Translating Mathematica expressions to High Performance Fortran

Anton Antonov

UNI-C, DTU, bldg.304, DK-2800 Lyngby, Denmark
email: Anton.Antonov.Antonov@uni-c.dk
phone: (+45) 3587 8965, *fax:* (+45) 3587 8990

Abstract

This paper introduces some ideas for translating the functional language *Mathematica* to the data-parallel language High Performance Fortran (HPF). It first discusses why we have the ability to do that. Then it gives some interpretations by Category Theory. Third the translating approach is presented for different *Mathematica* expressions that could be interpreted as specifications for parallel independence, reduction, task parallelism and subprogram's data mapping. Last is shown a simple executable program generated by the translator.

1 Introduction

Programming by functions is very natural and completely suitable for specifying mathematical problems quickly and elegantly. We can say that when we program with a functional language we directly use the mathematical meaning of the process we want to model and that leads to concise, elegant programs. This is one of the main reasons the functional languages to be developed [3].

Conciseness and elegance are highly desirable in the procedural languages too. That is why, for example, the modern FORTRAN 95 is more convenient for its users than the older FORTRAN versions. Consider the array processing and the array intrinsics in FORTRAN 90 and FORTRAN 95 [4] we can say that these languages are like object-oriented ones with just one static class which defines an abstract type for arrays.

We want also concise and elegant programs for the parallel computers. According to the two dual approaches in the paradigm of parallel programming, namely - the functional decomposition and the data decomposition, it seems natural from the functional languages to develop functional-parallel ones and from the "data" languages (like FORTRAN) to develop data-parallel ones. Examples for the first kind are Concurrent ML, Clean, NESL. For the second are HPF, Open MP, CoArray FORTRAN.

In this paper we will consider how we can translate expressions and programs written in a functional language (*Mathematica*) to programs in a data-parallel language (High Performance Fortran). In Section 2 we discuss the general reasons why the translation comes to us naturally. Then in Section 3 we give formal mathematical interpretation of these general reasons by Category theory. We show that the category[1] that models *the programming code* of High Performance Fortran could be imbeded in the category that models the functional languages[2]. The inverse image of this imbeding is the translator described in Section 4. In that section we show how typical constructions in functional programming correspond to typical constructions in data-parallel programming. The translator is written in *Mathematica*. Since it is just a set of rules how different types of *Mathematica* constructions are translated to HPF, Section 4 is naturally divided in subsections that show how are translated:

- data manipulation constructions
- pure functions
- second order functions
- function calls

The second order functions are translated to parallel loops. The translation of function calls should deal with data mapping in subprogram interfaces. The way the translator is implemented allows user's programs builded in *Mathematica* to be just rerun in it, in order the corresponding HPF code to be generated. Every kind of translation we describe is accompanied with examples. Also we do not assume that the reader is familiar with functional languages.

[1] category is a graph with special properties imposed on it.

[2] the cartesian closed category

Finally, a simple program is shown generated by the translator.

2 Why we have the ability to do that?

First let us observe that we have an advantage of using a particular programming language because of the constructions it has but we appreciate or find that language useful because of the code which is *not* in the programs written with it.

On the other hand the languages converge - they give us advantages in different ways but they become useful in the same way. We have different levels of that convergence. It could be convergence of paradigms or it could be convergence of structures. Let us look at the following examples:

Example 1. Object-Oriented(O-O) languages$\longleftrightarrow$Functional languages(FL)

"A functional language may be described roughly as one that gives the user some primitive types and operations and some constructors from which one can produce complicated types and operations."[1], page 19.
If we try to describe roughly an object-oriented language we will end up with the same sentence quoted above. Of course the constructions we use in these languages are completely different. Although they give as advantages in different ways, obviously, the way they are useful is the same. We will call that *paradigm convergence*.

Example 2. FORTRAN 95 $\longleftrightarrow$*Mathematica*

The following FORTRAN 95 code :

```
DO I=1,SIZE(A,1)
  A(I,:,:)=TRANSPOSE(A(I,:,:))
END DO
```

is equivalent to the *Mathematica* code:

```
A=Map[Transpose,A]
```

We will call similarities like that *structure convergence*.

As it was said, in the paradigm of parallel programming we have two dual approaches: functional decomposition and data decomposition. Because of this duality and the structure and paradigm convergence of the programming languages, it is natural to expect that we have the ability relatively easy to translate programs written in a functional language into programs of a data-parallel one.

Because functional languages are focused on the functions they should have advanced mechanisms to handle the data structures, since the domain and codomain of a function are parts of its definition. Let us take, as example, a bijective function $f : A \to B$. Mathematically that could be written like $f(A) = B$ or $B = f(A)$ and here we do not care how the image of A will be produced or how the sets A and B should be processed for that bijective mapping. If in the environment of a functional language like *Mathematica* we want the data structure B to be the image of the data structure A we can use a specification like `B=Map[f,A]` which has the same appearance and meaning as the corresponding mathematical notation. The structure of A and B is not reflected in that specification, they are handled implicitly. If it was not like that we would not have the elegance of specifying our problems by a functional language. Because we use directly (just) the mathematical meaning of the processes we specify, their parallel nature, if they have it, is preserved in the specification.

