On High-Level Low-Level Programming

Philippe Gerner
gerner@icps.u-strasbg.fr
LSIIT-ICPS, CNRS
Université Louis Pasteur
Strasbourg, France

Abstract

The current solution for efficient high-level parallel programming in the industry is to use *directives to the compiler*. However these directives pose two problems: first, they are often designed in an ad hoc manner and their subtleties are less easy to understand than the rest of the language; second, the degree of stringency of the directives is not fixed, so that evaluating the efficiency of the directives he writes is not easy for the programmer. This article proposes a methodology for addressing these issues. The data mapping directives of the language High Performance Fortran are used as an example; in particular, it is shown how the use of formal semantics can help clarify and structure the issues.

1 Introduction

The primary motivation for parallel programming is efficiency at execution (in terms of execution time). But in the current state of the art of parallel programming, "high-level" still implies "not efficient enough". In response to this, in the 1990's the notion of a *directive to the compiler* emerged, as in the language HPF (High Performance Fortran [19]) and the OpenMP library (e.g., [3]). The idea is for the programmer to write high-level code, yet help the compiler into finding an efficient parallel implementation for the code. The notion of a directive to the compiler is not new. Yet what is new is its importance in a programming language. Indeed, one may say that the essence of HPF and OpenMP is the language of directives they provide.

One problem with the directives is that as they touch on low-level issues, their meaning depends onto notions which are more implicit than explicit in the reference manual of the language. Thus their understanding by the programmer is not easy, and the meaning of the directives tends to be learned "the hard way".

Another problem is that more often than not, the directives are just advices to the compiler, so the programmer has to make special tests and use profilers in order to "learn his compiler", so as to write useful directives. This can lead him into writing directives that do produce efficient code only for his particular compiler. Thus non-portability of efficiency is introduced.

A solution to these two problems would be a documentation of directives which is both clear and explicit as to the constraints they put onto the compiler writer.

However, the current form of reference manuals inherits the traditional wisdom which says that it is more important to describe *what* a written program does, than *how* it actually does it. Thus notes as to the "how" tend to be relegated to "advice to implementors". However, when performance is an issue, which is often the case even in sequential programming, it is clear that these comments are eagerly read by the programmer as well. Thus we are interested in how a reference manual should be structured for having the "how" as first-class semantics, at the same level as the "what".

For structuring the methodology, we propose to use a formal semantics as a support. How to use formal semantics for lessening the ambiguities in a reference document is still an area of research. When used by a language designer, formal expression tends to induce coherence in the semantics of the language, as remarked by Milner [27]: we believe that, similarly, a formal framework for the notion of directives could help him construct a coherent language of directives.

Our methodology is illustrated with an exposition of the meaning of some of the key directives of the language HPF.

The paper is structured as follows.

In the next section, the function of the reference manual is discussed. Then the use of formal semantics is discussed in the light of low-level-conscious programming. Finally the case of the directives to the compiler is introduced, with a focus on directives in parallel programming.

The third section presents a formalisation of the semantics of some of the key data mapping directives of HPF. It shows that a formalization can help into both clarifying the semantics of the directives and structuring the exposition of the language in its entirety.

2 Reference and Actual Semantics

2.1 The Reference Manual

The semantics of industrial programming languages is defined by their *reference manuals*. As we concentrate on how the reference manual should provide the information as to the meaning of the directives, we first discuss the exact function of this manual.

A reference manual for a language L has two types of readers:

- the programmers in L, who refer to it is case of doubt as to the precise meaning of a particular construct.

- the implementers of a compiler for L, for whom the reference manual defines the precise meaning of each construct of the language.

(One may also consider a third kind of readers: the committee members for a standardization of the language. The document they eventually produce, which defines the norm, is "the ultimate reference". For simplicity, when the language is standardized, what we call "reference manual" is the document which defines the standard.)

The purpose of a reference manual is to ensure that for any two compilers C and C' for a language L, a program written in L will produce the same output data, for given input data, whether it has been compiled with C of with C'. Thus, L code can be portable. That is why "a standard is often described as 'a contract between the programmer and the implementer.' It describes not only what is "legal" source text, but also what a programmer can rely on in general and what behavior is implementation-dependent" [34, p. 81].

Because achieving portability is so important, the focus of reference manuals is on semantics matters being very clear, not on efficiency considerations. As says the draft for the C99 standard: "The semantic descriptions in this International Standard describes the behaviour of an abstract machine in which issues of optimizations are irrelevant." [21, p. 15].

Thus, the compiler writer is free to produce any actual semantics he likes, as long as he respect the few primary requirements, the main among which is, for C99: "At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced." [21, p. 15].

Of course, "an implementation might define a one-to-one correspondence between abstract and actual semantics," [21, p. 16], but the point is that is it not at all mandatory.

2.2 High-Level Low-Level Programming

In rapid prototyping, performance issues are not considered because only the functionality of the program (in the sense of "what is does") is important. But in other programming contexts, performance becomes an issue. An extreme case is scientific computing. As said about the C++ Matrix Template Library (MTL [1]): " To many scientific computing users, [...] the advantages of an elegant programming interface are secondary to issues of performance."

