Chapter 5

**ALGORITHMS FOR SIMD ARCHITECTURE**

Classically a vectorising compiler for a language such as Fortran, transforms serial programs that contain DO loops, into code that utilises available vector instructions of a target machine. If a loop contains no dependency cycles, i.e., the state during each iteration of the loop is independent of any previous states, then the loop is an abstraction of MAP, and can be vectorised. However, if the loop does contain dependency cycles then vectorisation may fail unless the compiler can spot that the body is an instance of a set of predefined templates such as reduction or recurrence operations—i.e.; it special cases fold and scan of operations such as the addition or maximum of two integers [44]. Such dependency cycles arise from assignment statements. Assignments can cause more serious problems if function calls are present in the body of a loop being vectorised. Vectorisation of the function call can only occur if the function is known to be referentially transparent [22]. The solutions to these problems fall into four camps, providing languages where:
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

- There are syntactic restrictions on the type of parallel for loops that can be expressed. This approach is exemplified by CMFortran's [19] forall statement that enables a single parallel assignment to be vectorised. Alternatively, the syntax of the for loop and assignment can be merged as with Fortran 90's [19] array operations;

- The user guarantees referential transparency as with the pure annotation for functions in High Performance Fortran [19].

- Functions are analysed using an abstract interpretation technique [16] to determine if they are referentially transparent. New developments have occurred in this field over the last couple of years. Talpin's "effects system" [74] extends a Hindley Milner type

\[
\begin{align*}
\text{MAP}_1 (\lambda x \rightarrow (2 + x) + 4) \text{ vec} \\
\Rightarrow \text{MAP}_2 (+) (\text{MAP}_1 (\lambda x \rightarrow 2 + x) \text{ vec}) (\text{MAP}_1 (\lambda x \rightarrow 4) \text{ vec}) \\
\Rightarrow \text{MAP}_2 (+) (\text{MAP}_2 (+) (\text{MAP}_1 (\lambda x \rightarrow 2) \text{ vec}) (\text{MAP}_1 (\lambda x \rightarrow z) \text{ vec})) (\text{MAP}_1 (\lambda x \rightarrow 4) \text{ vec}) \\
\Rightarrow \text{MAP}_2 (+) (\text{MAP}_2 (+) (\text{MAP}_1 (\lambda x \rightarrow 2) \text{ vec}) (\text{MAP}_1 (\lambda x \rightarrow 4) \text{ vec})) \\
\Rightarrow \text{MAP}_2 (+) (\text{MAP}_2 (+) \ll \ldots 2 \ldots \gg \text{ vec}) \ll \ldots 4 \ldots \gg \\
\Rightarrow \text{MAP}_2 (+) (\text{MAP}_2 (+) \ll \ldots 2 \ldots \gg \text{ vec}) \ll \ldots 4 \ldots \gg
\end{align*}
\]

Figure 5.1: Example vectorization
system in such a way that 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 therefore susceptible to vectorisation;

- The language does not contain assignment statements, so the problem cannot arise.

This is the approach taken by the SISAL community [26].

Non-strict functional languages contain no assignment statements as they would interact awkwardly with the semantic interpretation of the language. As a consequence, the last of the above approaches is adopted here, in conjunction with more classical vectorisation techniques. It will become apparent that without referential transparency, none of the program identities used to vectorise programs would hold. The inspiration for this work comes from two sources: Steele's [66] law that "α distributes over function calls" , and Bird's [8] MAP distributive law:

\[
\text{map} (f \circ g) = \text{map} f \circ \text{map} g
\]

Identity 3: "map distributive law"

Imperative data-parallel languages provide a large number of monolithic first-order operations that are a minor abstraction of the machine code instructions of a SIMD or vector machine. In contrast, a functional language "distills" these monolithic operations into a small but powerful collection of higher-order functions. Figure 5.1 shows an example of the kind of transformations propose to perform. The purpose of the transformation is to
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

rewrite programs written in the style of the left hand side of the MAP distributive law, into a form shown on the right hand side (the notation used in figure 5.1 will be described in detail in the following sections). Eventually the transformation produces functions of the form MAP₂ (+) which can be implemented directly in terms of the primitives of a data-parallel machine.

5.1 Program identities for vectorisation

In Chapter 3 a translation scheme for pod comprehensions was introduced that replaced comprehensions by the MAP, INDICES, SEND, and FETCH primitive parallel operations. Our objective now is to vectorise the MAP expressions into a form that is closer to how "real" data-parallel machines operate. One of the characteristics of the prior translation is that it converted "Haskell+pod comprehensions" into "Haskell+primitive parallel operations"-- a localised source-to-source translation. The translation scheme we are now going to use isn't quite so localised. Its effects will be "rippled" throughout the entire program being vectorised. The transformations required for vectorisation are performed in Peyton Jones STG language [58], a constrained, yet sugared version of the lambda calculus. A partial translation of Haskell to the STG language is given in Peyton Jones [58]. The extra translation needed on top of Peyton Jones's scheme are trivial by having the same parallel primitives in desugared Haskell and the STG language, the translation scheme would therefore apply the identity translation to the parallel primitives.
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

The formal operational semantics of the extended STG language expressed as a state transition system can be found in the next chapter. As with Peyton Jones's STG language, the important characteristics of the language are 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-vectorisation constructs that arise from desugaring Data Parallel Haskell programs containing pod comprehensions;
- Post-vectorisation constructs that are introduced by the vectorisation process.

The presentation of STG programs in this chapter doesn't always follow the syntax of figure 5.2 as braces are often omitted from lambda expressions. The reason for this is that we want the vectorisation rules to look similar to transformations on the lambda calculus, without minor syntactic clutter detracting from the nature of vectorisation.