Similar observation can be made for the O-O languages. The objects of the abstract types we define in these languages process implicitly the operations specified for them. Therefore we can define abstract types for the sets A and B above in order to be able to write code like `B.map(f,A)`. This is of course the paradigm convergence introduced in Example 1. All the arrays in FORTRAN 95 are objects of an abstract type for them. So great amount of the FORTRAN 95 code have common characteristics with the codes written with true O-O languages. One of the reasons that the translator is easy to implement is that HPF is based on FORTRAN 95.

3 Categorical Interpretation

In Section 2 we discussed the general reasons why the translation should be natural and easy. In this section we interpret these reasons with Category theory to give formal description of the translation from mathematical or mathematician's point of view. Then the translator described in Section 4 can be considered as a concrete implementation of that formal description.

It is natural to model different data types using cartesian products combined with corresponding morphisms and projections i.e. with *cartesian category*. On one hand FL have data types with operations between them and constructors applied to them. Clearly, they can be modeled by cartesian category. On the other hand FL have operations to produce derived data types and derived operations. It will be nice if we can model the later operations and the data types by cartesian products. Such model exists and is called *cartesian closed category*. It is derived from the cartesian category by addition of a new kind of set called *function space* and two new morphisms implied by this addition. The morphisms are called *application* (the function space is in its domain) and *curring* (the function space is in its codomain). These notions and their connections to the FL are described by Field and Harrison in [3], Chapter 13, Section 4, and by Barr and Wells in [1], Chapter 6. Their definitions can be found in the appendix of this paper.

The curring is of special importance since it is the way we make abstractions and hierarchies in functional languages. (In O-O languages we make abstractions and hierarchies by the polymorphism.) As Field and Harrison explain in [3], we could interpret the notation

```
f x y z = ...
```

as `f` being a function of three arguments i.e. like `f(x,y,z)` but it could be also interpreted as

```
((f[x])[y])[z]
```

i.e. the result of applying `f` to `x` is a function denoted `f[x]`, then this function is applied to `y`, then the function `(f[x])[y]` is applied to `z`. This idea of treating a function of n arguments as a concatenation of n single-argument functions is called *curring* after the mathematician H.B. Curry.

One way to express formally the analogy between FL and HPF, is to characterize the later[3] in terms of the notions above. As it was said FORTRAN 95 can be considered as an O-O language with just one class implemented and in Example 1 in Section 2 was said that the O-O languages and the FL have the same verbal (rough) formulation. Let us try to show that the category that models the FL, the cartesian closed category, could be used for description of an O-O language. An abstract type (a class) of an O-O language could be considered as a classical data record extended to one that can contain procedures called usually methods. The methods are applied to the data fields of the record and if we consider them as functions, the data fields can be in their domains or in their codomains. It is natural to consider each of the methods in some appropriate exponential of domains. May be that way we can find a category which describes an O-O language or, more precisely, a kind of the weaker "O-O-calculus" (by analogy with the λ-calculus), but since we are interested in making a translator, that is a little bit away from our purposes. Rather we should be interested in a category that models not the language we want to translate to, but the programming code written with it. So we should try to model the HPF code.

Great amount of the FORTRAN code concerns changing the values of array elements. Let us consider code which assigns to each element of an array a function of its indices (the result of the function might be the value of an element from other array with corresponding indices). Example 3 shows that writing that code in the context where the array and the function are defined could be considered as the result of a function of just two arguments - the name of the array and the name of the function.

Example 3.

Let `b` be an array of dimension `N`. Since FORTRAN 95 has an intrinsic function `SIZE(b,i)` which gives the size of `b` along the `i`-th axes, the code will be

```
DO i1=1:SIZE(b,1)
 DO i2=1:SIZE(b,2)
  ...
   DO iN=1:SIZE(b,SIZE(b,iN))
    b(i1,i2,...iN)=func(i1,i2,...,iN)
   END DO
  ...
```

[3] and FORTRAN 95 as a part of HPF

```
 END DO
END DO
```

or more shortly

```
FORALL(i1=1:SIZE(b,1),i2=1:SIZE(b,2),...,iN=1:SIZE(b,N)))
 b(i1,i2,...iN)=func(i1,i2,...,iN)
END FORALL.
```

To produce that code we should know the array `b`, the function `func`, and the dimension of `b` - `N`. The number `N` determines the size of the code as a text. If `N` is defined from the context via `b`, then the code above can be considered as a result of a function of two arguments (`b` and `func`).

Since we can consider the code that do the assignment as a function of two arguments, then that function is something like a specification similar to the specification for mapping a function to a structure in the FL. When we add HPF directives to that specification, it becomes complete in terms of its parallel execution.

Similarly, we can find out other kind of specifications for :

- actions like reduction of a row of objects to an object of the same kind,
- actions producing a reshaped copy of an array,
- assignments with different number of arrays,
- assignments of the results of nested functions,
- some useful combinations of the actions mentioned.

The completing of the specifications by HPF directives recalls the curring.

Example 4. an 5. bellow complete the analogy above. They show that there is a cartesian closed category that describes the HPF code that deal with assignments to array elements.

Example 4.

