Mapping Adl to the Bird-Meertens Formalism

Brad Alexander

September 13, 1994

Abstract

Bulk data operations such as *map and* reduce are an elegant medium for expressing repetitive computation over aggregate data structures. They also serve as a tool for abstraction: not all details of the computation, such as the exact ordering of the constituent operations, need to be specified by the programmer. A precise description of the behaviour of the bulk data operator is the preserve of the language implementor. If the implementation of these operators is parallel then they become a medium for expressing implicit data parallelism. There is a large body of work formally the relating bulk data operators to each other and to their underlying data types. Much of this research stems from Category Theory where a number of general properties of types and operators have been established. One theoretical framework in particular, the Bird-Meertens Formalism (BMF), has proved to be extremely useful. The BMF theory of a type provides a set of operators on that type and a set of algebraic identities on these operators. With these tools a programmer can build a program and then transform it to its equivalent forms by applying the identities. The focus of this report is the translation of a small, experimental data parallel language, Adl, into BMF. The translator is intended to be the first in a series of stages of a compiler targeted to a parallel distributed architecture. Most of the translator is described using inference rules in Natural Semantics. The report finishes with some worked examples of translation.

1 Introduction

Adl is one of a small number of languages and notations that specifies computation primarily through a handful of powerful combinators. APL and FP rely almost exclusively on such combinators. SISAL and Fortran 90, though more general also depend on related constructs to express the more computationally intensive parts of programs.

A reliance on combinators rather than recursion as a method to express bulk computation has its advantages. It has been noted that an absence of recursion (at least at the top level of a notation) aids the process of program transformation [3]. In this report I focus on the translation of code written in Adl into a non recursive form called the Bird-Meertens formalism.

1.1 Background of BMF

In his paper on FP[1] Backus demonstrated that programs couched in terms of the composition of a few combining forms were clearer and more amenable to algebraic transformation than those using more conventional notation. He noted a number of identities and described a system to use these identities for the purposes of program transformation. One of the advantages of the FP notation is the natural way in which highly parallel computation can be expressed.

The idea of using combining forms to express programs was further advanced by the application of category theory to computer science. It was shown that by expressing combining forms and type constructors as functions and functors respectively a large number of identities would be generated (an example of this work can be found in [23]). It was also proven that for certain useful types (Hagino types [11]) the generated identities would be sufficient to transform a program to all its equivalent forms [1](at least at the level of the aggregate operations).

There is a large body of literature dealing with the prospects for optimising programs using these identities. The Bird-Meerten's Formalism was defined in a series of lectures [4] detailing how algebraic identities can be used to greatly increase the efficiency of sequential programs. In this series the methods were applied to a variety of type constructors.

1.2 Applications of BMF to parallel programming

The results from Bird's lecture series are only partially applicable to parallel programming. Some of the transforms turn inefficient solutions based on reductions to efficient solutions based on scans. In the sequential case this transformation is more profitable than it is in the parallel case due to the fact that parallel scan has a larger number of operations than sequential scan.

A body of work has evolved to examine how parallel operations might be added to BMF programs. When operations that imply certain parallel behaviour on the part of the underlying machine are created new identities arise. These can form part of the optimisation process [21, 20].

Other work deals with forming new BMF theories based on different aggregate data types such as bags, lists and sets (part of the Boom hierarchy see [13]). Much new work involves the introduction of rectangular arrays of explicit dimension [12, 18]. Others use hybrid notations [2]. There is still much work proceeding in this area.

There is also a substantial field of research devoted to using a few standard combinators, often closely related to map and reduce as a means of mapping programs to parallel architectures. These standard combinators are often referred to as "skeletons". Some work is devoted to making use of a set of explicit combinators [9]. The alternative of detecting the implicit use of these combinators is also being explored with some success [6, 7].

Issues arising from using BMF theories as a means to express parallelism will not be covered in great depth in this report. The expression of parallelism through the use of BMF concatenate lists will be the subject of later stages of Adl's implementation. The bulk of this document shall concern the process of deriving a basic translator from Adl source to its equivalent BMF representation (for most constructs a relatively straightforward task) and a further refinement of this representation to a more manipulable subset.

The following section will address the level of correspondence between BMF notation and Adl

[1]In most cases there would, of course, be an infinite variety of equivalent forms for a program but most of these would be uninteresting.

source. Section 3 will define the target language. Section 4 describes the translation process. Section 5 demonstrates how the translator works on a small number of real examples.

2 Relevance to Adl

The closeness of the mapping between the Bird-Meertens formalism and the Adl language can be measured at two levels. At the coarse or macro scale we examine issues that pertain to both languages as a whole. At the micro level we examine details of constructs and problems that might be encountered when trying to establish mappings from individual constructs in Adl to the corresponding ones in BMF. At this point it is worth noting that BMF is not a language in the strictest sense but a set of theories. For the purposes of this report the BMF theory of concatenate lists as established by Bird in [4] will serve as the target langauge and any references to BMF (unless otherwise stated) will be to this.

2.1 Mapping at the macro level

At the macro level Adl and the BMF theory of concatenate lists correspond very closely. Adl is both functional (vitally important for equational reasoning) and non-recursive. This combination is quite rare in a language and matches very well with the basic properties of BMF.

One advantage in using BMF as an intermediate form to any functional language is that it's very flexible. It is possible to utilize a range of types and type constructors within the framework of BMF. A type to suit the primary parallel data type in the language can usually be found.

Perhaps the shared property that relates Adl and BMF most closely is Adl's lack of recursion. The absence of recursive functions in Adl allows function bodies to be in-lined into the BMF target code. All of the in-lined code is accessible to later stages of the compiler.

2.2 Mapping the details

The two representations are not as tightly bound at the level of details. Some constructs in Adl, notably `map`, `scan`, `reduce`, and `if` have direct translations to combinators in BMF notation. Others such as `let`, `while` and `iota` [2] have no direct counterpart.

The primitive vector indexing operation `!` is easily translated into it's target form. However once it is in this form it is not very easy to manipulate equationally.

None of these apparent conflicts are completely insurmountable. For instance `let`'s can be eliminated through the elimination of scope (`let` defines the value of variables and variables and target code of this translation doesn't have variables). Many instances of `!` and `iota` will be eliminated by the optimisation stage of the compiler (which is to be described in a later report) [3].

[2] Although `iota` has no traditional BMF counterpart in the literature it is not particularly difficult to define it using the tools available within BMF theories. `while` is much more foreign to the notation.

[3] The process of getting rid of explicit indexing is one of the major themes of this project

The match between Adl vectors and the most closely related data type in BMF ,concatenate lists[4], is also imperfect. In final form the constructs are similar, both are one dimensional with scope for nesting, but each comes with a different set of operators. In Adl there is one operator, `iota` for raising scalars types to vectors. In BMF the make-singleton function $[\cdot]$ is used to create a vector from a single scalar. The concatenate operator $+\!\!+$ is often used at this point to build lists from these singletons. However the choice of operator is by no means limited to concatenate. The potential number of operators is as large as the number of ways in which two lists (of abitrary length) may be permuted and interleaved. Members of this family of operators are legitimate components of BMF. In Adl the use of all forms of vector concatenation is discouraged. In these respects the range of operations offered over BMF concatenate lists is a superset of those offered over vectors in Adl.

The `while` operator is difficult to map into standard constructs in BMF. The problem is primarily due to the potential for `while` to loop forever. Some functions using the construct will not be total and totality is needed for many algebraic manipulations to be valid. Even when a function is total it is still possible to have a bound on the iteration that is very difficult to estimate (c.f. Ackerman's function and similar chaining algorithms).

In many cases it will be possible to establish a bound on the iteration of the `while`. When a bound can be established it will be possible (but not always practical) to use a more standard BMF constructs, corresponding to `map`, `reduce`, or `scan`.

Even when the limit of iteration is difficult to establish, the effect of a `while` construct can be isolated by regarding the code segment holding the `while`, as off-limits to the optimiser. This allows the optimiser to concentrate on the surrounding constructs. This mirrors the common practice, in the literature on BMF, of denoting arbitrary functions by a letter (usually f). As long as f is total its internal structure and its precise behaviour is not of any consequence. This convenient state of affairs arises because most BMF identities are defined over higher order functions operating over aggregate data structures. Names like f are used to indicate that these functions operate at a level below the higher order constructs and their exact behaviour can be divorced from higher level which is subject to manipulation. It is possible to envisage a scheme where an optimiser, can attempt to work at a level above that of a program segment containing a the target code for a troublesome construct such as `while`.

Similarly, it may in some cases be practical for the optimiser to work at the level below the difficult constructs. This may happen in a program such as an unbounded laplace iteration. The outer part of the program will be a `while` but the inner part may still be amenable to optimisation.

In summary, if we discount the presence of the `while` construct, the correspondence between Adl and BMF notation is surprisingly good. The task of translation and optimisation is substantially easier than it would be for a more conventional functional language. The source of this similarity lies in Adl's built-in higher order functions and lack of recursion. It has also been noted that it should be possible, in many cases to, hide the impact of `while`'s from the program optimiser.

[4]There are theories that provide a slightly better model for Adl vectors than concatenate lists, shapely types [15] and multidimensional arrays [18] provide two out of a number of alternatives. However, these types have the drawback of being slightly more complicated than concatenate lists and also (at the moment) of having far less written about them.

3 BMF Notation

This section enumerates some of the major BMF constructs and describes their use. Most of what follows is to familiarise the reader with the broad features of BMF. Consequently, much of what is covered is not of direct relevance to the translator, but to the project as a whole.

It should be kept in mind that BMF is an extremely flexible descriptive form. There are very few operations that could be described as "primitive". Certainly the best candidate for the title of primitive construct is the function compose operator, written $\cdot$.

BMF scripts are usually written as the composition of several operations. They usually take forms like..

$$< opn > \cdot < opn - 1 > \cdot \cdots \cdot < op2 > \cdot < op1 >$$

These scripts can be imagined as pipelines where data flows though the various transfoms from right to left. Composition retains all the properties generally associated with it in mathematics. One of the most useful properties it bears is associativity. So for all unary functions f,g and h we have:

$$f \cdot (g \cdot h) = (f \cdot g) \cdot h$$

It is easy to see the advantages this property has when the time comes to apply algebraic identities to these scripts.

Another almost universally used mechanism in BMF is function sectioning. This reduces the complexity of a function's argument by applying some of its components. For a lazy function this is very similar to currying (except that it is not always the first argument that is applied). For a strict language with tuples for arguments it reduces the arity of the tuple (or its components). Sectioning is expressed using brackets so $(/2)$ is a function that divides its argument by two and $(2/)$ is a function that divides two by its argument.

In common with most useful languages there is a form used in BMF to express conditional evaluation. It has the general structure:

$$(p \rightarrow c, a)$$

Which is very simliar to the form used in Backus' FP. As with all languages execution of conditional forms tend to weaken our knowledge of the data that flows from them. This makes reasoning about programs more difficult. The extent of the impact a conditional expression has on the optimisation process is dependent on its position within the program relative to where most of the optimisation is occuring.

One function that makes a very frequent appearance in BMF scripts is the identity function: id (which has the obvious meaning). The id function is not to be confused with the $id_{\oplus}$ value which denotes the identity element for a binary operator $\oplus$ [5]. Identity elements appear frequently in definitions of reductions, which shall be dealt with later. There are often seperate left and right identity operators for an operator (a property that has little to do with the associativity or otherwise of the operator).

[5]Identity elements don't exist for all operators e.g. the function fst on pairs.

3.1 Operations specific to types

Type constructors are central to BMF. Each theory has its own aggreate types (e.g. theories of trees, sets, bags, concatenate lists etc.) and the construction operations that go with them. The type constructor we are most interested in is the concatenate list constructor. Concatenate lists differ from ordinary lists in that they are built from the list concatenate ($+\!\!+$) and the make-singleton ($[\cdot]$) operations. This is in contrast to the single constructor, cons, which is (:) used to make ordinary lists. Lists built with cons can only be built in one order and the ordering is fundumentally sequential.

The concatenate operation doesn't impose any ordering on the construction of a list from its raw components. There are many orders in which the $+\!\!+$ operation can be applied to produce the same list. This a result of the associativity of $+\!\!+$.

There are three equations to define how concatenate lists are constructed; these are (taken from [22]):

$$\begin{aligned} [\cdot]a &= [a] \\ [\,] &= K_{[\,]} \\ [as]+\!\!+[bs] &= [as, bs] \end{aligned}$$

The first equation is relatively self explanatory it is the application of the make-singleton operator. The second line shows how the empty list and the function which takes anything as an argument and returns the empty list (the $K_{[\,]}$ function) are equivalent. The $K_{[\,]}$ function will arise again in the definition of the filter or $\triangleleft$ function. The third line shows how lists can be formed from smaller ones. The respective sizes of $[as]$ and $[bs]$ are left unspecified, a given list can be built up from its portions in a large number of ways.

The introduction of lists gives rise to a variety of operations that will work over lists independently of the types of the values they contain. Most of these operators can be constructed from two general ones. These are the map $(*)$ and the reduce $(/)$ operators[6].

Map peforms a single operation to each of the elements of a list. It changes the contents of the elements of a list without altering the form of the list. The use of a map in BMF is abbreviated to $< function > *$ so a program to square all the elements in a list of numbers (of any length) would be simply:

$$sqr\, * \quad \textbf{where} \quad sqr\ x = x * x$$

Maps can be nested simply by juxtaposing the $*$ symbols. The code to square all the elements of a two dimensional list is $sqr * *$.

Reduce (sometimes called "fold" or "insert") works by inserting a binary operator between the elements of a list. It alters the form of a list. The use of reduce is indicated by the form: $\oplus/_e$ where $\oplus$ is an abitrary associative binary operator. The e subscript is optional it indicates the value that is returned by the reduce on the occassion of it encountering an empty list. There are directed reductions $\not\leftarrow$ (right reduction) and $\not\rightarrow$ (left reduction) which will operate deterministically over non-associative operators. Right and left reduction will not be explored in much detail in this report since they are inherently sequential.

[6] Map and reduce (in some form) are primitive to all BMF theories.

3.1.1 Some identities

Both map and reduce over concatenate lists yeild several useful identities independent of the functions they are used with. This independence arises from properties that apply to all concatenate lists regardless of content.

One simple identity for maps is:

$$(f*) \cdot (g*) = (f \cdot g)*$$

Which shows that the composition of two maps is the same as a single map using a composition of the original functions. This identity may be used as a step in code optimisation to coalesce two loops with the same ranges i.e. it is essentially the same as loop fusion.

Some identities apply to both map and reduce when they are used in combination some trivial identities for empty lists are:

$$\begin{array}{rcl} f * \cdot K_{[]} & = & K_{[]} \\ \oplus/ \cdot K_{[]} & = & id_{\oplus} \end{array}$$

These are relatively self-explanatory. There are also corresponding identities for lists with one element:

$$\begin{array}{rcl} f * \cdot [\cdot] & = & [\cdot] \cdot f \\ \oplus/ \cdot [\cdot] & = & id \end{array}$$

The first equation shows that you don't really need to use a map on a list of one element. The second demonstrates how a combination of make singleton and reduce form the identity function.

More generally useful equalities apply to lists of arbitrary length. One simple identity is map promotion:

$$f * \cdot +\!\!+/ = +\!\!+/ \cdot (f*)*$$

This demonstrates how a single dimensional map can be promoted to two dimensions or vice-versa. This identity is useful for postponing the concatenation of lists within a list until after f has been mapped over all the data. The utility of this transformation depends on how the underlying machine interprets concatenation. If concatenation stands for the physical coalecsence of data from several processing nodes to one then there is much to be gained by delaying the operation.

Reduce-promotion is also relatively straightforward:

$$\oplus/.(\oplus/)* = \oplus/+\!\!+/$$

Reduce promotion may be used to decrease (or, used in reverse, increase) the number of dimensions a reduction occurs over. Again, the usefulness of this transformation depends on the target machine.

There is also a special relationship between the concatenate and make-singleton operator for lists of greater than unit-length:

$$+\!\!+/ \cdot [\cdot]* = id_*$$

which is effectively raising a special case (the case for $+\!\!+$) version of the $\oplus/ \cdot [\cdot] = id$ identity to work on lists.

These last three identities involve the use of the $+\!\!+$ operator. This function appears quite often in BMF identities associated with concatenate lists. The heavy use of $+\!\!+$ in BMF notation seems somewhat incongruous when the Adl language doesn't have a primitive to mirror its function [7]. The dichotomy is resolved by three factors. Firstly there are identities that generate list concatenates from other operations, as illustrated by the reduce promotion above.

The second, more important factor is the production of concatenates out of operations that Adl has but are less standard in BMF. As a small example, the zip operator, which turns a pair of lists into a list of pairs, is usually expressed in BMF using concatenate operations. In Adl it is expressed with a combination of `iota` and vector indexing. It's concievable that replacing combinations of `iota`s and indexing operations with more elegant forms will introduce the concatenate operator as part of the optimisation process.

Thirdly (and most importantly) a reduce with concatenate $/\!\!+\!\!\!+$ can be used as a part of the process of partitioning a function across a machine. A parallel program may be represented as the BMF code:

$$/\!\!+\!\!\!+ \cdot \ldots \cdot \mathit{partition}$$

Which uses the function *partition* (sometimes referred to as *split* or *spread*) to spread data across processors[8] from the first one. The rest of the program consists of a series of parallel operations followed by a $/\!\!+\!\!\!+$ (sometimes called gather) which recoalesces the data onto a single processor (whether this takes place in any literal sense depends on the compiler).

3.1.2 Scan

One important variety of operations on lists are accumulations or scans ($/\!\!/$). $\oplus/\!\!/$ takes a list and returns a list of the same length with the selected $\oplus$ operator applied to each each element of the initial sublist preceding and including the current element. Scan is not an entirely primitive operator in BMF. It can be defined as:

$$\oplus/ * \cdot \mathit{inits}^{+}$$

Where the inits^{+} function creates a list of initial sublists of the input list (excluding the empty sublist). So $\mathit{inits}^{+}\ [1, 2, 3, 4]$ would yield:

$$[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]$$

and $\oplus/ \cdot \mathit{inits}^{+}\ [1, 2, 3, 4]$ would yield:

$$[1, (1 \oplus 2), (1 \oplus 2 \oplus 3), (1 \oplus 2 \oplus 3 \oplus 4)]$$

A literal interpretation of this definition of scan would lead to a hopelessly inefficient implementation. Richard Bird demonstrates how this inefficient ($O(n^2)$) version can be converted to a sequential version that works in linear time.

A parallel version of scan also exists when the binary operator ($\oplus$) is associative. It takes the form:

$$\begin{array}{l} \otimes/ \cdot [\cdot]* \\ \quad \textbf{where} \\ a \otimes b = a +\!\!\!+ ((\mathit{last}\ a)\oplus) * b \end{array}$$

[7] The concatenate operator was left out of Adl in an attempt to discourage the dynamic allocation of data (or at least to make that allocation very explicit)

[8] A clever compiler might pre-spread the data, thus saving a lot of communication.

Where *last* is the function that returns the value of the last element of the list. Since $((last\ a)\oplus)$ is mapped over b the last value of a will need to be broadcast to all of the elements of b. Broadcast communication is not generally efficient.

There is a version of parallel scan that doesn't use broadcast communications (taken from [8].

$$
\begin{aligned}
&\ominus/ \cdot (id, id)^{\circ} \cdot [\cdot] \\
&\quad where \\
&a \ominus b = (loc1*, loc2*)^{\circ} \cdot (\Upsilon paira, \Upsilon pairb) \\
&\quad where \\
&loc1\ ((x1, sum1), (x2, sum2)) = (x1, sum1 \oplus sum2) \\
&loc2\ ((x1, sum1), (x2, sum2)) = (x1 \oplus sum2, sum1 \oplus sum2)
\end{aligned}
$$

Unfortunately this version requires certain orders of reduction and operates only on lists whose lengths are powers of two. There is a more complicated rendering that accepts lists of any length but it still assumes the list is reduced in certain orders.

The uses of scan are many and varied. Scan's offer parallel solutions to many problems where intuition would dictate otherwise (a good reference on how parallel scans may be applied to a variety of problems is [5]).

