An Approach for Defining the Semantics of Directives

Philippe Gerner

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

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, as their the degree of stringency is not fixed, the compiler writer has no obligation with respect to them, and can only implement them "as he believes is the best"; second, as a result, it is not easy for the programmer to evaluate the exact effect of the directives he writes. This article proposes a methodology for addressing this issue. The data mapping directives of the High Performance Fortran language are used as a case study.

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 [5]) and the OpenMP library (e.g., [1]). 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.

The problem with the directives is that more often than not, they 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 this problem would be a documentation of directives which is 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 indications about 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". The purpose of this article is to propose a methodology for doing this.

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, a formal expression tends to induce coherence in the semantics of the language, as remarked by Milner about his work on the semantics of Standard ML [17]: we believe that, similarly, a formal framework for the notion of directives could help him to construct a coherent language of directives. So we will use a formal framework: algebraic specifications, which provide us with the notion of models.

The paper is structured as follows. In the next section, The case of the directives to the compiler is introduced, and the relation of the norm to them is discussed. In the third section our methodology is presented, and is situated with some related works. The fourth section uses this methodology for a study of the key data mapping directives of HPF. It also introduces a proposition for a formalisation of the semantics of the directives.

2 The Problem

2.1 Directives to the Compiler

In rapid prototyping, performance issues are not considered. But in other programming contexts, performance becomes an issue. An extreme case is scientific computing. As a consequence, high-level programming languages provide some means to achieve performance "in spite of the high level". An example is imperative features in functional languages, as in the Caml language [15].

Directives to the compiler are an interesting form of high-level low-level programming, because instead of tuning his algorithm for a better performance by changing its form, the programmer just *adds* some instructions (the directives) for doing this tuning.

Directives for Parallel Programming 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 [8]) is still the choice when high performance is crucial. 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., [2]). 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.

2.2 The Documentation on Directives

The semantics of industrial programming languages is defined by their *norm*. As directives become an important tool, as in parallel programming, we examine in this article how the norm could explain the meaning of the directives. So we first discuss the exact function of this norm.

The documentation which defines the norm 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 norm defines the precise meaning of each construct of the language.

The purpose of the norm 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" [22, p. 81].

Because achieving portability is so important, the focus of the norm 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." [11, p. 15].

Thus, the compiler writer is free to produce any actual semantics he likes, as long as he respects 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." [11, p. 15]. Of course, "an implementation might define a one-to-one correspondence between abstract and actual semantics," [11, p. 16], but the point is that is it not at all mandatory.

Degree of Stringency The problem with the directives is that they are only indication to the compiler : they are not obligatory. But current norms have no intermediary levels between obligatory and non-obligatory : the function of the directives is exposed in an informal manner, and the norm thus imposes no clear standard onto the directives. The consequence is that the programmer then has to infer the effect of its low-level investment into writing directives, from interaction with his compiler.

We propose a methodology for defining the semantics of the directives. This methodology could be used by the writers of a norm, for explaining clearly the meaning of the directives. And it could also be used by the designers of a language, when they want to design a language of directives to go with the base language.

3 The Proposed Methodology

3.1 Three Semantics

In order to take explicitely into account the difference between abstract and actual semantics, we use in the following the expressions "*official semantics*" and "*actual semantics*". The official semantics is the abstract semantics which the norm defines.

The actual semantics, for us, is *the constraints on the compiler* which are given by the documentation. That is, it specifies when a software qualifies as a compiler for the language, in the sense of qualification norms: "many regulations, including many embodied in legislation, may be viewed as *qualification norms*, which specify the conditions under which some entity x counts as an entity of a particular type F" [10]. In practice, the actual semantics for a language *refers to* the official semantics on at least one point : if the official semantics defines the function of the program, then the official semantics must provide the same function. For exemple, for the C language, the data written into files must be the same.

And for what the directives speak about, we consider a *wished for actual semantics*. This wished for semantics *satisfies* the obligatory actual semantics : it is a refinement of this semantics. Thus, by transitivity, the wished for semantics also refers to the official semantics.