In this example we will use the notions shape and rank of an array. If the array `a` is defined in FORTRAN 95 as:

```
double a(4,3,6),
```

then the *shape* of `a` is the 3-tuple (4,3,6) and the *rank* of `a` is 3 (since `a` is 3 -dimensional).
The notions from the Category theory used in this example are defined in the appendix.
The assignment between the elements of two arrays can be described with the following category. An object of the category are all the arrays with the same shape. Since all arrays with the same shape have the same rank we can define the number $rank(A)$ as the rank of the arrays that constitute A. An arrow $f : A \to B$ is defined if $rank(A) = rank(B)$. Then f is a monomorphism from the index set of A in the index set of B. Composition of two arrows $f : A \to B$ and $g : B \to C$ is an arrow $h : A \to C$, $h = g \circ f$. We define $A \times B$ to be the arrays with a shape equal to the concatenation of the shape of A with the shape of B. It is easy to check, that at this point we have cartesian category. Bellow we will complete it to cartesian closed one.
Let us define $A \Rightarrow B$ in the following way. Every element $\widehat{f}$ of $A \Rightarrow B$ corresponds to an arrow $f : A \to B$, and $\widehat{f}$ have one of the forms:

```
FORALL(i1=l1:u1:s1, i2=l1:u1:s2)
 b(i1,i2)=a(m1(i1),m3(i2))
END FORALL
```

or

```
! some comment
FORALL(i1=l1:u1:s1, i2=l1:u1:s2)
 b(i1,i2)=a(m1(i1),m3(i2))
END FORALL
```

where m_i and the triplets `l1:u1:s1` and `l2:u2:s2` form the monomorphism f. Note that $\widehat{f}$ is a piece of programming code, a text. The arrow $App^{A,B} : (A \Rightarrow B) \times A \to B$ applied to the couple $\left\langle \widehat{f}, a \right\rangle$, is defined to give $f(a)$, where f is the arrow corresponding to $\widehat{f}$. There is also an object H={`""`,`"!HPF$ INDEPENDENT"`}. If $\widehat{f} \in A \Rightarrow B$ there are just two corresponding to $\widehat{f}$ elements, $\widehat{f'}$ and $\widehat{f''}$, that belong to $H \Rightarrow (A \Rightarrow B)$. The first one, $\widehat{f'}$, is equal to `""` concatenated with $\widehat{f}$, and the second one, $\widehat{f''}$, is equal to `"!HPF$ INDEPENDENT"` concatenated with $\widehat{f}$. Obviously $\widehat{f'}$ is equal to $\widehat{f}$. Let every arrow $\alpha : H \times A \to B$ be a monomorphism from the index set

of A to the index set of B. Obviously α corresponds to unique arrow $f : A \to B$. Let the arrow $\Lambda(\alpha) : H \to A \Rightarrow B$ is defined as $\Lambda(\alpha)(H) = \{\widehat{f}', \widehat{f}''\}$. Now we have the property that $\alpha = App^{A,B} \circ \langle \Lambda(\alpha) \circ Fst, Snd \rangle$. So we have a cartesian closed category that describes the action of an assignment between two arrays in the HPF code.

Example 5.

In order to describe the reductions we could define a category in which we have array objects like in Example 9 and an arrow $f : A \to B$ is defined if $rank(A) - 1 = rank(B)$. Clearly the two categories could be combined. To the object H in that combined category should be added the HPF directive for reduction.

We can say that FORTRAN 95 provides the first order specifications between the FORTRAN 95 array data structures and the HPF directives provide the second order specifications. These first order specifications correspond, of course, to the first order functions in the FL and the second order specifications to the second order functions. By that analogy it could be seen that some class of HPF programs could be imbeded in the cartesian closed category of a functional language since they could be described by category of the same kind. Since it is an imbeding (monomorphism) its inverse exists with the imbeding's codomain as a domain. This inverse mapping is the translator. The presentation above gives very good reasons to believe that making the translator will be an easy task. And that belief is true - the basic part of the translator was made for one day. The presentation above suggests that the translator could have two separate parts - one that translates higher order functions to HPF directives, and another one that translates first order functions to FORTRAN code. Well, it should be admitted that the current implementation is not like that since the translator was made first and the categorical interpretation last.

4 The translator

The ideal end result is a translator that takes a program written in *Mathematica* language, produces a corresponding HPF program, compiles it, and defines a *Mathematica* function which calls that (parallel) program. We will show bellow that we are close to that ideal end result - the translator bellow can provide an environment, helpful for moving a specification of a problem to a corresponding HPF program.

The strategy adopted is very simple: For each type of expression is made an overloading definition of the standard function `FortranForm` in order to get the corresponding HPF code. When a particular expression is translated the result of `FortranForm` is added to a variable called `FORTRANPROGRAM`[4]. Since in HPF the mapping directives should pass at the entry of a programming scope it is necessary to keep them in a variable different from that we accumulate the executive code to. That special variable is called `HPFSPECIFICATIONS`. Its contents will be prepended to the contents of `FORTRANPROGRAM` at the end of the translation. The executive HPF code is accumulated via the "hook" `$Pre`. `$Pre` is a global *Mathematica* variable and any function assigned to it will be automatically applied before the evaluation of any expression. The translation of an expression depends on the values assigned to the symbols in it.