As a consequence, high-level programming languages provide some means to achieve performance "in spite of the high level". An example is the introduction of imperative features in modern functional languages, as in the Caml language [25]. In fact, the degree of "low-level coding" in a program tends to be proportional to the "highness" of the language. For example, cut-free Prolog programs are more a rarity than the norm (the "`cut`" is a kind of directive to the interpreter [5, pp. 69-92], and is an essential tool for the Prolog logic programmer).

The Documentation on Performance Issues

The reference manual of the language documents some of the low-level facilities, but not necessarily in performance terms, since the reference semantics, for abstraction reasons, does not provide the relevant notions. Witness the reference semantics for the C `register` directive: "A declaration of an identifier with storage-class `register` suggest that access to the object be as fast as possible" ([21, p. 98]. No mention is made of the notion of a register, so that no notion of a register need be defined for the abstract machine, which is understandably convenient. Thus the real meaning of this directive is in fact implicit.

Moreover, there is often a non-certitude as to how the low-level feature is actually handled by the compiler. For the `register` directive, for example, the norm is that "the extent to which such suggestions are effective is implementation-defined" (ibid.). Indeed, the compiler may decide he has more efficient optimizations for this variable than putting it into a register.

The result of this state of affairs is that the programmer will tend to infer the effect of its low-level investment from interaction with his compiler. Sometimes he will even write programs "just for testing what the compiler does". Thus, he comes to know about the actual semantics of his programs more from the compiler than from the reference manual, and he is at risk of spending some time on some compiler-dependant performance tuning, which is bad investment when portability is considered. (Of course he can also introduce in this way some compiler dependency which does not even concern efficiency.)

2.3 Formal Semantics and Actual Semantics

For eliminating as much ambiguity as possible from the reference description, since the 1970's much research has been made for formally specifying the semantics of a programming language. But the definition of real programming languages is still written in natural language only (except for the grammar part). The only convincing exception we know of is the definition of Standard ML [27].

In the 80's, Yuri Gurevich introduced what became known as the *Abstract State Machines* [17], and Peter Mosses *Action Notation* [30]. Both put the emphasis onto description capability rather than onto adequacy for proving program correctness, and some of the more convincing language descriptions have used these methods, see, e.g., [22] (abstract state machines) and [18] (action notation).

But all these methods are designed for describing *one* semantics. But for describing the semantics of directives, we need to describe both the reference and the actual semantics, and the relation between them. Thus in the next section we will provide a method of our own.

The research which comes closer to our preocupation is in the field of *compiling research*. For example, work on correct compiling uses the notion of respecting the input-output behaviour of a program while changing its execution mode; but it focuses on adequacy for proving the correctness of a change, e.g., [23]. Also relevant is the *automatic parallelization* field, which provides an algebraic formulation of execution change through the notion of a *space-time mapping* [10]. But it is strongly tied to affine dependances, and too technical (it's integer programming) for being adequately used in a reference document to be used by any programmer.

2.3.1 Operational, Denotational

What is called the "*operational semantics*" of a programming language describes the computation mechanism which corresponds to a program (see, e.g., [32]), whereas *denotational semantics* [28] describes the relation between the input and output data of the program. It is natural to consider that the "essence" of a program is in how it works, rather than into what it does. But from the point of view of the function of the reference semantics, things are different. Indeed, in reference manuals, as for C99 above, the reference operational semantics is only a *descriptive means* for expressing which input-output behaviour the actual semantics of a program must have: this input-output behaviour is what must be preserved by the actual semantics, and there is no necessary relationship between the reference operational semantics and the actual semantics. Thus, from the point of view of the compiler, the reference operational semantics is just another way of expressing a denotational semantics. Operational semantics is a good *didactic* means to express the intented input-output behaviour of a program, as humans reason naturally in terms of temporal mechanisms. By constrast, denotational semantics is didactic only for (the "pure" part of) functional languages. A denotational semantics of C, for example, will not help the programmer much in understanding C.

2.4 Directives to the Compiler

Directives to the compiler are an interesting form of high-level low-level programming, because instead of tuning his algorithm in its very expression, the programmer keeps intact his algorithm, and just *adds* some instructions (the directives) for tuning its actual execution.

Directives exist in most real programming languages. They have been in use in academic circles also. For example, the Bulldog VLIW compiler [9] allows the programmer to indicate some invariants to the compiler. Also, the compiler for the functional language Opal [8] allows the programmer to indicate some algebraic properties for some of the functions in the program, which the compiler can use for optimization. Even the algebraic specification system *OBJ* [15] has a directive `memo`, for memoïzation.

2.4.1 Directives for Parallel Programming

As said in the introduction, directives to the compiler have found a new dimension in high-level parallel programming.

However HPF or OpenMP programming is still not the most frequent practice in parallel programming. Indeed, MPI (Message Passing Interface [16]) is still the choice when high performance is crucial. MPI programming is more low-level than HPF and OpenMP, as the code for the communication of data between the parallel tasks has to be written by the programmer, as opposed to the compiler for HPF and OpenMP. The reason for the persistence of the primacy of MPI is that HPF and OpenMP both are a bit ahead of their time in terms of compiler technology (see, e.g., [6]). That is, some of their directives can be so difficult to handle for the compiler, that the compiler might simply ignore them, not knowing how to handle them properly. That is why HPF programmers use debuggers like TotalView of profilers like pgprof for seeing "how our particular compiler has used the directives".

