Fortran Program Specialization

Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, Robert Glück[1]

University of Technology Vienna, Institut für Computersprachen,
Argentinierstraße 8, A-1040 Vienna, Austria
E-mail: e1802gab@vm.univie.ac.at

Abstract. We have developed and implemented a partial evaluator for a subset of Fortran 77. A partial evaluator is a tool for program transformation which takes as input a general program and a part of its input, and produces as output a specialized program. The goal is efficiency: a specialized program often runs an order of magnitude faster than the general program. The partial evaluator is based on the off-line approach and uses a binding-time analysis prior to the specialization phase. The source language includes multi-dimensional arrays, procedures and functions, as well as global storage. The system is presented and experimental results are given.

1 Introduction

Partial evaluation is a principle for program transformation which reconciles generality with efficiency by providing automated specialization and optimization of programs. It has proven its usefulness in numerous areas ranging from the specialization of scientific computations [7,8,6] to specialization of general scanners and parsers [24,20], and to automatic compiler generation [17,2]. Partial evaluation has been subject of a rapidly increasing activity over the past decade due to its recent advances both in theory and practice, see e.g. [9,13,18].

For example, specializing scientific computation algorithms by partial evaluation may give substantial savings. General programs for several diverse applications including orbit calculations and computations for electrical circuits have been sped up by specialization to particular planetary systems and circuits [7,8,6].

Despite the successful application of partial evaluation to declarative languages, such as Scheme or Prolog, only few attempts have been made to study partial evaluation of imperative languages and only few results have been reported so far. Our goal was to extend previous work on partial evaluation of imperative languages and to obtain more results in this important application area. In this paper we present a partial evaluator for a subset of Fortran and report our results for several numerical and non-numerical applications.

Our work differs from most previous works by applying partial evaluation in a more realistic setting. We use Fortran which, after all, is the most widely used language for scientific and engineering applications [14], a commercial Fortran compiler that generates high-quality machine code, and we implemented a partial evaluator for a subset of Fortran that is based on current partial evaluation technology. This extends previous works on the automatic partial evaluation of imperative languages. To the best of our knowledge this is the first partial evaluator fully implemented for this language.

Why Fortran? We chose this imperative language for several reasons. There is still a large gap between the state-of-the-art of current research in partial evaluation and its potential significance. Due to the widespread use of Fortran over a period of more than 30 years, a vast body of expertise is available in the form of standard libraries representing an enormous amount of human effort and investment. It is unlikely that existing applications and libraries will be ported to other languages in order to take full advantage of partial evaluation.

The paper is organized as follows. In Section 2 we review partial evaluation. In Section 3 the subset of Fortran 77 is defined. The system structure and the internal language CoreF are described in Section 4. Experimental results are given in Section 5. Related work is discussed in Section 6. Finally, Section 7 gives conclusions and directions for future work. This paper reports on work done during the period March 1992 to January 1993.

---

[1]Partially supported by the Austrian Science Foundation (FWF) under grant number J0780 & J0964. Current address: DIKU, Dept. of Computer Science, Univ. of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark. E-mail: glueck@diku.dk

2 Partial Evaluation

A partial evaluator is a tool for specializing programs with respect to a part of its input. Given a *source program* S and some part of its input, a partial evaluator generates a *residual program* R that returns the same result as S when given the remaining input. The ultimate goal of partial evaluation is to improve the efficiency of the source program by exploiting the known input in advance.

The basic methods of partial evaluation are the *specialization* of program points to known values, the *reduction* of partially known expressions, and the *evaluation* of known statements. Every statement (expression) in a program can be classified as computable either at specialization time (*static*) or at run time (*dynamic*). This classification can be done before specialization (off-line) or during specialization (on-line). On-line techniques for imperative languages have been suggested in [19]. Using off-line techniques, additional information such as the live of static variables may be given to the partial evaluator enabling a better control [1]. We use off-line techniques for the partial evaluation of Fortran.

Constant folding is a familiar compiler optimization and is clearly an instance of partial evaluation. Partial evaluation encompasses a number of traditional optimization techniques, as explained in [16].