A session with the translator could be the evaluation of the following steps:

the initial step of the translation

```
FORTRANPROGRAM={};HPFSPECIFICATIONS={}; $Pre=PreFunction;
```

some Mathematica programming code

```
bsize=4
image=Table[i,{i,1,8},{j,1,8}]
im1=Partition[image,bsize]
im2=Map[Transpose,im1]
```

the final step of the translation

```
$Pre=.;
FORTRANPROGRAM=Join[HPFSPECIFICATIONS,Drop[FORTRANPROGRAM,-1]];
```

the generated programming code printed with `Map[Print,FORTRANPROGRAM];`

[4] the translator is one-pass one.

```
!hpf$ PROCESSORS squad(2,1)
!hpf$ DISTRIBUTE image(BLOCKS(4),BLOCKS(8)) ONTO squad
!hpf$ ALIGN im1(i,*,:) WITH image((i-1)*4+1,:)
bsize=4
!hpf$ INDEPENDENT
forall(i=1:8,j=1:8) image(i,j)=i
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(image,1)/bsize
im1(i,*,:)=image((i-1)*4+1,:)
end do
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(im1,1)
im2(i,:,:)=transpose(im1(i,:,:))
end do
```

Now we could save the variable `FORTRANPROGRAM` in a file, to compile that file[5] and to use the compiled code as a *Mathematica* function as it is shown by Janhunen in [6].

The answer of *Mathematica* to some command (like MATLAB or Maple) is not necessarily assigned to some symbol or variable defined by the user, or variable he or she is aware of. We assume that all the expressions we will translate will be an assignments. This is the only restriction we take for the sake of simplicity of the translator. This assumption is very natural, nevertheless it can be overcome.

The translator is described bellow by separate descriptions what code is generated for assignments of different types. The notation **Mathematica expression $\longrightarrow$ HPF directives / FORTRAN 95 constructions** is used to designate the descriptions. The lighter gray boxes of the accompanying examples contain the user's input and the darker gray boxes contain the *Mathematica* response.

4.1 Translation of data manipulating structures

4.1.1 lhs=rhs $\longrightarrow$lhs=rhs

The assignment of an atomic object in *Mathematica* should be like the assignment in HPF. The definition is straightforward - to the FORTRAN form of the left hand side expression is assigned the FORTRAN form of the right hand side.

```
FortranForm[a1=5]
```

```
a1=5
```

4.1.2 lhs=Table[expr, iterators] $\longrightarrow$INDEPENDENT, FORALL statement

With the function `Table` we can make lists and lists of lists. We can do, for example, this:

```
Table[Binomial[i,j],{i,1,3},{j,1,i}]
```

```
{{1},{1,2},{1,2,3}},
```

but we restrict ourself to consider just lists with elements of equal size on each level i.e. arrays. The generation of such lists with `Table` could be done in parallel. Thus for the expression

```
image=Table[i+Sin[j],{i,1,3},{j,1,4}]
```

```
{{1+Sin[1],1+Sin[2],1+Sin[3],1+Sin[4]},
 {2+Sin[1],2+Sin[2],2+Sin[3],2+Sin[4]},
 {3+Sin[1],3+Sin[2],3+Sin[3],3+Sin[4]}}
```

the corresponding HPF code is

[5] for the code of this example we should put some additional work

```
FortranForm[image=Table[i+Sin[j],{i,1,3},{j,1,4}]]
```

```
!hpf$ INDEPENDENT
forall(i=1:3,j=1:4) image(i,j)=i + Sin(j)
```

4.1.3 Transpose $\longrightarrow$TRANSPOSE

`Transpose` is translated directly:

```
FortranForm[im2=Transpose[im1]]
```

```
im2=transpose(im1)
```

4.1.4 Partition$\longrightarrow$ PROCESSORS, DISTRIBUTE, ALIGN, INDEPENDENT, DO

The function `Partition` is very useful when we program with *Mathematica* because it gives the opportunity to make relevant domains for the functions we define. For example, if we want to simulate the parallel summation of a row of numbers, we can do that in *Mathematica* in the following way:

1.First we define a row of numbers

```
row=Table[i,{i,1,8}]
```

```
{1,2,3,4,5,6,7,8}
```

2.Second we partition it in lists of length 2

```
row2=Partition[row,2]
```

```
{{1,2},{3,4},{5,6},{7,8}}
```

3.Then we sum the elements of each of the lists of row 2 and assign it to the variable row

```
MyPlus[{x_,y_}]:=x+y
row=Map[MyPlus,row2]
```

```
{3,7,11,15}
```

4.We should repeat steps 2. and 3. until the result of step 3. is one element list.

The algorithm above could be expressed more concisely by the command:

```
NestList[Map[Apply[Plus, #1]&,Partition[#1,2]]&,row,3]
```

```
{{1,2,3,4,5,6,7,8},{3,7,11,15},{10,26},{36}}
```

We can see the successive reductions of the row. In the last command are used pure functions. Their translation to HPF is described in the subsection 4.2 bellow.