In summary, vectorisation performs two important transformations to the STG-language:

1. It transforms MAP\textsubscript{n} expressions into a form which can be directly implemented on a SIMD or vector machine;
2. It transforms algebraic data types into a form that is also susceptible to data parallel computation. The first twelve rules are concerned with the vectorisation process.
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

\[
\begin{align*}
\text{(program)} & \quad \text{prog} \quad \mapsto \quad \{\text{bind}\}^+ \\
\text{(binding)} & \quad \text{bind} \quad \mapsto \quad \text{var} = 1f \\
\text{(lambda)} & \quad \text{lf} \quad \mapsto \quad \lambda\text{vars} \to \text{expr} \\
\text{(expression)} & \quad \text{expr} \quad \mapsto \quad \text{letrec} \{\text{bind}\}^+ \text{ in expr} \\
&& \quad \text{case expr of alts default} \quad \text{local definition} \\
&& \quad \text{var atoms} \quad \text{case analysis} \\
&& \quad \text{prim atoms} \quad \text{function application} \\
&& \quad \text{consr atoms} \quad \text{primitive operation} \\
&& \quad \text{literal} \quad \text{constructor application} \\
&& \quad \text{MAP}_n \{\text{if} | \text{var} \text{ expr}_1 \cdots \text{expr}_n\} \quad \text{POD Map} \\
&& \quad \text{INDICES} \text{ var} \quad \text{Indices of a vector} \\
&& \quad \text{SEND} \text{ var var} \quad \text{send communication} \\
&& \quad \text{FETCH} \text{ var var} \quad \text{fetch communication} \\
&& \quad \text{CASE expr OF patls default} \quad \text{parallel case analysis} \\
\end{align*}
\]

\[
\begin{align*}
\text{(seq alt)} & \quad \text{alts} \quad \mapsto \quad \{\text{literal} \to \text{ expr }\}^* \quad \text{primitive alternative} \\
& \quad \mapsto \quad \{\text{constr} \{\text{var}\}^* \to \text{ expr }\}^* \quad \text{algebraic alternative} \\
\text{(par alt)} & \quad \text{patls} \quad \mapsto \quad \{\forall \text{ literal} \to \text{ expr }\}^* \quad \text{parallel primitive alt.} \\
& \quad \mapsto \quad \{\text{var}\}^* \{\forall \text{ constr} \to \text{ expr }\}^* \quad \text{parallel algebraic alt.} \\
\text{(default)} & \quad \text{default} \quad \mapsto \quad \text{var} \to \text{ expr} \quad \text{binding default} \\
& \quad \mapsto \quad \text{default} \to \text{ expr} \quad \text{wildcard default} \\
\text{(atom)} & \quad \text{atom} \quad \mapsto \quad \text{var} | \text{ literal} \\
& \quad \mapsto \quad \text{\cdots atom \cdots} \quad \text{constant infinite POD} \\
\text{(atoms)} & \quad \text{atoms} \quad \mapsto \quad \{\text{atom}_1, \ldots, \text{atom}_n\} \quad \text{where } n \geq 0 \\
\text{(vars)} & \quad \text{vars} \quad \mapsto \quad \{\text{var}_1, \ldots, \text{var}_n\} \quad \text{where } n \geq 0 \\
\end{align*}
\]

Figure 5.2: Syntax for a data-parallel extended STG language
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

of first-order functional programs. The extra two rules required to perform vectorisation of higher-order programs are given at the end of the chapter.

5.1.1 Vector form

One of the distinguishing features of the STG language is the presence of un-boxed data-types and primitive operations on those types. The Haskell integer data-type is different from the 32-bit integers manipulated by the ALU of a microprocessor, as it is an abstraction of a machine integer which has been lifted to include \( \bot \). This lifted property is exemplified by its boxed structure promoted by Peyton Jones and Launchbury [60]. Given that the type \texttt{Int#} represents a real 32-bit integer, then an Haskell \texttt{Int} is represented by the product type: \texttt{data Int = MkInt Int#}, where the boxing due to the constructor gives the type its lifted property. Given this data-type, the literal 42 can be represented as \texttt{MkInt 42#}; where 42# is the textual representation of a primitive 32-bit integer literal. As a consequence of using the lifted form of an integer, the evaluation of integer addition is performed by case analysis on the \texttt{MkInt} constructor. The following integer addition function mimics the denotational semantics of addition:

\[
\begin{align*}
\texttt{addInt } x \ y &= \texttt{case } x \ \texttt{of} \\
&\quad \texttt{MkInt } x# \rightarrow \texttt{case } y \ \texttt{of} \\
&\quad \quad \texttt{MkInt } y# \rightarrow \texttt{case } (x# \oplus y#) \ \texttt{of} \\
&\quad \quad \quad \texttt{res#} \rightarrow \texttt{MkInt } \texttt{res#}
\end{align*}
\]
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

The starting point for the vectorisation process is the assumption that for every primitive operation in the STG language, there exists an analogous primitive vector operation. For example given an un-boxed addition operation of type $(#)::<\text{Int#} \rightarrow \text{Int#}$ we assume there is a vector addition primitive of type $(#)::<<\text{Int#}>> \rightarrow <<\text{Int#>>}$ where the convention prim is used to represent the vector version of the primitive prim. This example assumes the knowledge of primitive vectors of integers (i.e., $<<\text{Int#>>}$), and begs the question-how do you unbox a parallel vector of boxed integers? Given the knowledge that such primitives exist, the basic program identity is the conversion of a primitive application into a vector form that can be directly implemented on a data-parallel machine.

\[
\begin{align*}
\text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow \text{prim } a_1 \ldots a_k) v_1 \ldots v_n \\
\Rightarrow \text{prim } (\text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow a_1) v_1 \ldots v_n) \\
\vdots & \quad \vdots \\
(\text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow a_k) v_1 \ldots v_n) \\
\end{align*}
\]

Vector form (I)

The goal of vectorisation is to simplify all the MAP n expressions in a STG program until they reach vector form. As a syntactic convention, each of the $a_1 \ldots a_k$ in rule I represents an atom that can either be a literal or a variable. Also, as STG programs are assumed to be well-typed, each of the expressions $v_1 \ldots v_n$ represent a primitively typed vector expression.
5.1.2 Basic rules for expressions