Example 1. Consider a program `power` computing the function `x` to the `n`th. The source program in Fortran is shown to the left, the specialized version to the right (assuming that `n` is static (`9`) and `x` is dynamic).

```
c Compute x to the n'th
      INTEGER n, x, power
      ...
      power = 1
1     IF (n .GT. 0) THEN
2        IF (MOD(n,2) .EQ. 0) THEN
            n = n/2
            x = x*x
            GOTO 2
         END IF
         n = n-1
         power = power*x
         GOTO 1
      END IF
      ...
```

```
c Specialized with respect to n=9
      INTEGER x, power
      ...
      power = 1
      power = power*x
      x = x*x
      x = x*x
      x = x*x
      power = power*x
      ...
```

Underlined constructs are dynamic; all other constructs are static. Computations depending only on the static `n` can be executed; the actions depending on the dynamic `x` are deferred. At specialization time the partial evaluator executes the non-underlined statements, and generates code for the underlined ones.

The specialized program in Example 1 can, of course, be further optimized, but this can be accomplished using an optimizing Fortran compiler. On the other hand, since Fortran compilers lack binding-time information, it is unreasonable to expect a compiler to execute static statements and produce specialized programs. The example also illustrates the trade off between speed and size (for huge static `n` the straightforward specialization may be undesirable).

3 A Subset of Fortran 77

We selected a subset of Fortran 77 which is large enough to include characteristic features of the language [5,14], and at the same time small enough to develop and fully implement the partial evaluator.

The subset, called F77, includes multi-dimensional arrays, functions and procedures, and `COMMON` statements (to specify global storage accessible within functions and procedures). Statements include assignments, nested conditionals `IF`, unconditional jumps `GOTO`, procedure calls `CALL`, the `RETURN` and the `CONTINUE` statement. The result of a function call has to be stored in a variable at the left-hand side of the call. Expressions include integer and real constants, identifiers, indexed arrays, arithmetic and relational operators. Simple input/output facilities are supported. Parameters are passed *call-by-reference* to functions and procedures (they may be modified in a function or procedure). The syntax of F77 is defined in Figure 1.

For simplicity, we assume that INTEGER and REAL are the only base types (other base types can be treated in a similar manner) and that not more than one COMMON region is used in a program. These assumptions can be generalized easily.

We require that all programs are 'well-typed', that is, the dimensions of formal and actual array parameters in procedure and function calls must match. For example, Fortran 77 allows programmers to pass, say, a 2-dimensional array to a parameter declared as a 3-dimensional array. This possibility is excluded in F77. The subset does not include the EQUIVALENCE statement for sharing storage units (it instructs the compiler to arrange for variables and arrays to access the same physical storage). These features have led to certain peculiarities in the style of Fortran programming. Dealing with such 'tricks' was beyond the scope of this work.

```
id     ∈ Identifiers             ifid   ∈ Intrinsic Function Identifiers   real  ∈ Reals
fid    ∈ Function Identifiers    label  ∈ Labels                           type  = INTEGER, REAL
sid    ∈ Procedure Identifiers   int    ∈ Integers

F77       ::= [PROGRAM sid] comdecl* vardecl* stmt+ END def*                   F77 program

comdecl   ::= COMMON id vardecl+                                               Declarations
vardecl   ::= type id   |   DIMENSION id '(' int+ ')'
def       ::= type FUNCTION fid '(' id* ')' comdecl* vardecl* stmt+            Functions
            | SUBROUTINE sid '(' id* ')' comdecl* vardecl* stmt+               Procedures

stmt      ::= label stmts  |  stmts                                            Statements
stmts     ::= lval '=' exp  |  id '=' fid '(' exp* ')'  |  GOTO label
            | DO label id '=' exp ',' exp [',' exp]  |  CONTINUE  |  RETURN
            | IF '(' exp ')' THEN stmt+ [ELSE stmt+] END IF  |  CALL sid '(' exp* ')'
            | END  |  READ '*,' lval  |  PRINT '*,' lval

exp       ::= int  |  real  |  id  |  id '(' exp+ ')'                          Expressions
            | ifid '(' exp+ ')'  |  exp bop exp  |  '(' uop exp ')'
lval      ::= id  |  id '(' exp+ ')'                                           Left-expressions
bop       ::= '+'  |  '-'  |  '*'  |  '/'  |  '**'                             Binary operators
            | '.LT.'  |  '.LE.'  |  '.GT.'  |  '.GE.'  |  '.NE.'  |  '.EQ.'
uop       ::= '-'                                                              Unary operators
```

Figure 1: Syntax of F77, a subset of Fortran 77.