The function `Partition` can be expressed in HPF by the executive directive `INDEPENDENT` and the mapping directives `DISTRIBUTE` and `ALIGN`. It would be nice to express `Partition` to HPF just with `INDEPENDENT` and `ALIGN`. For example if we take the variable `image` to be

```
image=Table[i,{i,1,8},{j,1,8}]
bsize=4;
```

```
{{1,1,1,1,1,1,1,1}, {2,2,2,2,2,2,2,2},
 {3,3,3,3,3,3,3,3}, {4,4,4,4,4,4,4,4},
 {5,5,5,5,5,5,5,5}, {6,6,6,6,6,6,6,6},
 {7,7,7,7,7,7,7,7}, {8,8,8,8,8,8,8,8}}
```

and we want the translation of `im1=Partition[image,bsize]` to make the assignments to `im1` without communication, the translation could be the following:

```
FortranForm[im1=Partition[image,bsize]]
```

```
!hpf$ ALIGN im1(i,:,:) WITH image(bsize*(i-1)+1:bsize*i,:)
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(image,1)/bsize
im1(i,:,:)=image(bsize*(i-1)+1:bsize*i,:)
end do,
```

where `bsize` should be defined as a parameter or instead of it to use its value (in this case 4) . The problem with translations of that kind is that the `ALIGN` constraints does not allow to specify a subscript triplet as a function of one or more align-dummies. One way to get out of this is the align directive to be:

```
!hpf$ ALIGN im1(i,*,:) WITH image((i-1)*bsize+1,:)
```

where each subarray `i` of the new dimension in `im1` is placed on the same processor with the element of the initial array (`image`), that should be first in that subarray. Now we should make an appropriate distribution of the initial array in order to be sure that no communications would arise in the assignment. This distribution is straightforward - `Partition[array,stride]` corresponds to the HPF's directive

```
!HPF$ DISTRIBUTE array(BLOCK(stride), BLOCK(SIZE(array,2)),...,
BLOCK(SIZE(array, SIZE(SHAPE(array))))) .
```

If we have several partitions in the *Mathematica* code it is possible to have distributee clashes. These clashes can be easily checked with a table generated during the compilation. We are even able to define special processor arrangements for every distribution and to cope successfully with the processor arrangement clashes. So, for the partitioning above we could have the translation:

```
FortranForm[im1=Partition[image,bsize]]
```

```
!hpf$ PROCESSORS squad(8/bsize,1)
!hpf$ DISTRIBUTE image(BLOCKS(bsize),BLOCKS(8)) ONTO squad
!hpf$ ALIGN im1(i,*,:) WITH image((i-1)*bsize+1,:)
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(image,1)/bsize
im1(i,:,:)=image(bsize*(i-1)+1:bsize*i,:)
end do
```

As it was explained at the beginning of the section, the mapping directives should be accumulated to a variable, say `HPFSPECIFICATIONS`, different from the variable `FORTRANPROGRAM`. So actually the result of `FortranForm[im1=Partition[image,bsize]]` is added to `FORTRANPROGRAM` via `$Pre` function but the corresponding mapping directives are added to `HPFSPECIFICATIONS` (by the function `FortranForm` itself).

The code generated for `FORTRANPROGRAM` is

```
HPFSPECIFICATIONS={}; FortranForm[im1=Partition[image,bsize]]
```

```
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(image,1)/bsize
im1(i,*,:)=image((i-1)*4+1,:)
end do
```

The code generated for `HPFSPECIFICATIONS` is

```
HPFForm[HPFSPECIFICATIONS]
```

```
!hpf$ PROCESSORS squad(8/bsize,1)
!hpf$ DISTRIBUTE image(BLOCK(bsize),BLOCK(8)) ONTO squad
!hpf$ ALIGN im1(i,*,:) WITH image((i-1)*bsize+1,:)
```

4.2 Pure Functions

Pure functions give very convenient way for specifying calculations. For example if we want to square all the elements in a list we can write the following:

```
Map[#1^2&,{4,5,{m,6,7},7,3,2,a}]
```

```
{16, 25, {m^2, 36, 49}, 49, 9, 4, a^2}
```

instead of defining first a square function and then mapping the list via it.

```
MySquare[x_]:=x^2
Map[MySquare,{4,5,{m,6,7},7,3,2,a}]
```

```
{16, 25, {m^2, 36, 49}, 49, 9, 4, a^2}
```

Pure functions are also convenient for expressions that have multiple entries of the same subexpression. For example the expression

```
If[Length[im1]==5,Rest[im1],im1];
```

could be written with a pure function as

```
If[Length[#1]==5,Rest[#1],#1]&[im1];
```

The benefit is more obvious if we try to imagine that instead of `im1` we have some complicated expression

```
If[Length[#1]==5,Rest[#1],#1]&[Transpose[Reverse[im1]]];
```

The examples above use one argument pure functions. We should distinguish between pure functions that are listable i.e. automatically threaded over list arguments[6], and nonlistable ones. Later ones are translated straightforward. Former ones, if they are applied to arrays, have a natural translation to FORTRAN 95 via the statement FORALL as it is shown in the next example:

```
FortranForm[im2=#1^2+Sin[#1]&[im1]]
```

```
forall(i1=1:size(im1,1),i2=1:size(im1,2),i3=1:size(im1,3))
im2(i1,i2,i3)=im1(i1,i2,i3)**2 + Sin(im1(i1,i2,i3))
end forall
```

The current translator translates only one argument pure functions. As is it shown in the next example nonlistable ones are not taken correctly.

```
FortranForm[im2=Transpose[#1]&[im1]]
```

```
im2=Function(transpose(Slot(1)))(im1)
```

4.3 Second order functions

It looks like there are two major kinds of loops - loops that have a body calculated independently of the other cycles in the loop and loops with a body calculated according to the previous one. May be any other loop can be expressed in terms of these two. The first kind of loop corresponds to the mathematical notion of mapping - we could consider it as mapping with a function defined by the loop's body depending of the control variables of the loop. The second one corresponds to the notion of superposition (or nesting) of a function. The *Mathematica* functions for specifying a mapping are `Map`, `MapAll`, `MapThread`. Those for specifying a nesting are `Fold`, `Nest`, `FixedPoint`. These functions a second order functions. On one hand `MapAll` and `MapThread` can be expressed by `Map`, and on the other hand `Nest` and `FixedPoint` can be expressed by `Fold`. Bellow are proposed HPF translations of `Map`, `Fold` and the combinations `Map[Fold[...]...]` and `Map[Composition[f1,f2],...]`.