Rule I for primitive applications is generalised in a manner similar to Steele's [66] \( \alpha \) distribution law. By pushing the MAP inside the arguments of the application, and pulling the function being MAPped outwards, more opportunities for simplification to vector form are exposed.

\[
\begin{align*}
\text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow f a_1 \ldots a_k) v_1 \ldots v_n \\
\Rightarrow \text{MAP}_n f (\text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow a_1) v_1 \ldots v_n) \\
\quad \vdots \quad \vdots \\
\quad \text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow a_k) v_1 \ldots v_n
\end{align*}
\]

Applications (II)

Rule II is rather wasteful when pushing the maps inside the arguments of the function \( f \), as each of the atoms \( a_1 : : : a_k \) is either a literal or a variable. If the atom is a variable then it will either be free in the lambda expression, or bound by one of the bindings \( x_1 \ldots x_n \). This observation leads to rules III & IV.

\[
\begin{align*}
\text{MAP}_n (\lambda x_1 \ldots x_n \rightarrow \text{lit}) v_1 \ldots v_n \\
\Rightarrow \ll \ldots \text{lit} \ldots \rr \\
\text{If } v_i \neq \bot; \text{ where } 0 \leq i \leq n
\end{align*}
\]

Constants (III)
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

Rule III is rather subtle as it would seem that the side condition would be extremely hard to satisfy. The reason for the side-condition is the expression

\[ \text{MAP}_1 (\lambda x \rightarrow 42#) \perp \] should reduce to \( \perp \), and not an infinite array of 42#'s. The rule is sound however, because we can guarantee that each of the arrays represented by the expressions \( v_1 \ldots v_n \) will never be \( \perp \), whenever such an expression is evaluated. In the implementation, arrays are wrapped inside a product type in a similar manner to the boxing of integers, and the compiler ensures that the "wrapping" is removed before its use by a MAP n expression. Effectively the compiler inserts an explicit check, which is safe because if the value of MAP n f arr is needed, then the value arr will be needed, regardless of f—consider e.g., length (MAP n (\( \lambda x \rightarrow 1 \)) arr).

\[
\text{MAP}_n (\lambda x_1 \ldots x_j \ldots x_n \rightarrow x_j) \; v_1 \ldots v_j \ldots v_n
\]

\[ \Rightarrow v_j \]

If \( v_i \not= \perp \); where \( 1 \leq i \leq n \)

Map simplification

Rule IV is a generalisation of the K-combinator simplification \( K \; y \; \perp = (\lambda a \; b \rightarrow a) \; y \; \perp \Rightarrow y \), in which the side condition follows in a similar way to rule III.
5.1.3 Mapping through the "fire break" of a local binding

The rules presented so far enable expressions such as the one shown in figure 5.1 to be reduced to vector form. However, vectorisation can easily be interrupted by the local bindings in a program, causing a "fire break" through which the algebraic identities cannot be applied. For example, the body of the letrec in the left of figure 5.3 can not be reduced by identities I-IV as the letrec gets in the way of vectorisation. This fire-break caused by the local bindings of a letrec is overcome by generating "new primitives" for each binding in the letrec. This idea was inspired by Hughes's [39, 57] observation that instead of converting a lambda calculus program into Curry's S, K, and I combinators [21] and then reducing the program containing those three combinators, the program can be specialised into a set of super-combinators which are tailored to the lambda calculus program in question. As with the vector form simplification where prim has the same semantics as MAP n prim, we specialise the function f to a function f which has the semantics MAP n f; where n is the natural arity of f.

Here f is the mapped version of the binding f, and wherever a MAPn f is encountered in a STG program, the MAP is replaced with f. Rule VI formalises this transformation process, and provides a bridge across the fire break.
5.1.4 Letrec: tidying-up vectorisation

A similar rule to V is required when a MAP_n expression is found in the body of the letrec, and no map ever enclosed the letrec as a whole. This scenario is a consequence of the user writing a pod comprehension which has subsequently been desugared into a MAP_n expression. Rule VII follows in a similar manner to rule V, by generating new versions of the bindings in the letrec.

\[
\text{letrec } f_1 = \lambda x_1,x_1 \ldots x_1,n \rightarrow \text{expr}_1 \\
\vdots \\
f_k = \lambda x_k,1 \ldots x_k,m \rightarrow \text{expr}_k \\
in \text{MAP}_i (\lambda z_i \ldots z_i \rightarrow \text{expr}) v_1 \ldots v_i
\]

\[
\Rightarrow \text{letrec } \overline{f}_1 = \lambda x_1,1 \ldots x_1,n \rightarrow \text{MAP}_n (\lambda z_1,1 \ldots z_1,n \rightarrow \text{expr}_1) z_1,1 \ldots z_1,n \\
\vdots \\
\overline{f}_k = \lambda x_k,1 \ldots x_k,m \rightarrow \text{MAP}_m (\lambda z_k,1 \ldots z_k,m \rightarrow \text{expr}_k) z_k,1 \ldots z_k,m \\
in \text{MAP}_i (\lambda y_i \ldots y_i \rightarrow \text{expr}) v_1 \ldots v_i
\]
5.1.5 Vectorising top level bindings

Rule VIII is like the previous two rules for letrec-expressions, providing similar vectorisation for top-level bindings in a STG program. Unlike local bindings of a letrec, the sequential form of a binding is retained as well a generating a parallel version of the binding.

\[
\begin{align*}
  f_1 &= \lambda x_{1,1} \cdots x_{1,k} \to \text{expr}_1 \\
  \vdots \\
  f_k &= \lambda x_{k,1} \cdots x_{k,m} \to \text{expr}_k \\
  \Rightarrow f_1 &= \lambda x_{1,1} \cdots x_{1,n} \to \text{expr}_1 \\
  f_1 &= \lambda x_{1,1} \cdots x_{1,n} \to \text{MAF}_n \left( \lambda x_{1,1} \cdots x_{1,n} \to \text{expr}_1 \right) x_{1,1} \cdots x_{1,n} \\
  \vdots \\
  f_k &= \lambda x_{k,1} \cdots x_{k,m} \to \text{expr}_k \\
  f_k &= \lambda x_{k,1} \cdots x_{k,m} \to \text{MAF}_m \left( \lambda x_{k,1} \cdots x_{k,m} \to \text{expr}_k \right) x_{k,1} \cdots x_{k,m}
\end{align*}
\]