4 The Partial Evaluator for F77

The input and output of the partial evaluator are programs written in F77. The partial evaluator is *off-line*, meaning that before a source program is specialized with respect to the given input, it is binding-time analyzed. Source and residual programs can be compiled directly into executable machine code using any Fortran 77 compiler. The partial evaluator itself is fully implemented in Fortran 77 and is therefore highly portable. The system has three main phases (Figure 2):

- The *preprocessing phase* translates an F77 source program into an intermediate language CoreF and the binding-time analysis (BTA) annotates all statements (expressions) in the source program as either static or dynamic corresponding to the static/dynamic classification (S/D) of its input. The output of the preprocessing phase is an annotated CoreF program.
- The *specialization phase* takes the annotated CoreF program and the static data as input and specializes the program with respect to the static data. This phase, the kernel of the system, contains an interpreter (INT) for the evaluation of static CoreF statements. The output of the specialization phase is a specialized CoreF program.
- The *postprocessing phase* translates a specialized CoreF program into F77.

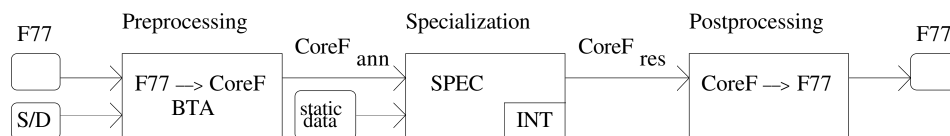

Figure 2: The structure of the Fortran specializer.

4.1 The Internal Language CoreF

The partial evaluator uses an internal language, called CoreF. The syntax is defined in Figure 3. The main difference to F77 is that all statements (expressions) are marked by a keyword, such as `VAR` and `ASSIGN`. The conditional is simplified and includes only two labels for jumping to the then- and the else-branch. The relational operators are replaced by calls to external functions and `DO` statements are translated into corresponding combinations of `IF`s and unconditional jumps. The language is equivalent to F77 in all other respects.

```
id     ∈  Identifiers              label  ∈  Labels      type   =  INTEGER, REAL
fid    ∈  Function Identifiers     int    ∈  Integers
sid    ∈  Procedure Identifiers    real   ∈  Reals

CoreF     ::=  vardecl* def+                                               CoreF program

vardecl   ::=  type id        |   DIMENSION id '(' dim+ ')'                Declarations
dim       ::=  ICST int
def       ::=  FUNCTION type fid '(' par* ')' vardecl* lstmt+              Functions
           |   SUBROUTINE sid '(' par* ')' vardecl* lstmt+                 Procedures

lstmt     ::=  label stmt                                                  Statements
stmt      ::=  ASSIGN lval '=' exp      |   IF '(' exp ')' label label
           |   SCALL sid '(' exp* ')'   |   FCALL id '=' fid '(' exp* ')'
           |   GOTO label   |   CONTINUE   |   RETURN   |   END

exp       ::=  ICST int     |   RCST real  |   VAR id                      Expressions
           |   ARRAY id '(' exp+ ')'   |   ECALL fid '(' exp* ')'
           |   BOP exp bop exp          |   UOP uop exp
par       ::=  VAR id                                                      Parameters
lval      ::=  VAR id   |   ARRAY id '(' exp+ ')'                          Left-expressions
bop       ::=  '+'   |   '-'   |   '*'   |   '/'   |   '**'                Binary operators
uop       ::=  '-'                                                         Unary operators
```

Figure 3: Syntax of the internal language CoreF.

4.2 The Preprocessing Phase

The preprocessing phase of the system takes as input a source program written in F77 and a static/dynamic classification of its input, and returns as output an annotated CoreF program (Figure 4). The preprocessing phase includes a translator from F77 to CoreF and a binding-time analysis. The binding-time analysis takes a CoreF program and a static/dynamic classification of its input, and returns a CoreF program in which every statement (expression) is annotated as either static or dynamic. The translator uses conventional methods and is not described here. The binding-time analysis is described below.

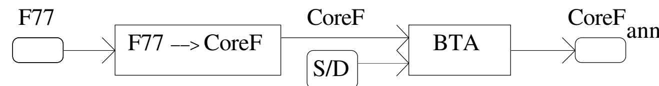

Figure 4: Structure of the preprocessing phase.