[6] e.g. `f[{a,b}]` becomes `{f[a],f[b]}`

4.3.1 lhs=Map[func, array] $\longrightarrow$INDEPENDENT, DO

`Map[f, {a,b,c,...}]` gives `{f[a],f[b],f[c],...}`. It is natural to translate commands like that to independent `DO` or `FORALL` loops - `Map` specifies that `f` is applied separately to each of the elements in the list. We can implement that separate application of `f` by `DO` loop and since it should be done separately we can put the `INDEPENDENT` directive. In *Mathematica* we can specify different levels the function `f` to be applied on. Since our general assumption is that all the structures in the *Mathematica* code we translate are arrays, in the implemented translation of `Map[f,expr]` the function `f` is applied to the last level of the structure `expr`, i.e. to the elements of the array `expr`.

The first example is the function `Map` with a pure function to be applied to.

```
FortranForm[im2=Map[#1^2&,im1]]
```

```
!hpf$ INDEPENDENT, NEW(i,i1,i2)
do i=1,size(im1,1)
forall(i1=1:size(im1(i,:,:),1),i2=1:size(im1(i,:,:),2))
im2(i,i1,i2)=im1(i,i1,i2)**2
end forall
end do
```

We can take also some of the standard functions but their FORTRAN translation should be predefined.

```
FortranForm[im2=Map[Transpose,im1]]
```

```
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(im1,1)
im2(i,:,:)=transpose(im1(i,:,:))
end do
```

The compiler does not take, for example, correctly this:

```
FortranForm[im2=Map[Sin,im1]]
```

```
!hpf$ INDEPENDENT, NEW(i<>If[NewVariables[Sin[righthand$2535]]
!= , ,<>NewVariables[Sin[righthand$2535]],
NewVariables[Sin[righthand$2535]]]<>)
do i=1,size(im1,1)
Evaluate(lefthand$2535)=Sin(Evaluate(righthand$2535))
end do
```

but it takes the following completely satisfying version

```
FortranForm[im2=Map[Sin[#1]&,im1]]
```

```
!hpf$ INDEPENDENT, NEW(i,i1,i2)
do i=1,size(im1,1)
forall(i1=1:size(im1(i,:,:),1),i2=1:size(im1(i,:,:),2))
im2(i,i1,i2)=Sin(im1(i,i1,i2))
end forall
end do.
```

4.3.2 lhs=Fold[func, initial value, array] $\longrightarrow$INDEPENDENT, REDUCE, DO

Since the meaning of `Fold[f,x0,{a,b,c,d}]` is `f[f[f[f[x0,a],b],c],d]` we can translate it to an independent, reduce loop if the function `f` is associative. The only reduction functions allowed in HPF are `+, *, MAX, MIN, IAND, IOR, IEOR`. So it is relevant the translator to check, does the function to be folded, is one of them. If it is not we can translate the whole expression as pipelining as it is described in the sub-subsection 4.3.4. In the current translator this feature is not implemented.

```
FortranForm[im3=Fold[Plus,x0,im1]]
```

```
im3(:,:)=x0
!hpf$ INDEPENDENT, NEW(i), REDUCTION(im3)
do i=1,size(im1,1)
im3(:,:)=im1(i,:,:) + im3(:,:)
end do
```

4.3.3 lhs=Map[Fold[func, initial value, #1]&, array]⟶INDEPENDENT, REDUCE, DO

We are lucky that HPF allows array variables to appear in the `REDUCTION` clause. We are lucky even more, because we are also able to put array sections as reduction variables. Because of the later feature we have translations like the following:

```
FortranForm[im4=Map[Fold[Plus,x0, #1]&,im1]]
```

```
!hpf$ INDEPENDENT, NEW(i,i1,i2)
do i=1,size(im1,1)
im4(i1,:)=x0
!hpf$ INDEPENDENT, NEW(i), REDUCTION(im4(i1,:))
do i=1,size(im1(i1,:,:),1)
im4(i1,:)=im1(i1,i,:) + im4(i1,:)
end do
end do
```

In the last example the variable `x0` should be conformable with the array section `im4(i,:)`.