5.1.6 Converting sequential objects to infinite pods

Rule IX is a generalisation of rule III for literals. If the x in the body of the lambda expression is free, then the effect of rule IX is to convert x into an infinite vector in which every element contains x.
In the implementation of the compiler for the DAP, this seemingly trivial rule is usually the sole culprit for programs failing to vectorise! The problem is that there is no general method of transforming an object of type $\alpha$ into an infinite vector of $\alpha$'s—the conversion is ad hoc for each type. One solution to this problem is to fail vectorisation whenever a troublesome conversion is encountered. Although this was the technique actually implemented (due to time constraints), better solutions are available. As the compiler is based upon the Glasgow Haskell compiler [42], a natural choice would be to use Haskell style overloading. Unfortunately this cannot be simply incorporated into the compiler as vectorisation occurs as a compiler pass after overloading has been resolved. The compiler has to therefore re-implement the overloading mechanism to implement rule VIII. Another technique would be to push the problem down into the runtime system. In this scenario, each object (i.e., closure) would know how to convert itself into an infinite parallel object. This could be implemented by adding an extra code pointer to a specialised conversion routine to each closure of the runtime system.
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

5.1.6.1 An aside: O(1) update of pods

The variable vectorisation rule IX is used in the vectorisation of the following DPHaskell updatePod function:

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

This function provides yet another solution to the problem of providing a O(1) update of an array-like data-structure in a purely functional language [63], i.e., given a vector vec, the value at processor i is replaced with the value x. This is achieved here by creating an infinite vector of xs which is then merged with the vector to be updated at processor i (see §5.2.1 on updating). Here we have used the rather large hammer of data-parallelism to achieve a O(1) update, at the cost of O(N) space complexity.

5.1.7 Conversion

What these program identities have shown is that by pushing any MAP n expressions inside the object being mapped gives rise to a program that contains primitive vector functions. There is another approach to this problem. If we consider \( \lambda \rightarrow \text{expr} \) to be a thunk [A thunk is a non-functional suspended computation. For example, the closure for the expression (1 + 2) is a thunk. In the context of the STG language, a thunk is a closure in which the lambda expression has no formal parameters.], then \( \text{MAP}_0 (\lambda \rightarrow \text{expr}) \) can be considered as a parallel thunk that represents an infinite
vector in which each element contains \( \text{expr} \). The current translation rules deal with such a thunk by vectorising \( \text{expr} \), as in the following simple example:

\[
\text{MAP}_1 \left( \lambda y \mapsto \text{letrec } z = \lambda \mapsto 2 + 3 \right) \text{ vec}
\]

\[
\Rightarrow \text{letrec } y = \text{vec}
\]

\[
\begin{array}{c}
\Xi = \lambda \mapsto \text{MAP}_0 \left( \lambda \mapsto 2 + 3 \right) \\
\text{in } \text{MAP}_1 \left( \lambda y \mapsto z \right) \text{ vec}
\end{array}
\]

\[
\Rightarrow \cdots 2 \cdots \triangleright \text{vec} \cdots \cdots \triangleright
\]

In contrast to this vectorisation process, it is possible to short-circuit the vectorisation process for parallel thunks. The idea is to convert a thunk into a parallel thunk by using a rule similar to III for constants, i.e.,

\[
\text{MAP}_0 \left( \lambda \mapsto \text{expr} \right) \varnothing : : \text{expr} : : \text{AE}.
\]

The example expression would now be transformed into:

\[
\text{MAP}_1 \left( \lambda y \mapsto \text{letrec } z = \lambda \mapsto 2 + 3 \right) \text{ vec}
\]

\[
\Rightarrow \cdots 2 + 3 \cdots \triangleright
\]

This transformation is basically similar to rule IX (and encompasses all the problems associated with that rule) with the following extension: if any of the formal parameters of a lambda expression applied by \( \text{MAP}_n \) are used in the RHS of a letrec, then the vectorisation rule for letrec loses the values of the sequential bindings during vectorisation. We have incorporated these ideas in the compiler for the DAP, and it
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

amounts to performing as much work sequentially as possible. The down-side of the translation is that the transformation isn't clean and therefore cannot be easily expressed in terms of the vectorisation rules described here. The motivation behind the translation is that given $\text{MAP}_0 (\lambda \rightarrow \text{expr})$, it is far more efficient to evaluate expr and then convert it into a parallel object, than to let the vectorisation process turn expr "inside-out" producing a form susceptible to vectorisation.

5.1.8 Mapping through a primitive case expression

We have been liberal with the truth in the examples given so far. Close inspection of figure 5.1 reveals the example to be syntactically incorrect as the syntax of figure 5.2 does not allow nested expressions. Although case expressions can be syntactically laborious (as was seen to be the case in the addInt example), they form the heart of vectorisation. They are important because:

- As with the STG language, all evaluation be it sequential or data parallel is forced by case expressions;
- With the onset of evaluation by case expressions, and transformations to algebraic data-types case expressions manipulate, we become specific about the representation of algebraic data-types on a data-parallel machine;
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

All the transformations have been source to source, and there has been no need for any syntactic support for data parallelism. Parallelism has been encapsulated by primitive operations that have a parallel semantics. This now changes as we introduce a parallel case statement as the result of vectorisation.

Primitive case expressions provide a similar role to C-style switch statements, with the evaluation of the discriminant of a case expression producing a primitive unboxed data type as a result. As a starting point we assume that for every primitive type there is an associated vector form of the type. For Int#, Char#, and Float# the associated parallel types are <<Int#>>, <<Char#>>, and <<Float#>> which are the primitive vector types of a SIMD or vector machine.

