HPFfe: a Front-end for HPF

SCCS-771

Parallel Software Systems Group
NPAC at Syracuse University
PACT Laboratory of CS Dept at Harbin Institute of Technology
PACT Group of CS Dept at Peking University

Octobor 1, 1996

Abstract

This is a general description of a compiler front-end for High Performance Fortran Version 1.0, developed by a joint effort of the three groups designated in the author part of this document. Specifically, the following sections are included in this document.

- A global view of the package. It gives the reader a functional view as well as a physical directory structure of the system. The completeness of coverage of HPF 1.0 grammar is emphasized.
- Current status and testing results. The test procedure and testsuites used are discussed together with testing result, which demonstrate the quality of the front-end to some extent.
- More words on the functional modules.
- A brief on the intermediate representation.
- Predecessors of the system.
- People involved in this work.
- Conclusions

1 A global view of the package

The target of this front-end is full HPF 1.0 as specified in [3]. By definition, it includes full Fortran 90 as specified in [1] and almost full Fortran 95 as specified in [6].

Externally, four functional modules are provided

- Parsing – transform an HPF program into an intermediate representation (IR).
- Semantics checking – check various semantic correctness according to the constraints stated in language specification.

- Showing the IR – produce a readable form of the intermediate representation. This is useful for debugging and further compiler development.
- Unparsing – produce an equivalent source program from intermediate representation. At present, this is useful for debugging and verification. It is also useful for node program generation for further compilation work.

Physically, the package is organized in the following directory structure (we follow GNU conventions in managing software packages).

```
hpffe-x.xx/
   INSTALL
   README
   Makefile
   configure
   configure.in
   acconfig.h
   config/
   bin/
      dumpdep
      hpf2dep
      hpfsc
      unparse
   src/
      Makefile
      Makefile.in
      basicop/
      hpf2dep/
      hpfsc/
      tools/
         Makefile
         Makefile.in
         dump/
         unparse/
      include/
   include/
   testsuite/
   lib/
      libbasic.a
      libhpf.sl
```

2 Current status and testing results

By its performance on a comprehensive testing, we claim the front-end is completed at time of this writing, though some bugs remain and some known limitations exist.

We adopted a rigorous testing procedure, which is divided in the following two parts.

Positive testing Select a program; make sure DEC HPF compiler can compile it (this is taken as our criterion for a valid HPF program, though DEC compiler itself may contain some bugs and limitations); put it through our parser, semantics routine, and unparser in sequence. A success is claimed if the output program can be compiled by DEC HPF compiler again. Five sets of programs are designed/selected. They are catalogued as

- **Simple tests** 50 small testing programs, each tests some specific language features. 1000 lines.
- **Comprehensive tests** A testsuite specifically developed based on the syntax rules of Fortran 90 and HPF extensions. The principle in designing the testsuite is that each syntax rule is at least touched once (we call it *exhaustive first order testing.* 9 programs, 3000 lines. See [5] for a report on the design of the testsuite.
- **Typical Fortran 90 programs** The set of example programs from Kerrigan's book, Migrating to Fortran 90. 14 programs, 1250 lines.
- **Real HPF application** The set of HPF programs developed by Blackhole community. 36 programs, 1500 lines.
- **Real Fortran 90 application** The Runge-Kutta solver code from NAG. 9 programs and 4000 lines.

Negative testing Select a program with some syntax or semantics error; let it be input to our front-end. A success is claimed if the front-end complains about the program, though it may not locate the error exactly. For negative test, we designed 50 programs, each contains one or more syntax or semantic errors.

With some known limitations stated below, we see very good outcome from this testing procedure. Especially, we note the following

- If we take a valid program and let it go through the process,

  ```
  syntax-semantics-unparse-syntax-semantics-unparse
  ```

  we'll see the output of two unparses are identical (checked by diff). Although this does not necessarily prove the correctness of our front-end, it does tell something about its quality.

- The blackhole package, after going through the syntax-semantics-unparse process, can be successfully recompiled and run by DEC HPF compiler without any change.

The system has been successfully installed on the following platforms with various combinations of cc, gcc, lex, flex, yacc, and bison.

- IBM RS/6000, AIX 3.2.5
- Sun SparcStation 1+, OS 4.1.1
- Sun Sparc 10, OS 4.1.4
- Sun Sparc workstation, Solaries 2.4
- Sun Sparc workstation, Solaries 2.5
- DEC Alpha, OSF/1, V3.0
- HP, HP-UX
- SGI INDY, IRIX 5.3
- Linux

Current limitations

- Do not support multi files, i.e., a module to be used in a program unit must be put in the same file as the program unit. Our parser can not automatically look for the module files, though we have a separate preprocessor to do this.
- Keywords are reserved.
- Optional comma allowed in rule 507

  ```
  R507 length-selector is ([LEN=] type-param-value)
                       or * char-length [,]
  ```

  can not be used. Note, this use of comma is considered obsolete by Fortran 95.