This scheme is summarized by the following diagram, where the semantics which is mandatory for the compiler is underlined, and where the semantics which is wished for by the programmer is written in italics :

$$
\begin{tikzcd}
\text{official sem.} & & \textit{wished for actual sem.} \arrow[ll, "\text{refers to}"'] \arrow[dl, "\text{satisfies}"] \\
& \underline{\text{actual sem.}} \arrow[ul, "\text{refers to}"] &
\end{tikzcd}
$$

Fig. 1. The three semantics and their relation

The norm is conceived as the *sum* of these three semantics. The principal difference between the three semantics is their **status**, w.r.t. to obligation for the compiler. Official semantics says nothing about the implementation. Actual semantics defines it. The wished semantics is a wished refinement of the actual semantics.

The point of this approach is that a document which describes the norm should make a clear distinction between the three status, and present the features of the langage according to which semantics they belong to. For the directives, this implies that they be considered as first-class citizens, as explained below.

The Preference of the Programmer In the above scheme, directives are viewed as constraints over the wished for semantics, but constraints which have no mandatory status. However, the programmer clearly has a *preference* as regards how the compiler will use them: this can be seen as a particular form of constraint onto the compiler. In the next section we will show how the notion of preference can be used to give a formal meaning to the directives, so that the norm can speak about them in a precise manner.

3.2 Related Works

Formal Semantics It is usually considered that for stating clearly the semantics of a language, using a formal method is a good approach. However, all the formal methods known to us which have been designed for describing the semantics of programming languages are designed for describing *one* semantics: the compiler is not considered, so one can consider that they allow one to define the official semantics of a language. By constrast, we are interested in expressing both an official and an actual semantics, and the relationship between the two.

For example, what is called the "*operational semantics*" of a programming language describes the computation mechanism which corresponds to a program (see, e.g., [19]), whereas *denotational semantics* [18] 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 norm, things are different. Indeed, in norms, as for C99 above, the official operational semantics is only a *descriptive means* for expressing which input-output behaviour the actual semantics of a program must have, and there is no necessary relationship between the official operational semantics and the actual semantics. Thus, from the point of view of the norm, the official operational semantics is just a convenient way of defining a denotational semantics.

Research in Compiling The research which comes closer to our preoccupation of relating official and actual semantics 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, see, e.g., [13]. But the focus is on optimisations, and the aim of these optimisations is left mainly implicit. By constrast, we want to explicitate this aim (for the wished for semantics), while the optimisation techniques are not the preoccupation of the norm.

Also relevant is the *automatic parallelization* field, which provides an algebraic formulation of execution change through the notion of a *space-time mapping* [4]. But it is tied to affine dependances, and too technical (it's integer programming) for being adequately used in a norm document.

Formal Preference The notion of preference in the programming field can be found in recent work on constraint programming ("soft constrains", see, e.g., [20]), and on "preference logic programming" [7]. These works focus on resolving the constraints. By constrast, we only aim at stating in a simple way (for inclusion in the norm document) the preference over the directives. But these works provide useful hints at *how* the compiler could then proceed to handle the directives: while compiling techniques are not the preoccupation of the norm, it is important for the designer of the language to be conscious of the current techniques, so that he can conceives directives which are within the reach of a compiler.

4 Case Study : HPF

Here we discuss the latest version of HPF : HPF 2 [5], which is an extension to Fortran 95 [12].

4.1 The Shanks Transform

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. An example of a data parallel program is given by the following coding, in HPF, of the Shanks transform [21]. The Shanks transform is a numerical means for accelerating the speed of convergence for a convergent series $(S_n)_{n\in\mathbb{N}}$. The transform is defined by :

$$T(S_n) = \frac{S_{n+1}S_{n-1} - (S_n)^2}{S_{n+1} - 2S_n + S_{n-1}} \quad .$$

The series $(T(S_n))_{n\in\mathbb{N}}$ converges towards the same limit as $(S_n)_{n\in\mathbb{N}}$, but quicker. The transform can be applied several times, so that the series $(T^K(S_n))_{n\in\mathbb{N}}$ $(K \geq 1)$ converges still quicker. The HPF program below applies the transform K times, from the $N+1$ terms of the initial series $(S_n)_{n\in\mathbb{N}}$, which are stored into input array `A`. The result of the transform is put into output array `B`. The intermediary series are put into a two-dimensional array `T`.

```
INTEGER N,K
...
```

```
DOUBLE PRECISION A(0:N), B(0:N-(2*K))
DOUBLE PRECISION T(0:K,0:N)

!HPF$  ALIGN A(I)      WITH T(0,I)
!HPF$  ALIGN B(I)      WITH T(K,I+K)
!HPF$  TEMPLATE LINE(0:N)
!HPF$  ALIGN T(*,J) WITH LINE(J)

FORALL (I=0:N)
  T(0,I) = A(I)
END FORALL

DO I=1, K
  FORALL (J=I:N-I)
    T(I,J) = SHANKS_OP(T(I-1,J-1), T(I-1,J), T(I-1,J+1))
  END FORALL
END DO

FORALL (I=0:N-(2*K))
  B(I) = T(K,I+K))
END FORALL
```