The important feature of the translation of case is that it introduces parallel case expressions. Abstractly, when mapping a lambda expression over a vector where the body of the lambda contains a case expression with two alternatives, then some of the vector elements will evaluate to the first alternative, and some to the second. The distinctive feature of a parallel case is that it has a linear complexity in relationship to the number of alternatives in the case - this mirrors the different control paths of the case alternatives.

\[
\text{eqInt} = \lambda \, x \, y \rightarrow \text{case} \, (x \# y) \text{ of} \\
0\# & \rightarrow 1\# \\
\text{default} & \rightarrow 0\#
\]

\[
\text{eqRx} = \lambda \, x \, y \rightarrow \text{case} \, (x \# y) \text{ of} \\
0\# & \rightarrow \ldots 1\# \ldots \\
\text{default} & \rightarrow \ldots 0\# \ldots 
\]

Figure 5.4: Vectorising a binding containing a primitive case
Figure 5.4 gives an example vectorisation of a binding in a STG program. Informally, the operational semantics of the parallel case are:

- The vector represented by the discriminant "x -# y" is evaluated in parallel;
- Every vector element of the discriminant that matches against the literal 0# evaluates the first alternative evaluate the default alternative;
- The vector that encapsulates the entire 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 case alternatives. This priority scheme mimics the priorities inherent in a case expression, and is a consequence of Haskell's pattern matching equations.

This latter point, the merging of vectors, is problematic, and as a consequence the solution adopted to address merging has a great impact on the representation of algebraic data types (see §5.2). In a similar vein to the rules presented so far, when vectorisation encounters a map which surrounds a case expression, the solution is to push the MAP inside the alternatives of the case.
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

Rule X introduces a parallel case statement with the semantics given informally above. The $\forall$ is used as a syntactic reminder that the result of a parallel case is a vector in which all the elements which match $\text{lit}_i$, evaluate the alternative $\text{expr}_i$. If a variable default binding is used in a primitive case expression, then the translation of the default in rule X is changed to:

\[
\begin{align*}
\text{MAP}_i \left( \lambda y_1 \ldots y_i \rightarrow \begin{array}{c}
\vdots \\
\vdots \\
\vdots
\end{array} \\
\text{in expr} \\
\end{array} \right) f_k = \lambda z_{k,1} \ldots z_{k,m} \rightarrow \text{expr}_k \quad v_1 \ldots v_i
\end{align*}
\]

\Rightarrow \text{letrec } y_1 = v_1; \ldots ; y_i = v_i

\begin{align*}
\overline{f}_1 = \lambda z_{i,1} \ldots z_{i,n} \rightarrow \text{MAP}_n \left( \lambda z_{i,1} \ldots z_{i,n} \rightarrow \text{expr}_1 \right) z_{i,1} \ldots z_{i,n} \\
\vdots \quad \vdots
\overline{f}_k = \lambda z_{k,1} \ldots z_{k,m} \rightarrow \text{MAP}_m \left( \lambda z_{k,1} \ldots z_{k,m} \rightarrow \text{expr}_k \right) z_{k,1} \ldots z_{k,m} \\
\text{in MAP}_i \left( \lambda y_1 \ldots y_i \rightarrow \text{expr} \right) v_1 \ldots v_i
\end{align*}

5.2 Vectorising algebraic data types

Before case analysis of algebraic data-types can be considered, their representation on a data-parallel machine needs to be resolved. Vectorisation 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), and the scrutiny of such forms by case analysis is guided by following machine constraints:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Constraint</td>
<td>Some data-parallel machines (e.g., 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. Other SIMD machines such as the Maspar MP-1 allow limited forms of parallel pointers, yet these forms aren't general enough to implement vectors of ADT's.</td>
</tr>
<tr>
<td>Parallelism Constraint</td>
<td>When a parallel case expression such as the one shown in figure 5.4 is evaluated, potentially all the alternatives of the case need to be evaluated. The result of each of these alternatives may be a head normal form that contains as yet unevaluated closures. These head normal forms need to be “merged” into a single vector that encapsulates the meaning of the entire case.</td>
</tr>
</tbody>
</table>

Fortunately since primitive vectors are in normal form, there is no problem with merging as it can be implemented as a masked parallel assignment-e.g., tgt(mask) = src of DAP Fortran [19]. However, it should be noted that the result of a primitive algebraic case statement is independent of the "flavour" of case being performed-this is because of the fact that the evaluation of a primitive case doesn't guarantee that a primitive expression will be returned as a result!

Our goal in the following sections is to provide a feasible yet efficient runtime representation of algebraic data types. As a running example the vector shown in figure
5.5 is transformed into a series of representations with varying degrees of suitability for a data-parallel machine.

5.2.1 Case closures: 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 $\langle (\alpha_1; \text{adt}_1); \ldots; (\alpha_n; \text{adt}_n) \rangle$, consists of a finite list of tuples containing a primitive 1-bit boolean vector (this is different from a vector of Trues and Falses), and an ordinary sequential algebraic data-type. Each element of the 1-bit boolean vector $\alpha_i$ that contains True represents a processor that logically contains the associated algebraic data-type $\text{adt}_i$. Figure 5.6 shows the case closure representation of the

![Diagram](image-url)
exemplar vector.

One problem with this representation of an ADT is that it has the distinct feeling we are cheating! The representation fulfils the hardware constraint because vectors of 1-bit truth values can be implemented on a data-parallel machine—this is exactly the kind of datastructure these machines are designed to manipulate efficiently. The representation also fulfils the parallelism constraint; if the processors identified by $\alpha$ evaluate the first alternative of a case expression to the head normal form $a$, and all the other processors identified by $\alpha$ evaluate the second alternative to $b$, then the two alternatives can be merged by creating the case closure $< (\alpha; a); (\beta; b) >$ As case closures are returned by the evaluation of case expressions, and case expressions maybe nested, then either a case closure or an ADT may be delivered from the evaluation of a case. We skate over this
distinction here, the important point is that a tree-like structure of case closures with algebraic data-types at its leaves could easily be created from the evaluation of a case.