3 More words on the functional modules

As mentioned earlier, there are four high level functional modules developed in this package, namely a parser (hpf2dep), a semantics routine (hpfsc), an IR renderer (dumpdep), and a unparser (unparse). We've also partially developed a Motif package which allows the user to visualize and trace the internal representation.

The parser is built with traditional lex/yacc tools. Although we considered some more advanced tools (such as Eli), we have settled on the old fashion, betting on its popularity/availability and general acceptance.

The semantics routine is probably our 'invention' in constructing a research parallel compiler product – people normally do source-to-source translation and leave the semantics part for native commercial compiler to handle. Our semantics routines conduct an extensive checking based on various constraints stated in language specification.

The IR renderer is modified/extended from the same module provided in Sage. It's main purpose is for compiler developer to heck the code.

Unparser is also modified/extended from the same module provided in Sage. For now, it helps to develop the front-end. It will also be directly useful for node program generation in our next step towards an HPF compiler and then a general language/compiler/runtime testbed.

4 A brief on intermediate representation

As acknowledged in the next section, we have adopted the intermediate representation of Sage system as our starting point. We give a brief introduction here. For more information, please refer to [2] [4] or [7].

An abstract syntax tree (AST) and a symbol table constitute the major part of our intermediate representation for HPF program files.

There are two types of nodes in the syntax tree. One is called *bif node*, corresponding roughly to HPF statements; the other is called *low-level node*, corresponding to the expressions in each statement.

Macroscopically, we may only consider the *bif* nodes and their relations in the AST, which is based on program control constructs. The root is called *global node* and each node has potentially two distinct children, the *control children*. One is called *true branch* and the other is called *false branch*. Except pure conditional construct such as *if-then-else*, most *bif* node has empty *false branch*.

Each branch may represent a list of statements in the same level, as can be seen in the following example.

Besides other information, there are two important pieces of informations contained in each node. One is called *identifier*, which is a unique integer designating the node among others. The other is called *type*, which is a character string telling the nature of the node. Thus, we may use the notation *N[type]* to refer to the node id and its type. To describe the control children, we will use an expression of the form

```
( N[type]  (true-branch) (false-branch))
```

The following simple example is to illustrate this structure.

```
      PROGRAM MAIN
      REAL A(100), X
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P

      FORALL (i=1:100) A(i) = X + 1

      CALL FOO(A)

      END

      SUBROUTINE FOO(X)
      REAL X(100)
!HPF$ INHERIT X

      IF (X(1).EQ.0) THEN
        X = 1
      ELSE
        X = X + 1
      END IF
      RETURN
      END
```

The *bif* nodes for this program are structured as follows:

```
(1[GLOBAL]                    ! has two components in true-branch
   ((2[PROG_HEDR]             ! has six components in true branch
      (3[VAR_DECL]
       4[PROCESSORS_STMT]
       5[DISTRIBUTE_DECL]
       (6[FORALL_STMT]
          (7[ASSIGN_STAT]
           8[CONTROL_END]
          )
          NULL
       )
       9[PROC_STAT]
       10[CONTROL_END]
      )
      NULL
    )
    (11[PROC_HEDR]
       (12[VAR_DECL]
        13[INHERIT_DECL]
        (14[LOGIF_NODE]        ! both branchs are non empty
           (15[ASSIGN_NODE]
            16[CONTROL_END]
           )
           (17[ASSIGN_NODE]
            18[CONTROL_END]
           )
        )
        19[RETURN_STAT]
        20[CONTROL_END]
       )
       NULL
    )
   NULL
)
```

The integer identifier associated with each node is the key to accessing the rest of the properties of the node.

Other information contained in a statement node include:

- the name and line number of the source file containing this statement,
- the identifier of the control parent of this node,
- a symbol reference such as a do loop parameter in Fortran or a subroutine name in a call statement,
- two or three expressions associated with the node (see the following sections for details),
- the control children of the node.

Among them, expressions give rise to one level of refinement of our syntax tree. A statement node can have up to three expressions nodes associated with it, we also call them *low level expressions.*

Expression means lists or algebraic expressions of variables, constants or functions. For example, a Fortran assignment statement has a left hand side expression and a right hand side expression.

Besides an integer *identifier*, each expression node has four components. The first is its type. The second is an optional symbol reference. The third and fourth components are the *left* and *right* operand expression nodes. To describe expressions we can use the following notation.

```
(N[TYPE,symbol_id]  left_subtree  right_subtree)
```

where N is the id, TYPE is its type, symbol_id is for possible associated symbol table identifier. If both the left_tree and right_tree are empty, we may simply denote the expression node as `N[TYPE,symbol_id]`

Thus, the expression `X+1` in the above example program takes the form

```
(1[ADD_OP]  2[VAR_REF, X]  3[INT_VAL, 1])
```

As indicated above, an IR of a program consist of the AST and a symbol table, which records various attributes about each symbol appearing in the program.