The lines that begin with "`!HPF$`" are HPF directives. They are also Fortran 95 commentaries: HPF is an extension of Fortran 95, but does not extends the language itself: it only defines special commentaries. The motivation for these directives is that when a distributed memory parallel machine is used, it is not enough that potentially parallel instructions like the above `forall` instructions are executed in parallel: if the elements of the arrays are distributed in a random way onto these memories, these instructions will generate many inter-processor communications. As communications are costly in time, performance of the program depends on minimizing the number of communications: the programmer can help the compiler into this difficult task by giving him indications as to how to distribute.

4.2 Formalizing the Semantics

We want to express three distinct semantics, and preferably in a formal way, so as to be able to express the semantics of the directives. As said above, no method exists for for that. So we propose a method. It is not necessarily "the" method, but it provide us with a means to expose the above methodology, and to formally define the meaning of the directives.

The Choice of Rewriting Logic Our method uses *rewriting logic* (RWL) [16]. RWL is algebraic specifications plus transitions. As for algebraic specifications, the semantics of RWL is given through the notion of the *satisfaction* of a theory by 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* [14] with *functors* between them. Our choice of the RWL framework is motivated by the following factors:

- the notion of satisfaction allows for coding each directive as an *equation*, which can then be satisfied or not by an algebra.
- a category (in a model) is defined by objects and arrows, which is interesting when we want to express *execution graphs* : the arrows can be used to model the transitions.
- theory morphisms help structure the semantics [6]. Thus the three semantics can be defined in separate modules. Also, some modules can be extensions of others.

The RWL theories in this article are written using the *CafeOBJ* notation [3].

The Models of Rewriting Logic Here we recall here the basics of rewriting logic. This section can be skipped on a first reading without loss of comprehension of the principal ideas in the following.

For simplicity we treat only RWL with unconditional equations and transitions. The reader can find a treatment of the conditional case in [16]. 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 $s_1 \ldots s_n, s$ 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 $x_1, \ldots, x_n$, and having the same sort s. Any instanciation $t(u_1, \ldots, u_n)$ of term t with ground terms $u_1, \ldots, u_n$ (where $u_1, \ldots, u_n$ substitute for the variables $x_1, \ldots, x_n$) denotes an object $[\![t(u_1, \ldots, u_n)]\!]$ from category $[\![s]\!]$. That is, t denotes a functor from a product category (the domain for the substitution of the variables $x_1, \ldots, x_n$) to $[\![s]\!]$. Equation $t = t'$ states the equality of functors $[\![t]\!]$ and $[\![t']\!]$, and hence the equality, for any substitution $(u_1, \ldots, u_n)$, between objects $[\![t(u_1, \ldots, u_n)]\!]$ and $[\![t'(u_1, \ldots, u_n)]\!]$ 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 $x_1, \ldots, x_n$. 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 $u_1, \ldots, u_n$ of the variables, an arrow $r_{(u_1,\ldots,u_n)}$ from object $[\![t(u_1, \ldots, u_n)]\!]$ to object $[\![t'(u_1, \ldots, u_n)]\!]$.