Binding-Time Analysis

The binding-time analysis (BTA) annotates each statement (expression) in the source program as either static or dynamic. The task of the BTA is to determine a *congruent* division of the variables in the source program, that is, to classify all variables as static or dynamic. A division is called congruent if static variables depend only on static values. Any variable that depends on a dynamic value is classified as dynamic. A variable which one time depends on a static value and another time on a dynamic value must be classified as dynamic because its value cannot always be computed during specialization. The goal of the BTA is to find a 'good' division: a static/dynamic classification which is congruent and enables as many static computations as possible.

The BTA is implemented by an approximative algorithm which iterates over the static/dynamic division until a stable classification is reached. The analysis terminates because there are only finite many variables in the source program, finite many different annotations, and no annotation is changed

back from dynamic to static. The BTA uses an analysis based on abstract interpretation. The analysis is similar to the one described in [25]. For a comprehensive discussion of the BTA see [18].

The output of the BTA is an annotated CoreF program. To represent static/dynamic annotations a *two-level* version of CoreF is used in which every keyword appears in either of two versions: in a normal version (= static) and in an underlined version (= dynamic). An underlined keyword is written with an underscore in front (i.e. '_'). The use of this syntax implies that every statement (expression) in the source program can only be given one annotation. The BTA is *monovariant*.

Example 2. The two-level version of the `power` program written in CoreF is shown below. The declaration `INTEGER n` denotes a static variable; the declaration `_INTEGER x` denotes a dynamic variable. Similarly, `ASSIGN` is a static assignment, and `_ASSIGN` is a dynamic assignment, and so on.

```
c Annotated CoreF version of power
     INTEGER n
     _INTEGER x
     _INTEGER power
     ...
3    _ASSIGN _VAR power = ICST 1
4    IF (ECALL GT (VAR n, ICST 0)) 5 12
5       IF (ECALL MOD (VAR n , ICST 2 )) 9 6
6          ASSIGN VAR n = BOP VAR n / ICST 2
7          _ASSIGN _VAR x = _BOP _VAR x * _VAR x
8          GOTO 5
9          ASSIGN VAR n = BOP VAR n - ICST 1
10         _ASSIGN _VAR power = _BOP _VAR power * _VAR x
11         GOTO 4
12   ...
```

4.3 The Specialization Phase

The input to the specialization phase is the annotated source program (in the two-level version of CoreF) and the static values. The output is a CoreF program in which all static computations are executed. The specialization phase follows the annotations made by the binding-time analysis: it executes static statements, reduces partially static expressions, and specializes dynamic basic blocks with respect to static values. For the specialization of source programs a variant of the *polyvariant specialization* is used [11].

Static Statements

The specialization of a sequence of statements is simple: the specializer runs through the statements step-by-step generating code for dynamic statements and executing static ones. For executing static statements (expressions) the specialization phase includes an interpreter for CoreF which modifies the *static storage* provided for static values. Note that only those statements (expressions) are annotated as static which do not depend on dynamic values. That is, their effect can be computed at specialization time. Static `IF` statements are handled by evaluating the test expression and jumping to the corresponding branch. Static `GOTO`s, functions, and procedures are handled in a likewise manner.

Dynamic Conditionals

The partial evaluator generates code for both branches of a *dynamic conditional*. An `IF` statement is annotated as dynamic if the test expression is dynamic. This implies that the test cannot be decided at specialization time and the partial evaluator generates specialized code for both branches. This is similar to a compiler which also generates code for both branches of a conditional. However, in a partial evaluator both branches have to be specialized with respect to the same static storage. The effect of both branches has to be computed on a copy of the same static storage (each branch may have different effects on the static storage).