During the construction of the program IR, these records will be linked to the bif nodes and the low-level nodes described above So when traveling along the parse tree, all information about the symbols can be obtained.

5 Predecessors of the system

The front-end, as a product, is cultivated from two previous systems.

One of them is Sage++, developed by University of Indiana. Its data structure for internal representation is our starting point for our IR (after comparing with a different philosophy such as employed in ParaSoft Fortran 90 front-end, which is heavily language dependent.) The major rationals for this decision include

- It supports interoperability among different languages.
- Hopefully we may be able to use existing Sage utilities and library.

though this has brought us some problems, particularly in the complexity in defining specific data structure for each language construct/element. We need to greatly extend the current Sage definitions (it supports F77 and partially F90) while maintaining compatibility with it.

Another one is a Fortran 90 to FORTRAN 77 translator developed at Harbin Institute of Technology(HIT). The translator takes as input Fortran 90 source code, and translates it (without building an internal representation) into FORTRAN 77 code. Although some limitations existed in the translation (such as recursive procedures), Fortran 90 syntax is covered pretty well in its grammar rules. We did not adopt Sage's parser since we consider (1) its yacc rules does not covered enough grammar; (2) its lexer is written in C, which is hard to extent.

Once such decisions were made, the technical approaches are straightforward.

- Derive a specification of internal representation of HPF 1.0 program, taking into account the compatibility with Sage's implementation. (PACT of Peking University.)
- Extent the Fortran 90 parser to HPF 1.0 parser. (PACT of Harbin Institute of Technology.)
- Construct IR based on the specification and the parser. (PACT of Harbin Institute of Technology.)
- Develop a semantics routine. (PACT of Peking University.)
- Modify Sage's utilities for IR rendering (PACT of HIT) and unparsing (NPAC of SU).
- System integration. (NPACT)

During the process, we have kept our intermediate representation as compatible as possible to that of Sage++, so that the original data structure is consistent with a statement-level representation of the PDG for structured program [2], which makes it possible to do some future work using the library provided by Sage++.

6 People involved in this work

This work is a result of collaboration, in about 6 months period, of three parties designated in author part of this document. Many people are involved from time to time. This section is dedicated to them.

- NPAC at Syracuse University: D. B. Carpenter, Kivanc Dincer, Geoffrey C. Fox, Don M. Leskiw, Xiaoming Li, Guansong Zhang.
- PACT at Harbin Institute of Technology: Binxing Fang, Zhaojun Gu, Xiaoming Li Xinying Li, Guansong Zhang, Limei Zhang, Qiang Zheng, Jidong Zhong.
- PACT at Peking University: Xu Cheng, Wenkui Ding, Xiaobing Feng, Xiaoming Li, Yu Li, Zhenlin Wang, Zhuoqun Xu, Mi Yan, Gengbin Zheng.

Incidentally, we see a 'virtual' institute **NPACT** has been virtually formed along with this collaboration: `NPACT = NPACT + PACT`.

7 Conclusion

The process and result of this work have at least the following significance.

- Reached a quality public domain HPF 1.0 front-end, which sets up a sound basis for further compilation work. At time of this writing, we do not see any claimed public domain HPF front-end as complete as this one, and as thoroughly tested as this one. Clearly, the front-end is not only good for HPF, but also good for Fortran 90 and Fortran 95.

- Gained experience in conducting an international collaboration on a voluntary basis. The central technical interface for this collaboration is our intermediate representation. Collaboration on such an interface, which itself is changing over time, is not a easy task, don't mention people only communicate via email.
- Educated people. Most of the people involved are graduate students. They certainly learned a lot, especially in the methodology of non-toy software construction and remote collaboration with other people.

As far as the front-end itself is concerned, we see the following points for improvement.

- Further debugging. Thus, bug report is welcome from the community.
- Producing dedicated Fortran 90 or Fortran 95 front-end from it.
- Providing more utilities around it. In particular, we plan to port and extend Sage++ library on top of intermediate representations of HPFfe.

References

[1] International Standards Organization, ISO/IEC 1539: 1991, Information Technology — Programming languages — Fortran, Geneva, 1991.

[2] A Class library for Building Fortran 90 and C++ Restructuring Tools, Nov. 1993, http://www.extreme.indiana.edu/sage/index.html.

[3] HPFF, High Performance Fortran Language Specification 1.0, May. 1993, Rice University, Houston, Texas.

[4] D. Gannon, J. K. Lee, et al, "SIGMA II: A tool kit for building parallelizing compilers and program analysis systems," IFIP Transactions, A-11, Programming Environments for Parallel Computing, N. Topham, et al, Ed. North-Holland, 1992, pp17-36.

[5] D.B. Carpenter, PCRC Fortran 90 and HPF Syntax Test Suite, 1996

[6] ISO, Fortran 95 Standard, final working document (X3J3/95-007R2), Nov. 1995.

[7] NPACT, The Implementation of HPF Front-end, internal technical report, NPAC, February, 1996.