4.3 Representation of the program

We define an abstract syntax for HPF with a RWL theory `HPF-CONSTRUCTS`. Then the Shanks program (without the directives, for the moment) corresponds to the following theory, which uses the constructs defined in `HPF-CONSTRUCTS` :

```
module! SHANKS {
  using (HPF-CONSTRUCTS)
  op algo1    : -> Algo
  ops N K     : -> Integer
  ops A B T   : -> ArrayName

  var x : VarName
  eq input-var  x = x == A .
  eq output-var x = x == B .

  eq dom A = [ 0 : N ] .
  eq dom B = [ 0 : N - (2 * K) ] .
  eq dom T = [ 0 : K , 0 : N ] .
  eq dom LINE = [ 0 : N ] .

  eq algo1 = (on dom1 : T := A get1) ;
             (do_from 1 to K :
               (on pdom1 : T := T pget1 pget2 pget3 op1)) ;
             (on dom2 : B := T get2) .

  vars I J : Integer
  eq dom1 = [ 0 , 0 : N ] .
  eq pdom1 ( I ) = < I , I : N - I > .
  eq get1 < 0 , I > = < I > .
  eq get2 < I > = < K, I + K > .
  eq pget1 ( I ) < J > = < I - 1 , J - 1 > .
  eq pget2 ( I ) < J > = < I - 1 , J     > .
  eq pget3 ( I ) < J > = < I - 1 , J + 1 > .
  vars v1 v2 v3 : Value
  eq op1 (v1, v2, v3) =
      ((v3 * v1) - (v2 * v2)) / ((v3 - (2 * v2)) + v1) .
}
```

Functions `pget1`, ..., are parametered: the parameter is the current loop index. Similarly for the parametered index domain `pdom1`.

4.4 Official Semantics

The official semantics of HPF is defined by the Fortran 95 norm [12], since HPF directives are just Fortran 95 commentaries. Indee, the HPF norm says: “data mapping is specified using directives that can aid the compiler in optimizing parallel performance, but have no effect on the semantics of the program” [5, p. 5]

We describe the official semantics by transitions over a machine state. We define a machine, whose job is to execute instructions :

```
module* OFFICIAL-SEM-MACHINE {
  [ MachineName ]
  op M : -> MachineName

  [ Inst ] [ Trace ]
  [ Inst < Trace ]
  op noInst : -> Inst
  op (_;_)  : Trace Trace -> Trace  { assoc id: noInst }

  [ MachineState ] [ MAlgo ]

  op (_wTrace:_wAlgo:_) :
      MachineName Trace MAlgo       -> MachineState
  op (_wTrace:_toDo:_wAlgo:_) :
      MachineName Trace Inst MAlgo  -> MachineState
  op (_wTrace:_done:_wAlgo:_) :
      MachineName Trace Inst MAlgo  -> MachineState

  var tr : Trace   var inst : Inst    var algo : MAlgo
  trans M wTrace: tr toDo: inst wAlgo: algo
     => M wTrace: tr done: inst wAlgo: algo .   ** [ e ]

  eq    M wTrace: tr done: inst wAlgo: algo
     =  M wTrace: (tr ; inst) wAlgo: algo .

  [ Address ] [ Value ]
  op (_.val_)  : MachineState Address -> Value
}
```

This machine is defined by its set of states, from the sort `MachineState`. A state is defined by the name of the machine—“`M`” here—, a trace of the instructions it has already executed, and the algorithm which is still to be executed. This machine has a memory: this is coded by the last operation, `.val`, from which one can observe the value attached to an address, from any state of the machine. The working of the machine is illustrated with its use, in the following.

Official Execution The official execution is defined the following way :

```
module OFFICIAL-SEM {
  using (OFFICIAL-SEM-MACHINE)
  protecting (HPF-CONSTRUCTS)
  [ Algo < MAlgo ]
  op <_> : Assignt -> Inst

  var tr : Trace    var algo : Algo   var assgnt : Assignt
```

```
  trans M wTrace: tr wAlgo: (assgnt ; algo)
     => M wTrace: tr toDo: < assgnt > wAlgo: algo .
  ...
}
```

That is, HPF algorithms (of sort `Algo`) become algorithms for the machine. Assignments become the instructions of the machine. When the algorithm which is still to be executed begins with an assignment, this assignement is given as a `doTo` instruction. The mechanism is then the following: the transition, labeled "`e`", defined in the `OFFICIAL-SEM-MACHINE` theory will make the state transit to a state where the instruction is `done`: this represent execution of this instruction. This instruction is then put into the trace, and the machine is then ready for a new instruction. We do not detail the working of the do loop, but the mechanism is the same: a sequence of assignments will be generated. We do not detail either the effect of transition on the values at the adresses: it simply states that executing an assignement "makes this assignment".