The specialization of dynamic functions and procedures is depth-first. It is necessary to specialize a function or procedure before the statements succeeding the call are specialized. The code generation for statements is given in pseudo-code (Figure 5). The static versions of `RETURN` and `END` do not appear in Figure 5 because they occur only in fully static functions and procedures which are handled by the interpreter during specialization.

```
repeat
  switch(stmt_at(label))
    case  ASSIGN:   <evaluate expression & update lval>; label=next(label);
    case _ASSIGN:   gen_assign(<reduce expression>); label=next(label);
    case  IF:       if (<evaluate expression>)
                    then label=<then label>
                    else label=<else label>;
    case _IF:       gen_if(<reduce expression>);
                    add_pending(<then label>);
                    add_pending(<else label>); stop=TRUE;
    case  GOTO:     label=<goto label>;
    case _GOTO:     gen_goto();
                    add_pending(<goto label>); stop=TRUE;
    case  SCALL:    <perform static procedure>; label=next(label);
    case _SCALL:    gen_scall(<reduce expressions>);
                    <specialize procedure>; label=next(label);
    case  FCALL:    <perform static function & update variable>; label=next(label);
    case _FCALL:    gen_fcall(<reduce expressions>);
                    <specialize function>; label=next(label);
    case  CONTINUE: label=next(label);
    case _CONTINUE: gen_continue(); label=next(label);
    case _RETURN:   gen_return(); stop=TRUE;
    case _END:      gen_end(); stop=TRUE;
until (stop);
```

Figure 5: Code generation for statements.

Specializing Dynamic Basic Blocks

A *dynamic basic block* (DBB) [1] is a collection of basic blocks $B_0 \ldots B_n$ such that there is

(i) a dynamic transition to B_0 (the first basic block);
(ii) all transitions between $B_0 \ldots B_n$ are static;
(iii) all control statements (`IF`, `GOTO`) in the leaf basic blocks are dynamic.

Recall that a *basic block* is a sequence of statements such that there is a distinguished entry point (the first statement), and the last statement is a control statement.

All reachable DBBs are specialized with respect to the static storage. A *done-* and a *pending-list* are used to keep track of already specialized blocks and those remaining to be specialized. Each time a dynamic conditional is reached the done-list is searched whether the 'target' DBB was already specialized with respect to the same static storage. If so, the corresponding jump label is inserted. Otherwise, the DBB has to be specialized with respect to the static storage.

Consider a dynamic `IF`-statement. The specializer generates code for the dynamic `IF`-statement and searches the done-list in order to check whether the corresponding dynamic basic block was already specialized with respect to the same static values. If so, the corresponding label is used in the `IF`-statement. If not, the then-branch is specialized with respect to the static values and this is recorded in the done-list (the same is done with the else-branch).

Note that the same DBB may get specialized with respect to different static storage. This provides a polyvariant specialization of DBBs (the BTA is monovariant with respect to static/dynamic annotations). The same method is used in the C-specializer [1].

4.4 The Postprocessing Phase

The postprocessing phase of the partial evaluator is a translator from CoreF to F77. The result is a residual program which can be compiled directly into machine code by a Fortran 77 compiler. Other transformations such as additional unfolding can be performed during this phase. The translation uses conventional methods and is not described here.

5 Experiments

We used the Fortran partial evaluator for a number of numerical and non-numerical experiments. We present our results using the cubic splines interpolation, a general scanner, and an interpreter simulating a stack machine. The last two examples were originally proposed by Pagan [22,24]. We compare our results with those reported by Andersen [1]. More results for the specialization of numerically oriented Fortran programs can be found in [6].

The *run times* are given in user seconds using the Lahey Fortran compiler (version 5.01) and an IBM AT386/25MHz. The *size* of the programs is *lines* of 'pretty printed' Fortran source code and *KBytes* of compiled code. These results have been obtained with the latest version of the system. Note that a fair comparison with the run times given in [1] is not possible because the Fortran compiler uses other optimizations and generates machine code for a different platform.

5.1 Specialization of a Numerical Program

Interpolation is an important numerical method used in a wide range of scientific and engineering applications. The *cubic splines interpolation* approximates the values between two measured values with a cubic polynomial. To avoid 'edges' two consecutive polynomials must have the same values and derivatives at the point they meet. The intended result is a continuous function that connects all measured values.

Different range conditions restrict the possible behavior of the curve at the beginning and the end. (i) *Natural range conditions* make the bending in the endpoints zero, that means the second derivative disappears. (ii) *Periodical range conditions* imply the first and the last value to be equal and are used in case the function is known to be periodical. (iii) *De Boor range conditions* require the first and last two polynomials to be equal. In each of these cases the construction of the derivatives requires solving a tridiagonal equation.