The utility of scan from the point of view of BMF is that a number of inefficient solutions expressed in terms of reductions over nested lists (e.g. $inits^{+}\ [1, 2, 3, 4]$) can be expressed more efficiently using scans. Much of the seminal work on BMF [4] is devoted to this sort of conversion. The work deals primarily with sequential reductions and sequential scans but the methods can be applied to parallel scans if associative operators are used.

3.1.3 Miscellanous operators

There are some other operators which are less commonly used in BMF but are nonetheless required in the initial stages of the translation of Adl programs. Perhaps most important of these operators is *iota*. The *iota* operator is a direct translation of the primitive Adl function of the same name. The need for *iota* arises from two, slightly overlapping, sources. The first source is the requirement for an operator that maps from a domain of scalars (in this case the integers) to the domain of lists. There is no accepted best way to do this for a particular data type but *iota* is a good candidate because it produces a list whose contents are unique, ordered and successive.

The second source of demand for *iota* is more closely associated with the Adl language than any fundumental need. Most uses of `iota` in Adl are for generating indices. As an example a program for matrix add in will have to use `iota`'s which produce vectors corresponding to the length of a particular dimension of one of the input arguments. These `iota`'s are present to allow the programmer to operate within the constraints of the Adl language. As I shall illustrate later, these `iota`'s do not have to appear in the BMF version.

A more widely used construct is the $^{\circ}$ or "all-applied-to" operator. This operator takes a list of functions and applies each of them to the input value to produce a list of values. For example $[sqr, sqrt]^{\circ}\ 4$ will produce the list $[2, 16]$.

A similar and, for our purposes, more useful construct is all applied to for product types. For example $(sqr, (>3))^{\circ}\ 4$ evaluates to $(3, true)$. The all applied to construct for product types is very useful for duplicating arguments or applying them to different operators to yield results of possibly different types. The all applied to operator for products will appear quite frequently in initial translations of Adl programs as means to explicitly pass and group data.

The $()^{\circ}$ operator is useful for tuple creation. There is a mirrored need to access elements of a product type or tuple. In this report I shall be, for notational convenience, using mainly pairs. The functions π_1 and π_2 shall be used to access the first and last elements of a pair respectively. Sometimes it will be necessary to compose several uses of π_1. Instead of writing this in the verbose way i.e. $\pi_1 \cdot \pi_1 \cdot \cdots \cdot \pi_1$ I shall abbreviate this to π_1^n where n is the length of the chain of π_1's.

Finally there is a need for an index operator as a means to access elements of lists. The index operator is denoted by the symbol !. It is always used in the context: $\ldots \cdot ! \cdot exp$ (the part to the left of ! is optional) where exp is a BMF expression yielding a pair. In the rare cases where the index operator takes its inputs directly from the outside world its application has the form $\ldots \cdot !$.

Indexing is not commonly found in standard expressions of BMF. Expressions incorporating indexing operations are hard to manipulate. Formalisms such as BMF and Ruby [16] were constructed as a means to dispense with array indexing where possible. In line with this approach it is intended that array indexing be present mostly in the initial stages of translation from Adl to BMF (it is not generally possible, nor always desirable, to eliminate all array indexing operations).

It is in the nature of using BMF that any list of possible operators is not going to be exhaustive. Those listed so far are the closest to being primitive operators i.e. not able to be defined in terms of others. The functions defined from here on should (for the most part) be easily defined in terms of the operators above.

3.2 Homomorphisms

Thus far I have shown a small set of identities that apply to concatenate lists regardless of their contents. Obviously there is scope for more identities than these. The problem is that these identities will not rely just on properties of lists but on the properties of functions used in BMF programs.

It would be tedious to derive the identities associated with functions on a case by case basis. This would mean a user would have to specify the properties of a new function to the optimiser or we would have to limit function-specific manipulations to a small set of primitive functions. This leads to the same ad-hoc approach encountered in the building of most optimisers.

Fortunately it is possible to establish general properties of functions. If a function is found to abide by a general property then a pre-established set of identities can be applied to it.

One very useful set of functions for which general properties and hence common identities can be established are homomorphisms. A homomorphism is a transformation between one ordered grouping of objects and another that in some way preserves the integrity of the grouping.

The grouping of objects can be described as a monoid. A monoid is sometimes expressed as a

4-tuple [9]. of the form:

$$(\alpha, \oplus, id_\oplus, i)$$

Where α is a type, $\oplus$ is a binary operator, $id_\oplus$ is the identity element of $\oplus$ and i is an operation to apply to each element of the group of objects.

An example of a monoid is the monoid vfor concatenate lists. To describes a group of objects as a concatenate list you apply the monoid:

$$([\alpha], +\!\!+, [\,], [\cdot])$$

Which states (for our purposes) the elements can be of any type (α). They are transformed to singletons via $[\cdot]$ and they are joined together using $+\!\!+$ with the empty list for an identity element to be used (if required) in the process.

A homomorphism can be looked upon as a transformation between a monoid of the form $(\alpha, \oplus, id_\oplus, i)$ to a monoid $(\beta, \otimes, id_\otimes, j)$. A function h, is a homomorphism if it abides by the following identities:

$$\begin{array}{lcl} h\ id_\oplus & = & id_\otimes \\ h\ (i\ a) & = & j\ a \\ h(x \oplus y) & = & h\ x \otimes h\ y \end{array}$$

Where a is an abitrary object and x and y are abitrary groupings of objects.

A simple example of a homomorphism is the list length function (#). Length does the following conversion between monoids:

$$([\alpha], +\!\!+, [\,], [\cdot]) \xrightarrow{\#} (int, +, 0, K_1)$$

It is easily demostrated that this conversion abides by the required properties for homomorphisms:

$$\begin{array}{lcl} \#\,[\,] & = & 0 \\ \#\,[a] & = & K_1\ a \\ \#(x +\!\!+ y) & = & \#x + \#y \end{array}$$

One very useful property of monoids is that for each target monoid for conversion from a list there is a unique homomorphism. Moreover for each unique homomorphism from $[\alpha]$ (shorthand for the list monoid above) to $(\beta, \otimes, id_\otimes, j)$ there is a canonical form for describing it. The canonical form of this homomorphism is: $\otimes/ \cdot j*$. The simplicity of this expression and the ease with which it can be manipulated makes it attractive to treat functions as homomorphisms wherever possible.

Another identity associated with all homomorphisms from $([\alpha], +\!\!+, [\,], [\cdot])$ to $(\beta, \otimes, id_\otimes, j)$ is:

$$h \quad = \quad \otimes/ \cdot (h.[\cdot])*$$

Which basically states that applying a homomorphism is identical to applying the homomorphism to each element of the list (wrapped up by a make-singleton function) and reducing the result with the binary operator from the target monoid.

Using the forms outlined above the homomorphism: # could be written as $+/ \cdot K_1*$ or perhaps, as $+/ \cdot (\#.[\cdot])*$.

[9]Some articles describe monoids using 3-tuples $(\alpha, \oplus, id_\oplus)$ but these either have less general properties or make implicit references to a fourth element

3.2.1 Other useful homomorphisms

The following is an outline of some of the more useful functions that can be written as homomorphisms. The list is by no means comprehensive there are a very large number of functions expressible as homomorphisms from lists to some other monoid.

First we examine the sort function. Sort can be expressed as the conversion between the two monoids:

$$([\alpha], +\!\!+, [\,], [\cdot]) \xrightarrow{sort} ([\alpha], \wedge\!\!\!\wedge, [\,], [\cdot])$$

Where the $\wedge\!\!\!\wedge$ symbol stands for the binary merge operator that takes a pair of ordered lists and merges them into a single ordered list (in linear time). The operation of this form of sort is almost exactly the same as the traditional merge sort with best case efficiency of $O(n \log n)$. This solution can be expressed as $\wedge\!\!\!\wedge/ \cdot [\cdot]*$, describing what is normally thought of as a binary recursive solution in terms of a very straightforward map and reduction.

Another useful function is reverse. The monoid conversion for reverse is:

$$([\alpha], +\!\!+, [\,], [\cdot]) \xrightarrow{reverse} ([\alpha], \underset{\sim}{+\!\!+}, [\,], [\cdot])$$

Where the only component of the monoid to change is the $+\!\!+$ symbol. This has been replaced in the second monoid by $\underset{\sim}{+\!\!+}$ which stands for concatenate in reverse. The reverse function can be written in BMF as $\underset{\sim}{+\!\!+} / \cdot [\cdot]*$.

Any partially applied map function $(f*)$ is also a homomorphism. The monoid conversion for $f*$ is:

$$([\alpha], +\!\!+, [\,], [\cdot]) \xrightarrow{f*} ([\beta], +\!\!+, [\,], ([\cdot] \cdot f))$$

Note that only the first and last components change. These are the only components of a concatenate list monoid that are affected by a homomorphism consisting solely of a map.

The map homomorphism can be expressed in BMF as $+\!\!+/ \cdot ([\cdot] \cdot f)*$ which can be simplified in the following manner:

$$
\begin{array}{ll}
 & +\!\!+/ \cdot ([\cdot] \cdot f)* \\
= & \{(f \cdot g)* = g * \cdot f*\} \\
 & +\!\!+/ \cdot [\cdot] * \cdot f* \\
= & \{+\!\!+/ \cdot [\cdot]* = id_*\} \\
 & id_*.f* \\
= & \{eliminating\ identity\} \\
 & f*
\end{array}
$$

Reduction is a homomorphism. Any reduction of the form $\oplus/$ can be framed as a transformation between the monoids:

$$([\alpha], +\!\!+, [\,], [\cdot]) \xrightarrow{\oplus/} (\alpha, \oplus, id_{\oplus}, id)$$

Reduce can be expressed in BMF as $\oplus/ \cdot id*$ which is trivially simplified to $\oplus/$.

A more complicated homomorphism is a sectioned cross product $(x\mathsf{X}_{\oplus})$. The use of sectioning ensures that the operator is unary (its first argument is already supplied) and it can be cast as a homomorphism. $(x\mathsf{X}_{\oplus})$ can be written as the transform between the monoids:

$$([\alpha], +\!\!+, [\,], [\cdot]) \stackrel{(x\mathsf{X}_{\oplus})}{\longrightarrow} ([\beta], +\!\!+, [\,], j)$$
$$\textbf{where}$$
$$j\ a = (\oplus a) *\ x$$

The sectioned cross product basically maps the $\oplus a$ operator over the sectioned argument x for every a and then concatenates all of the components of the result. So for example the application:

$$([x, y]\mathsf{X}_{\oplus})[a, b, c]$$

would yield:

$$[x \oplus a, y \oplus a, x \oplus b, y \oplus b, x \oplus c, y \oplus c]$$

This small list of homomorphisms is only a sample of the possible range of functions that fall under the category of homomorphisms. The biggest advantage of using homomorphisms on lists stems from our ability to break the function into its map and reduce components. This allows the identities associated with map and reduce to be applied and increases the chance of useful points of program optimisation being found.

An interesting aside is the fact that the general usefulness of homomorphisms expressed in the form $\oplus/ \cdot f*$ has been at least partly recognised before its use in BMF. The main parallel construct in the SISAL language [10], the **forall** loop, takes this form. The fact that it mirrors a general homomorphism might go some way toward explaining its usefulness. Unfortunately the SISAL **forall** construct does not allow the user to define an associative binary function for the reduce step. Instead the programmer must choose from a given set of built-in operators.

At present, homomorphisms that don't use one of the built-in operators in their reduction stage must be expressed as binary recursive functions (where it is harder to recognise opportunities for optimisation). However, it may be that the built-in operators are sufficient to cover the most frequently encountered parallel problems (the **forall** loops are highly parallel) in SISAL .

3.3 Completeness

The grounding of the type constructor for concatenate lists in category theory provides us with a set of identities. A core of these identities, derived from the properties of the list monoid, $([\alpha], +\!\!+, [], [\cdot])$, are sufficient to derive all the other possible identities on concatenate lists. I have not outlined all of the core identities in the preceeding section. This is because a few e.g. $id_* = (id)*$ are intuitively obvious. The ones that are listed form the bulk of the core equations.

Once the core equations for a type constructor are known all BMF scripts based around that type constructor can be transformed using those equations to all their equivalent forms. If a program can not be transformed as desired it is not due to an inadequacy in our knowledge of the data type, it will be a problem in our specification of the problem at hand.

With the knowledge that a program can be transformed to all its equivalent forms (at least at the level of operations over agreggate data structures) the process of optimisation is greatly enhanced. The application of the identities will always yield correct programs. If the optimiser makes a move that gives a less efficient result [10] that move may be easily reversed. The optimising part of a compiler could be based on a search strategy. The completeness result ensures that the entirety of the search space is reachable and the optimality of a solution depends on the search algorithm.

3.4 Some examples of BMF in use

I will use the following subsection to describe some BMF solutions to some simple problems. Some of the programs will be coded in Adl in the next section so these BMF scripts may be viewed as target code (after some optmisation has taken place).

3.4.1 Matrix addition

Matrix addition is a fairly trivial operation using BMF. However in order to describe the operation the auxillary function: zip Υ_{pair}. Zip takes a pair of lists and makes them into a list of pairs. The lists must be of the same length. The *pair* subscript signifies that the corresponding elements of the two lists will be combined into a pair (assuming that they are not fed into the zip function as a tuple anyway). Υ_{pair} will act in the following way:

$$\Upsilon_{pair}\ ([a,b,c],[d,e,f]) = [(a,d),(b,e),(c,f)]$$

The BMF code for matrix addition with this in mind is:

$$+**.\Upsilon_{pair}*.\Upsilon_{pair}$$

The two instances of zip are used to form pairs out of different levels of the matrix. After the data has passed the $\Upsilon_{pair}*$ term it would be a two-dimensional array containing pairs. The nested map of addition adds the components of each of these to form the result.

The matrix is represented by a list of concatenate lists. There is nothing implicit in the concatenate-list constructor to insure the matrices are the same shape. Wrong shaped arrays will cause a run time-error inside the zip function (unless of course there is enough information to work out the shape of the arrays at compile time). There are type constructors available (shapely types) that register a type-error if a matrix is used in an improper shape context [11].

3.4.2 Matrix Multiply

A matrix multiplication algorithm is composed of a block of dot products. The dot product operator ($\odot$) is a binary function that multiplies the corresponding elements of a pair of vectors to form a scalar. $\odot$ can also be expressed as $\Upsilon_{\times}$ (zip with times).

Two additional combinators are also needed to bring together the vectors for the dot product. These are distribute from right (*distr*) and distribute from left (*distl*). Distribute from right operates in the following manner:

$$distr\ ([a_1,\ldots,a_n],b) = [(a_1,b),\ldots,(a_n,b)]$$

[10] Skillicorn has noted that estimating the efficiency of programs written in BMF is a relatively easy task.

[11] If the shape of the input matrix is not known at compile-time then the type checking will occur at load-time. This scheme is similar to the one found in compilers implementing Hindley-Milner type system.

The following identity also holds:

$$distr = \mathsf{X}_{pair} \cdot (id, [\cdot] \cdot \pi_2)^{\circ}$$

The operation of distribute from left is similar:

$$distl\ (a, [b_1, \ldots, b_n]) = [(a, b_1), \ldots, (a, b_n)]$$

as is the identity:

$$distl = \mathsf{X}_{pair} \cdot ([\cdot] \cdot \pi_1, id)^{\circ}$$

Using these definitions, the BMF form for matrix multiply is:

$$\odot ** \cdot distl * \cdot distr \cdot (\pi_1, trans \cdot \pi_2)^{\circ}$$

Where *trans* is the transpose function on lists of lists.

3.5 Standard Deviation

The last example to be considered here is the derivation of the standard deviation from a sample of values. Statistical applications often exhibit embarassing amounts of parallelism. This is true even of some of the simple operators. Standard deviation is a measure of the variance in a sample. The formula for Standard deviation or σ is:

$$\sigma(x_1, \ldots, x_n) = \sqrt{\frac{1}{N-1} \sum_{i=1}^{n} (x_i - \overline{x})^2}$$

Parallelism arises at two points in this equation. Firstly in the calculation of the mean: $\overline{x}$ and, secondly, in the summation part.

The mapping of the various items in this expression to their BMF equivalents is very simple. The data set can be modelled by a concatenate list:

$$[x_1, \ldots, x_n]$$

The derivation of $\overline{x} = \sum_{i=1}^{n} x_i/n$ will translate to something like[12]:

$$div \cdot (+/, \#)^{\circ}$$

Where the summation is represented by $+/$. There is no zero element included since the standard deviation for a list of length zero is undefined. The value: n is derived from the length of the input list.

The summation expression will consist of a map to calculate a list of $(x - \overline{x})^2$ and a $+/$ to perform the summation. Each component of the sum will need access to both x_i and $\overline{x}$. To ensure that this is the case the input list to the summation section will have to look like:

$$[(x_1, \overline{x}), \ldots, (x_n, \overline{x})]$$

[12]As might be expected with a general purpose notation, there is no canonical way of expressing a function. Any translation of an Adl program is only an instance of a number of equally valid translations.

To create this list we need to use the code:

$$distr \cdot (id, div \cdot (+/, \#)^{\circ})^{\circ}$$

Which distributes the mean over the value returned by id. Since id always returns its input and the input to this program is the list of data elements $[x_1, \ldots, x_n]$ the result list is the value of $\overline{x}$ distributed over the $[x_1, \ldots, x_n]$ (which is what we want).

The next step is to define the code for the summation expression:

$$\sum_{i=1}^{n}(x_i - \overline{x})^2$$

This divides into two components, firstly a map expression to evaluate the individual $(x_i - \overline{x})^2$ terms and, secondly, a reduction for the summation part. The BMF code for the combination of these is:

$$+/ \cdot ((\hat{\ }2) \cdot - \cdot (\pi_1, \pi_2)^{\circ})*$$

Where $\hat{\ }2$ is a sectioned operator that raises its operand to the power of two. Lastly we have to translate the expression for $1/N - 1$ and the square-root operator that surrounds the whole expression:

$$sqrt \cdot \times \cdot (div \cdot (1, - \cdot (\#, 1)^{\circ})^{\circ}, exp)^{\circ}$$

Where exp stands for the summation part composed with the code for producing and distributing the mean. The code for the entire program is:

$$sqrt \cdot \times \cdot (div \cdot (1, - \cdot (\#, 1)^{\circ})^{\circ}, +/ \cdot ((\hat{\ }2) \cdot - \cdot (\pi_1, \pi_2)^{\circ}) * \cdot distr \cdot (id, div \cdot (+/, \#)^{\circ})^{\circ})^{\circ}$$

4 Translation to target language

This section provides an annotated, semi-formal description of the mapping of Adl constructs to their equivalent BMF code. Though this process is quite simple at the large scale it is quite involved at the level of details.

The translator's rules will be defined using Natural Semantics [17]. This notation was chosen for its brevity and its flexibility. A formalism, very similar to Natural Semantics, is used as the basis of the Centaur language development system [14]. If the reader is unfamiliar with notations with the same general flavour as Natural Semantics then it's advisable to, perhaps, read only the sections on the source and target syntax. Then skip to section 5. After a general understanding has been gained from the examples there, the meaning of the rules in this section will become clearer.

4.1 Definition of the translation

Translation is a mapping between two syntactic domains. In this case objects in Adl syntax are transformed into corresponding objects in BMF syntax. For some constructs the translation yields additional information that will be needed at a later stage. This information is incorporated into an environment. The environment also forms part of the target syntax even though is only used in the intermediate stages of translation i.e. no trace of the environment will appear in the target code.

I have chosen to describe the translation rules using natural semantics. This choice is based on the concise way notions such as environment and the evaluation of sub-expressions can be expressed. There will be at least one rule for each major construct. The description to follow will be coarse grained. Some details will be omitted. Most notably, details on the translation of primitive pointwise operators such as `+` into their BMF equivalent, $+$ are left out. However, where the translation of an item is unspecified it means that the operator will be mapped to its exact equivalent in the target language.