In the following, we call *initial states* of the machine those with an empty trace (`noInst`); and *final states* those with an empty algorithm (`noAlgo`).

4.5 Actual Semantics

As the directives are Fortran 95 commentaries, the actual semantics of HPF is that of Fortran 95. Which is : the effect of the execution of the program onto output files is the same *as if* the program is executed according to the official semantics.

For expressing this, we begin by defining the actual machine: it is a copy of the official one, with two renamings: the name of the machine becomes "`AM`" ('actual machine'), and the sort of machine states "`ActMachineState`".

```
module* ACTUAL-SEM-MACHINE {
  protecting (OFFICIAL-SEM-MACHINE * {
      sort MachineState -> ActMachineState, op M -> AM })
}
```

Actual execution Actual execution of an algorithm is "in one step":

```
module ACTUAL-SEM {
  using (ACTUAL-SEM-MACHINE)
  protecting (HPF-CONSTRUCTS)
  [ Algo < MAlgo ]
  op <<_>>   : Algo -> Inst

  var algo : Algo
  ceq AM wTrace: noInst wAlgo: algo
    = AM wTrace: noInst toDo: << algo >> wAlgo: noAlgo
      if algo =/= noAlgo .
```

```
  ...
}
```

That is, no details are given as to the "how". The dots are for the following two equations. The first one states that the values of input variables on the initial states of this actual machine are the same as on the initial states of the official machine. The second equation states that the values of the output variables on the final states of the actual machine after execution of an algorithm `algo` are equal to the values on the final state which is reached by the official machine when it executes the same algorithm.

```
  vars tr tr' : Trace   var x : ArrayName    var i : Index

  ceq (AM wTrace: noInst wAlgo: algo) .val [ x, i ]
    = (M  wTrace: noInst wAlgo: algo) .val [ x, i ]
            if input-var x .

  ceq (AM wTrace: tr' wAlgo: noAlgo) .val [ x, i ]
    = (M  wTrace: tr  wAlgo: noAlgo) .val [ x, i ]
    if (AM wTrace: noInst wAlgo: algo) ==> (AM wTrace: tr' wAlgo: noAlgo)
   and (M  wTrace: noInst wAlgo: algo) ==> (M  wTrace: tr  wAlgo: noAlgo)
   and  output-var x .
```

4.6 Wished for Actual Semantics

The norm of HPF says: "The explanations given [in this document] do not necessarily reflect the actual compilation process" [5, p. 6]. What should be understood of course, is that it is *wished* (by the programmer) that they do reflect the actual compilation process. Here we define this wished semantics.

It is wished that a program in HPF executes onto a distributed memory parallel machine : so we define accordingly the wished machine:

```
module* WISHED-FOR-ACTUAL-SEM-MACHINE {
  protecting (ACTUAL-SEM-MACHINE * {
      sort ActMachineState -> WActMachineState, op AM -> WAM })

  [ Proc ]
  [ TemplateName ] [ ArrayName < TemplateName ]
  op (_.loc__)  : WActMachineState TemplateName Index -> Proc
}
```

It is a copy of the actual machine, which is *extended* as follows. First a set of *processors* is declared. Then *Templates* are defined. About them, the norm says: "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)." [5, p. 40] Each array can be considered as having an underlying template, which is stated by "`ArrayName < TemplateName`". The distribution of each template "indexed position" onto the processors is then declared, with operation `.loc`. The meaning of this is that if a value is attached to the position (which is the case for an array), then this value is distributed onto the corresponding processor. Templates then serve as convenient intermediaries, as the template `LINE` for the values of array `T` in program Shanks (see below).

Wished for Execution We do not reproduce here the definition of the wished execution of the algorithm. It begins by stating that the relation between input and output values is the same as for the actual semantics, with the same expression scheme as between official and actual semantics. Then it refines the single-step execution of the actual semantics into a more complex sequence of steps. In particular, every array assignment "becomes again" an instruction—as it was in the official semantics.