However, for the parts of the program for which the programmer does not provide some directives, the compiler has no less hard a job. This is well expressed by the fact that the task of providing the best compromise between optimal scheduling of the instructions and optimal data distribution for communication minimization is NP-complete. So directives can help the compiler a lot.

3 Semantics of the HPF Directives

We propose a methodology for defining the meaning of the directives to the compiler, in the context of their description by a reference manual.

Our method uses a formal description of the semantics of the language as a support for structuration and clarity of the reference manual. We mentionned above that no formal semantics method known to us has been designed specifically for expressing the relationship between reference and actual semantics. So in this next section we use a formalization of our own, based on *rewriting logic* [26] (see below).

We explain our method through applying it to the case of the HPF data mapping directives. However, the method is not dependent upon the HPF language. What a formal support brings in clarity is shown by how the HPF *template* construct is handled.

3.1 The HPF Data Mapping Directives

The idea of the data parallel programming model which is followed by HPF is to organize the program as a sequence of parallel operations, where a parallel operation is the concurrent application of a same operation onto a bunch of data. For example :

```
C = A + B
```

where `A`, `B`, and `C` are three arrays, is a parallel operation.

Notice that the fact that it should execute in parallel (which would justify the relevance of this programming model for performance) is not discussed in the HPF reference manuals [19, 11]. One possible reason is that it probably considered "obvious" that HPF array assignments should be made in parallel if possible. Another is that HPF itself does not define array assignments: it imports them from Fortran 90 for HPF version 1, or from Fortran 95 for HPF version 2. About operations on arrays, the Fortran 90 Standard says: "these features can significantly facilitate optimization of array operations on many computer architectures" [20, p. xiii]. So the particular interpretation of the array assignments in HPF, as being more on the level of actual semantics than the reference one, has not been emphasized.

However such parallel operations are not enough for efficiency. If a distributed memory parallel machine is used, then the elements of the arrays are distributed onto these memories, and a random distribution will likely generate more inter-processor communications (for requiring the data which are needed for the computations) than a thoughtful one. As communication time is not negligeable, performance of the program depends on minimizing the number of communications. Thus an important job for the compiler is to find an efficient ditribution of the data onto the memories. This is no easy task, and that is why the HPF model provides the programmer with the means to write some *data* *mapping directives*. An example HPF data mapping directives is given below (this toy example will be used throughout the section):

```
      INTEGER I, J, K, L, M, N
      INTEGER, DIMENSION(4) :: A, B, C, D

!HPF$ TEMPLATE T(4)
!HPF$ ALIGN A(:) with T(:)
!HPF$ ALIGN B(:) with T(:)
!HPF$ ALIGN C(:) with T(:)

!HPF$ PROCESSORS, DIMENSION(2) :: MY_PROCS
!HPF$ DISTRIBUTE (BLOCK) ONTO MY_PROCS :: T

      K = I + J
      N = L + M
      C = A + B
      D = C + N
```

The data mapping directives are the lines beginning with "`!HPF$`". Their meaning is exposed below, together with the formalization of this meaning. In this example, array `D` has not been aligned. This is not an error, as fortunately HPF does not constrain the programmer to give a data mapping directive for every array in the program.

3.2 The Choice of Rewriting Logic

As a tool for formally defining the meaning of the directives, we use rewriting logic [26]. Rewriting logic (RWL) is algebraic specifications plus transitions. As for algebraic specifications, the semantics of RWL is given through the notion of *satisfaction*, inherited from model theory [33]: the semantics of a RWL theory is given as the set of its *models*. The difference with classical algebraic specifications is that a model is not a family of sets together with functions between them, but a family of *categories* [24] with *functors* between them. More on these models can be found in appendix A, but the following of this section is self-contained.

Our choice of the RWL framework is motivated by the following factors:

- theory morphisms help structure the semantics [14]. This is used for defining the meaning of the language without directives, and then adding directives in a way that just "enrich" the previous definition.
- notion of satisfaction allow coding each directive as an *equation* which can be satisfied or not by an algebra.
- a category (in a model) is defined by objects and arrows, so that categories have a natural *graphical* quality, which is interesting when we want to include a formal semantics as a digestible part of a reference manual.

The Readers of a Formal Description As programmers are not in general theoretical computer scientists, they will probably not consider that a rewriting logic description provides a clarification of the semantics of the language. In the case of HPF, which is used for scientific computing, the programmers often are not even professional programmers, but rather users of numerical computing like theoretical physicists, engineers, or meteorologists.

Notice that this problem is not restricted to RWL descriptions. Action Semantics, for example, has its operational semantics defined in [29] with the Structured Operational Semantics method [32], and a programmer will probably won't buy this document in order to better understand the programming language he's using, as the notation is intuitive enough (by its very purpose).