4.2 Conventions

When using inference rules to define a process or semantics we must adopt an unambiguous naming convention. The approach taken by the Centaur system is a good one. Phyla names (written in capital letters) are used to designate a class of objects i.e. an object that may take on more than one form (a union type). Objects with structure are called operators. A phyla will contain one or more operator names and an operator will incorporate zero or more phyla names into a structure (e.g. tuple, list, signature etc.).

The phyla names are used to declare the type of each judgement (or rule type) in the semantics. Every rule conclusion that fits into the template of the judgment is interpreted within the context of the type declared for the judgement and the operators instantiated in the rule itself. As an example: imagine the type of the rules forming the top level semantics of an Adl interpreter. The main judgement for the interpreter is:

$$ENV \vdash EXP \Rightarrow EXP$$

From this we can gather that any rule in this set that has a conclusion with the format:

$$A \vdash B \Rightarrow C$$

must have the variables A of type ENV, B of type EXP and C of type EXP. If we have abstract syntax of the operator `binop` declared:

```
binop -> OP EXP EXP
```

Then we'd know when we looked at the rule conclusion:

$$A \vdash \texttt{binop}(B, C, D) \Rightarrow E$$

that A is of type ENV, B is of type OP and C,D, and E are of type EXP. This set of conventions allows the rules to be defined in a very precise and unambiguous manner.

The rules used in this section follow the conventions used in Centaur with four differences:

1. There is no seperate section for operators in the syntax. Instead the structure of the operator is defined in-situ (i.e. in the clause of the phylum definition). This means that an operator such as "binapp" which would normally be defined in the phylum as:

$$E ::= \ldots \mid \text{binapp} \mid \ldots$$

with the corresponding operator definition:

$$\text{binapp} -> \ E\ BINOP\ E$$

is now defined simply as:

$$E ::= \ldots \mid \text{binapp}(E\ BINOP\ E) \mid \ldots$$

2. Items of Adl concrete syntax are now allowed in the definition so instead of:

$$E ::= \ldots \mid \text{let} \mid \ldots$$

where "let" is defined as the operator:

$$\text{let} -> \ DECLS\ E$$

we now adopt the notation:

$$E ::= \ldots \mid \texttt{let}\ DECLS\ \texttt{in}\ E\ \texttt{endlet} \mid \ldots$$

This convention makes the rules more readable and saves us from the trouble of having to define lots of new operator names[13].

3. The notation for lists of objects is slightly modified. The normal convention is to distinguish between an object and a sequence of these objects by using different names. For example the definition of bound variable patterns in Adl function declarations might be:

$$\begin{array}{ll} PAT & ::= PATLIST \mid X \\ PATLIST & ::= \text{pattern}\ PAT^* \end{array}$$

Where the term PAT^* stands for zero or more occurences of PAT. This definition is not quite as good as it could be. Pattern lists are isomorphic to tuples, they should have at least two members. To indicate this using the forms above we would need to replace the *PATLIST* term with the more verbose:

$$\text{patlist}(PAT : \text{patlist}(PAT : PATLIST))$$

The approach taken in this report is to declare the arity of an object using superscripts, so the declaration of patterns will be:

$$PAT ::= PAT^{2+} \mid X$$

The term, PAT^{2+} stands for a list of patterns of length two or greater. Its type is: PAT^* (a sequence of patterns). The use of the superscript,$*$, in an expression A^* is overloaded. It is used to indicate that A is raised to a sequence type. It is also used to indicate there are zero or more occurences of A in this sequence. This quantity measure is not really a part of the type and the notations should really be kept more distinct. Fortunately, the overloading doesn't obscure the meaning of the semantics in practice.

[13] In some instances operators are kept completely implicit (no tags or pieces of concrete syntax). Examples of this inclued the function and variable environment operators. In all such cases the operator used is very obvious from the context.

4. There is a rigid system of variable naming. A variable in a rule corresponding to a declaration sequence must have the name *decls*. The case and the font are important (to distinguish the variable, decls, from the phylum, ($DECLS$), and the operator, decls[14]. If there is more than one occurrence of a variable of the same type in a rule then subscripts and accents are used to distinguish them.

These alterations to the conventions used in Centaur are cosmetic. It should be a trivial matter to convert the syntax description and the rules based on them to the format required of Metal files by the Centaur system.

4.3 Adl syntax

The outline of the syntax of the source language for the rules consists of two major parts the first is the definition of the phyla or syntactic classes. The second part consists of the abstract syntax proper.

The name of every phylum is written in upper-case italics (e.g. DEC). The phyla names used in the source language and the objects they correspond to are shown below.

Phyla Definitions

PR ↦ *Programs*, *DECLS* ↦ *Declaration sequences*, *DEC* ↦ *Declarations*,
E ↦ *Expressions*, *PRAPP* ↦ *Primitive applications*
BINOP ↦ *Binary operators*, *UNOPP* ↦ *Unary operators*,
CONST ↦ *Constants*, *PAT* ↦ *Patterns*, *INTEGER* ↦ *Integers*,
REAL ↦ *Reals*, *X* ↦ *Identifiers*.

The second component of the definition is the abstract syntax specification. The phyla definitions are listed first (those lines using the (::=) symbol). Items of concrete syntax are differentiated by the use of `typewriter` font.

Abstract syntax

PR	::= pr*(DECLS DEC)*
DECLS	::= DEC^*
DEC	::= vdec*(X* `:=` *E)* | fdec*(X PAT* `:=` *E)*
E	::= app*(X E)* | prapp*(PRAPP)* | *X* | const*(CONST)* | `if` *E* `then` *E* `else` *E* `endif` | `let` *DECLS* `in` *E* `endlet` | binapp*(E BINOP E)* | unapp*(UNOP E)* | `[` E^* `]` | `(` $E^{+}2$ `)`
PRAPP	::= `reduce` x e^2 | `reducep` x e | `reducel` x e^2 | `reducer` x e^2 |

[14] This is a departure from Centaur's convention of using uppercase names to denote variables

`reducelp` x e | `reducerp` x e | `map` x e | `scan` x e^2 |
`scanp` x e | `scanl` x e^2 | `scanr` x e^2 | `scanlp` x e |
`scanrp` x e | `while` x^2 e

BINOP ::= `>` | `<` | `=` | `~=` | `<=` | `>=` | `&` | `|` | `mod` | `^` | `-` | `+` |
`*` | `div` | `/` | `!`

UNOPP ::= `iota` | `#` | `~` | `acos` | `asin` | `atan` | `round` |
`trunc` | `float` | `cos` | `sin` | `tan`

CONST ::= `true` | `false` | *INTEGER* | *REAL*

PAT ::= PAT^{2+} | X

Where the superscripts $*$, $+$ stand for zero-or-more or one-or-more occurrences respectively. The superscript $2+$ stands for two or more occurrences. Numeric superscripts stand for exactly that number of occurrences. There is an extra restriction that two identifiers with the same name may not appear in a single pattern (PAT) object.

4.4 BMF syntax

BMF syntax is less hierarchical than Adl syntax. The syntactic categories are divided into two parts. The first is for the target language itself. The second is for the environment used during the translation process.

The conventions used for BMF syntax are the same as those for Adl. The main difference is that items of concrete syntax are mostly in san-serif rather than `typewriter` font. Another difference is in the naming scheme for variables in the rules, the name of a variable that ranges over, ENV, the environment phylum is ρ rather than env. All other variable names follow the same conventions as before, i.e. they are lower-case versions of the names of the phylums they range over.

The objects forming the BMF syntax can be divided into two parts. The first part is the syntax of the target language. The second part is the scaffolding needed to complete the translation process i.e. various types of environments and the sub-components within.

In the following I outline the target language syntax at the top of the phyla and abstract syntax definitions. The auxiliary components are noted separately.

Phyla definitions

Target language

$EXP \mapsto$ *Expressions*, $OP \mapsto$ *Operators*,
$K \mapsto$ *Constants*, $INT \mapsto$ *BMF integer constant*, $RL \mapsto$ *BMF real constant*, $N \mapsto$ *Integer value*.

Auxiliary definitions

ENV $\mapsto$ *Environment*, *VARBIND* $\mapsto$ *Variable environment*,
VOD $\mapsto$ *Identifier or Dummy*, *V* $\mapsto$ *Identifier*, *FUN* $\mapsto$ *Function environment*,
FB $\mapsto$ *Function binding*, *CLOS* $\mapsto$ *Closure*.

Note the presence of two classes of integers. The first, under the phylum, INT represents the class of BMF constant functions. These must be applied to something before they yield a value. They are (at least in theory) typed. The second phylum, N denotes ordinary integer values. These values are suitable for performing arithmetic on (whilst only the values yielded by objects in INT can be subject to arithmetic manipulation)[15].

Abstract syntax

Target language

$$
\begin{array}{lll}
EXP & ::= & EXP \cdot EXP \mid K \mid (EXP^{2+})^{\circ} \mid [EXP^{*}]^{\circ} \mid OP \\
 & & \mid (EXP)/_{EXP} \mid (EXP)/ \mid (EXP)\nrightarrow \mid (EXP)\nleftarrow \mid (EXP)\nrightarrow_{EXP} \\
 & & \mid (EXP)\nleftarrow_{EXP} \mid (EXP)/\!\!/_{EXP} \mid (EXP)/\!\!/ \mid (EXP)\mathrel{/\!\!/\!\!\!\!\rightarrow} \mid (EXP)\mathrel{/\!\!/\!\!\!\!\leftarrow} \\
 & & \mid (EXP)\mathrel{/\!\!/\!\!\!\!\rightarrow}_{EXP} \mid (EXP)\mathrel{/\!\!/\!\!\!\!\leftarrow}_{EXP} \mid \pi_1 \mid \pi_2 \mid {}^{\mathrm{N}}\pi_{\mathrm{N}} \mid \mathsf{id} \mid \mathsf{if}(EXP, EXP, EXP) \\
 & & \mid (EXP)* \mid \mathsf{while}(EXP, EXP, EXP)
\end{array}
$$

$$
\begin{array}{lll}
OP & ::= & ! \mid \mathsf{iota} \mid \mathsf{distl} \mid \\
 & & \mathsf{gt} \mid \mathsf{ge} \mid \mathsf{lt} \mid \mathsf{le} \mid \mathsf{ne} \mid \mathsf{or} \mid \& \mid \\
 & & \text{-} \mid + \mid \times \mid \div \mid \mathsf{div}
\end{array}
$$

$$
K \quad ::= \quad \mathsf{true} \mid \mathsf{false} \mid INT \mid RL
$$

Auxiliary definitions

$$
\begin{array}{lll}
ENV & ::= & \mathsf{env}(VAR, FUN, N) \\
 & & \\
VAR & ::= & VARBIND^{*} \\
 & & \\
VARBIND & ::= & \mathsf{vb}\ (VOD,\ EXP) \\
VOD & ::= & \mathsf{dummy} \mid V \\
 & & \\
FUN & ::= & \mathrm{FUNBIND}^{*} \\
 & & \\
FUNBIND & ::= & \mathsf{fb}(V,\ CLOS) \\
 & & \\
CLOS & ::= & \mathsf{clos}(E, N, VAR, VAR, FUN)
\end{array}
$$

At one point (inside the definition of $CLOS$) there is a reference to an expression: E from the Adl syntax. This appears because a function closure incorporates an Adl expression for later processing.

[15] This distinction will be dwelt on at slightly more length in section 4.5.2

Set *Translate* **is** $\boxed{\vdash PR \rightsquigarrow EXP}$

Programs

$$\frac{\begin{array}{c} Remove\_name\_clashes(\vdash decls\, pat\ e \rightarrow pat' e') \\ Extract\_bvar(pat' \rightarrow var\ n) \qquad Length\_var(var \rightarrow n) \\ \mathtt{env}(var, [\,], n) \vdash \mathtt{let}\ decls\ \mathtt{in}\ e'\ \mathtt{endlet} \Rightarrow exp \end{array}}{\vdash \mathrm{pr}(decls, \mathrm{fdec}(x\ pat\ \mathtt{:=}\ e)) \rightsquigarrow exp} \quad (1)$$

Figure 1: Translator rule for programs

Since this reference appears in the auxiallary section and none of the syntax in the auxiliary section appears in the final translation we can be assured that the final BMF program will be free of Adl syntax.

4.5 The rules of translation

I describe the translator by using inference rules. A particular piece of code will match the bottom line (or sequent) of one or more rules. When more than one rule matches the conditions and the premises provide a mechanism for discrimination between alternatives (i.e. if either the condition or the premises cause the proof of a rule to fail then the next matching rule can be consulted). The general layout of an inference rule is:

$$\frac{(antecedent \vdash expression \Rightarrow exp)^*}{antecedent \vdash expression \Rightarrow exp}\ condition^*$$

Where the superscript $*$ is again used to denote zero or more occurrences of the object. The part of the rule below the line forms a conclusion. The conclusion is proved true if all of the premises above the line and the conditions beside the line are met.

The part of the conclusion to the right of the $\Rightarrow$ symbol represents the code produced by the translator. The type of arrow used depends on the judgement the rule belongs to. I happen to use the $\Rightarrow$ symbol as an example because it appears most frequently in the top level rules of the translator.

Rules without premises are axioms. Axioms are used when the object being processed is simple enough to be processed without reference to further steps in a set of premises e.g. constants, empty-sequences etc.

4.5.1 Programs

The set forming the top-level rules of the translator consists of several different rule types called judgements. The judgement for handling programs has only one rule it is shown in figure 1. Rule 1 is the only rule that matches a complete Adl program. As such it forms the sole entry point to the system of rules which follow. The type of the judgement is shown in the top right hand corner of the figure. The $\rightsquigarrow$ symbol is used to distinguish this judgement from the others in the *Translate* set.

The rule's conclusion will match only a (possibly empty) declaration sequence followed by a single function declaration. Programs that fail to fall into this pattern will fail to translate.

The declaration sequence (*decls*) is used to denote the top-level declarations in the Adl program preceding its entry point. The function forming this entry point is defined by the fdec(x *pat* `:=` e) pattern.

The rule has four premises. First of these is *Remove_name_clashes*. This rule-set collects the names of all the variables declared in *decls*. It then checks to see if these names clash with any inside *pat*. If they do then the names in *pat* and the corresponding names in e are changed. The results are held in e' and pat'. Both of these are items of Adl's abstract syntax. *Remove_name_clashes* is the only genuine source-to-source transformation in the translation. The reasons for needing it will be explained after the other premises are described.

The second is a call to a judgement in another rule set called *Extract_bvar* which is formally defined in section 4.9. *Extract_bvar* is normally used to help create a closure out of a function declaration. Its role is to take the pattern: *pat* and use it to create a variable environment: *var*. A third rule set, *Length_var* finds the length of *var*, placing it in the integer offset n.

The fourth premise takes the values, *var* and n and injects them into a complete environment. The declaration sequence: *decls* and the body of the function declaration, e, are injected into a `let` expression. Finally the `let` expression is evaluated in the new environment using this translators judgment for Adl expressions (we can tell this because the $\Rightarrow$ symbol is used and the types of all the parameters are correct). The final result of the translation is the BMF expression denoted by *exp*.

It is the injection of the elements of the program into a `let` expression which gives rise to the need for *Remove_name_clashes*. By turning the program node into a let expression with a pre-defined environment containing the bound variables of pat' we are turning the program into a form, more palatable for the rest of the translator. Unfortunately the transformation from a program to `let` cannot always take place without having to compensate for the changes in scope it introduces. To illustrate consider the following Adl program:

```
     a := 3;
     b := 4;
f(a,b) := a + b
```

It would be incorrect to tranform this code into the expression:

```
f(a,b) := let
             a:= 3;
             b:= 4
          in
             a + b
          endlet
```

Remove_name_clashes has the task of renaming the bound variables of `f` so that they don't clash with the variable declarations. A properly transformed version of the program might look like:

$$\boxed{\begin{array}{c} ENV \vdash E \Rightarrow EXP \\ \vdash CONST \triangleright EXP \end{array}}$$

Constants

$$\frac{\vdash const \triangleright exp}{\rho \vdash const \Rightarrow exp} \tag{2}$$

Booleans

$$\vdash \texttt{true} \triangleright \mathsf{true} \qquad \vdash \texttt{false} \triangleright \mathsf{false} \tag{3}$$

Reals and Integers

$$\vdash real \triangleright rl \qquad \vdash integer \triangleright int \tag{4}$$

Variable names

$$\frac{\mathsf{env}(var, fun, n) \vdash Lookup\_var(var \vdash x \rightarrow exp)}{\mathsf{env}(var, fun, n) \vdash x \Rightarrow exp} \tag{5}$$

Figure 2: Translator rules for handling constants and variables

```
f(_a,_b) := let
                a:=3;
                b:=4
              in
                _a + _b
              endlet
```

4.5.2 Constants and variables

The rules for the translation of Adl constants and variable names into their equivalent BMF are given in figure 2

These handle the canonical objects in Adl syntax. Rule 2 strips the constant of its "constant" tag and invokes the judgement used for handling constants. Rules 3 and 4 are the components of this judgement. They convert boolean, integer and real constants into their equivalents in BMF. Some details of the translation of integers and reals have conveniently been ignored. The rules handling these should specify that the *int* and *rl* values produced map to corresponding values in *real* and *integer*.

int and *rl* are not quite ordinary integer and real values. In BMF these items are functions. The function *int* can take on the meanings: $\mathsf{0}, \mathsf{1}, \mathsf{2} \ldots$. These are overloaded. When they are applied to any value at all, the integer value corresponding to their name is returned. Ordinary integer values, $0, 1, 2 \ldots$ cannot be applied to anything [16].

Rule 5 takes variable names and converts them to their equivalents in the BMF target code. This task is not trivial. The target language does not admit the use of variable names. The only way

[16] Overloading the functions $\mathsf{0}, \mathsf{1}, \mathsf{2} \ldots$ to operate on any type of value is not very pleasing from a formal viewpoint. A thorough treatment would subscript each function with the type it expects to receive. I will leave the subscripts out as a matter of convenience.

$\boxed{ENV \vdash E \Rightarrow EXP}$

Conditionals

$$\frac{\rho \vdash e_1 \Rightarrow exp_1 \qquad \rho \vdash e_2 \Rightarrow exp_2 \qquad \rho \vdash e_3 \Rightarrow exp_3}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{endif} \Rightarrow \mathsf{if}(exp_1, exp_2, exp_3)} \quad (6)$$

Figure 3: Translator rule for Adl conditionals

accessing a value is by using functions to retrieve it from the values passed into the section of the program which is using the variable. These functions generally take the form of product-type accessing functions e.g. π_1,π_2 (often referred to as fst and snd in the literature). If product types forming the input to the current section of the program are nested then these access functions may be composed (e.g. $\pi_2 \cdot \pi_1 \cdot \pi_1$).

The rules to generate the appropriate sequence of functions to access an address are defined in the rule set *Lookup_var* in section 4.6. The rules which make sure that the values are passed around the executing target code using tuples of the expected forms are called by the code which handles variable declarations and, to a lesser extent, function application (function applications must contain some code for passing their bound variables into their bodies). These issues are dealt with, at least in part, in sections 4.5.4 and 4.5.5 respectively.

4.5.3 Conditionals

The rule for the translation of conditionals is shown in figure 3. It is very straightforward. First, three subcomponents are translated. If this step succeeds the results of these translations are placed inside the BMF if( ) construct. The syntax used for the conditional differs from its common usage in the literature (which is $(exp_1 \rightarrow exp_2, exp_3)$). I've used a different convention because the $\rightarrow$ may be confusing in the context of inference rules and the if form is closer to the format the construct will take in the translator's implementation.

4.5.4 Let

The rules for let are shown in figure 4. There are two rules for `let` expressions. The first handles the trivial case where the declaration sequence is empty. When this happens the translation of the `let` expression is exactly the same as the translation of its body: e. This case only ever occurs at the top level of the program when rule 1 generates an empty declaration sequence because no declarations preceed the program's entry point.

The more routine `let` expression is handled by rule 8 which states that to translate a `let` you first attempt to process its declarations. Then you must process its body in the environment existing after the declarations have been handled.

