hadn't been compiled with vectorisation at all. (Nevertheless, some expression bodies may have been selectively vectorised.)

4.2 Interface files

GHC uses *interface files* to communicate exported type declarations and function signatures between separately compiled modules. Whenever a module `A` is compiled, an interface file `A.hi` is generated from its type information. Whenever a module `B` imports `A`, the compiler reads `A.hi` to obtain type information for all imported entities.

If optimisation is enabled (and vectorisation is a form of optimisation), GHC emits further information into interface files. This additional information includes information computed by code analysis (such as strictness information) as well as the right-hand sides of function definitions that are considered for cross-module inlining.

When a module is compiled with vectorisation, GHC includes the following additional information in the interface file:

- For each value or function binding `x`, GHC indicates whether $\texttt{x}_\text{V}$ exists.
- For each type constructor `T`, GHC indicates whether $\texttt{T}_\text{V}$ exists, and if so, whether `T` itself serves as $\texttt{T}_\text{V}$. If $\texttt{T}_\text{V}$ exists, we may also have the type-specific conversion functions `toV` and `fromV`.
- For each type constructor `T`, we have a type family instance defining the representation of the non-parametric array representation described in Section 3.1. This always exists as we use parametric, boxed arrays as a fallback for types where we cannot derive an optimised representation.

5 Related work

There is a long list of work concerning vectorisation, some of which we have mentioned throughout this paper. We have discussed much of this previous work in [8] and will refrain from repeating this discussion. To the best of our knowledge, none of this previous work has mentioned partial vectorisation. In particular, NESL [3, 7] was implemented as a whole program compiler performing wholesale vectorisation.

The Proteus system [14] promised a combination of data and control parallelism, but Proteus had a particular focus on manual refinement of algorithms and where data parallel components were automatically vectorised, this again was a complete whole-program transformation. Moreover, the system was never fully implemented.

Manticore [15] supports a range of forms of parallelism including nested data parallelism. Manticore employs some of the same techniques as we do, but it does not seem to use vectorisation in the same form. The Manticore implementation is, at the time of writing, work in progress.

6 Conclusion

We argued that vectorisation for a fully fledged functional language needs to be partial; i.e., only part of a program is vectorised. We presented and in part formalised a partial vectorisation transformation that vectorises sub-expressions

selectively and uses conversions where necessary to integrate vectorised and unvectorised code. At the time of writing, we are working on completing a first version of vectorisation for GHC, which includes our partial vectorisation strategy. All code is publicly accessible from the GHC HEAD repository at `http://darcs.haskell.org/ghc/`.

References

1. Blelloch, G.E.: Programming parallel algorithms. Communications of the ACM **39**(3) (1996) 85–97
2. Blelloch, G.E., Sabot, G.W.: Compiling collection-oriented languages onto massively parallel computers. Journal of Parallel and Distributed Computing **8** (1990) 119–134
3. Blelloch, G.E.: Vector Models for Data-Parallel Computing. The MIT Press (1990)
4. Keller, G., Chakravarty, M.M.T.: Flattening trees. In Pritchard, D., Reeve, J., eds.: Euro-Par'98, Parallel Processing. Number 1470 in Lecture Notes in Computer Science, Berlin, Springer-Verlag (1998) 709–719
5. Chakravarty, M.M.T., Keller, G.: More types for nested data parallel programming. In Wadler, P., ed.: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM Press (2000) 94–105
6. Leshchinskiy, R., Chakravarty, M.M.T., Keller, G.: Higher order flattening. In: Third International Workshop on Practical Aspects of High-level Parallel Programming (PAPP 2006). Number 3992 in LNCS, Springer-Verlag (2006)
7. Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. Journal of Parallel and Distributed Computing **21**(1) (April 1994) 4–14
8. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data Parallel Haskell: a status report. In: DAMP 2007: Workshop on Declarative Aspects of Multicore Programming, ACM Press (2007)
9. Chakravarty, M.M.T., Keller, G., Lechtchinsky, R., Pfannenstiel, W.: Nepal—nested data parallelism in Haskell. In Sakellariou, R., Keane, J., Gurd, J.R., Freeman, L., eds.: Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference. Number 2150 in Lecture Notes in Computer Science, Berlin, Germany, Springer-Verlag (2001) 524–534
10. Barnes, J., Hut, P.: A hierarchical $O(n \log n)$ force calculation algorithm. Nature **324** (December 1986)
11. Leshchinskiy, R.: Higher-Order Nested Data Parallelism: Semantics and Implementation. PhD thesis, Technische Universität Berlin (2005)
12. Peyton Jones, S., Santos, A.: A transformation-based optimiser for Haskell. Science of Computer Programming **32**(1–3) (1998) 3–47
13. GHC Team: Type families. `http://haskell.org/haskellwiki/GHC/Type_families` (2007)
14. Mills, P., Nyland, L., Prins, J., Reif, J.: Software issues in high-performance computing and a framework for the development of hpc applications. In: Computer Science Agendas for High Perfromance Computing, ACM Press (1994)
15. Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Status report: The manticore project. In: 2007 ACM SIGPLAN Workshop on ML, ACM Press (2007)