4.3.4 lhs=Map[Composition[func1,func2], array]⟶ TASK REGION, ON

Until now we have discussed just the translation to the standard HPF 2.0. Here we will consider the approved extensions of HPF 2.0 for task parallelism, the directives TASK REGION and ON. We will take the approach described by Schreiber in [7]. It is clear that `Composition[f1,f2]` stands for `f1[f2[argument]]`. This composition is a fear invitation for pipelining - when `f1` works on the result of `f2`, the later could work on the next piece of data to be processed. Apparently this approach could be extended to any amount of composed functions. This generalization leads us to pipelined code when we translate `Fold` with an arbitrary function to be folded. Here is an example how the translator works on compositions. The *Mathematica* code is very concise.

```
FortranForm[r1=Composition[Sin[#1]&,Cos[#1]&,Tan[#1]&,#1^2&][im2]]
```

```
!hpf$ TASK_REGION
!hpf$ ON HOME(v1Func), RESIDENT, BEGIN
forall(i1=1:size(im2,1),i2=1:size(im2,2),i3=1:size(im2,3))
v1Func(i1,i2,i3)=Sin(im2(i1,i2,i3))
end forall
!hpf$ END ON
v2Func=v1Func
!hpf$ ON HOME(v2Func), RESIDENT, BEGIN
forall(i1=1:size(v1Func,1),i2=1:size(v1Func,2),i3=1:size(v1Func,3))
v2Func(i1,i2,i3)=Cos(v1Func(i1,i2,i3))
end forall
!hpf$ END ON
v3Func=v2Func
!hpf$ ON HOME(v3Func), RESIDENT, BEGIN
forall(i1=1:size(v2Func,1),i2=1:size(v2Func,2),i3=1:size(v2Func,3))
v3Func(i1,i2,i3)=Tan(v2Func(i1,i2,i3))
```

```
end forall
!hpf$ END ON
v4Func=v3Func
!hpf$ ON HOME(v4Func), RESIDENT, BEGIN
forall(i1=1:size(v3Func,1),i2=1:size(v3Func,2),i3=1:size(v3Func,3))
v4Func(i1,i2,i3)=v3Func(i1,i2,i3)**2
end forall
!hpf$ END ON
r1=v4Func
!hpf$ END TASK_REGION
```

The complete translation of `Map[Composition[func1,func2], array]` is not finished yet but is close.

4.3.5 Translating function calls

This feature is not implemented in the translator but clearly from the presentation above it is more than reachable. For example, a *Mathematica* function defined as

`InhFunc[arg_, dummy_]:= ...`

could be considered as a subprogram with inherited argument mapping.

A function defined as

`DescrFunc[arg:{{_,_,_}...}, dummy_Number]:=...`

could be considered as a subprogram with descriptive argument mapping.

Of course, there are some issues like: Where the code of the generated subprograms should be placed? Should translator have special definitions for the user defined functions, that appear in the main second order functions? These questions are postponed to be solved, eventually, in the future.

5 Working Example

We translate the following *Mathematica* assignments into HPF code.

```
bsize=4
line=Table[i,{i,1,8}]
image=Table[i,{i,1,8},{j,1,8}]
im1=Partition[image,bsize]
im2=Map[Transpose,im1]
im2=Map[#1^2&,im2]
x0=Table[0,{i,1,Dimensions[im1][[2]]},{j,1,Dimensions[im1][[3]]}]
im3=Fold[Plus,x0,im1]
x1=Table[0,{i,1,Dimensions[im1][[3]]}]
im4=Map[Fold[Plus,x1, #1]&,im1]
```

To the code generated we add the FORTRAN 95 definitions of the data structures `line, image ,im1, im2, im3, im4, x0` and we get the program bellow.

```
program fprog
integer, parameter :: bsize=4
integer :: i, i1, i2, i3, j
integer :: image(8,8), line(8)
integer :: im1(2,4,8), im2(2,8,4), im3(4,8), x0(4,8)
integer :: im4(2,8), x1(8)
!hpf$ PROCESSORS squad(2,1)
!hpf$ PROCESSORS cube(2,1,1)                                    (1)
!hpf$ DISTRIBUTE image(BLOCK(4),BLOCK(8)) ONTO squad
!hpf$ DISTRIBUTE im1(BLOCK(1),BLOCK(4),BLOCK(8)) ONTO cube     (2)
!hpf$ ALIGN im1(i,*,*) WITH image((i-1)*4+1,1)                 (3)
!hpf$ INDEPENDENT
forall(i=1:8) line(i)=i
!hpf$ INDEPENDENT
```

```
forall(i=1:8,j=1:8) image(i,j)=i
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(image,1)/bsize
im1(i,:,:)=image(bsize*(i-1)+1:bsize*i,:)
end do
!hpf$ INDEPENDENT, NEW(i)
do i=1,size(im1,1)
im2(i,:,:)=transpose(im1(i,:,:))
end do
!hpf$ INDEPENDENT, NEW(i,i1,i2)
do i=1,size(im1,1)
forall(i1=1:size(im2(i,:,:),1),i2=1:size(im2(i,:,:),2))
im2(i,i1,i2)=im2(i,i1,i2)**2
end forall
end do
!hpf$ INDEPENDENT
forall(i=1:4,j=1:8) x0(i,j)=0
im3(:,:)=x0
!hpf$ INDEPENDENT, NEW(i), REDUCTION(im3)
do i=1,size(im1,1)
im3(:,:)=im1(i,:,:) + im3(:,:)
end do
!hpf$ INDEPENDENT
forall(i=1:8) x1(i)=0
!hpf$ INDEPENDENT,NEW(i,i1,i2)
do i1=1,size(im1,1)
im4(i1,:)=x1(:)
!hpf$ INDEPENDENT,NEW(i1), REDUCTION(im4(i1,:))
do i=1,size(im1(i1,:,:),1)
im4(i1,:)=im1(i1,i,:) + im4(i1,:)
end do
end do
end program fprog
```

That program was compiled on IBM SP with the HPF compiler `pghpf`, Rel 2.4, and started on two processors. It looks like the compiler does not handle row (2) of the program correctly. If we replace row (2) with the rows (1) and (3) the program gives the same results as the corresponding expressions in *Mathematica*.