The point is that, like Action Semantics or Abstract State Machines, we make a difference between an accessible notation and the "pure" model. Our model below is at the level of rewriting logic proper, and a complete methodology would provide a notation based on this model.

3.3 Coding the Language Semantics

We describe a subset of HPF (which we call mini-HPF) which is tiny but has the key data mapping directives.

The semantics of the language is given by assigning a precise meaning to each program, rather than to each language construct as is usually done. The semantics of a particular program is a given as a *rewrite theory*, which uses some predefined theories which do not depend on a particular program. The description of the tool which does the translation from a mini-HPF program to its assigned RWL theory would correspond to a description of the semantics of mini-HPF. However, for making the presentation more digestible, we prefer below to illustrate this translation on a particular instance of a program.

3.4 Reference Operational Semantics

The following RWL theories in this section are written using the well-defined *CafeOBJ* notation [7]. This notation is used by the *CafeOBJ system* [31], a successor to the well-know OBJ specification system [15].

We first define the reference operational semantics of our HPF program, when directives are not taken into account yet. It is given by the following theory, which is the translation, in our methodology scheme, of the algorithm of the HPF program.

```
module! ALGO1 {
  using (MHPF-SEM)
  ops I J K L M N : -> SName
  ops A B C D     : -> ArrayName
  eq length A = 4 .  eq length B = 4 .
  eq length C = 4 .  eq length D = 4 .
  op algo1 : -> Algo
  eq algo1 =  (K := I + J) ;
              (N := L + M) ;
              (C := A + B) ;
              (D := C + N) .
}
```

This theory imports (with the CafeOBJ keyword `using`) another theory, `MHPF-SEM` (for "semantics of mini-HPF"). `MHPF-SEM` defines the reference operational semantics of (a tiny subset of) HPF, and it imports itself the theory `MHPF-SEM-CONSTRUCTS` which defines the sorts and operations to be used in the underlying rewrite theories of mini-HPF programs, as in `ALGO1`. `MHPF-SEM-CONSTRUCTS` is defined as follows:

```
module MHPF-SEM-CONSTRUCTS {
  [ SName ] [ SAssignt ]
     op _:=_+_  :
```

```
   SName SName SName      -> SAssignt

  [ ArrayName ] [ ArrayAssignt ]
  protecting (INT)
     op length_  : ArrayName -> Int
     op _:=_+_  :
   ArrayName ArrayName SName      -> ArrayAssignt
     op _:=_+_  :
   ArrayName ArrayName ArrayName  -> ArrayAssignt

  [ SAssignt ArrayAssignt < Assignt < Algo ]
     op noAlgo  :               -> Algo
     op _;_     : Algo Algo   -> Algo
                 { assoc id: noAlgo }
}
```

The common semantics basis for each mini-HPF program is defined by theory `MHPF-SEM`:

```
module* MHPF-SEM {
  protecting (MHPF-SEM-SYNTAX)

  [ MachineName ]
  op M : -> MachineName

  [ MachineState]
  op (_wTrace:_wAlgo:_) :
      MachineName Algo Algo -> MachineState

  vars tr algo : Algo   var assgnt : Assignt
  trans M wTrace: tr wAlgo: (assgnt ; algo)
     => M wTrace: (tr ; assgnt) wAlgo: algo .

  op (_.val_)   : MachineState SName          -> Int
  op (_.val__)  : MachineState ArrayName Int  -> Int

  ...
}
```

The dots at the end are for the semantics of assignment (that is, defining that executing "`_:=_+_`" does make the intended addition). This can be looked at in appendix B.

The theory defines *machine states*. Operation "`_wTrace:_wAlgo:_`" says that a machine state is characterized by a machine name and two algorithm texts. There is only one machine name in our theory: `M`. It is there to make the specification easier to read. The first algorithm text is a trace of what the machine has already executed, and the second one the algorithm which is yet to be executed by the machine.

Execution of the first assignment from the algorithm to be executed is coded by the unique transition of our whole specification, from one machine state to another: `trans ....` This transition makes the assignment instruction go "through the barrier of the "`wAlgo:`" and be appended to the trace: this represents execution of the instruction.

The transitions between machine states from the assignment instructions follow the order of these intructions in the algorithm. This is illustrated below by how in the text of the HPF program becomes, in a model, arrows from machine states to machine states:

$$\begin{tikzcd} z_0 \arrow[d, "\{\mathsf{a}_1\}"] \\ z_1 \arrow[d, "\{\mathsf{a}_2\}"] \\ z_2 \arrow[d, "\{\mathsf{a}_3\}"] \\ z_3 \arrow[d, "\{\mathsf{a}_4\}"] \\ z_4 \end{tikzcd}$$

where $z_0 = [\![$`M wTrace: noAlgo wAlgo: algo1`$]\!]$,
"a_1," "a_2," "a_3" and "a_4" are names for the following terms of sort `Assignt`:

$\mathsf{a}_1 =$ `K := I + J`,
$\mathsf{a}_2 =$ `N := L + M`,
$\mathsf{a}_3 =$ `C := A + B`,
$\mathsf{a}_4 =$ `D := C + N`;

and "$\{\mathsf{a}_i\}$" as an arrow name indicates that the transition is generated by this assignment instruction a_i going through barrier "`wAlgo:`". Thus, machine state z_5 is:
$[\![$`M wTrace:` a_1 ; a_2 ; a_3 ; a_4 `wAlgo: noAlgo`$]\!]$

We must complete the description of the reference semantics by explicitating the meaning of the above formalization. Indeed, in algebraic specification, the (most often infinite) set of models of a theory does not provide in itself any meaning. Thus, a *meaning attribution* has to be made onto them. (A treatment of an example of such meaning attribution can be found in [12].) In our method we take care to explicitate which meaning is to be attributed to the models. Thus, our reference semantics says that:

- sequence in the diagram (following the direction of the arrows) corresponds to sequence in time at the execution;
- every transition that involves a assignment of sort `ArrayAssignt`, like a_3 above, has to be imagined as involving parallel computations on the scalar values contained in the right-hand side arrays, and then parallel assignment to the stores of the left-hand side array. For example, for $\mathsf{a}_3 =$ `C := A + B`, there are 4 computations (because the arrays have length 4): $[\![\mathtt{.val}]\!](z_2, [\![\mathtt{A}]\!], 1) + [\![\mathtt{.val}]\!](z_2, [\![\mathtt{B}]\!], 1), \ldots,$ $[\![\mathtt{.val}]\!](z_2, [\![\mathtt{A}]\!], 4) + [\![\mathtt{.val}]\!](z_2, [\![\mathtt{B}]\!], 4)$. There are executed in parallel. Similarly, the resulting values are stored in parallel into the 4 stores of array `C`.

3.5 Actual Operational Semantics

For modeling in a simple way the relationship of the reference semantics to the actual one, we remark that the documentation on the directives has to say something about notions that belong to the actual semantics, like processors. So the documentation must provide an *abstraction* of the actual semantics. A formal support is given for this with, again, the notion of a *theory*.

We provide a theory of virtual and real processors:

```
module* PROCS-THEORY {
  [ Proc ]
  protecting (INT)
  op nbProcs : -> Int

  [ VProc ]
  op map_ : VProc -> Proc

  [ TemplateName ]
  [ ProcArrayName ]
  op <_,_>   : TemplateName  Int -> VProc
  op <_,_>   : ProcArrayName Int -> Proc
}
```

The meaning which is to be attributed to this theory is now described.

Sort `Proc` declares some *processors*. These are the processors of the parallel machine onto which a HPF program will execute. `nbProcs` designates the number of processors actually used by the program.

The notion of a *template* has been introduced with the HPF directives model. The reference manual says about this construct: "a template is simply an abstract space of indexed positions; it can be considered as an 'array of nothing' (as compared to an 'array of integers', say)" [11, p. 40]. This notion is not an easy one conceptually. In our model we manage to avoid the notion of an "array of nothing" by distinguishing between the notion of a template (a name plus some integer indices) and array proper (a name plus a "`.val`" operation).

As shown in our HPF program example, a template is to be "distributed" onto processors. As a template contains no values, it is as if no data is distributed. The notion of a *virtual processor* helps explain the scheme: a couple formed with a template name and an integer refers to a virtual processor, and the "distribution" is then simply a mapping of some virtual processors onto the real ones. Sort `VProc` declares a sort of virtual processors. That each virtual processor will eventually be mapped onto a real one is expressed is declared with operation `map`.

Real processors are refered to with a scheme similar to that used for the virtual processors, with a name of sort `ProcArrayName` and an integer. The complete virtual/real processors-plus-mapping scheme serves in HPF to express the respective alignment of data: this is illustrated below.

Remark Our real processors correspond to what the HPF documentation calls *abstract processors*, as contrasted with the *physical processors* of the machine. This is in accord with our method, since our theory of real processors is an abstraction from the notion of physical processors.

3.5.1 Alignment and Distribution

Now we can show which RWL theory the mini-HPF program complete with directives is translated into:

```
module! ALGO1-ALIGNTS-DISTRD {
  protecting (ALGO1)
  using (PROCS-THEORY)

** alignments:
  [ ArrayName < TemplateName ]
  op T : -> TemplateName
  var i : Int
  eq < A, i > = < T, i > .
  eq < B, i > = < T, i > .
  eq < C, i > = < T, i > .

** distribution:
  op my-procs : -> ProcArrayName
  eq nbProcs = 2 .
  eq map < T, i > =
         < my-procs, ((i + 1) quo (nbProcs)) > .
}
```

That is, it uses the translation of the program without the directives, imports the theory of virtual and real processors, and then translate the directives proper.

Care is taken, with keyword `protecting`, to indicate that the models of the previous theory `ALGO1` are not modified by the new theory. That is, we garantee that the new theory just *adds* some new information "around" the preceding theory: we don't modify the operational semantics of the program as given in `ALGO1`. For example, the above drawing with four arrows is still valid.