We implemented the cubic splines interpolation in Fortran 77. The program returns the value of the y-coordinate at the desired x-coordinate. In our experiments we assumed that the number of measured values, their distance and the type of range conditions (e.g. *type=periodical*) are known before run time. The measured values and the x-coordinate of the value to be computed are assumed to be unknown. For all measurements the program was executed 1000 times.

By specializing the program to *natural* and *de Boor* range conditions a speedup of 3.3 is obtained. The speedup of the program specialized to *periodical* range conditions is 4.6. This speedup is higher because one of the two invocations of the tridiagonal equation solver becomes fully static and can be executed at specialization time (due to our monovariant binding-time analysis a copy of the equation solver had to be used).

This example demonstrates that a general purpose program, in our example the cubic splines interpolation, can be specialized with respect to specific arguments resulting in large speedups. It is obvious that it is easier to specialize the general purpose program, than to write each specialized version by hand, especially if some of the input assumptions may change.

Cubic Splines Interpolation	Run time	Speedup	Lines	KBytes
Splines (type=natural)	1.26		257	44.6
$Splines_{type=natural}$	0.38	3.3	161	41.2
Splines (type=periodical)	1.76		257	44.6
$Splines_{type=periodical}$	0.38	4.6	172	41.4
Splines (type=de Boor)	1.26		257	44.6
$Splines_{type=de\ Boor}$	0.38	3.3	163	41.3

5.2 Specialization of a General Scanner

This example uses a general scanner written in Fortran 77. The general scanner takes as input a token description (*table*) and a stream of characters. By specializing the general scanner with respect to a token description, a lexical analysis tailored for the given set of tokens is produced.

The static input in this example is the same as in Pagan and Andersen: a token description of 8 tokens and a stream of 16 characters (repeated 64000 times). The run times represent the performance of the scanners and do not include the time required for reading the input stream from a file. For a detailed description of the general scanner see [24, Sect. 4.2]. The speedup of the specialized program

is 4.2. Andersen reports a speedup of 1.2 using the C-specializer and the same input. If a table involving more backtracking operations is used, a larger speedup is obtained.

Using a table for recognizing 36 tokens of a small Fortran-like imperative language, and an input stream of 1100 characters (repeated 100 times) we get a speedup of 5.3.

General Scanner	**Run time**	**Speedup**	**Lines**	**KBytes**
Scan1 (table1, stream1)	40.97		35	52.1
$Scan1_{table1}$ (stream1)	9.77	4.2	165	53.0
Scan2 (table2, stream2)	6.05		35	52.1
$Scan2_{table2}$ (stream2)	1.15	5.3	1530	64.6

5.3 Specialization of an Interpreter

In the following example an interpreter for a 'polish-form' assembler language is given which is the same as in [22]. The interpreter simulates a straightforward stack machine with instructions like `STORE`, `LOAD`, and `ADD`. The 'polish-form' program *primes* which is given to the interpreter reads a number *n* and computes the first *n* prime numbers.

The specialization of the interpreter with respect to this program yields a primes program in Fortran. That is, specializing the interpreter with respect to a 'polish-form' program corresponds to a translation from 'polish-form' into Fortran. The results show that the specialization pays off. The speedup of the specialized program is 14.6. Andersen reports a speedup of 7.0 using the C-specializer and the same input. The speedup is obtained because the specializer removes the interpretative overhead at specialization time.

A comprehensive discussion of the translation achieved by partial evaluation of an interpreter can be found in [18]. However, this effect has not been studied in practice in the context of imperative languages because only few partial evaluators have been build for imperative languages so far.

'Polish-form' Interpreter	**Run time**	**Speedup**	**Lines**	**KBytes**
Interpreter (primes, 500)	813.51		141	48.6
$Interpreter_{primes}$ (500)	55.75	14.6	134	46.6

5.4 Performance of the Partial Evaluator

To illustrate the performance of the system, the run times of the binding-time analysis and the specialization phase are shown. The times are tentative since the phases were not optimized for speed (the numbers do not include the time required to read/write files from disk).