Close inspection of the top premise of rule 8 reveals that the format of the predicate (or judgement) is different from the normal top level judgement of the translator. This judgment takes a declaration sequence as its argument (rather than a generic expression). It uses a different arrow symbol. Lastly it produces both an environment, ρ and an expression rather than just an expression.

$$\boxed{\begin{array}{l} ENV \vdash E \Rightarrow EXP \\ ENV \vdash DECLS \rightarrow ENV\ EXP \end{array}}$$

Let expressions

$$\frac{\rho \vdash e \Rightarrow exp}{\rho \vdash \texttt{let}\ []_{DEC}\ \texttt{in}\ e\ \texttt{endlet} \Rightarrow exp} \tag{7}$$

$$\frac{\begin{array}{c}\rho \vdash dec^{+} \rightarrow \rho'\ exp \\ \rho' \vdash e \Rightarrow exp'\end{array}}{\rho \vdash \texttt{let}\ dec^{+}\ \texttt{in}\ e\ \texttt{endlet} \Rightarrow exp' \cdot exp} \tag{8}$$

Declaration sequences

$$\rho \vdash [\,]_{DEC} \rightarrow \rho\ \mathsf{id} \tag{9}$$

$$\frac{\begin{array}{c}Handle\_vdec(\rho \vdash vdec \mapsto \rho', exp) \\ \rho' \vdash dec^{*} \rightarrow \rho''\ exp'\end{array}}{\rho \vdash (vdec : dec^{*}) \rightarrow \rho''\ exp' \cdot exp} \tag{10}$$

$$\frac{\begin{array}{c}Handle\_fdec(\rho \vdash fdec \mapsto \rho') \\ \rho' \vdash dec^{*} \rightarrow \rho''\ exp'\end{array}}{\rho \vdash (fdec : dec^{*}) \rightarrow \rho''\ exp'} \tag{11}$$

Figure 4: Translator rules for Let expressions

The judgement in this premise is used solely to translate declaration sequences and update the environment accordingly.

The three rules for this judgement are rules 9, 10 and 11. The three cases they deal with are an empty declaration sequence, a sequence with a variable declaration at the front, and a sequence with a function declaration at the front respectively.

The rule for handling empty sequences can only be invoked after one or more variable or function declarations have been processed. This is because the pattern in rule 8 specifies a declaration sequence of dec^{+}, which means the original sequence must contain one or more variable or function declarations. The rule for the empty sequence returns the same environment as it received and returns the identity function: id.

The rule for handling a sequence with a variable declaration at the front makes a call to the rule set: $Handle\_vdec$ which processes $vdec$ and the old environment: ρ into exp and ρ'. The second premise processes the rest of the sequence under the new environment (ρ'). It is worth noting that exp forms part of the final result of the rule. This indicates that variable declarations not only change the environment but add to the code. The details of this process are hidden inside $Handle\_vdec$ (in section 4.8) but it is useful to include an example of its action here.

Assume we have the following Adl declaration embedded in a declaration sequence:

```
a := 4
```

Since this is a $vdec$ it will be processed by $Handle\_vdec$. The environment ρ which is fed into the rule set will have the format: $\mathsf{env}(var, fun, n)$. Since this is a variable declaration only the var and n components will be affected. The var environment consists of a list of variable bindings. The format of a binding is:

$$\mathsf{vb}(v, exp)$$

Where v stands for a variable name (which is only ever used in translation, it doesn't appear in the target code) and exp stands for an address function e.g π_1,π_2, id etc. $Handle\_vdec$ adds a binding onto the front of the var list. In the case of the declaration above the new var entry would look like:

$$(\mathsf{vb}(a, \mathsf{id}) : old\_var)$$

The address portion is always id for variable declarations. This part represents the function that would be used to locate the value of a once this entry in the environment has been reached. Since a and all left-hand-sides of declarations in Adl are singleton values rather than tuples only the function id (i.e. no function at all) is necessary to reach a's value once the part of the environment a is stored in is reached. The exp part contains other addresses when the time comes to access components of a tuple. Names can only be assigned to tuple elements by binding a value to the tuple forming the input parameter to a function. This binding only happens upon function application. The generation of these address will be covered in the description of the rules for function application [17].

The n component of the environment is incremented to keep a record of how much the environment has changed. This record is used by the rules for function application to determine the amount of change between the environment when the function was declared and when it is used.

$Handle\_vdec$ also generates an expression. For the declaration above this is simply:

$$(\mathsf{id}, 4)^\circ$$

The id function is generated for any variable declaration. It has the effect of passing data from previous sections of the program onto the next. The value 4 is the translation of the right hand side of the declaration. The surrounding $(\ )^\circ$ construct stands for all-applied-to for tuples. It takes its argument splits it into the number of components within the tuple (in this case two) and applies the function at each position to them to produce a tuple of values of the same arity. This function is a very simple higher order construct[18].

The code produced by the translation of a sequence of declarations resembles:

$$(\mathsf{id}, exp_n)^\circ \cdot \ldots \cdot (\mathsf{id}, exp_2)^\circ \cdot (\mathsf{id}, exp_1)^\circ \ldots$$

this code would produce the tuple:

$$((\ldots(((\ldots), exp_1), exp_2), \ldots), exp_n)$$

upon execution. This tuple contains all of the values generated by the declaration sequence. These values may then be accessed by the body of the `let`.

Now we consider the rule for function declarations. The rule for a declaration sequence, starting with an $fdec$, is very similar to the rule for a declaration sequence starting with a $vdec$, except that the first premise is the call:

$$Handle\_fdec(\rho \vdash fdec \mapsto \rho')$$

[17] These types of variable bindings are also described in section 4.6 containing the rules for translating variable names.

[18] The $(\ )^\circ$ notation I use is non-standard. The literature contains a few variations on this construct. The one I have chosen is an adaptation of Richard Bird's applied-to-all construct for lists.

$\boxed{ENV \vdash E \Rightarrow EXP}$

Function applications

$$\frac{\rho \vdash e \Rightarrow exp \qquad Lookup\_funenv(\rho \vdash x \rightarrow \mathsf{clos}(e', n', var', var'', fun')) \quad Build\_new\_env(n, n', var', var'', fun' \mapsto \rho') \qquad \rho' \vdash e' \Rightarrow exp'}{\mathsf{env}(var, fun, n) \vdash \mathrm{app}(x\ e) \Rightarrow exp' \cdot (\mathsf{id}, exp)^\circ} \tag{12}$$

Figure 5: Translator rule for function application

The rule set $Handle\_fdec$ doesn't return an expression it just returns a changed environment. Only the fun component of the environment is changed. The new fun environment contains an extra function binding of the form:

$$\mathsf{fb}(v, \mathsf{clos}(e, n, var, var', fun))$$

Where v is a function name and cl is a closure. The closure contains a copy of the function's body, a copy of the environment offset, a copy of the var as it is when the function's declared (a product of Adl's static scoping) a copy of the function's bound variables (arranged to look like an environment), and finally a copy of the current function environment.

The details on the construction of the closure are presented in section 4.9. The details on its use are presented with the rules for function application.

4.5.5 Function Application

The rule for function application is shown in figure 5 It is, arguably, the most complex in the translation semantics. It contains four premises as well as some minor post processing in the conclusion.

The first premise translates the function x's actual parameters into the BMF expression exp. This code is wrapped up in an $(id, exp)^\circ$ expression in the rule's conclusion to ensure the data that could be accessed by the free variables in the function body is passed into where the function body may reach them.

The second premise uses the rule set $Lookup\_funenv$ to search the function environment for the closure corresponding to x. As might be imagined, the rules contained contained in $Lookup\_funenv$ are quite simple. They are defined in section 4.7.

The third premise uses the rule set: $Build\_new\_env$ and most of the components of the closure to construct a new environment in which to carry out the translation of the function body: e'. The fourth premise handles the translation of e'.

A large amount of detail is hidden in the reference to $Build\_new\_env$. There is only one rule in this set. It is shown in figure 6.

Rule 13 uses four components from a function closure as input and uses them to create a new

Set *Build_new_env* **is** $\boxed{N, N, VAR, VAR, FUN \mapsto ENV}$

Environment building

$$\frac{\begin{array}{c} n' - n = n'' \qquad Repeat\_dummy(n'' \mapsto var'') \\ append\_varenv(var'', var' \rightarrow var''') \\ append\_varenv(var, var''' \rightarrow var'''') \end{array}}{n, n', var, var' fun \rightarrow \mathsf{env}(var'''', fun, n'+1)} \tag{13}$$

End *Build_new_env*

Figure 6: Rule for fabricating an environment for function application

environment for the evaluation of the function body contained in that closure. It's main purpose is to adjust the variable environment as it stood at the time of closure creation (represented by var') to bring it up to date with the environment of values as it stands now.

To abide by the conventions of static scoping, no variables defined after the front entry on var' was defined should be visible in the new environment. The tuple passed into this part of the program will almost certainly contain some of these values and some mechanism must be created so that they can be ignored.

The first three premises build such a mechanism. The first premise just subtracts the environment offset at the time of the building of the closure from the offset at the point when the function was called. This gives the translator an indication of how many declarations have been made since this closure was built. The number n'' will be then number of entries in the input tuple to the function body that the program will have to ignore before it gets to the first entry of the original environment var''. The second premise creates an environment, of length n'', containing only dummy entries:

$$\mathsf{vb}(\mathsf{dummy}, \mathsf{id})$$

The third premise appends this dummy environment to the front of var'. Now any access to a variable in the function's body will generate the original address of the value in var' plus the address generated by the dummy entries in the environment.

The last premise adds the bound variable environment: var to the front of var''. This allows the translator to generate addresses for accessing the bound variables of the function's body.

The first premise simply subtracts the n from the time of the functions declaration from the n' value as it presently stands. This yields n'', the length of the gap between the environments that must be filled in.

n'' is used by the judgement in the second premise: *Repeat_dummy* (not formally defined here), to create an environment of n'' entries. These entries all take the form $\mathsf{vb}\,(\mathsf{dummy,current})$. Their purpose is to pad out the environment so that *search_list* can generate the correct addresses.

The third premise appends the old variable environment from the closure to the dummy environment. The last premise adds the bound variable environment to the other end.

4.5.5.1 Short example The way in which the environment is built and used may still be obscure. This is a good point at which to present an illustrative example of the variable environment and how it is used to help build the translation of a function's body. Consider the function declaration:

```
f (x,y) := x + y + b
```

by the time this code is reached we assume that some variable declarations have been processed. They have been used to form the variable environment:

$$[\mathsf{vb}(c, \mathsf{id}), \mathsf{vb}(b, \mathsf{id}), \mathsf{vb}(a, \mathsf{id})]$$

Where a was the first value to be declared and c the last. Call this environment var'. This environment will be integrated into the function closure:

$$\mathsf{clos}(\texttt{x + y + b}, n, [\mathsf{vb}(x, \pi_1), \mathsf{vb}(x, \pi_2)], var', fun)$$

Where the value of n is 3 and the third item of the closure is the variable environment formed from the bound variables x and y (I will refer to this environment as var).

Now the translator moves on to the place where the function is to be applied. There have been some other function declarations. The current variable environment is now:

$$[\mathsf{vb}(e, \mathsf{id}), \mathsf{vb}(d, \mathsf{id}), \mathsf{vb}(c, \mathsf{id}), \mathsf{vb}(b, \mathsf{id}), \mathsf{vb}(a, \mathsf{id})]$$

The function `f` should not be allowed to directly access the values of d and e. However the execution of the translated BMF code leading up to the function application would yield a tuple that looks like:

$$(((((val\_a, val\_b), val\_c), val\_d), val\_e), (val\_x, val\_y))$$

where the tuple at the end of the environment: (val_x, val_y) are the values created by the code produced by the first premise of the function application rule 12 (i.e. the functions actual parameters). The job of the translator at this stage is to create code for the body of `f` which takes account of the space in the tuple consumed by d and e without allowing the body to reference them.

The solution is contained in *Build_new_env*. The first premise takes the new environment offset which is 5 and takes the environment offset n (equal to 3) from it. This tells us the gap to be filled in is of length 2. The rule set *Repeat_dummy* creates the environment:

$$[\mathsf{vb}(\mathsf{dummy}, \mathsf{id}), \mathsf{vb}(\mathsf{dummy}, \mathsf{id})]$$

This is appended to var' and then to the bound variable environment to give us the environment in which `f`'s body is to be translated:

$$[\mathsf{vb}(x, \pi_1), \mathsf{vb}(x, \pi_2), \mathsf{vb}(\mathsf{dummy}, \mathsf{id}), \mathsf{vb}(\mathsf{dummy}, \mathsf{id}), \mathsf{vb}(c, \mathsf{id}), \mathsf{vb}(b, \mathsf{id}), \mathsf{vb}(a, \mathsf{id})]$$

This environment gives us the correct offsets for all the values in the tuple whilst effectively rendering the values of e and d invisible. For the sake of interest the final translation of the body of `f` is:

$$+ \cdot (+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, \pi_2 \cdot \pi_1^3)^\circ$$

Where *x_val* is accessed by the function $\pi_1 \cdot \pi_2$ and *y_val* is accessed by $\pi_2 \cdot \pi_2$. The value: *b_val* is accessed by $\pi_2 \cdot \pi_1^3$ where π_1^3 is syntactic sugar for $\pi_1 \cdot \pi_1 \cdot \pi_1$.

$$\boxed{ENV \vdash E \Rightarrow EXP}$$

Binary operators

$$\frac{\rho \vdash e_1 \Rightarrow exp_1 \qquad \rho \vdash e_2 \Rightarrow exp_2 \qquad Convert\_op(binop \rightarrow op)}{\rho \vdash \mathrm{binapp}(e_1\, binop\, e_2) \Rightarrow op \cdot (exp_1, exp_2)^{\circ}} \tag{14}$$

Unary operators

$$\frac{\rho \vdash e \Rightarrow exp \qquad Convert\_op(unop \rightarrow uop)}{\rho \vdash \mathrm{unapp}(unop\, e) \Rightarrow uop \cdot exp} \tag{15}$$

Figure 7: Translator rules for binary and unary operators

$$\boxed{ENV \vdash E \Rightarrow EXP}$$

Tuple literals

$$\frac{Eval\_explist(\rho \vdash (e : (e : e^*)) \mapsto exp^{2+})}{\rho \vdash \texttt{(}\,(e : (e : e^*))\,\texttt{)} \Rightarrow (exp^{2+})^{\circ}} \tag{16}$$

Vector literals

$$\frac{Eval\_explist(\rho \vdash elist \mapsto exp^*)}{\rho \vdash \texttt{[}e^*\texttt{]} \mapsto [exp^*]^{\circ}} \tag{17}$$

Figure 8: Translator rules for tuple and vector literals

4.5.6 Binary and Unary Operators

The rules for the translation of binary and unary operators are contained in figure 7. Both rules are very simple. Rule 14 handles binary operator application. It first translates the rule's operands and then converts the operator to its BMF equivalent (usually with the same name). The operands are paired into an all-applied-to for tuples function. The converted operator is composed with this.

Rule 15 is even simpler. There is only one operand and it is composed directly with the converted operator.

4.5.7 Tuples and Vector literals

The rules for translating tuple and vector literals are shown in figure 8. Tuples are translated to an all-applied-to (for tuples) function of the same arity as the original tuple. The items inside the tuple are individually translated using the *Eval_explist* rule set. The definition of *Eval_explist* is shown in figure 9

In this version of the inference rules I have chosen to represent an expression list as a con-list containing expressions. A cons-list representation, due to its recursive construction, is a convenient medium for a natural-semantic description to work with. A much more efficient representation would very likely be used in a final implementation.

Set *Eval_explist* **is** $\boxed{ENV \vdash E^* \mapsto EXP^*}$

Empty explist

$$\rho \vdash []_E \mapsto []_{EXP} \tag{18}$$

Multiple items

$$\frac{\rho \vdash e \Rightarrow exp \qquad \rho \vdash e^* \mapsto exp^*}{\rho \vdash (e : e^*) \mapsto (exp : exp^*)} \tag{19}$$

End *Eval_explist*

Figure 9: The rule set for $Eval\_explist$.

$\boxed{ENV \vdash E \Rightarrow EXP}$

Map expressions

$$\frac{\rho \vdash x\ e \Rightarrow exp' \cdot exp}{\rho \vdash \mathtt{map}\ x\ e \Rightarrow exp' * \cdot \mathsf{distl}.exp} \tag{20}$$

Figure 10: Translator rule for map expressions

An expression list is not a fully-fledged Adl expression, it is a sub-component of an all-applied-to expression for either tuples or for vector literals.

Rule 18 caters for empty expression lists. The subscripts indicate which domains the empty lists come from.

Rule 19 handles lists of one or more items. Its premise is a recursive call to this judgement.

Vectors are catered for by rule 17. The (possibly empty) expression list is evaluated by a call to the rule set: *Eval_explist*. The resulting expression sequence (exp^*) is then inserted into an all-applied-to operator for vectors $[\,]^\circ$.

4.5.8 Map

The rule for map is shown in figure 10. At first, rule 20 seems to be quite straightforward. This simplicity stems from a reasonably subtle “trick”. Close inspection of the rule shows that to yield the correct function body, exp' the function x must be applied to the equivalent of a single element of $\mathsf{distl}.exp$ (noting that the rules for function application ensure that exp has the form (id, exp'') so that all items of data can be passed on by the id). The rule does not do this instead it applies x directly to e.

The rule remains correct due to the fact that x, when applied to anything in the environment ρ, will produce the same code for the function body. That is, the value of exp' depends only on the environment ρ. Anything could be substituted for e on the top row of the rule and the function body derived from x's closure would be the same.

Moreover the insertion of distl (distribute left element) between exp and $exp'*$ does not change the

$\boxed{ENV \vdash E \Rightarrow EXP}$

Undirected reductions

$$\frac{\rho \vdash e_1 \Rightarrow exp_1 \qquad \rho \vdash f\ e_2 \Rightarrow exp_2' \cdot exp_2}{\rho \vdash \texttt{reduce}\ f\ e_1\ e_2 \Rightarrow \pi_2 \cdot ((\pi_1^2, (exp_2' \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/_{exp_1} \cdot \mathsf{distl} \cdot exp_2} \quad (21)$$

Figure 11: Translator rule for undirected reductions

offset component of ρ. So as long as $\mathsf{distl} \cdot exp$ produces the output that $exp'*$ expects (which it does) the correctness of the whole sequence is guaranteed.

4.5.9 Reduce

There are several forms of `reduce`. The rule for the undirected reduction of vectors of length zero or greater is shown in figure 11. The translation is defined by a single rule. The first step is to evaluate the expression forming the second argument into the BMF expression, exp_1. This code is inserted directly into the result code as the subscript to the `reduce` expression. The second step applies the list expression to the function forming the first argument, x. As with map, the actual contents of the argument e_2 will not influence the code produced for the body of x. The target code for x resides in exp_2' while the translation of e_2 is placed in exp_2. In common with the results of all function applications exp_2 will have the format $(\mathsf{id}, exp_2'')^\circ$ where exp_2'' is the direct translation of e_2 and the surrounding all-applied-to expression is provided to ensure that the rest of the environment is passed on.

The last step of `reduce` translation is encapsulated in the conclusion to rule 21. A distl term is added to distribute the contents of the current environment (passed on by id in the expression: $(\mathsf{id}, exp_2'')^\circ$) over the vector generated by e_2.

The second component of the translation is the insertion of an "adjustment-term" in the body of `reduce`. This term ensures that data reaches exp_2' in the form it expects. exp_2' (like any translation of an Adl function) expects its input environment to be in the form of the tuple:

$$((free\_vars), bound\_vars)$$

but (due to the nature of `reduce`) receives it in the form:

$$(((free\_vars), bound\_var_1), ((free\_vars), bound\_var_2))$$

Obviously there needs to be some pre-adjustment of this term. The function used to select the appropriate items is:

$$(\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ$$

Where the π_1^2 selects the first copy of the $free\_var$ environment (the choice of $free\_var$ is arbitrary, the pre-adjust function would be just as valid by using a $\pi_1 \cdot \pi_2$ in its place since the copies of $free\_vars$ are identical). The other half of the pre-adjustment function selects the items for the input tuple for the bound variables of exp_2'.