Alignment First, array names are considered as template names ("`ArrayName < TemplateName`"). This means that for any array name X and integer i, $[\![\texttt{<\_,\_>}]\!](X,i)$ is a virtual processor. The intended meaning is that this virtual processor has a (virtual) memory attached to it, which contains the store reserved for the value $[\![\texttt{.val}]\!](S,X,i)$ (on any machine state S). It is also intended that this value will eventually be stored onto the memory attached to the real processor $[\![\texttt{map}]\!]([\![\texttt{<\_,\_>}]\!](X,i))$ onto which the virtual processor is mapped. Thus, the virtual/real/map scheme serves to express the distribution of data.

The three HPF `ALIGN` directives are translated as three equations about virtual processors. By default, every virtual processor $[\![\texttt{<\_,\_>}]\!](X,i)$ is distinct from virtual processor $[\![\texttt{<\_,\_>}]\!](Y,j)$, when $X \neq Y$ or $i \neq j$. This default rule holds because the so-called *initial semantics* for the models of the theory is chosen (this is a classical tool of algebraic specification): this choice is indicated by the CafeOBJ code for it—the exclamation mark "`!`" attached to the declaration of the theory: "`module!`". Thus the equations serve to state the equality of some virtual processors which would otherwise be distinct. For example, virtual processors $[\![\texttt{< A, 1 >}]\!]$ and $[\![\texttt{< T, 1 >}]\!]$ are equal.

As it is also the case that $[\![\texttt{< C, 1 >}]\!] = [\![\texttt{< T, 1 >}]\!]$, by transitivity we have $[\![\texttt{< A, 1 >}]\!] = [\![\texttt{< C, 1 >}]\!]$. Thus, for any machine state `S` values $[\![\texttt{S .val A 1}]\!]$ and $[\![\texttt{S .val C 1}]\!]$ are associated with the same virtual processor. This means that they will eventually be associated with the same real processor and be stored onto its associated memory.

Thus we see how HPF template names which are not array names (as `T` here) are just a facility for puting the values of different arrays onto a same memory, which is called "aligning some data".

The motivation for aligning is minimizing communications. The point is that we explicitely assume that our machine follows the so-called *Owner Computes Rule*: if an array element has been assigned a particular memory, then its computation is made by the processor which is attached to this memory. Thus, when instruction `C := A + B` is executed, the value of `C` at index 1 is computed (see appendix B for the formal expression) using the values of `A` and `B` at index 1. As the value for `A` is on the same abstract processor, no (virtual) inter-processor communication is required.

Distribution The semantics we give to the `PROCESSORS` directive of our example is that is declares a `ProcArrayName` name, `my-procs`. The number of processors is declared equal to the size of the HPF array `MY-PROCS`.

The distribution by blocks of the `DISTRIBUTE` directive is translated into its meaning: an equation which puts a constraint onto function $[\![\texttt{map}]\!]$.

The number of processors declared by the `PROCESSORS` directives *does* correspond to the actual number of processors used to run the program [11, p. 38], so that from alignments onto a template to distribution onto an array of processors we have reached *a faithful abstraction of what actually happens.*

Remark 1 The `DISTRIBUTE` construct also allows to distribute an array, like `A` above, *directly* onto an array of processors. The meaning we attribute to that it is `A` *taken as a template* (through "`ArrayName < TemplateName`") which is distributed.

Remark 2 The HPF `DISTRIBUTE` directive allows in fact to have a virtual processor mapped onto *several* processors. We have simplified above for the sake of a short exposition. Extension to a codomain for `map` whose elements are the subsets of the set of processors is straitforward.

3.5.2 Degree of Stringency

"HPF directives appear as structured comments that suggest implementation strategies or assert facts about a program to the compiler. When properly used, they affect only the efficiency of the computation performed, but do not change the value computed by the program" [11, p. 4]. Thus, if the semantics of HPF is to be described, a degree of uncertainty as use of the directives by the compiler has to be coded.

This is the subject of future work. It will use models that satisfy only some of the equations that correspond to the directives. It will be based on some existing works like [13].

3.6 Other Data Mapping Models

We have given above a model of data distribution. Other models have been proposed in the literature. For comparing them to our model, we find it convenient to re-use the notion of virtual processors and templates as defined in our model.

- The language C* [35], the only "real" language in this list, is an adaptation of the C language for the Connection Machine. Its data distribution relies on the notion of a *shape*, that corresponds to a template in our model. Aligning two arrays `A` and `B` can only be done by having them have the same shape: so they must have the rank and size. This distribution is at the level of virtual processors, and mapping the virtual processors onto the real ones is left to the compiler.
- Luc Bougé's language $\mathcal{L}$ [4] is based on arrays, but more like HPF than C*. Each index in an array indentifies a value *and* a processor. Thus, this model works also at the level of virtual processors. There is no alignment of arrays.
- The model PEI [36] has a notion of *data field* that corresponds to an array distributed onto a global (non-array) template. Thus the distribution is at the level of virtual processors. There is a single global template, and *every* array is aligned with it. The respective alignment of the arrays with this template is induced automatically by any operation on arrays. For example, an assignment `C = A + B` automatically induces an alignment of the three arrays. A drawback of this automatic alignment is that any non-trivial program soon involves contradictory alignments. As the elements of the global template are not allowed to be indentified for resolving the contradiction, the only way to resolve it is to express the algorithm differently.