4.7 The Semantics of the Program

The norm of the language is defined as the sum of the three semantics :

```
module NORM {
  protecting (OFFICIAL-SEM)
  protecting (ACTUAL-SEM)
  protecting (WISHED-FOR-ACTUAL-SEM)
}
```

Then the wished semantics of the Shanks program is given by the combination of the algorithm and the norm, plus the template declaration:

```
module SHANKS-SEM {
  using (NORM)
  protecting (SHANKS)
  op  LINE : -> TemplateName
}
```

Preference As to the `ALIGN` directives, they translate into the following equations about the wished for actual semantics of SHANKS:

```
  var S : WActMachineState   vars I J : Int

  ceq S .loc A < I > = S .loc T < 0 , I >
      if 0 <= I and I <= N .

  ceq S .loc B < I > = S .loc T  < K, I + K >
      if 0 <= I and I <= (N - (2 * K)) .

  ceq S .loc T < I , J > = S .loc LINE < J >
      if 0 <= I and I <= N and 0 <= J and j <= N .
}
```

These equations state the equality of distribution between different values: this is the meaning of the "ALIGN" directive.

But as they correspond to directives, these equations do not have to be satisfied by the models of the theory SHANKS-SEM. However, "if two objects are mapped (via the two-level mapping of alignement and distribution) to the same abstract processor, it is *a strong recommendation* to the implementation that they *ought* to reside in the same physical processor" [5, p. 20]. This corresponds also to the preference of the programmer.

We modelise this with an order relation on the set $\mathrm{MOD}(\texttt{SHANKS-SEM})$ of these models: this construction is inspired by the models of *deontic logic* [9]. Deontic logic is the logic of obligations—in the legal sense, which means that it is possible that the obligations are not satisfied. If D is the set of equations the directives translates into, then we consider the relation $\preccurlyeq_D$ on $\mathrm{MOD}(\texttt{SHANKS-SEM})$, with :

$$M \preccurlyeq_D M' \iff (\forall d \in D \; : \; M \models d \implies M' \models d) \enspace .$$

That is, given a set of directives which the compiler knows how to satisfy (by a model M), then if he (the compiler) can satisfy one more and go to a model M' of the program, then this is prefered by the programmer. For other languages of directives, it will be possible to define other preference relations between the models.

5 Conclusion

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 directives to the compiler are in the present state of the art of parallel compiling the most promising solution, as they provide a means for a clear separation of concerns between high- and low-level considerations. But if they are to become a common tool, is is important that they can be defined properly by documentations.

We have described a methodology for including a proper description of the meaning of directives into a norm. This methodology implies that three semantics of different status be indentified in a norm: official, actual, and wished for actual semantics.

We have used rewriting logic as an expository tool: this allowed us to describe the relationship between the three theories; and to define the semantics of directives by a preference relation onto the models of the theory which represents a program.

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

References

1. *OpenMP Fortran Application Program Interface*, November 2000. Version 2.0.
2. 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.
3. Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. Submitted to *Theoretical Computer Science*, 2000.
4. Paul Feautrier. *Automatic Parallelization in the Polytope Model*, volume LNCS 1132, pages 79–103. Springer-Verlag, May 1996.
5. HPF Forum. *High Performance Fortran Language Specification*, 2.0 edition, January 1997.
6. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.
7. Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Preference logic programming. In *International Conference on Logic Programming*, pages 731–745, 1995.
8. William Groppa, Ewing Lusk, and Anthony Skjellum. *Using MPI*. Scientific and Engineering Computation. MIT Press, 2nd edition, November 1999.
9. Sven Ove Hansson. *The Structure of Values and Norms*. Cambridge University Press, 2001.
10. Andrew J.I. Jones and Marek Sergot. *On the Characterisation of Law and Computer Systems: The Normative Systems Perspective*, chapter 12. John Wiley and Sons, 1993.
11. ISO/IEC JTC1/SC22/WG14. Programming languages – C – Committee draft. January 1999.
12. ISO/IEC JTC1/SC22/WG5. X3j3/96-007 – Fortran 95 – Committee draft. March 1996.
13. 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.
14. Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 2nd edition, 1998.
15. 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`.
16. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
17. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
18. Peter D. Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 575–631. Elsevier Science Publishers, 1990.
19. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
20. F. Rossi S.Bistarelli, U. Montanari. Soft concurrent constraint programming. In *Proceedings of ESOP 2002 (European Symposium on Programming)*, volume LNCS. Springer Verlag.
21. D. Shanks. Nonlinear transformations of divergent and slowly convergent sequences. *J. Math. Phys.*, 34:1–42.
22. Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, January 1994.