Now for the downside. Matching a case closure of size $n$ against a case expression with $m$ alternatives potentially requires $m \times n$ combinations to test which of the alternatives needs evaluating. More seriously, different alternatives of the case expression may be evaluated $n$ times; where $n$ can be much larger than the number of alternatives in the case expression.

As an example, the case closure of figure 5.6 contains the lists "Cons (MkInt 42#) Nil)" and "Cons (MkInt 8#) Nil". A naive implementation of case would precede by testing each element of the case closure against all the alternatives of the case expression. With the two Cons' above, if an alternative of a case contains a Cons constructor, then both of the Cons's will have to evaluate the same case alternative independently-in a multiprocessor environment this could be achieved by forking-off the evaluation of the two, yet we don't have such luxuries in a SIMD environment. This phenomenon is particularly worrying for the complexity measurements of algorithms, as map will not have the constant complexity we were hoping for. As this requirement is the foundation for our work, a better representation of an ADT is needed!

5.2.2 The "inside-out" transformation

The solution to the adverse effects of case closures is to apply a transformation similar to the one used in the vectorisation of MAPn. Vectorisation can be interpreted as pulling a function used inside a mapped lambda expression outwards. The way we turn ADT's
inside-out is to notice that case closures provide a mechanism for expressing vectors of algebraic data-types. One way of turning this representation inside out is to merge the representation of the case closure and sequential ADT by reaching inside the case closure and pulling a parallel version of the ADT outwards. As the sequential ADT is being pulled outwards, it needs to "catch" some of the parallel information from the primitive 1 bit vector of the case closure to ensure the final representation is sound. Considering the vector of lists in the running example, this can be achieved by reaching inside the case closure and pulling all the Cons's outwards to form a big parallel Cons.

Another way of thinking about the problem is that in a vector containing a sum-of-product type, some of the vector elements will contain one of the products from the sum-of-products, and other processors will contain a different product. By flattening the sum-of-products into a single product type, we can form a single representation of all the different constructors that can exist simultaneously. This whole transformation, shown in the example of figure 5.7 can be described in two stages:

- A sum-of-product type is flattened into a single product type that encapsulates all the alternatives of the sum-type;
- As the single product type encapsulates all the possibilities of constructors within a single sum-of-product, we can safely convert a vector containing thousands of these super-product types into a single product type containing vectors as its elements.
5.2.3 Flattening a sum of product type

A list can be defined by the following sum of product type:

```
> data List a = Nil
>
| Cons a (List a)
```

This can be flattened into a product type by providing disjoint "slots" in the resulting product for every slot in the original sum type. A particular instance of a constructor in the original sum-type is identified by tagging the flattened product type with an enumerated type. We assume enumerated types can be implemented by a technique over and above the method used for sum-of-products. An implementation could use N unique integers for a sum-type with N-products, or a \( \log N \)-bit word. In our example this would be:
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

As another example of this transformation, we convert a labelled binary tree into a product type. This transformation highlights the use of the multiple slots in the transformed type:

> data LabelTree a b = Leaf a
    | Branch (LabelTree a b) b (LabelTree a b)

A tree of type LabelTree αβ contains leaves which store data of type α, and a binary Branch which is labelled with the type β. The flattened form of the tree is:

As can be seen from the type, every slot of the original sum-of-product type occurs in the final flattened product type. If the slots in different products of a single sum-type have the same type, then the slots could be reused. This is because the standard interface to the type guarantees that the slots will never be in use at the same time. The transformed type is more general than the original, and hence an instance of the type which mimics the original is padded with 1's-using appropriate error messages, as shown below, was found to be an invaluable debugging aid for the abstract machine:
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

> mkLeaf :: a -> LabelTree a b
> mkLeaf x = LabelTree Leaf# x (error "Leaf(Branch): Not a left-branch")
> = (error "Leaf(Branch): Not a label")
> = (error "Leaf(Branch): Not a right-branch")

> mkBranch :: LabelTree a b -> b -> LabelTree a b -> LabelTree a b
> = LabelTree Branch# (error "Branch(Leaf): Not a leaf") left label right

5.2.4 Vectorising a flattened sum-of-product type

The idea now is to convert a flattened data structure of type \(<<\text{List } \alpha >>\) into a form that is semantically equivalent, and can be implemented on a data-parallel machine. The motivation behind this transformation is to abstract all the pointers from a slot in the implementation of a vector of ADT's into a single big pointer to an object that is semantically equivalent this fulfils the hardware constraint. This is analogous to transforming all the functions in the expression MAPn \(f\) into a single function \(f\) that encapsulates the infinite number of applications of \(f\) to each element of a potentially infinite vector. For example, given a type \(<<\text{List } \alpha >>\), transforming the flattened sum-of-product type for a list produces:

\[
data <<\text{List } \alpha >> = <<\text{List } \text{ListTags } \alpha (\text{List } \alpha) >>
\Rightarrow \text{iota data VecList } \alpha = \text{List } <<\text{ListTags } \cup \text{"Not Here"}>> \llangle \alpha \rrangle (\text{VecList } \alpha)
\]

If this transformation is applied to all the types in a program, then a vector of algebraic Data-types will be represented by either:

- A primitive vector type such as \(<<\text{Int}# >>\) that has a natural implementation on a data-parallel machine;

139
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

- An ordinary sequential implementation of a product type;
- A vector containing \( N \) different tags which can be represented as a \( \log N \)-bit vector. Such a representation will probably be more efficient than a primitive vector type such as \(<\text{Int}^\#>>\). Notice how we add an extra "Not Here" tag as a representation of data being omitted from a processor (see figure 5.7, where "Not Here" was used for processor two which doesn't contain an integer). The tag for "Not Here" comes from the case closure \(<(\alpha; a)>\), such that wherever \(\alpha\) is false then the tag will be "Not Here", and wherever it is true it will be one of the tags of the ADT.