Remark Reflecting on the example of PEI, one might ask if it would not be possible to write contradictory distributions in HPF (as two real processors can't be equal). In fact, HPF resolves this in a simple way: it is forbidden to align or distribute an array more than one time.

4 Conclusion

We have described a methodology for including a proper description of the meaning of directives to the compiler into a reference manual. We have used rewriting logic as a formal framework: this allowed us to describe the actual semantics as a theory that enriches the theory which codes the classical reference semantics; to give a clear account of the notion of a template; and to code each directive as an equation in the enriched theory. In this model, respect of a directive corresponds to an algebra satisfying the corresponding equation. Full formalization of the degree of stringency for satisfaction of the directives by the compiler is the subject of future work, and will follow this framework.

While use of parallel machines and environments develop, high-level programming becomes all the more needed for portability reasons; but efficient use of these machines and environment then can't be left to the compiler alone, so the directives to the compiler are in the present state of the art of parallel compiling the most promising solution. The advent of the computational GRID [2] will bring a similar situation, as parallel programming for the GRID should be high-level and at the same time exploit efficiently the GRID architecture. If ad hoc directive constructs and ad hoc explanations of them in the documentation are to be avoided, a proper theory of the combination of high-level and low-level notions in a programming model has to be developped. Directives to the compiler (or execution environment) provide a separation of concerns between the two levels while relating them, so the concept of a directive could have a rich future in computing.

Acknowledgments I have benefited from discussions with Michel Salomon on the meaning of the HPF directives; Éric Violard on data parallelism and the programmer-compiler contract; and Pascal Schreck on algebraic specifications.

References

[1] `http://http://www.osl.iu.edu/research/mtl/`.

[2] *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999. Edited by Ian Foster and Carl Kesselman.

[3] *OpenMP Fortran Application Program Interface*, November 2000. Version 2.0.

[4] Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: Semantics and implementation. *Future Generation Computer Systems*, 8:363–378, 1992.

[5] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, third edition, 1987.

[6] Fabien Coelho, Cécile Germain, and Jean-Louis Pazat. *State of the Art in Compiling HPF*, volume LNCS 1132, pages 104–133. Springer-Verlag, May 1996.

[7] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. Submitted to *Theoretical Computer Science*, 2000.

[8] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and implementation of an algebraic programming language. In *Programming Languages and System Architectures*, pages 228–244, 1994.

[9] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1985. YALEU/DCS/RR-364.

[10] Paul Feautrier. *Automatic Parallelization in the Polytope Model*, volume LNCS 1132, pages 79–103. Springer-Verlag, May 1996.

[11] HPF Forum. *High Performance Fortran Language Specification*, 2.0 edition, January 1997.

[12] Philippe Gerner. A note on computational meaning attribution in rewriting logic. Technical Report ICPS RR 02-07, Université Louis Pasteur, France, May 2002. `http://icps.u-strasbg.fr/pub-02/rr-07-02.ps`.

[13] Joseph Goguen. An introduction to algebraic semiotics, with application to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphor, Analogy and Agents*, LNAI, Vol. 1562, pages 242–291. Springer Verlag, 1999.

[14] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.

[15] Joseph Goguen, Timothy Winkler, José Meseguer, Prof. Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Grant Malcolm, editor, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[16] William Groppa, Ewing Lusk, and Anthony Skjellum. *Using MPI*. Scientific and Engineering Computation. MIT Press, 2nd edition, November 1999.

[17] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide*, pages 9–36. Oxford University Press, 1995.

[18] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Selected papers from CSL'92 (Computer Science Logic)*, volume LNCS 702, pages 274–308. Springer Verlag, 1992.

[19] HPF Forum. *High Performance Fortran Language Specification*, 1.0 edition, 1993.

[20] ISO. *Fortran 90*. May 1991. ISO/IEC 1539: 1991 (E).

[21] ISO/IEC JTC1/SC22/WG14. Programming languages – C – Committee draft. January 1999.

[22] P.W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, May 1997.

[23] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Symposium on Principles of Programming Languages*, pages 283–294, 2002.

[24] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 2nd edition, 1998.

[25] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.04 : Documentation and user's manual*, December 2001. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[26] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[27] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[28] Peter D. Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 575–631. Elsevier Science Publishers, 1990.

[29] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992. CTCS 26.

[30] Peter D. Mosses. Theory and practice of action semantics. Technical Report BRICS RS-96-53, December 1996.

[31] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ User's Manual –ver.1.4–*.

[32] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[33] Bruno Poizat. *A Course in model theory – an introduction to contemporary mathematical logic*. Springer-Verlag edition, 2000.

[34] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, January 1994.

[35] Thinking Machines Corp. *C* Programming Guide*, November 1990.

[36] Eric Violard and Guy-René Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.

A The Models of Rewriting Logic

The basics of rewriting logic are recalled here. For simplicity we treat only RWL with unconditional equations and transitions. The reader can find a treatment of the conditional case in [26].