$\boxed{ENV \vdash E \Rightarrow EXP}$

Undirected reductions for non-empty vectors

$$\frac{\rho \vdash x\ e \Rightarrow exp' \cdot exp}{\rho \vdash \texttt{reducep}\ x\ e \Rightarrow \pi_2 \cdot ((\pi_1^2, (exp' \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/ \cdot exp} \quad (22)$$

Figure 12: Translator rule for undirected reductions on positive length vectors.

After exp'_2 is evaluated it is paired with another copy of the $free\_vars$ environment. This pairing is necessary because during a reduction the output of one function application could be used as one of the inputs (either left or right in the case of `reduce`) to another. When this happens this input will have to contain a copy of the free variable environment for the body of the function to access.

Finally there is a post-adjustment function for the whole `reduce`: π_2 which selects the second element of the result of the reduction. This is to preserve the semantics of `reduce` which is designed to produce a single, unaccompanied value as a result.

Much of the detail shown in the conclusion of the `reduce` rule could have been hidden. If this were the case the behavior implied by the distl and the pre and post-adjustment functions would have to be implicit in the behaviour of the `reduce` primitive. I have chosen to make this processing explicit so the semantics of the BMF $/_e xp$ function itself can be as simple as possible. In cases where x is a combinator (not a rare occurrence), it should be possible for lateer stages of the compiler to strip away the pre and post-adjust sections of the `reduce`. If the pre and post-adjustment code of `reduce` were implicit these functions would not be accessible to optimisation.

`reduce` is unordered i.e. the order of reduction of the expression exp_2 is left unspecified. To produce deterministic results the operator exp'_2 must be associative [19]. There are sequential forms of `reduce`, `reducel` and `reducer` to perform reduction from the left and right respectively. In addition to these forms there are the equivalent functions `reducep`, `reducelp` and `reducerp` that don't need the user to specify the base value argument, e_1. Understandably, these functions are undefined on empty lists.

The inference rules for all forms of `reduce` are very similar to rule 21. The main difference being in the rules for the the reductions without base arguments which take the shorter form shown in figure 12.

The conversions of the various forms of reduce to their equivalent BMF symbols are shown in table 4.5.9. The exp' term stands for the both the pre and post-adjustment code as well as the direct translation of the body of `f`

[19] It is the programmer's responsibility to ensure that the operator used is associative. If it isn't results will be unpredictable.

Adl syntax	*BMF symbols*
`reduce` $x\ e\ e'$	$\pi_2 \cdot (exp')/_{exp''} \cdot \mathsf{distl} \cdot exp$
`reducel` $x\ e\ e'$	$\pi_2 \cdot (exp')\not\rightarrow_{exp''} \cdot \mathsf{distl} \cdot exp$
`reducer` $x\ e\ e'$	$\pi_2 \cdot (exp')\not\leftarrow_{exp''} \cdot \mathsf{distl} \cdot exp$
`reducep` $x\ e'$	$\pi_2 \cdot (exp')/ \cdot \mathsf{distl} \cdot exp$
`reducelp` $x\ e'$	$\pi_2 \cdot (exp')\not\rightarrow \cdot \mathsf{distl} \cdot exp$
`reducerp` $x\ e'$	$\pi_2 \cdot (exp')\not\leftarrow \cdot \mathsf{distl} \cdot exp$

Table 1: Mapping from reduce expressions in Adl syntax to their equivalents in BMF

$$\boxed{ENV \vdash E \Rightarrow EXP}$$

Scans on possibly empty vectors

$$\frac{\begin{array}{c}\rho \vdash e_1 \Rightarrow exp_1 \\ \rho \vdash x\ e' \Rightarrow exp'_2 \cdot exp_2\end{array}}{\rho \vdash \mathsf{scan}\ x\ e_1\ e_2 \Rightarrow \pi_2 \cdot ((\pi_1^2, (exp'_2 \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/\!\!/_{exp_1} \cdot \mathsf{distl} \cdot exp_2} \tag{23}$$

Scans on non-empty vectors

$$\frac{\rho \vdash x\ e \Rightarrow exp' \cdot exp}{\rho \vdash \mathsf{scanp}\ x\ e \Rightarrow \pi_2 \cdot ((\pi_1^2, (exp' \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/\!\!/ \cdot \mathsf{distl} \cdot exp} \tag{24}$$

Figure 13: Two basic forms of translator rules for scan.

4.5.10 scan

The translation rules for the two basic forms of scan are shown in figure 13. The translation rule for `scan` is almost identical to that for `reduce`. Like `reduce` there are six variations of `scan`. These are, `scan`, `scanl`, `scanr`, `scanp`, `scanlp`, and `scanrp`.

The variations ending with `p` have only two arguments and are undefined on empty vectors. The rule for these variations follows the form shown in rule 24. The mappings from the various scan combinators to their equivalent BMF forms is shown in table 4.5.10

Adl syntax	*BMF symbols*
`scan` $x\ e\ e'$	$(exp')/\!\!/_{exp''} \cdot \mathsf{distl} \cdot exp$
`scanl` $x\ e\ e'$	$(exp')\not\!\not\rightarrow_{exp''} \cdot \mathsf{distl} \cdot exp$
`scanr` $x\ e\ e'$	$(exp')\not\!\not\leftarrow_{exp''} \cdot \mathsf{distl} \cdot exp$
`scanp` $x\ e'$	$(exp')/\!\!/ \cdot \mathsf{distl} \cdot exp$
`scanlp` $x\ e'$	$(exp')\not\!\not\rightarrow \cdot \mathsf{distl} \cdot exp$
`scanrp` $x\ e'$	$(exp')\not\!\not\leftarrow \cdot \mathsf{distl} \cdot exp$

Table 2: Mapping from scan expressions in Adl syntax to their equivalents in BMF

$\boxed{ENV \vdash E \Rightarrow EXP}$

While expressions

$$\frac{\rho \vdash x_1\ e \Rightarrow exp'' \cdot exp \qquad \rho \vdash x_2\ e \Rightarrow exp' \cdot exp}{\rho \vdash \texttt{while}\ x_1\ x_2\ e \Rightarrow \mathsf{while}\,(exp'', exp') \cdot exp} \tag{25}$$

Figure 14: Translator rule for `while`.

4.5.11 while

The last rule of the translator handles `while` expressions. It is shown in figure 14 The `while` construct has three arguments. The first two are function names, x_1 representing the body of the while loop, x_2 representing the predicate. The expression e is the expressions input data (this component becomes implicit and is fed in through a composition after translation).

This translation rule is rather wasteful since it evaluates the input expression, e, twice. This is the price paid for making use of the rule for function application in such a naive way. A more efficient version could be arrived at by using some simple dummy expression e.g. the integer constant `1` in place of e for one of the evaluations. Since the actual value of e won't affect the translation of the function it's applied to, this approach will suffice to produce the correct result. A more efficient version of the above rule is:

$$\frac{\rho \vdash f_1\ \mathtt{1} \Rightarrow exp'' \cdot \mathtt{1} \qquad \rho \vdash f_2\ e \Rightarrow exp' \cdot exp}{\rho \vdash \texttt{while}\ f_1\ f_2\ e \Rightarrow \mathsf{while}\,(exp'', exp') \cdot exp}$$

This version will always be more efficient under the assumption that the cost of translating the integer constant, `1` is always less than the cost of translating the expression, e.

4.6 Rules for creating variable names

In the definition of rule 5 in section 4.5.2 we refer to the rule set *Lookup_var* as a means of generating code to access variables. This section contains the definition of *Lookup_var* and a number of auxiliary rules.

4.6.1 Top level rules for variable search

Figure 15 contains the rules defining *Lookup_var*. Rule 26 handles the case where the variable environment contains only one entry and that entry matches x. Since the variable we are searching for comes from the domain X and the variable we want to match it against comes from the domain V we must use a judgment to convert from one format to another. The rule set: *convert* (not shown

Set *Lookup_var* **is** $\boxed{ENV \vdash X \to EXP}$

Lookup for Singleton environment

$$\frac{Convert(x \succ v)}{\mathsf{env}([\mathsf{vb}(v, addr)], fun, n) \vdash x \to \mathsf{id}} \tag{26}$$

Lookup for longer environments

$$\frac{\begin{array}{c} Convert(x \succ v) \\ Search\_list(var \vdash v \to exp) \\ Remove\_ids(\vdash exp \to exp') \end{array}}{\mathsf{env}(var, fun, n) \vdash x \to exp} \quad if\ Length(var) > 1 \tag{27}$$

End *Lookup_var*

Figure 15: Top level rules for searching the variable environment

here) fills this role. It changes the nominal type of the identifier without changing its contents[20].

When there is only element in the variable environment and that is the one being accessed it is treated as a special case. The code produced is the function id (since id is the function you'd use to obtain the entire input of a function). The translation rules will fail if the variable environment is empty or the name being searched for doesn't exist.

The second rule handles searches when there is more than one item in the environment. In these cases the translator must create functions to select the appropriate part of the input. The first premise uses *Convert* to change the type of the identifier. The second is a call to the rule set: *Search_list*. This set is responsible for traversing the variable environment (when its length is greater than one) to generate addresses. The third premise is a post-processing function to strip the result of *Search_list* of superfluous id functions. The rules for both *Search_list* and *Remove_ids* are given in the next section.

4.6.2 Searching the variable environment

The rules defining *Search_list* are shown in figure 16. Rule 28 fits the case where the desired variable is encountered as the last element of the environment. Since the translator produces code that structures the environment as nested tuples in the following manner:

$$((\ldots(val_1, val_2), \ldots), val_n)$$

Where the most recently produced item is outermost, it follows that the function to access val_1 will terminate in a π_1 term. The variable binding: $[\mathsf{vb}(v, exp)]$ in rule 28 corresponds to val_1. The expression: exp is a possible sub-address. Sub-addresses occur if the variable in question was defined as a part of a tuple. This will only happen if the variable was bound by a tuple in a function

[20] *Convert* would be unnecessary but for the distinction between variable names in Adl and those in BMF. The distinction is used out of a desire to keep the sets of syntax completely seperate. As it is I have only partly succeeded; closures still contain Adl expressions (however closures or variables won't ever appear in the target code so the question is a minor one).

Set *Search_list* **is** $\boxed{VAR \vdash V \to EXP}$

Value found

$$[\mathsf{vb}(v, exp)] \vdash v \to exp \cdot \pi_1 \tag{28}$$

$$(\mathsf{vb}(v, exp) : var) \vdash v \to exp \cdot \pi_2 \tag{29}$$

Not yet found

$$\frac{var \vdash v \to exp'}{(\mathsf{vb}(v', exp) : var) \vdash v \to exp' \cdot \pi_1} \quad (v \neq v') \tag{30}$$

End *Search_list*

Figure 16: Rule for address generation

Set *Remove_ids* **is** $\boxed{\vdash EXP \to EXP}$

Identity function removal

$$\frac{\vdash exp \to exp'}{\vdash exp \cdot \mathsf{id} \to exp'} \tag{31}$$

$$\frac{\vdash exp \to exp'}{\vdash \mathsf{id} \cdot exp \to exp'} \tag{32}$$

$$\frac{\begin{array}{c}\vdash exp_1 \to exp'_1 \\ \vdash exp_2 \to exp'_2\end{array}}{\vdash exp_1 \cdot exp_2 \to exp'_1 \cdot exp'_2} \quad (exp_1 \neq \mathsf{id}) \wedge (exp_2 \neq \mathsf{id}) \tag{33}$$

End *Remove_ids*

Figure 17: Rules for removing excess id functions

application. In most cases exp will be id (bound by a variable declaration). After *Search_list* has executed there will usually be a number of *id* functions embedded in the address. These do no harm but can make code harder to read. That is why *Remove_id* exists.

Now to the second rule (number 29). This copes with the (not very remote) possibility of the variable binding for v occurring before the end of the environment. In this case a π_2 function is generated followed by the v's sub-address.

The third rule is the one that enables the translator to incrementally build addresses. When the first element of the variable environment is not v there will be a need to search further into the environment. This is conceptually the same as searching further into the left part of a tuple environment in during the execution of a BMF program. To indicate that this action is required a π_1 function is placed at the front of the code generated by the rules premise. The premise itself is a search of the remainder of the environment.

If the variable name cannot be matched to an entry in the environment then all three rules will fail and the translation will abort (which is to be expected).

The rule set *Remove_ids* is defined in figure 17. Rules 31 and 32 eliminate the identity function from the left and right hand sides, respectively, of a function composition. Both rules check the

$$\boxed{\begin{array}{l} ENV \vdash X \rightarrow CLOS \\ FUN \vdash V \mapsto CLOS \end{array}}$$

Set *Lookup_funenv* **is**

Top level rule

$$\frac{Convert(x \succ v) \qquad fun \vdash v \mapsto clos}{\mathsf{env}(var, fun, n) \vdash x \rightarrow clos} \tag{34}$$

Item found

$$(\mathsf{fb}(v, clos) : fun) \vdash v \mapsto clos \tag{35}$$

Item not found yet

$$\frac{fun \vdash v \mapsto clos}{(\mathsf{fb}(v', clos') : fun) \vdash v \mapsto clos} \qquad (v \neq v') \tag{36}$$

End *Lookup_funenv*

Figure 18: Rules for searching the function environment

remainder of the expression in their premise. In event of the input expression being: $\mathsf{id} \cdot \mathsf{id}$ either of the first two rules may be applied and one of the id expressions is left intact (which is a safe strategy).

If neither of the first two rules apply but expression of concern is still a composition then it is possible that some of the sub-expressions on either side are compositions containing id. The two premises of rule33 perform this checking.

Excess identity functions can be removed by means other than by defining *Remove_id*. Clauses could have been inserted in the rules of *Search_list* to remove them as they were created. The reason *Remove_id* was defined instead lies in the fact that it is a general purpose rule set. There is scope for its use not only in the translator but in other stages of compilation.

4.7 Searching the function environment

The rule set for searching the function environment is shown in figure 18 The translator will enter this set at rule 34 since this is the only rule in the judgement referred to by the top level of the translator. Its purpose is to prepare the input values for further processing. It projects fun out of its environment and converts x to its equivalent v.

Rule 35 handles the case where the name inside the function binding at the start of the environment matches the name being searched for. In this circumstance the closure bound to the name in the function environment is returned.

Rule 36 caters for a mismatch between the binding at the start of the environment and the search argument. When this happens the remainder of the environment is searched (a task performed by the rule's first premise).

Set *Handle_vdec* **is** $\boxed{ENV \vdash VDEC \mapsto ENV, EXP}$

Variable declarations

$$\frac{convert(x \succ v) \qquad Translate(\mathsf{env}(var, fun, n) \vdash e \Rightarrow exp)}{\mathsf{env}(var, fun, n) \vdash x \,\texttt{:=} e \mapsto \mathsf{env}((\mathsf{vb}(v, \mathsf{id}) : var), fun, n+1), (\mathsf{id}, exp)^{\circ}} \quad (37)$$

End *Handle_vdec*

Figure 19: Translator rule for individual variable declarations

Set *Handle_fdec* **is** $\boxed{ENV \vdash FDEC \mapsto ENV}$

Function Declarations

$$\frac{Extract\_bvar(\vdash pat \rightarrow var') \qquad Convert(x \succ v)}{\mathsf{env}(var, fun, n) \vdash x\ pat\ \texttt{:=}\ e \mapsto \mathsf{env}(var, (\mathsf{fb}(v, \mathsf{clos}(e, n, var', var, fun)) : fun), n)} \quad (38)$$

End *Handle_fdec*

Figure 20: Translator rule for individual function declarations

4.8 Handling variable declarations

The rule for processing variable declarations is shown in figure 19. The judgement takes an environment and an expression and returns the new environment and a new expression. The environment is changed by adding a new binding, $\mathsf{vb}(v, \mathsf{id})$, to the front of the original variable environment. The second element of the binding is always id for variable declarations (for reasons outlined in section 4.5.4). The variable name: v, is produced by converting the type of x using the *Convert* rule set.

The expression part of the declaration is transformed to a BMF expression by the use of *Translate*. The resulting expression, exp is embedded inside an all-applied-to for tuples with an id function as its first element. This expression serves as a means of forwarding data from previous declarations onto the remainder of the program.

4.9 Handling function declarations

The rule for *Handle_fundec* is shown in figure 20. *Handle_fdec* is responsible for creating a new entry in the function environment to correspond to a single function declaration. In contrast to *Handle_vdec* there is no BMF expression produced. The creation of an expression is delayed until the section of the code where the function is applied is reached. At that point the code in e will be translated in the context of the appropriate environment and in-lined.

All the information needed to build this code is stored in a function closure. The purpose of *Handle_fdec* is to create this closure, then insert it into the environment.

The closure consists of:

Set *Extract_bvar* **is**

$$\boxed{\begin{array}{l} \vdash PAT \rightarrow VAR \\ N\ N \vdash PAT \mapsto VAR \end{array}}$$

Singleton patterns

$$\frac{convert(x \succ v)}{x \rightarrow [\mathsf{vb}(x, \mathsf{id})]} \tag{39}$$

Tuple patterns

$$\frac{Length(pat^{2+} \rightarrow n) \qquad n\ 1 \vdash pat^{2+} \mapsto var}{\vdash pat^{2+} \rightarrow var} \tag{40}$$

Large tuples

$$\frac{\begin{array}{c} \vdash pat \rightarrow var \qquad Prefix(\vdash {}^{n}\pi'_{n}\ var \rightarrow var') \\ n\ (n'+1) \vdash pat^{*} \mapsto var'' \qquad Concat\_env(var', var'' \rightarrow var''') \end{array}}{n\ n' \vdash (pat : pat^{*}) \mapsto var''} \tag{41}$$

Minimal tuples

$$n\ n' \vdash [\,]_{PAT} \rightarrow [\,]_{VAR} \tag{42}$$

End *Extract_bvar*

Figure 21: Rules for creating a variable environment out of *fundec* patterns.

1. The original expression e from the body of the function declaration.
2. The original environment offset n.
3. A variable environment derived from pat.
4. The original variable environment.
5. The original function environment.

The inclusion of the last two is problematic in terms of efficiency. The growth of the environment using this scheme is exponential in the number of functions in scope. Clearly, only the unbound (or free) functions and variables used in e need to be stored these components, so there is much room for improvement. However, this model suffices for a semantic description. A much more efficient (but slightly less descriptively clear) scheme would be used in a real implementation.

Of the elements contained in the closure, all but var' is available in its required form in the rule's inputs. The value of var' is created by using the rule set: *Extract_bvar*

Extract_bvar takes all of the variable names out of a pattern and places them into a variable environment with addresses corresponding to their position within the pattern.

The rules for *Extract_bvar* are defined in figure 21. *Extract_bvar* contains two judgements. The first judgement (instantiated by rules 39 and 40) handles the top level of a pattern. The rules in the second judgement (41 and 42) are used to traverse the elements of a tuple pattern.

Patterns can take two forms, identifiers and tuples. Rule 39 handles identifiers. An identifier is translated into the variable environment containing the converted identifier and the id function.

Set *Prefix* **is** $\boxed{\vdash EXP\ VAR \to VAR}$

Non-empty environment

$$\frac{\vdash {}^{n}\pi'_{n}\ var^{+} \to var'}{\vdash {}^{n}\pi'_{n}\ (\mathsf{vb}(v, exp) : var^{+}) \to (\mathsf{vb}(v, exp \cdot {}^{n}\pi'_{n}) : var')} \tag{43}$$

Empty environment

$$\vdash {}^{n}\pi'_{n}\ [\,]_{VAR} \to [\,]_{VAR} \tag{44}$$

End *Prefix*

Figure 22: Prefixing a tuple-access function to all components of a variable environment.

The handler for tuples is much more complex. Rule 40 takes the tuple and finds its length using the *Length* rule set (not defined here). The length will be important later on for determining which family of tuple access functions to use to access elements of a tuple. As an example a pattern: $(pat_1, pat_2, pat_3, pat_4\ )$ will be accessed by the functions: ${}^4\pi_1$, ${}^4\pi_2$, ${}^4\pi 3$, ${}^4\pi_4$ while the pattern: (pat_1, pat_2, pat_3) will need the functions: ${}^3\pi_1$, ${}^3\pi_2$, ${}^3\pi_3$.