The technique just presented for representing vectors of ADT's in an inside-out form isn't new, but a technique often used when programming vector or data-parallel machines. Jouret [47] used the same technique in his implementation of a non-strict data-parallel language, where he noticed that a vector of trees can be turned inside out and represented as a tree of vectors. What we have done is to provide a formal description of the conversion process, that forms a proof of correctness for the technique.

5.2.5 The program identity for constructors

Rule XI for constructors follows from the definition of the constructor functions such as mkLeaf, mkEmpty, and mkCons of §5.2.3. Given a constructor constr \(a_1 \ldots a_n\) in the STG language, the rule creates the analogous constructor formed after flattening the sum-of-product, and converting it to an inside out representation that involves padding any unused slots with
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

\[
\text{MAP}_w(\lambda x_1 \ldots x_n \rightarrow \text{constr}_i \, a_{i,1} \ldots a_{i,n}) \, v_1 \ldots v_n
\]

\[\Rightarrow \ldots \text{constr}_i \ldots \Rightarrow \bot_1 \ldots \bot_n\]

\[
\begin{align*}
(\text{MAP}_w (\lambda x_1 \ldots x_n \rightarrow a_{i,1}) \, v_1 \ldots v_n) \\
& \vdots \\
(\text{MAP}_w (\lambda x_1 \ldots x_n \rightarrow a_{i,n}) \, v_1 \ldots v_n)
\end{align*}
\]

Constructors (XI)

where \( \text{data } T \, \alpha_1 \ldots \alpha_n = \text{constr}_1 \, T_{1,1} \ldots T_{1,n} \)

\[
\vdots \\
\vdots \\
\vdots \\
\text{constr}_n \, T_{n,1} \ldots T_{n,n}
\]

\( t = \text{total number of slots in the sum-of-product type.} \)

\( s = \text{number of slots before constr}_i. \)

This rule encapsulates the inside-out transformation, for a single constructor of an inside-out type. The novel feature of the inside-out transformation is the representation of multiple constructors by a single flattened product type. However, this rule only creates one construct, all the others are padded with \( \bot \)-the benefits of the inside-out type are therefore lost.

The creation of multiply merged inside-out objects is a consequence of merging the different alternatives of a case expression. This mechanism is controlled by the abstract machine,

141
and is therefore outside the scope of the translation rules here. An introduction is given here, with more details given in the next chapter.

5.2.6 An introduction to merging

The running example of a vector of lists of figure 5.7 was rather selective as the data structure was already in a normal form. The problem with non-strict languages is that data-structures are rarely in a normal form. For example, those returned from case alternatives may contain head normal forms which by definition contain thunks. The problem to be addressed is how to merge these head normal forms. For example, given the STG function:

```haskell
> funny x ys = case x of
>    D8 -> Cons (Nklat 16) (sort ys)
>    default -> Cons (Nklat 08) (reverse ys)
```
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

The sort and reverse on the RHS of the alternative will be represented in the abstract machine by an unevaluated thunk. With a little tinkering, the Cons and MkInt can be merged into their inside-out representation, but the tail of the Cons represents two different unevaluated closures. Pictorially this is shown in figure 5.8, where the tail of the list contains two thunks (represented by light-bulbs in the figure). For a case with N alternatives, a single pointer in the inside-out transformation could point to N different thunks which need merging together. The sort and reverse expressions are thunks and may not need evaluating, so their merging has to be suspended as well. This is shown in figure 5.8 by the special MERGE node. Like the conversion of serial objects into infinite vectors, the merging of two vectors can be adhoc. One solution is to re-implement overloading, another is to provide each object with the knowledge of how to merge itself with others (see discussion of §5.1.6). The solution adopted by Jouret [47] was to relegate merging to an auxiliary function of the abstract machine. We believe merging to be far too important, as it can make or break the implementation. This is because without merging, the inside-out structure vital to the implementation will not be formed. We therefore reuse "case closures" from the naive implementation, but make the manipulation of these objects a central part of the abstract machine. We believe that merging is an important feature of the implementation and should be brought out into the open. This is similar to Peyton Jones' observation that naive implementations of non-strict languages needlessly update a redex at every reduction step. The solution Peyton Jones adopts is to make updating an integral part of the abstract machine, whereby a
program is analysed and needless updates are eliminated [60]. As we shall see, making
merging part of the abstract machine provides the opportunity to optimise programs and
therefore eliminate needless suspended merges.

5.2.7 Case analysis

Case analysis of algebraic types is changed to model the inside-out representation used in
the parallel implementation. As constructors all have the form constr tags a₁ ... aₖ, where
constr is automatically generated from a sum-of-product type, there is no need to clutter
the rule with the product-type constructor constr. The variables w₁,..., wₖ in rule XII
represent the values of all the slots in the flattened algebraic data-type. Each of the constrᵢ
where 1 ≤ i ≤ k are the possible values denoted by the vector of tags of the flattened type.
Informally evaluation of a case precedes by:

- Evaluating the discriminant of the case which always produces a flattened
  product type;
- Bind w₁,..., wₖ to each of the slots in the type;
- Evaluate the first case alternative for those processors that have constr₁ present
  in the vector of tags;
- This is repeated for each constructor-case analysis is therefore performed by tag
  checking in parallel;
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

- Evaluate the default for those processors which are not defined for each of the constructors.
- Merge the result of all the alternatives together by creating a case closure of each of the boolean bit vectors that determines which processors matched, and therefore evaluated each alternative, along with the head normal form representing the RHS.

\[
\text{case expr of}
\begin{align*}
\text{constr}_1 w_{1,1} \ldots w_{1,j} & \rightarrow \text{expr}_1 ; \\
& \vdots \\
\text{constr}_k w_{k,1} \ldots w_{k,j} & \rightarrow \text{expr}_k ; \\
\text{default} & \rightarrow \text{expr}_d 
\end{align*}
\]

\[
\Rightarrow \text{CASE } (\text{MAF}_n (\lambda x_1 \ldots x_n \rightarrow \text{expr}) v_1 \ldots v_n ) \text{ CF}
\]

\[
\text{where } s = \text{total number of slots in the flattened product type.}
\]

mapListAndPod = \lambda \text{fn pod } zs \rightarrow \text{let }\pod' = \text{MAF}_1 \text{ fn pod } \text{ in MkTuple } \pod' zs'

Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

If a variable binding is used in the default position of the alternative, the last line of rule XII is replaced by:

\[ z \rightarrow \text{expr}_d \Rightarrow \overline{\text{MAF}}_{k+t}(\lambda z \ldots z_n \rightarrow \text{expr}_d) \overline{v}_1 \ldots \overline{v}_k \]

One point that is potentially worrying about this rule is that all the slots \( w_1 \ldots w_s \) are live during the evaluation of each of the alternatives, although only a subset will be live at a given time. After evaluation of a case alternative for a particular constructor is complete, the associated slots for that constructor in the flattened product type can be stubbed to prevent unnecessary space leaks.