6 Conclusion

We have shown how we can translate in a natural way expressions of a functional language to HPF statements for parallel execution. The translator presented is implemented in *Mathematica*. It translates second order functions, pure functions and some procedural constructions for data manipulations. Ideas for translating function calls and compositions of functions had been also shown. In the translator should be added some more features and one of them is the generation of the FORTRAN 95 definitions of the structures used in the translated code. The translator could be programmed to respond to some contradictions during the translation. It should be pointed out how far we were able to go with this translator without many efforts and the translator in its current stage can be used for generating tedious HPF code from some very concise *Mathematica* expressions. If the translator is made correctly we will have the additional benefit that the code generated would be without errors.

Acknowledgments: This paper is based on the author's report for the Ph.D. course "High-Performance Programming", held at DTU, Denmark, organized by Prof. Per Christian Hansen during the period Sep-Dec, 1998. More information about the course can be found at the site http://www.imm.dtu.dk/~pch/hpp.html. The lectures were held by leading experts in high-performance computing: Prof. Jack Dongarra, Dr. William Gropp, Dr. Sven Hammarling, Prof. Nicholas J. Higham, Prof. Bo Kågström, Dr. Ramesh Menon, Prof. John Reid, Dr. Robert Schreiber. Prof. Stig Skelboe and Prof. Per Christian Hansen gave some introductory

lectures. The ideas in this paper was inspired by the lecture of Dr. Ramesh Menon for Open MP, and the lecture of Dr. Robert Schreiber for HPF. I consider myself lucky to have had the opportunity to attend the course therefore I am very grateful to Prof. Per Christian Hansen for organizing it.

Appendix

In this appendix we define category, cartesian category and cartesian closed category as Field and Harrison in [3], Chapter 13, Section 4.

A *category* is a collection of objects, $obj(\underline{C})$ such that

1. Given $A, B \in obj(\underline{C})$, here is a collection of arrows from A to B, denoted by $A \to B$, and if f is in $A \to B$ we say that th domain of f is A, and the codomain of f is B, or $dom(f) = A$, $cod(f) = B$.
2. For all objects A, B, C there is a composition operation, $\circ$, defined on the arrows, which is associative, i.e. if f is in $B \to C$ and g is in $A \to B$ then $f \circ g$ is in $A \to C$, and $(f \circ g) \circ h = f \circ (g \circ h)$ for all arrows f, g, h such that $cod(h) = dom(g)$, $cod(g) = dom(f)$.
3. For each object A there is an identity arrow, id^A in $A \to A$, such that for all arrows f, g with $cod(f) = dom(g) = A$, $id^A \circ f = f$ and $g \circ id^A = g$.

In a *cartesian category*, $\underline{C}$, there is a product construction, defined as follows:
For all objects A, B there exists an object $A \times B$ and arrows $Fst^{A,B} : A \times B \to A$, $Snd^{A,B} : A \times B \to B$, called projections, with he property that any object C and arrows $f : C \to A$, $g : C \to B$, there exists an unique arrow $C \to A \times B$, denoted by $\langle f, g \rangle$ and called a *pair* of f and g, with the property that $f = Fst \circ \langle f, g \rangle$, $g = Snd \circ \langle f, g \rangle$.
The *cartesian closed category* is cartesian category in which for all objects A, B, there exists an object, $A \Rightarrow B$, and an arrow $App^{A,B} : (A \Rightarrow B) \times A \to B$, called application, with the property that for any object C and an arrow, $f : C \times A \to B$ there exists an unique arrow $C \to A \Rightarrow B$, denoted $\Lambda(f)$, and called *curring* of f, such that $f = App^{A,B} \circ \langle \Lambda(f) \circ Fst, Snd \rangle$.

The object $A \Rightarrow B$ represents the *functional space* defined on A and B, and the pair $\langle A \to B, App^{A,B} \rangle$ defines the *exponential* of A and B. The functional space is the set of all functions from A to B. If $c \in C$, $\Lambda(f)(c)$ is a function which for $a \in A$ is $f(c, a)$ i.e. $\Lambda(f)(c)(a) = f(c, a)$.

References

[1] Michael Barr, Charles Wells, Category Theory for Computer Science, second edition, Prentice Hall 1996.
[3] A.J.Field, P.G.Harrison, Functional Programming, Addison-Wesley, 1988
[4] Wilhelm Gehrke, Fortran 95, Language guide, Springer 1996
[5] High Performance Fortran Forum, High Performance Fortran Language Specification, January 31, 1997 ,Version 2.0
[6] Pekka Janhunen, Fortran Definitions, Finnish Meteorological Institute Geophysics Dept. 14.06.90., available on http://www.mathsource.com/Content/Enhancements/Interfacing/Fortran/0202-172
[7] Robert Schreiber, High Performance Fortran, Version 2, Parralel Processing Letters Vol.7 No.4(1997) 437-449