In all cases, the length of the pattern tuple determines the value of the left superscript of every operation used to access an element of that pattern. The value in the right subscript of the access function depends solely on the relative position of the element within the tuple.

The rules of the second judgement are responsible for the generation of this subscript and the insertion of the access functions into the environment.

Rule 41 takes two integer values: n and n'. The first one is the value to be used in the left superscript, it is equal to the length of the original tuple. The second value: n' is the value of the right subscript. Its value increases as we traverse the tuple.

The first premise is a recursive call to the first judgement. It takes the first pattern in the tuple and converts it into an environment. Since patterns can be nested to arbitrary depths the environment var can have more than one entry. The next premise refers var to the rule set *Prefix* to prefix the addresses of var with ${}^n\pi'_n$, the tuple-access operation generated from the values n and n'. The rules for *Prefix* are shown in figure 22. Rule 43 caters for a non-empty variable environment. The address component of the first binding of the environment is composed with the ${}^n\pi'_n$. Since this operation is placed on the right hand side of the composition and since data flows through composed functions from right to left the operation ${}^n\pi'_n$ could be considered as a prefix to the other part of the address expression. The premise of this rule uses the same judgement to process the remainder of the list.

Rule 44 handles the terminal case where the variable environment is empty. This rule will never be executed first since all patterns must generate an environment with at least one component [21].

[21] If I were sure that the rules in *Extract_bvar* were the only ones to call prefix then a more correct version of rule 44 would cater for singleton lists. This would cause the translation to fail when *Prefix* is presented with an empty list (since there would be no matching case). However there is a chance in future that *Prefix* might be called from other rule sets. In any case the acceptance of empty environments is not an error because the rules of *Extract_bvar* prevent the creation of such environments.

Referring back to rule 41 we consider the next premise:

$$n\,(n'+1) \vdash pat^* \mapsto var''$$

This rule translates the rest of the pattern tuple. The value for the right hand subscript is incremented so the appropriate address will be generated for the next pattern. The last premise is a call to the rule set: *Concat_env* which takes two variable environments and joins them (the order of entries in the environments is maintained). The rules for *Concat_env* aren't shown here.

Rule 42 simply handles the case of an empty tuple. This will never be the first rule invoked in *Extract_bvar* because the syntax of the source language rejects tuples with an arity of less than two.

The set should produce a correct (and ordered) sequence of sub-addresses on an arbitrary pattern. As an example consider the pattern:

```
( a,(b,c,d),e,(f,g))
```

Processing the individual components (through the first premise of rule 41) will yield the following four variable environments:

$$[\mathsf{vb}(a,\mathsf{id})]\ [\mathsf{vb}(b,\mathsf{id}\cdot{}^3\pi_1),\mathsf{vb}(c,\mathsf{id}\cdot{}^3\pi_2),\mathsf{vb}(d,\mathsf{id}\cdot{}^3\pi_3)]\ [\mathsf{vb}(e,\mathsf{id})]\ [\mathsf{vb}(f,\mathsf{id}\cdot\pi_1),\mathsf{vb}(g,\mathsf{id}\cdot\pi_2)]$$

Where the appearance of all the id functions is an artifact of the rule for handling patterns consisting of a single identifier. When impact of prefix is taken into account the environments become:

$$\begin{array}{l}
[\mathsf{vb}(a,\mathsf{id}\cdot{}^4\pi_1)]\\
[\mathsf{vb}(b,\mathsf{id}\cdot{}^3\pi_1\cdot{}^4\pi_2),\mathsf{vb}(c,\mathsf{id}\cdot{}^3\pi_2\cdot{}^4\pi_2),\mathsf{vb}(d,\mathsf{id}\cdot{}^3\pi_3\cdot{}^4\pi_2)]\\
[\mathsf{vb}(e,\mathsf{id}\cdot{}^4\pi_3)]\\
[\mathsf{vb}(f,\mathsf{id}\cdot\pi_1\cdot{}^4\pi_4),\mathsf{vb}(g,\mathsf{id}\cdot\pi_2\cdot{}^4\pi_4)]
\end{array}$$

Which are then concatenated to form the final environment.

5 Examples of translation

In this section we examine three examples of how the translator maps Adl source expressions into BMF target code. Each example is drawn from a slightly different area.

This section has been written so that each example is relatively self-contained (though they refer to each other when it is of interest). The examples do not describe every detail of the translation (to do so would have doubled the length of this section). Instead, the explicit descriptions of how the rules are applied descends only to the level where the expressions invoke relatively complex rules. Below this level the results of the translation are given without the accompanying explanation.

Several conventions are adopted to abbreviate the descriptions of the rule applications. One common one is the use of ellipsis (...) to fill in code sections that are unimportant or obvious. Another is the use of names for source expressions. For example the body of the source program in the first example: `ma` can be referred to as e_{ma}. Where the subscripted term is the standard variable used to range over an expression type in the Adl's abstract syntax from section 4.3. Thus a term like e_x could be used to refer to an expression: `x` while dec_x would be used to refer to a declaration: `x`.

```
ma(a: vof vof real, b:vof vof real) :=
   let
      f x :=
         let
            g y  := a!x!y + b!x!y
         in
            map (g,iota (# a!x))
         endlet
   in
      map (f,iota (# a))
   endlet
```

Figure 23: Adl code for matrix add

Lastly, in most cases, all of the environment except for the variable environment is left implicit. This usually does not affect the clarity of the discourse. The variable environment is stylised to reflect the structure of the tuple that would be passed around at that point of the program. So an environment may be written ss the nested tuple $((a, b), x), y)$ rather than as a straight list of variable bindings: $[\mathsf{vb}(y, \mathsf{id}), \ldots]$ as would normally happen.

5.1 Numerical Applications: Matrix Add

Of all the bulk data operations encountered in numerical applications, matrix addition would be one of the simplest. In spite of this simplicity this operation is still interesting from the point of view of implementation, especially on distributed parallel architectures. It is one of the more basic operations worth distributing across the machine. As such it is an example worth studying in this context, i.e. the translation of a language targeted mainly at distributed memory architectures.

A possible Adl implementation of matrix add is given in figure 23. The matrices are represented by nested vectors. The program itself consists of two nested maps. The outer map handles each row. The inner map handles each element of each row. Each scalar addition is performed within the function: `g`. The program will only work if the dimensions of `a` are identical to `b`. If `b` is smaller indices into `b` will run out of range at run-time. If `a` is smaller results will be incorrect. There is no way through Adl's type systems of guaranteeing that `a` and `b` are rectangular.

All of these shortcomings are preserved in the translation i.e. the translator does not make explicit any of the implicit assumptions in the Adl code.

Now we consider the translation process. The rule set: *Translate* has only one rule to match entire programs. The application of this rule to `ma` is given in figure 24 part (I). The top level rule is designed to cater for programs where the main declaration (in this case: `ma`) may be preceeded by auxiliary function and variable declarations. None of the rules of the translator can handle such and agglomeration of functions so it is the task of rule 1 to cast the program components into the more familiar surrounds of a let expression. In this example this processing is of little consequence since there are no auxiliary definitions, nonetheless we consider it here.

(I) Rule 1: Programs(exp_{ma})

$$\frac{\begin{array}{c} \mathit{Remove\_name\_clashes}(\ldots) \\ \mathit{Extract\_bvar}(\texttt{(a,b)} \rightarrow (a,b)) \qquad \mathit{Length\_var}(\ldots) \\ (a,b) \vdash \texttt{let}\,[\,]\,\texttt{in}\, e_{ma}\, \texttt{endlet} \Rightarrow exp_{ma} \end{array}}{\vdash \mathrm{pr}([\,],\texttt{ma (a:vof vof real, b: vof vof real) :=}\ e_{ma}) \leadsto exp_{ma}}$$

(II) Rule 7: Trivial let (exp_{ma})

$$\frac{data \vdash e_{ma} \Rightarrow exp_{ma}}{data \vdash \texttt{let}\,[\,]\,\texttt{in}\, e_{ma}\, \texttt{endlet} \Rightarrow exp_{ma}}$$

Figure 24: Top level calls for finding translation of whole program: exp_{ma}

The first premise is a call to the rule set: *Remove_name_clashes* (not defined in this report). This performs some necessary renaming in case some of the auxiliary definitions' names clash with `ma`'s formal parameters. In the absence of auxiliary definitions it does nothing. *Extract_bvar* takes the parameter list and injects it into an empty environment. Normally the environment consists of a variable part, a function part and a natural offset value. In this example we will only show the first part, the others can be left invisible without obscuring the workings of the example.

Length_var is used to generate the natural offset value which is normally used to help generate addresses to replace variable names. Variable name generation will not be considered in these examples so the offset is left implicit.

The last premise takes the body of the main declaration of the program: e_{ma} (the let expression making up the right hand side of `ma`) and injects it into a synthesised let expression. This expression is then immediately evaluated using the normal rules of the translator.

Figure 24 part (II) shows the first step of this evaluation. The new let expression is applied to rule 7 which exists as means of processing `let`'s with an empty declaration section. The premise of the rule takes the body of the expression: e_{ma} and evaluates it in the original environment. The result of this evaluation is the same as the result for the whole rule. This rule effectively states that the translation of the let expression is the same as the translation of its body.

The translator now has to apply the rule matching the outermost part of e_{ma}. This part is another let expression. This time it has a non-empty declaration part, allowing a different rule to be applied. Figure 25 shows the relevant rule being applied.

There is only a single function declaration in e_{ma}'s declaration part. This simplifies matters greatly. A function declaration alone does not cause the translator to create any code and it does not alter the variable environment. As a result we see the first premise leaves the visible part of the environment unchanged and returns only an `id` function as a code result. The second premise takes the body of the let expression and evaluates it under the new environment (which now contains a closure for `f`). The result of this evaluation: exp_{ma_body} is composed with the `id` from the first

(I) Rule 8: Let expressions($exp_{ma\_body} \cdot exp_{ma\_decls}$)

$$\frac{\begin{array}{c}(a,b) \vdash \texttt{f x := ...} \rightarrow (a,b)\ \mathsf{id} \\ (a,b) \vdash e_{ma\_body} \Rightarrow exp_{ma\_body}\end{array}}{(a,b) \vdash \texttt{let f x := ... in } e_{ma\_body}\ \texttt{endlet} \Rightarrow exp_{ma\_body} \cdot \mathsf{id}}$$

Figure 25: Evaluating body of `ma` to: exp_{ma}

(I) Rule 20: Map (exp_{ma_body})

$$\frac{(a,b) \vdash \texttt{f iota (\# a)} \Rightarrow exp_f \cdot exp_{f\_args}}{(a,b) \vdash \texttt{map (f, iota (\# a))} \Rightarrow (exp_f) * \cdot \mathsf{distl} \cdot exp_{f\_args}}$$

(II) **Rule 12: Function application** ($exp_f \cdot exp_{f\_args}$)

$$\frac{\begin{array}{cc}(a,b) \vdash \texttt{iota (\# a)} \ \Rightarrow \mathsf{iota} \cdot \# \cdot \pi_1 & \mathit{Lookup\_funenv}(\ldots \texttt{f} \rightarrow \ldots e_f \ldots) \\ \mathit{Build\_new\_env}(\ldots \mapsto ((a,b),x)) & ((a,b),x) \vdash e_f \Rightarrow exp_f\end{array}}{(a,b) \vdash \texttt{f iota (\# a)} \ \Rightarrow exp_f \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^{\circ}}$$

Figure 26: Translating `map (f, iota (# a))` into exp_{ma_body}

premise to form the result. We can see that:

$$\begin{aligned} exp_{ma} &= exp_{ma\_body} \cdot exp_{ma\_decls} \\ &= exp_{ma\_body} \cdot \mathsf{id} \\ &= exp_{ma\_body} \end{aligned}$$

We should now examine how the value for exp_{ma_body} is derived.

exp_{ma_body} is translated from the expression e_{ma_body}. The appropriate rule for this expression is the map rule. The application of this rule to e_{ma_body} is shown in figure 26 part (I). The structure of the rule is quite simple. The first premise translates the application of `f` to its arguments to yield the code for the function body: exp_f and the code for the arguments exp_{f_args}. This application is actually badly typed. You cannot apply a scalar function: `f` to a vector expression: `iota (# a)`. The reason the rule gets away with this is that we are translating expressions rather than evaluating them. The reasons for adopting this strange way of handling map expressions are outlined in section 4.5.8.

Once the the values of exp_f and exp_{f_args} have been produced by the premise the conclusion injects them into the standard target code context for a map expression.

The methods for obtaining the values of exp_f and exp_{f_args} are shown in figure 26 part (II). The first premise of this rule translates the expression: `iota (# a)`, producing the expression:

$$exp_{f\_args} = \mathsf{iota} \cdot \# \cdot \pi_1$$

Rule 8: Let expressions (exp_f)

$$\frac{\begin{array}{c}((a,b),x) \vdash \texttt{g y := ...} \rightarrow ((a,b),x)\,\mathsf{id} \\ ((a,b),x) \vdash e_{f\_body} \Rightarrow exp_{f\_body}\end{array}}{((a,b),x) \vdash \texttt{let g y :=... in}\, e_{f\_body}\, \texttt{endlet} \Rightarrow exp_{f\_body} \cdot \mathsf{id}}$$

Figure 27: Translating e_f to exp_f

Rule 8: Map expressions (exp_{f_body})

$$\frac{((a,b),x) \vdash \texttt{g iota (\# a!x)} \Rightarrow exp_g \cdot exp_{g\_args}}{((a,b),x) \vdash \texttt{map (g,iota (\# a!x))} \Rightarrow (exp_g) * \cdot \mathsf{distl} \cdot (\mathsf{id}, exp_{g\_args})^{\circ}}$$

(II) Function application ($exp_g \cdot exp_{g\_args}$)

$$\frac{\begin{array}{cc}((a,b),x) \vdash \texttt{iota (\# a!x)} \Rightarrow \mathsf{iota} \cdot \#\cdot ! \cdot (\pi_1^2, \pi_2)^{\circ} & \mathit{Lookup\_funenv}(\ldots \mathsf{g} \rightarrow \ldots e_g \ldots) \\ \mathit{Build\_new\_env}(\ldots \mapsto (((a,b),x),y)) & (((a,b),x),y) \vdash e_g \Rightarrow exp_g\end{array}}{((a,b),x) \vdash \texttt{g iota (\# a!x)} \Rightarrow exp_g \cdot \mathsf{iota} \cdot \#\cdot ! \cdot (\pi_1^2, \pi_2)^{\circ}}$$

Figure 28: Translating the map expression: e_{f_body} to exp_{f_body}

The second premise is a call to the rule set: *Lookup_funenv* which searches the function environment for the function binding of `f` to its closure. Part of this closure is the Adl code forming the body of `f`, called: e_f.

The third premise takes the the values in the current environment and the appropriate parts of `f`'s closure (parts which are not shown here) and builds up the new variable environment. In this case the name of the functions formal parameter is added. The last premise evaluates e_f in the context of this new environment. The conclusion composes this value with the code for exp_{f_args}.

The body of `f` is another let expression. Figure 27 shows the derivation of exp_f from this body. As with the expression contained within `ma` the declaration part of `f` has only a single function declaration. The first premise produces an unchanged variable environment (though the function environment has, of course, been updated) and a code section of `id`.

The second premise evaluates the let expressions body: e_{f_body} to yield: exp_{f_body}. This target code is composed with the target code for the first premise in the rules conclusion. It can be noted that:

$$\begin{aligned} exp_f &= exp_{f\_body} \cdot exp_{f\_decls} \\ &= exp_{f\_body} \cdot \mathsf{id} \\ &= exp_{f\_body} \end{aligned}$$

To derive a value for exp_{f_body} the translator must apply the e_{f_body} to the map rule. This application is shown in figure 28 part (I). The premise translates the function: `g` applied to its arguments into $exp_g \cdot exp_{g\_args}$. The conclusion injects these results into a target-code template for map.

Figure 28 part (II) shows the function application invoked in the premise of the map rule. The first premise derives the value of exp_{g_args}. Due to the simplicity of its derivation the result is given in-situ. *Lookup_funenv* uses the function environment to derive the closure for `g`. *Build_new_env* creates a new environment from this closure and the current environment. The new variable environment has an entry for `g`'s formal parameter: y. The last premise evaluates the expression from `g`'s closure:e_g into the target code: exp_g. Finally the code from the function's body and its arguments are composed together in the conclusion.

exp_g is the scalar expression that is applied pointwise to corresponding entries of `a` and `b`. Its production is quite straightforward but detailed. We won't cover the rules used to create it here. The value the translator arrives at for exp_g is:

$$+ \cdot (! \cdot (! \cdot (\pi_1^3, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ, ! \cdot (! \cdot (\pi_2 \pi_1^2, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ)^\circ$$

This is the last unknown part of the translation of `ma` into exp_{ma} now all that remains is to embed the lower level expressions into the appropriate places at the higher level. So we have:

$$\begin{aligned} exp_g \cdot exp_{g\_args} \quad &= + \cdot (! \cdot (! \cdot (\pi_1^3, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ, ! \cdot (! \cdot (\pi_2 \pi_1^2, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ)^\circ \\ &\quad \mathsf{iota} \cdot \# \cdot ! \cdot (\pi_1^2, \pi_2)^\circ \end{aligned}$$

Which is embedded in exp_{f_body} the target code for `map (g, iota (# a!x))`:

$$\begin{aligned} exp_{f\_body} &= exp_f \\ &= (+ \cdot (! \cdot (! \cdot (\pi_1^3, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ, ! \cdot (! \cdot (\pi_2 \pi_1^2, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ)^\circ) * \cdot \\ &\quad\ \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot ! \cdot (\pi_1^2, \pi_2)^\circ)^\circ \end{aligned}$$

Which, in turn, is a part of the expression: $exp_f \cdot exp_{f\_args}$:

$$\begin{aligned} exp_f \cdot exp_{f\_args} \quad &= (+ \cdot (! \cdot (! \cdot (\pi_1^3, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ, ! \cdot (! \cdot (\pi_2 \pi_1^2, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ)^\circ) * \cdot \\ &\quad\ \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot ! \cdot (\pi_1^2, \pi_2)^\circ)^\circ \cdot \\ (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ & \end{aligned}$$

The final result is this expression embedded in a map template to form exp_{ma_body}:

$$\begin{aligned} exp_{ma} &= exp_{ma\_body} \\ &= ((+ \cdot (! \cdot (! \cdot (\pi_1^3, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ, ! \cdot (! \cdot (\pi_2 \pi_1^2, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ)^\circ) * \cdot \\ &\quad\ \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot ! \cdot (\pi_1^2, \pi_2)^\circ)^\circ) * \cdot \\ &\quad\ \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ \end{aligned}$$

This solution varies somewhat from the much more parsimonious code shown at the end of section 3.4.1. The difference lies in the way the aggregate structures are referenced. The earlier solution performs bulk data movement using nested zip with pair: Υ_{pair} operations. These bring the data together in the appropriate way. In the solution given here the operands are brought together by copying array values using distl and using indexing operations into these copies. The second version is (at least potentially) much less efficient than the first. Methods have been formulated to convert the second program through algebraic manipulation into the first. These optimisation methods will be the subject of a later report.