5.3 Higher-order vectorisation

The vectorisation rules given so far are incomplete because they are unable to cope with higher-order arguments that are either passed as arguments to functions, or stored in algebraic data-types to be applied at a later date. As an example, when the right-hand side of mapListAndPod is vectorised, two versions of the function \( \text{fn} \) are required; one for the ordinary sequential representation of the function, and another for the vectorised form of the function.
We obtain a solution to this problem by passing the tuple \((m; fn)\) to a function whenever
the function \(fn\) is used as an higher-order argument.

\[
\begin{align*}
\text{higher-order application (XIII)}
\end{align*}
\]

\[
f \{x_1, \ldots, x_i, \ldots, x_n\}
\]
\[
\text{iff } x_i \text{ has a functional type}
\]
\[
\Rightarrow \text{ let } hof = \text{MkTuple}\{x_i; z_i\}
\]
\[
in \ f \{x_1, \ldots, x_{i-1}, hof, x_{i+1}, \ldots, x_n\}
\]

Similarly in the Lambda-form of a function, when an argument has a functional type, then
a tuple will be used in the vectorised program, so the tuple will need to be taken apart.

Two similar rules, not given here, are also required when a function is used in a

\[
\begin{align*}
\text{higher-order function (XIV)}
\end{align*}
\]

\[
\lambda \{x_1 \ldots x_i \ldots x_n\} \rightarrow \text{expr}
\]
\[
\text{iff } x_i \text{ has a functional type}
\]
\[
\Rightarrow \lambda \{x_1 \ldots x_i \ldots x_n\} \rightarrow \text{case } x_i \text{ of }
\]
\[
\text{MkTuple} \{x_i; z_i\} \rightarrow \text{expr}
\]

constructor application, and the scrutiny of such applications by case analysis.

**5.4 Programs that fail vectorisation**

The purpose of this chapter has been the conversion of programs into a form that can be
executed on a data-parallel machine. The outstanding problem with the vectorisation rules, is that if a program contains nested parallelism, then the vectorisation process creates an ill-defined program, e.g.,

\[
\text{let } \text{inc} = \lambda \text{z} \rightarrow \text{case } \text{z of }
\begin{align*}
\text{MkInt } \text{z#} &\rightarrow (+) \ 1\# \ 1\# \\
\text{in }
\text{MAP}_1 (\lambda p \rightarrow \text{MAP}_1 (\lambda y \rightarrow \text{inc } y) \ p \) \ \text{vec}
\end{align*}
\]

As the body of the \text{letrec} can be vectorised by rule 5.1.2 into the expression \text{inc vec}, one solution would be to generate a binding \text{inc} that has the semantics \text{MAP}_1 (\text{MAP}_1 \text{inc}), but when will such nesting stop? Code cannot be generated for every conceivable level of nesting, therefore we leave nested parallelism as an open question. It seems that its presence in a language has a fundamental impact on the abstract machine, for example see Blellochs NESL [13]. Another solution would be to reimplement any nested parallel expressions sequentially. We do not discuss nested parallelism any further in this thesis.
Chapter 5. ALGORITHMS FOR SIMD ARCHITECTURE

5.5 Summary

The vectorisation rules given in this chapter transform programs into a style that can be implemented on a data-parallel machine. The next chapter defines a data-parallel abstract machine, that evaluates programs that have been vectorised by these rules. A summary of the vectorisation rules is shown in table 5.2.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Section</th>
<th>Name/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>5.1.1</td>
<td>Transform a primitive application into a <em>vector form</em> that can be implemented directly on a data-parallel machine.</td>
</tr>
<tr>
<td>II</td>
<td>5.1.2</td>
<td>Transform an application by turning it inside out.</td>
</tr>
<tr>
<td>III</td>
<td>5.1.2</td>
<td>Convert a constant into an infinite vector.</td>
</tr>
<tr>
<td>IV</td>
<td>5.1.2</td>
<td>Simplify a mapped lambda expression that contains a variable bound by the formal parameters of the lambda.</td>
</tr>
<tr>
<td>V</td>
<td>5.1.3</td>
<td>Vectorise through the &quot;fire break&quot; of a letrec expression by generating new bindings.</td>
</tr>
<tr>
<td>VI</td>
<td>5.1.3</td>
<td>Vectorise a mapped function bound by a vectorised letrec-binding.</td>
</tr>
<tr>
<td>IX</td>
<td>5.1.6</td>
<td>Convert a mapped lambda expression that contains a variable not bound by the formal parameters of the lambda into an infinite vector.</td>
</tr>
<tr>
<td>X</td>
<td>5.1.8</td>
<td>Vectorise a primitive case expression by generating a primitive parallel case expression.</td>
</tr>
<tr>
<td>XI</td>
<td>5.2.5</td>
<td>Generate a parallel inside-out data-structure from an algebraic constructor.</td>
</tr>
<tr>
<td>XII</td>
<td>5.2.7</td>
<td>Vectorise algebraic case expressions into a parallel case expression that manipulates the inside-out data-structures.</td>
</tr>
<tr>
<td>XIII</td>
<td>5.3</td>
<td>Vectorise higher-order functions.</td>
</tr>
</tbody>
</table>

Table 5.2: Summary of the vectorisation rules