When one writes a rewrite theory, one describes its *models*. A model of a rewrite theory is a family of *categories* [24], functors, and natural transformations, which satisfies the description. The relationship between a rewrite theory and a model of it is stated in the following. Let $[\![.]\!]$ be the denotation function with respect to this chosen model. A rewrite theory consists of:

- *Sort* declarations. A sort s is a name which denotes a category $[\![s]\!]$ in the model.
- *Operation* declarations. An operation σ is a name, and has an arity $s_1 \dots s_n, s$ of sorts. It denotes a *functor* $[\![\sigma]\!]$ from the product category $[\![s_1]\!] \times \cdots \times [\![s_n]\!]$ to category $[\![s]\!]$.
- *Equations.* An equation has form $t = t'$, with t and t' being two terms constructed from some operations and from some variable names $x_1, \dots, x_n$, and having the same sort s. Any instanciation $t(u_1, \dots, u_n)$ of term t with ground terms $u_1, \dots, u_n$ (where $u_1, \dots, u_n$ substitute for the variables $x_1, \dots, x_n$) denotes an object $[\![t(u_1, \dots, u_n)]\!]$ from category $[\![s]\!]$. That is, t denotes a functor from a product category (the domain for the substitution of the variables $x_1, \dots, x_n$) to $[\![s]\!]$. Equation $t = t'$ states the equality of functors $[\![t]\!]$ and $[\![t']\!]$, and hence the equality, for any substitution $(u_1, \dots, u_n)$, between objects $[\![t(u_1, \dots, u_n)]\!]$ and $[\![t'(u_1, \dots, u_n)]\!]$ of $[\![s]\!]$.
- *Transitions.* A transition has form $r : t \to t'$, where r is a *label*, and with t and t' being, as above, two terms of same sort, constructed from some operations and some variable names $x_1, \dots, x_n$. The transition states that there is a *natural transformation*, which is named r, from functor $[\![t]\!]$ to functor $[\![t']\!]$. This implies that there is in category $[\![s]\!]$, for any subtitution $u_1, \dots, u_n$ of the variables, an arrow $r_{(u_1, \dots, u_n)}$ from object $[\![t(u_1, \dots, u_n)]\!]$ to object $[\![t'(u_1, \dots, u_n)]\!]$.

We have given no label to the transition in our specification in section 3, first because the CafeOBJ notation does not provide a formal way for doing this, and then because there is only one transition anyway.

B Semantics of Assignments

The remaining of the `MHPF-SEM` theory is given below. It says that execution of an addition-plus-assignment instruction *does* an addition and an assignment.

```
vars tr algo : Algo
vars x y z w : SName
vars X Y Z W : ArrayName
var i : Int

ceq (M wTrace: (tr ; (z := x + y)) wAlgo: algo) .val w
 =   ((M wTrace: tr wAlgo: algo) .val x)
   + ((M wTrace: tr wAlgo: algo) .val y)
   if w == z .

ceq (M wTrace: (tr ; (Z := X + y)) wAlgo: algo)
                                              .val W i
  =   ((M wTrace: tr wAlgo: algo) .val X i)
    + ((M wTrace: tr wAlgo: algo) .val y)
    if W == Z
       and-also 1 <= i and-also i <= (length W) .

ceq (M wTrace: (tr ; (Z := X + Y)) wAlgo: algo)
                                              .val W i
  =   ((M wTrace: tr wAlgo: algo) .val X i)
    + ((M wTrace: tr wAlgo: algo) .val Y i)
    if W == Z
       and-also 1 <= i and-also i <= (length W) .
```

```
** negative cases:

  ceq (M wTrace: (tr ; (z := x + y)) wAlgo: algo) .val w
   =  (M wTrace: tr wAlgo: algo) .val w
      if not (w == z) .

  ceq (M wTrace: (tr ; (Z := X + y)) wAlgo: algo)
                                                  .val W i
    = (M wTrace: tr wAlgo: algo) .val W i
      if not (W == Z) .

  ceq (M wTrace: (tr ; (Z := X + Y)) wAlgo: algo)
      .val W i
    = (M wTrace: tr wAlgo: algo) .val W i
      if not (W == Z) .

  var ar-assgnt : ArrayAssignt
  eq (M wTrace: (tr ; ar-assgnt) wAlgo: algo)     .val w
   =  (M wTrace: tr wAlgo: algo) .val w .

  var s-assgnt : SAssignt
  eq (M wTrace: (tr ; s-assgnt) wAlgo: algo)    .val W i
    = (M wTrace: tr wAlgo: algo) .val W i .


** values are the same in every initial state:

  var algo' : Algo
  eq (M wTrace: noAlgo wAlgo: algo)  .val x
   = (M wTrace: noAlgo wAlgo: algo') .val x .

  eq (M wTrace: noAlgo wAlgo: algo)  .val X i
   = (M wTrace: noAlgo wAlgo: algo') .val X i .
```

In this specification, the values of array elements when the index is out of bounds is simply left undefined.

We call an *initial state* a machine state with a null trace ("`noAlgo`"). The last two equations say that in any model, all initial states must have the same values. This makes valid the preceding equations, which define the values at one state from the values at some other states, "poping" along the way from the trace (if the trace is viewed as a stack).