5.2 Statistics: Standard Deviation

In this section we outline the translation of an Adl program for calculating the standard deviation of a vector of numbers. The formula for standard deviation is shown in section 3.5. A possible rendering of this formula in Adl is shown in figure 29.

```
sd(data:vof real) :=
   let
      plus(x,y) := x + y;
      mean_data := reducep(plus,data)/ (# data);
      sub_mean_sqr x := (x - mean_data)^2;
      mean_var := reducep(plus,map(sub_mean_sqr, data)
   in
      (1/((# data)-1)* mean_var)^0.5
   endlet
```

Figure 29: Adl code for deriving the standard deviation of a vector of values

(I) Rule 1: Programs(exp_{sd})

$$\frac{\begin{array}{c} \textit{Remove\_name\_clashes}(\ldots) \\ \textit{Extract\_bvar}(\texttt{data} \rightarrow \textit{data}) \qquad \textit{Length\_var}(\ldots) \\ \textit{data} \vdash \texttt{let}\ [\,]\ \texttt{in}\ e_{sd}\ \texttt{endlet} \Rightarrow exp_{sd} \end{array}}{\vdash \mathrm{pr}([\,], \texttt{sd (data:vof real) :=}\ e_{sd}) \leadsto exp_{sd}}$$

(II) Rule 7: Trivial let (exp_{sd})

$$\frac{\textit{data} \vdash e_{sd} \Rightarrow exp_{sd}}{\textit{data} \vdash \texttt{let}\ [\,]\ \texttt{in}\ e_{sd}\ \texttt{endlet} \Rightarrow exp_{sd}}$$

Figure 30: Top level calls for finding translation of whole program: exp_{sd}

Most of the computation in `sd` takes place in the calls to `map` and `reduce` inside the declaration part. The declarations of `mean_data` and `mean_var` in particular are pivotal. They will produce the most substantial proportion of the target code.

Now we examine the translation from this source to its target code. As always the translator invokes rule 1 as the entry point. This rule is responsible for wrapping up any auxiliary [22] declarations and the main function into a contrived let expression. Since there are no auxiliary declarations in this program the *decls* part of the let expression must remain empty. The application of rule 1 to `sd` is shown in figure 30 part (I).

The first three premises attempt to construct the appropriate environment in which to translate the body of the main function: e_{sd}. Their workings have been greatly abbreviated (in this case they don't do that much anyway). Most important is *Extract_bvar* which takes the main function's formal parameter list and creates a variable environment of names from it. The fourth premise injects the main function's body: e_{sd} into a let expression and evaluates it under this new variable environment. The declarations preceeding the main function are injected into the declaration part of this `let` but in this case the declaration list is empty.

[22] Auxiliary meaning any declaration other than the main one which is `sd` in this example.

(I) Rule 8: Let expressions$(exp_{sd\_body} \cdot exp_{sd\_decls})$

$$\frac{\begin{array}{c} data \vdash dec^{+}_{sd\_decls} \rightarrow \rho_{sd\_decls}\ exp_{sd\_decls} \\ \rho_{sd\_decls} \vdash e_{sd\_body} \Rightarrow exp_{sd\_body} \end{array}}{data \vdash \texttt{let}\ dec^{+}_{sd\_decls}\ \texttt{in}\ e_{sd\_body}\ \texttt{endlet} \Rightarrow exp_{sd\_body} \cdot exp_{sd\_decls}}$$

Figure 31: Evaluating body of `sd` (a let expression): exp_{sd}

Now the translator must cope with the new `let`. There are two rules to handle let expressions. One for `let`'s containing non-empty declaration sequences and one for handling empty ones.

Part (II) of figure 38 shows how the rule for handling `let`'s with empty declaration lists is used to translate e_{sd}. Not surprisingly this rule doesn't do much. A `let` with an empty declaration sequence has exactly the same semantics as the code contained in the let body in isolation. This rule reflects this fact by projecting the body of the `let` (in this case e_{sd}) out of the expression and evaluating it in isolation to produce the rule's result.

e_{sd} is itself a let expression. Unlike the synthesized `let` just considered, it has a non-empty declaration list. Rule 8 is used to translate expressions of this type. Figure 31 shows how e_{sd} (now more fully transcribed for convenience) is handled. The declaration sequence (referred to here as of $dec^{+}_{sd\_decls}$) has to be processed before the body of the `let`: e_{sd_body} can be translated. The translator produces both code and an updated environment whilst processing a declaration sequence. Clearly a different judgement is required in order to return two values. The first premise makes a call to such a judgement (differentiated by the use of the $\rightarrow$ symbol). The translation of $dec^{+}_{sd\_decls}$ (remembering that a $+$ or $*$ superscript is used to indicate a list of objects rather than a single instance of an object) by this judgement, will be examined in detail after we consider the other parts of this rule.

The second premise takes the modified part of the environment (denoted here by ρ_{sd_decls}) and uses it as the context for the evaluation of the body of the let expression: e_{sd_body}. As a last step the code solutions for the first and second premises are composed to form the eventual answer:

$$exp_{sd} = exp_{sd\_body} \cdot exp_{sd\_decls}$$

For the program `sd` the segment of code represented by exp_{sd_decls} is much larger than the code for exp_{sd_body}. The derivation of this first expression and the corresponding environment shall be discussed next.

5.2.1 Translating the declaration sequence

There are three rules in the judgement for declaration sequences. Rule 10 handles a sequence that starts with a variable declaration. Rule 11 caters for sequences starting with a function declaration. Both rules work by stripping off the first declaration, processing it, and then making a recursive call to the judgement. By this process the declaration sequence shrinks by one declaration at a time until it is empty. Rule 9 is used to handle the empty sequence.

(I) Rule 11:Function Declarations$(\rho_{sd\_decls}\ exp_{sd\_decls})$

$$\frac{\begin{array}{c} Handle\_fdec(data\ [\,] \vdash \texttt{plus (x,y):=}\ldots \mapsto data\ [plus]) \\ data\ [plus] \vdash dec^*_{mean\_data} \rightarrow \rho_{sd\_decls}\ exp_{md\_decls} \end{array}}{data\ [\,] \vdash [\texttt{plus (x,y):=}\ldots, dec^*_{mean\_data}] \rightarrow \rho_{sd\_decls}\ exp_{md\_decls}}$$

(II) Rule 10: Variable declarations$(\ \rho_{sd\_decls}\ exp_{md\_decls})$

$$\frac{\begin{array}{c} Handle\_vdec(data\ [plus] \vdash \texttt{mean\_data :=}\ldots \mapsto (data, mean\_data)\ [plus]\ \ exp_{mean\_data}) \\ (data, mean\_data)\ [plus] \vdash dec^*_{sub\_mean\_sqr} \rightarrow \rho_{sd\_decls}\ exp_{sms\_decls} \end{array}}{data\ [plus] \vdash [\texttt{mean\_data := }\ \ldots, dec^*_{sub\_mean\_sqr}] \rightarrow \rho_{sd\_decls}\ exp_{sms\_decls} \cdot exp_{mean\_data}}$$

Figure 32: Calls for handling first two declarations in `sd`

The rule for handling function declarations will not be of great direct interest. Function declarations only cause the function environment to be updated. Variable declarations produce both target code and changes to the variable environment. Since both of these entities are explicit in the examples most discussion will centre on variable declarations.

The first declaration in the list is for the function `plus`. This will cause rule 11 to be applied and the function environment updated. The second declaration, `mean_data`, triggers rule 10. The application of both rules is shown in figure 32.

Part (I) outlines the evaluation of `plus`. The first premise calls the rule set *Handle_fdec* which takes the function body and creates an entry in the function environment. In order to make these changes visible we must (at least temporarily) show the function environment. Instead of showing the full binding of a function name to its closure we can adopt the simple convention of just listing the names. Thus with this first premise takes the empty function environment and augments it with an entry for *plus*.

The second premise takes the remainder of the declaration sequence and evaluates it under the environment (*data* [plus]). The two results are the environment created by the whole of the declaration sequence (ρ_{sd_decls}) and the expression created by the remainder of the sequence (exp_{md_decls}). Finally the code and the new environment produced by the second premise are combined in the result: *data* [*plus*] and $exp_{md\_decls} = exp_{sd\_decls}$.

Part (II) shows the application of rule 10 to $dec^*_{mean\_data}$ under the environment: *data* [*plus*] to create ρ_{sd_decls} and exp_{md_decls}. The first premise handles the individual declaration of `mean_data :=`... by issuing a call to *Handle_vdec*. This rule set translates the right hand side of the declaration into exp_{mean_data} and appends the name *mean_data* to the variable environment.

To see how the right hand side of `mean_data` is evaluated we need to look inside the *Handle_vdec* rule set. *Handle_vdec* consists of only one rule (rule 37). This rule's application to `mean_data` is shown in figure 33.

The first premise is a call to the *Convert* rule set. *Convert* takes a name from the abstract syntax of Adl and converts it to a name in BMF. In a real implementation this step may involve little or

(I) Rule 37:Variable Declaration$(\rho_{mean\_data}\ exp_{mean\_data})$

$$\frac{\begin{array}{c} Convert(\texttt{mean\_data} \succ mean\_data) \\ Translate(data\ [plus] \vdash \texttt{reducep (plus,data)/ (\# data)} \Rightarrow exp_{mean\_data}) \end{array}}{\begin{array}{ll} data\ [plus] \vdash & \texttt{reducep (plus,data)/ (\# data)} \mapsto \\ & (data, mean\_data)\ [plus]\ \ (\mathsf{id}, exp_{mean\_data\_b})^{\circ} \end{array}}$$

Figure 33: Rule in *Handle_vdec* used to translate `mean_data`

(I) Rule 14:Binary operators $(exp_{mean\_data\_b})$

$$\frac{\begin{array}{c} data\ [plus] \vdash \texttt{reducep (plus,data)} \ \Rightarrow exp_{reducep} \\ data\ [plus] \vdash \texttt{(\# data)} \Rightarrow \# \\ Convert\_op(\texttt{/} \rightarrow \div) \end{array}}{\begin{array}{ll} data\ [plus] \vdash & \texttt{reducep (plus,data)/ (\# data)} \Rightarrow \\ & \div \cdot (exp_{reducep}, \#)^{\circ} \end{array}}$$

Figure 34: Rule in *Handle_vdec* used to translate `mean_data`

no processing.

The second premise calls *Translate* (the starting point of the entire translator) to evaluate the body of `mean_data` into $exp_{mean\_data\_b}$.

Finally the conclusion of the rule updates the variable environment with the name: *mean_data* and injects $exp_{mean\_data\_b}$ into the expression $(\mathsf{id}, exp_{mean\_data\_b})^{\circ}$.

The outer-expression in the body of `mean_data` is the real-division operator. Rule 14 handles this, and all other binary operators. Figure 34 shows this first step.

The first two premises are used to evaluate arguments. The description of the derivation of $exp_{reducep}$ is fairly involved so we defer this until later. The right hand side is trivial so the target code for this # has been inlined. In most cases # would be composed with an addressing function derived from variable name: `data`. The target code for `data` is `id` so it has been omitted.

The last premise converts the Adl operator (`/`) into its BMF equivalent: $\div$. Finally, the conclusion composes the pair of expressions translated from the operands with the divide operator to form the result (i.e. exp_{mean_data}).

Now we examine the translation of `reducep (plus,data)` into the BMF expression $exp_{reducep}$. The rule of interest here is rule 4.5.9 (undirected reduction for non-empty vectors). The application of this rule is shown in figure 35. In spite of the seeming complexity of the conclusion the workings of the rule are quite straightforward. The only premise produces the code for the binary operator applied to the input argument. This would seem inappropriate. `data` is a vector and `+` is an operator to handle two scalars. The translator gets away with this mismatch because the rule for

(I) Rule 4.5.9:Reduction, non-empty vectors $(exp_{reducep})$

$$\frac{data\ [plus] \vdash \texttt{plus data} \Rightarrow + \cdot \mathsf{id}}{data\ [plus] \vdash \ \texttt{reducep (plus,data)} \Rightarrow \pi_2 \cdot ((\pi_1^2, (+ \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/ \cdot \mathsf{id}}$$

Figure 35: Translation of `reducep(plus,data)`

function application does not allow the translation of the operator to be affected by the type or contents of its operand. The rationale for using this trick is given in the description of the rule for map in section 4.5.8. The rule's conclusion injects the results of this application into a standard (but detailed code segment). An explanation of this code is given in section 4.5.9.

With the code for the reduce part of the expression in place we now can now produce the code for exp_{mean_data} the expression is:

$$\begin{aligned} exp_{mean\_data} &= (\mathsf{id}, exp_{mean\_data\_b})^\circ \\ &= (\mathsf{id}, \div \cdot (exp_{reducep}, \#)^\circ)^\circ \\ &= (\mathsf{id}, \div \cdot (\pi_2 \cdot ((\pi_1^2, (+ \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/ \cdot \mathsf{id}, \#)^\circ)^\circ \end{aligned}$$

Which is the code formed by the declaration of `mean_data` in the first premise in the rule application in figure 32 part (II).

The second premise of this rule is a recursive call to process the remainder of the declaration sequence: $dec^*_{sub\_mean\_sqr}$. This code consists of two declarations the first: `sub_mean_sqr` is a function that subtracts `mean_data` from its argument and squares the result. The second: `mean_var`, is a value formed by mapping `sub_mean_sqr` over `data` and summing the result.

These two declarations follow a similar pattern to the first two. Since they don't contain any new features, details of their translation will be ignored. Only their final results are given here.

The expression part of the result is:

$$\begin{aligned} exp_{sms\_decls} = \ & (\mathsf{id}, \pi_2 \cdot ((\pi_1^2, (+ \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/ \cdot \\ & (\,\hat{}\, \cdot (- \cdot (\pi_2, \pi_2 \cdot \pi_1)^\circ, 2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \pi_1) \end{aligned}$$

The final environment, including the function names, is:

$$\rho_{sd\_decls} = ((data, mean\_data), mean\_var)\ [plus, sub\_mean\_sqr]$$

The BMF expression yielded from the entirety of the declaration part: $dec^+_{sd\_decls}$ is:

$$\begin{aligned} exp_{sd\_decls} &= exp_{sms\_decls} \cdot exp_{mean\_data} \\ &= (\mathsf{id}, \pi_2 \cdot ((\pi_1^2, (+ \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/ \cdot \\ &\quad (\,\hat{}\, \cdot (- \cdot (\pi_2, \pi_2 \cdot \pi_1)^\circ, 2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \pi_1) \cdot \\ &\quad (\mathsf{id}, \div \cdot (\pi_2 \cdot ((\pi_1^2, (+ \cdot (\pi_1^2, (\pi_2 \cdot \pi_1, \pi_2^2)^\circ)^\circ))^\circ)/ \cdot \mathsf{id}, \#)^\circ)^\circ \end{aligned}$$

5.2.2 Translating the `let` body

The above code forms the first part of the final translation. The second part is the translation of the let expression body:

$$e_{sd\_body} = (\texttt{1}/((\#\ \texttt{data})\ -\texttt{1}) *\ \texttt{mean\_var})\hat{}\ \texttt{0.5})$$

Which is a series of applications of simple binary and unary operators. The direct translation of this term is:

$$exp_{sd\_body} = \hat{}\cdot(\times\cdot(\div\cdot(1, -\cdot(\#\cdot\pi_1^2, 1)^\circ)^\circ, \pi_2)^\circ, 0.5)^\circ$$

When this is composed with the code for the declaration section we obtain the code for the entire program:

$$\begin{aligned} exp_{sd} &= exp_{sd\_body}\cdot exp_{sd\_decls} \\ &= \hat{}\cdot(\times\cdot(\div\cdot(1, -\cdot(\#\cdot\pi_1^2, 1)^\circ)^\circ, \pi_2)^\circ, 0.5)^\circ\cdot \\ &\quad (\mathsf{id}, \pi_2\cdot((\pi_1^2, (+\cdot(\pi_1^2, (\pi_2\cdot\pi_1, \pi_2^2)^\circ)^\circ))^\circ)/\cdot \\ &\quad (\hat{}\cdot(-\cdot(\pi_2, \pi_2\cdot\pi_1)^\circ, 2)^\circ) * \cdot\mathsf{distl}\cdot(\mathsf{id}, \pi_1)\cdot \\ &\quad (\mathsf{id}, \div\cdot(\pi_2\cdot((\pi_1^2, (+\cdot(\pi_1^2, (\pi_2\cdot\pi_1, \pi_2^2)^\circ)^\circ))^\circ)/\cdot\mathsf{id}, \#)^\circ)^\circ \end{aligned}$$

This code bears little superficial resemblance to the hand-produced code for standard deviation at the end of section3.5. The main points of divergence are in the use of the sectioned operator ($\hat{}2$) instead of its prefixed version and the much briefer rendering of the reduce operators in the original version. The earlier version also makes use of the *sqrt* operator. This is replaced with the power operator with an exponent of 0.5.

The programs can be made to converge through a series of algebraic manipulations to exp_{sd}. These changes will form part of the optimisation process in a stage of the compiler. The exact nature of this stage will be the subject of a later report.

5.3 Image Processing: The Sobel Operator

Most image processing applications apply a series of transformations to very large two dimensional grids of pixels. Scope for parallelism arises when a transform can be decomposed into a series of independent operations on different sections of the grid.

Edge detection algorithms tend to fall into this category. In most cases, at least one part of the edge detection routine will involve the independent application of an operator to each pixel. The example we use here is the Sobel operator.

The Sobel operator (one of several described in [19]) is one of a number of possible techniques for finding the intensity gradient at a pixel. This operator works by means of two templates covering the area surrounding each pixel. The templates and the method for calculating the value for ∇ is shown in figure 36 The first template (shown in part (a)) yields an estimate for ∇_1, the vertical intensity gradient. It works by laying central box of the template over the pixel at which the local gradient is to be computed. The intensities of the eight adjoining pixels are then multiplied by the factors that overlay them in the template. For template (a), only values with vertical offsets from the centre pixel are factored into the calculation. Once all of the weights have been applied the resulting values are summed to create the value ∇_1.

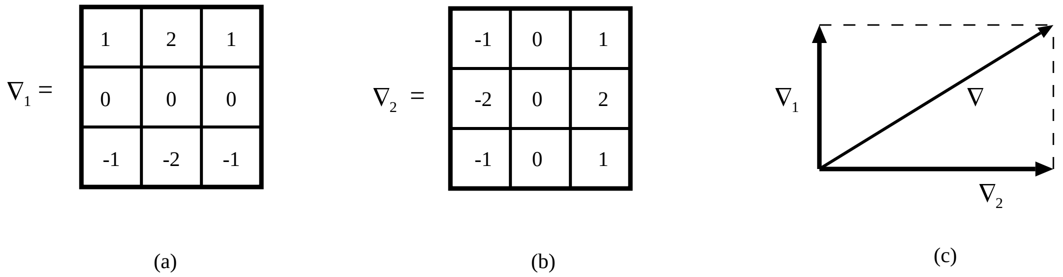

Figure 36: Templates and method for calculating final gradient for the Sobel operator

Part (b) of figure 36 calculates the horizontal gradient, ∇_2 in an analogous manner. Part (c) of figure 36 demonstrates the method of deriving the final value of the pixel gradient: ∇ from the values of ∇_1 and ∇_2. The formula used here is:

$$\nabla = \sqrt{\nabla_1^2 + \nabla_2^2}$$

To produce a complete map of local gradients the Sobel operator must be applied to every pixel in the grid (with the exception of boundary pixels).

This algorithm is easily represented in Adl. A possible solution is shown in figure 37. Two nested mappings apply the sobel operator over the most of the elements of the program's input parameter, `grid`. The outer `map` applies `f` to each row of `grid` (excepting the first and last). Each inner `map` applies `g` to every element (again excepting the first and last) of its row. Every invocation of `g` applies the sobel operator to its element.

The translation of `sobel` is quite straightforward but some code is rather verbose. As with the previous two examples the application of the more trivial rules will be left implicit. Source code will be abridged by ellipsis or represented by e variables.

The Application of the first two rules is given in figure 38. The term: exp_{sobel} is a placeholder for the translated version of the entire program. At this stage of translation its value is undefined.

The term e_{sobel} is merely an abbreviation for the entire contents of the source function: `sobel`. Again due to the lack of any declarations (other than `sobel`) these rules act mainly as bookkeeping operations.

Part (I) embeds the body of `sobel` in a let expression and its parameter, `grid`, in a variable environment. The second premise of this rule then directs the translator to evaluate this artificially created expression.

Part (II) shows the application of the rule which matches this program fragment. It states that to evaluate the let expression with an empty declaration part you only have to evaluate the body. The body in this case is e_{sobel} and the result is still the expression exp_{sobel}.

e_{sobel} is also a let expression. The declaration part of e_{sobel} is non-empty so a different rule of the translator applies. Figure 39 shows this application. For the sake of brevity the evaluation of all of the variable and function declarations is left implicit. The results of this evaluation are shown as the solution to the first premise. An augmented variable environment: (*grid,xrange*) is the first part of this result. The new function environment containing the closure for `plus1` and `f` is one of the things left implicit. The code for generating `xrange` is the second part of the solution.

```
sobel (grid:vof vof real) :=
   let
      plus1 x := x+1;
      xrange  := map (plus1, iota ((# grid) - 2));
      f x     :=
         let
            yrange := map (plus1, iota (# (grid!x)-2));
            g(y)   :=
               let
                  del_1 := grid!(x-1)!(y+1) + 2*(grid!(x-1)!y) +
                           grid!(x-1)!(y-1) - (grid!(x+1)!(y+1) +
                           2*(grid!(x+1)!y) + grid!(x+1)!(y-1));
                  del_2 := grid!(x-1)!(y+1) + 2*(grid!x!(y+1)) +
                           grid!(x+1)!(y+1) - (grid!(x-1)!(y-1) +
                           2*(grid!x!(y-1)) + grid!(x+1)!(y-1))
               in
                   (del_1^2 + del_2^2)^ 0.5
               endlet
         in
            map (g,yrange)
         endlet
   in
      map(f,xrange)
   endlet
```

Figure 37: Adl code for applying the Sobel operator across a pixel grid.

(I) Rule 1: Programs(exp_{sobel})

$$\frac{\begin{array}{c} Remove\_name\_clashes(\ldots) \\ Extract\_bvar(\texttt{grid} \rightarrow grid) \qquad Length\_var(\ldots) \\ grid \vdash \texttt{let}\,[\,]\,\texttt{in}\; e_{sobel}\; \texttt{endlet} \Rightarrow exp_{sobel} \end{array}}{\vdash \mathrm{pr}([\,], \texttt{sobel (grid:vof vof real) :=} e_{sobel}) \rightsquigarrow exp_{sobel}}$$

(II) Rule 7: Trivial let (exp_{sobel})

$$\frac{grid \vdash e_{sobel} \Rightarrow exp_{sobel}}{grid \vdash \texttt{let}\,[\,]\,\texttt{in}\; e_{sobel}\; \texttt{endlet} \Rightarrow exp_{sobel}}$$

Figure 38: Top level calls for finding translation of whole program: exp_{sobel}

Rule 8: Let (exp_{sobel})

$$\frac{\begin{array}{l} grid \vdash \texttt{plus1 x := ...xrange :=...} \rightarrow \\ \quad (grid,xrange)\ (\mathsf{id}, (+ \cdot (\pi_2, 1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ)^\circ \\ \quad\quad (grid,xrange) \vdash e_{sobel\_body} \Rightarrow exp_{sobel\_body} \end{array}}{\begin{array}{l} grid \vdash \texttt{let plus1 x := ...xrange := ...in}\ e_{sobel\_body}\ \texttt{endlet} \Rightarrow \\ \quad\quad exp_{sobel\_body} \cdot (\mathsf{id}, (+ \cdot (\pi_2, 1)^\circ) * \cdot \ldots)^\circ \end{array}}$$

Figure 39: Translating the body of `sobel` to exp_{sobel}

Rule 20: Map (exp_{sobel_body})

$$\frac{(grid,xrange) \vdash \texttt{f xrange} \Rightarrow exp_f \cdot exp_{f\_args}}{(grid,xrange) \vdash \texttt{map (f,xrange)} \Rightarrow (exp_f) * \cdot \mathsf{distl} \cdot exp_{f\_args}}$$

Figure 40: Translating `map (f,xrange)` into exp_{sobel_body}

The second premise evaluates the body of the let expression e_{sobel_body}. The entire translation could be expressed as:

$$exp_{sobel} = exp_{sobel\_body} \cdot (\mathsf{id}, (+ \cdot (\pi_2, 1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ)^\circ$$

where the value of exp_{sobel_body} must be derived from e_{sobel_body} and has yet to be translated.

e_{sobel_body} is a `map` expression. Figure 40 shows the application of this expression to the `map` rule. There is only one premise, a call to the rule for function application to establish the code for `map`'s body: exp_f and the argument exp_{f_args}. The result of this evaluation gives us the value:

$$exp_{sobel\_body} = (exp_f) * \cdot \mathsf{distl} \cdot (\mathsf{id}, exp_{f\_args})^\circ$$

where exp_f and exp_{f_args} are still unknown. They are evaluated in the function application rule shown in figure 41.

The argument part of the application, `xrange`, is just a variable name. exp_{f_args} is simply the translation of this name: `xrange`. The standard variable-lookup rules in the translator produce the

Rule 12: Function application $(exp_f \cdot exp_{f\_args})$

$$\frac{\begin{array}{c} (grid,xrange) \vdash \texttt{xrange} \Rightarrow \pi_2 \qquad Lookup\_funenv(\ldots \texttt{f} \rightarrow \ldots e_f \ldots) \\ Build\_new\_env(\ldots \mapsto ((grid, xrange), x)) \qquad ((grid, xrange), x) \vdash e_f \Rightarrow exp_f \end{array}}{(grid,xrange) \vdash \texttt{f xrange} \Rightarrow exp_f \cdot (\mathsf{id}, \pi_2)^\circ}$$

Figure 41: Translating the function application: `f xrange` to $exp_f \cdot exp_{f\_args}$

Rule 8: Let (exp_f)

$$\frac{\begin{array}{c}((grid, xrange), x) \vdash \texttt{yrange:=}\ldots \rightarrow \\ (((grid, xrange), x), yrange)\ (\mathsf{id}, (+ \cdot \pi_2) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\#\cdot! \cdot (\pi_1^2, \pi_2)^\circ, 2)^\circ)^\circ)^\circ \\ (((grid, xrange), x), yrange) \vdash e_{f\_body} \Rightarrow exp_{f\_body}\end{array}}{\begin{array}{rl}((grid, xrange), x) \vdash & \texttt{let yrange:=}\ldots\texttt{in}\ e_{f\_body}\ \texttt{endlet} \Rightarrow \\ & exp_{f\_body} \cdot (\mathsf{id}, (+ \cdot \pi_2) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\# \ldots)^\circ)^\circ\end{array}}$$

Figure 42: Translating body of f `f` to exp_f

value: π_2 for this so we have:

$$exp_{f\_args} = \pi_2$$

Finding the translation of e_f is more complicated. The second premise: *Lookup_funenv* is responsible for finding the closure of `f`. Only the Adl expression part of the closure, represented by e_f, is shown here. Other parts of the closure are used to help build the new environment in the third premise: *Build_new_env*. Finally this new environment, now containing an entry for `f`'s formal parameter: `x` is used in the translation of e_f. The final result being stored in exp_f.

e_f is a let expression. Its translation is shown in figure 42. The declaration-part is shown in abbreviated form. The result of the first premise is the result of the first premise. As usual this result contains the new environment:

$$(((grid, xrange), x), yrange)$$

and new code, this time for `yrange` (which is the only variable declared in the sequence). The second premise finds the code for e_{f_body} evaluated under the new environment.

Finally, the conclusion of this rule returns the code:

$$exp_f = exp_{f\_body} \cdot (\mathsf{id}, (+ \cdot \pi_2) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\#\cdot! \cdot (\pi_1^2, \pi_2)^\circ, 2)^\circ)^\circ)^\circ$$

where exp_{f_body} has yet to be established and the remainder of the code is from the declaration of `yrange`.

e_{f_body} is short hand for the expression: `map (g,yrange)`. Since this is a `map` expression rule 20 applies. Its application to e_{f_body} is shown in figure 43. The code produced for exp_{f_body} is:

$$exp_{f\_body} = (exp_g) * \cdot \mathsf{distl} \cdot (\mathsf{id}, exp_{g\_args})^\circ$$

where exp_g and exp_{g_args} are derived from the function application in the premise.

The function application is shown in the second part of the figure. The argument, `yrange`, a simple variable, translates to π_2 so we have:

$$exp_{g\_args} = \pi_2$$

The rest of the premises extract the code for `g`'s body: e_g and evaluate it (within the new environment containing its formal parameter: `y`) to obtain exp_g.

Rule 20: Map (exp_{f_body})

$$\frac{(((grid, xrange), x), yrange) \vdash \texttt{g yrange} \Rightarrow exp_g \cdot exp_{g\_args}}{(((grid, xrange), x), yrange) \vdash \texttt{map (g,yrange)} \Rightarrow (exp_g) * \cdot \mathsf{distl} \cdot (\mathsf{id}, exp_{g\_args})^\circ}$$

Rule 12: Function application $(exp_g \cdot exp_{g\_args})$

$$\frac{\begin{array}{cc}(\ldots, yrange) \vdash \texttt{yrange} \Rightarrow \pi_2 & Lookup\_funenv(\ldots g \to \ldots e_g \ldots) \\ Build\_new\_env(\ldots \to ((\ldots, yrange), y)) & ((\ldots, yrange), y) \vdash e_g \Rightarrow exp_g\end{array}}{(((grid, xrange), x), yrange) \vdash \texttt{g yrange} \Rightarrow exp_g \cdot \pi_2}$$

Figure 43: Translating `map (g,yrange)` to exp_{body_f}

Rule 8 Let expression (exp_g)

$$\frac{\begin{array}{c}(\ldots, y) \vdash \texttt{del\_1:=}\ldots \to (((\ldots, y), del\_1), del\_2)\,(\mathsf{id}, del\_2)^\circ \cdot (\mathsf{id}, del\_1)^\circ \\ (((\ldots, y), del\_1), del\_2) \vdash e_{g\_body} \Rightarrow exp_{g\_body}\end{array}}{\begin{array}{ll}((((grid, xrange), x), yrange), y) \vdash & \texttt{let del\_1:=}\ldots\texttt{in}\, e_{g\_body}\, \texttt{endlet} \Rightarrow \\ & exp_{g\_body} \cdot (\mathsf{id}, del\_2)^\circ \cdot (\mathsf{id}, del\_1)^\circ\end{array}}$$

Figure 44: Translation of `g` to exp_g

Lastly, the conclusion combines the results of the premises into the code:

$$exp_g \cdot exp_{g\_args} = exp_g \cdot \pi_2$$

The unknown, e_g is a let expression so rule 8 applies. This application is shown in figure 44

The first premise translates the declaration part. In this case it contains the variable declarations for `del1` and `del2`. Both of these values translate to straightforward, but rather unwieldy, BMF expressions. Instead of quoting them verbatim the names *del_1* and *del_2*, are used as place-holders. As before, the first premise also yields an updated environment.

The second premise evaluates e_{g_body} in the updated environment. The conclusion of the rule combines the code for the declarations and the code for e_{g_body} into the expression:

$$exp_g = exp_{g\_body} \cdot (\mathsf{id}, del_2)^\circ \cdot (\mathsf{id}, del_1)^\circ$$

where exp_{g_body} is unknown.

e_{g_body} stands for the relatively simple expression: `(del_1^2 + del_2^2)^0.5`. Its translation is correspondingly simple:

$$\hat{} \cdot (+ \cdot (\,\hat{}\, \cdot (\pi_2 \cdot \pi_1, 2)^\circ,\ \hat{}\, \cdot (\pi_2, 2)^\circ)^\circ, \mathsf{0.5})^\circ$$

Now all of the constituent unknowns have been translated. All that remains is to consolidate the results.

Firstly we have the expression for exp_g:

$$exp_g = \hat{}\cdot(+\cdot(\hat{}\cdot(\pi_2\cdot\pi_1, 2)^\circ,\ \hat{}\cdot(\pi_2, 2)^\circ)^\circ, 0.5)^\circ\cdot(\mathsf{id}, del_2)^\circ\cdot(\mathsf{id}, del_1)^\circ$$

This is a component of the expression for exp_{f_body}:

$$\begin{aligned} exp_{f\_body} = \ & (\hat{}\cdot(+\cdot(\hat{}\cdot(\pi_2\cdot\pi_1, 2)^\circ,\ \hat{}\cdot(\pi_2, 2)^\circ)^\circ, 0.5)^\circ\cdot(\mathsf{id}, del_2)^\circ\cdot(\mathsf{id}, del_1)^\circ)*\cdot \\ & \mathsf{distl}\cdot(\mathsf{id}, \pi_2)^\circ \end{aligned}$$

Which, in turn forms the first part of exp_f:

$$\begin{aligned} exp_f = \ & (\hat{}\cdot(+\cdot(\hat{}\cdot(\pi_2\cdot\pi_1, 2)^\circ,\ \hat{}\cdot(\pi_2, 2)^\circ)^\circ, 0.5)^\circ\cdot(\mathsf{id}, del_2)^\circ\cdot(\mathsf{id}, del_1)^\circ)*\cdot \\ & \mathsf{distl}\cdot(\mathsf{id}, \pi_2)^\circ\cdot \\ & (\mathsf{id}, (+\cdot\pi_2)*\cdot\mathsf{distl}\cdot(\mathsf{id}, \mathsf{iota}\cdot -\cdot(\#\cdot!\cdot(\pi_1^2, \pi_2)^\circ, 2)^\circ)^\circ)^\circ \end{aligned}$$

exp_f is the function part of the map expression forming exp_{sobel_body}:

$$\begin{aligned} exp_{sobel\_body} = \ & ((\hat{}\cdot(+\cdot(\hat{}\cdot(\pi_2\cdot\pi_1, 2)^\circ,\ \hat{}\cdot(\pi_2, 2)^\circ)^\circ, 0.5)^\circ\cdot(\mathsf{id}, del_2)^\circ\cdot(\mathsf{id}, del_1)^\circ)*\cdot \\ & \mathsf{distl}\cdot(\mathsf{id}, \pi_2)^\circ\cdot \\ & (\mathsf{id}, (+\cdot\pi_2)*\cdot\mathsf{distl}\cdot(\mathsf{id}, \mathsf{iota}\cdot -\cdot(\#\cdot!\cdot(\pi_1^2, \pi_2)^\circ, 2)^\circ)^\circ)^\circ)*\cdot \\ & \mathsf{distl}\cdot(\mathsf{id}, \pi_2)^\circ \end{aligned}$$

exp_{sobel_body} is the last component of the let declaration represented by e_{sobel}. The translation of this (exp_{sobel}): is:

$$\begin{aligned} exp_{sobel} = \ & ((\hat{}\cdot(+\cdot(\hat{}\cdot(\pi_2\cdot\pi_1, 2)^\circ,\ \hat{}\cdot(\pi_2, 2)^\circ)^\circ, 0.5)^\circ\cdot(\mathsf{id}, del_2)^\circ\cdot(\mathsf{id}, del_1)^\circ)*\cdot \\ & \mathsf{distl}\cdot(\mathsf{id}, \pi_2)^\circ\cdot \\ & (\mathsf{id}, (+\cdot\pi_2)*\cdot\mathsf{distl}\cdot(\mathsf{id}, \mathsf{iota}\cdot -\cdot(\#\cdot!\cdot(\pi_1^2, \pi_2)^\circ, 2)^\circ)^\circ)^\circ)*\cdot \\ & \mathsf{distl}\cdot(\mathsf{id}, \pi_2)^\circ\cdot \\ & (\mathsf{id}, (+\cdot(\pi_2, 1)^\circ)*\cdot\mathsf{distl}\cdot(\mathsf{id}, \mathsf{iota}\cdot -\cdot(\#, 2)^\circ)^\circ)^\circ \end{aligned}$$

which is the final result of the program.

6 Conclusion

This document describes a first step in the process of converting a simple source language into efficient BMF code and then onto a target architecture. The preceeding pages outline only the initial translation from Adl to its equivalent BMF form. Though the description is thoroughgoing there are still two issues that have not been canvassed. I examine these briefly here.

6.1 Efficiency

If interpreted literally the BMF code produced by the translator is very inefficient. This is primarily due to the large amount of copying implied by the use of the distl function and the explicit passing of all values regardless of their use.

It is possible, with the addition of a few extra combinators to eliminate the great majority of this waste. An added benefit of this process is that the code also becomes more succinct. This optimisation process (referred to as "Data Movement Optimisation") is quite involved. The number of inference rules used to describe it is approximately double the number in the initial translator. As previously mentioned, code optimisation is described in a later report.

6.2 Type Checking

The BMF code produced by the translator contains no variable names. As a result the only type information present in the target code is attached to the operators. The higher order operators such as $*$, $/$ and $(\ ,\ )^\circ$ are polymorphic they aren't instantiated to concrete types. The only operators conveying type information are the scalar operators.

The final stage of the Adl type checker instantiates all of these operators to their concrete types. This makes the task of the translator very simple. The rule set *Convert_op* just has to change the Adl operator into its equivalent BMF operator. No other type information needs to be propagated[23].

The distinctions between different concrete instances of the same overloaded operator have been glossed over in this report. Instead of just using: $+$ as I've done throughout this text I should have used the appropriate concrete instance ie one of:

$$+_{int\to int\to int} \qquad +_{int\to real\to real} \qquad +_{real\to int\to real} \qquad +_{real\to real\to real}$$

The main motivation for not doing this was to save space.

6.2.1 Implementation

The Adl to BMF translator, like the stages of the project that preceeded it will be implemented in C. If experience with other sections of the project is mirrored here the parts of the translator with the most thorough natural semantic description will be the easiest to code and maintain. This perhaps reflects the worth of a formal sematic description as a software engineering tool.

References

[1] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, August 1978.

[2] C R Banger and D B Skillicorn. A foundation for theories of arrays. Obtainable at `ftp.qucis.queensu.ca:/pub/skill/`, January 1993.

[3] R S Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[4] Richard S Bird. Lectures in constructive functional programming. In *Constructive Methods in Computing Science*, pages 151 – 216. NATO ASI Series, Vol F55, 1989.

[5] Guy E Blelloch. *Vector Models for Data-Parallel Computing.* The MIT press, Cambridge Massachusetts, 1990.

[6] Tore A Bratvold. Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 213–226, September 1992. Aachener Informatik-Berichte Nr 92-19.

[23] It might be wise to include some representation of the expected input types of the program for the convenience of the writer of later stages of the compiler. Even if the input types were not passed on they could always be derived from the types of the first operators of the program.

[7] David Busvine. Implementing recursive functions as processor farms. *Journal of Parallel Computation*, 19:1141–1153, 1993.

[8] B. Carpentieri and G. Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIG Plan Notices*, 26(10):19–28, October 1991.

[9] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *PARLE'93: Parallel Architectures and Languages Europe*. Springer Verlag, June 1993. LNCS 694.

[10] J T Feo and D C Cann. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, (10):349–366, 1990.

[11] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science: Edinburgh U.K.*, pages 140–157. Springer-Verlag, September 1987. Appears in, Lecture Notes in Computer Science Vol 283.

[12] Gaetan Hains and Lenore M. R. Mullin. Parallel functional programming with arrays. Listed for publication in the Computer Journal, 1993.

[13] Paul F. Hoogendijk and Roland C. Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Science of Computer Programming*, 22:67–105, 1994.

[14] Ian Jacobs and Laurence Rideau. A centaur tutorial. Technical Report 141, INRIA, France, August 1992.

[15] Barry Jay. Matrices, monads and the fast fourier transform. Technical report, Sydney University of Technology, 1993.

[16] Geraint Jones and Mary Sheeran. The study of butterflies. In *Collecting butterflies*. Oxford University Computing Laboratory, 1991.

[17] G. Kahn. Natural semantics. In *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*. Springer-Verlag, 1987.

[18] Richard Miller. *A constructive theory of Multidimensional Arrays*. PhD thesis, Lady Margret Hall, Oxford, February 1993.

[19] John A. Richards. *Remote Sensing Digital Image Analysis*. Springer Verlag, 1993.

[20] Paul Roe. Derivation of efficient data parallel programs. In *ACSC-17*, pages 621 – 628, January 1994.

[21] David B. Skillicorn. Parallelism and the Bird-Meertens formalism. Obtained via ftp from Queens university, Canada, April 1992.

[22] David B. Skillicorn. the Bird-Meertens formalism as a parallel model. In *Software for Parallel Computation*, June 1992.

[23] Mike Spivey. A categorical approach to the theory of lists. In *Mathematics of Program Construction*, pages 400–408. Springer-Verlag, June 1989. Found in: Lecture Notes in Computer Science Vol 375.