Program \ *Phase*	***BT Analysis***	***Specialization***
Cubic Splines (natural)	1.65	1.38
Cubic Splines (periodical)	1.65	1.49
Cubic Splines (de Boor)	1.65	1.39
Scan1 (table1)	0.11	1.37
Scan2 (table2)	0.11	14.99
Interpreter (primes)	0.38	1.53

Partial evaluation is advantageous if one input (*table*, *primes*) to a source program (*scanner*, *interpreter*) changes less frequently than another (*stream*, *n*). It gives substantial savings when a residual program is used several times. But partial evaluation can even pay off in a single run if the specialization time plus the residual run time is faster than the run time of the source program. This is the case in the interpreter example.

Note, that the annotated version of a source program can be reused as long as the static/dynamic classification of the input does not change. For example, in the case of the cubic splines interpolation the binding-time analysis needs to be performed only once (not three times). While the run time of the binding-time analysis depends on the size of the source program and the initial static/dynamic classification, the specialization phase depends on the annotated program and the static values. The run time of the specialization phase depends on the size of the generated code. Depending on the static values the generated residual program can be several times larger than the original source program (due

to the polyvariant specialization). In our experience the binding-time analysis usually requires the smallest portion of the total run time.

6 Related Work

Ershov and his group were the first who investigated imperative languages and mixed computation [15,16,11]. Pagan [24] describes several manual methods similar to the partial evaluation of imperative languages and studied the specialization of general parsers [23]. The examples are taken from his works. The partial evaluator presented in this paper automates several of these techniques.

Meyer describes an on-line partial evaluator for a Pascal-like language [19], allowing potentially more specialization during partial evaluation than an off-line specializer. Recently Nirkhe and Pugh [21] developed an off-line system for a similar language for the specialization of hard real-time problems. Gomard and Jones report a self-applicable partial evaluator for a small untyped flow-chart language with S-expressions from Lisp as the only data structure [17].

Andersen developed the first self-applicable partial evaluator for a large subset of an imperative language, namely the programming language C [1-3]. His system handles recursive functions and procedures, and multi-dimensional arrays. Recently it was extended to handle pointers [4]. On the one hand, there are no pointers and recursions available in Fortran 77, but on the other hand it is necessary to deal with other Fortran specific problems. Both systems use the off-line technique.

The application of partial evaluation to software maintenance is discussed by Blazy and Facon [10]. Their partial evaluator aims at improving the readability of Fortran programs, but not at improving the efficiency of the programs. Coen-Porisini et al. [12] suggest program specialization using symbolic simplification, theorem proving and other optimizations in an interactive environment in order to support the reuse of software in the context of Ada.

We investigate the application of partial evaluation to numerically oriented computation in scientific and engineering applications in [6]. The results demonstrate that existing partial evaluation technology is strong enough to improve the efficiency of a large class of numerical programs.

7 Conclusion and Future Work

We developed and fully implemented a partial evaluator of a subset of Fortran 77. The results compare favorably with the results reported for other partial evaluators, including those for declarative programming languages. This demonstrates that partial evaluation and imperative languages do not contradict and that their integration is on the agenda.

As next step it will be interesting to investigate the specialization of larger applications, in particular, programs for scientific and engineering applications. We expect that the speed-ups will be smaller, but still large enough to be useful in practice. Another promising direction is to exploit the parallelism exposed by partial evaluation on parallel computing systems, e.g. by using Fortran parallelizers.

There is still room for improvements. The partial evaluator could exploit more information about dynamic data. A combination of partial evaluation and more traditional compiler methods would be useful, such as algebraic identities and life/dead analysis. Using partial evaluation as a development tool in practice remains a challenging problem. Industrial-strength partial evaluators will have to tackle a variety of existing programs and libraries. To obtain good results one will need to deal with certain peculiarities and various programming styles. For example, passing arrays of different dimensions as arguments and sharing the same physical storage is frequently used in existing Fortran programs.

Last but not least, one should note that the implementation costs for the Fortran partial evaluator were higher compared with similar systems for declarative programming languages since Fortran is notoriously weak in symbol manipulation. However, the partial evaluator is now fully implemented in Fortran.

Acknowledgments. Special thanks are due to Lars Ole Andersen for his continued interest and encouragement, and for thorough comments on an earlier version of this paper. This has been a great help. We would like to thank Peter Kowatsch and Wolfgang Lair from *MSB Software* Vienna for their technical assistance and – as well as Hans Moritsch – for many valuable discussions about Fortran.

References

1. Andersen L. O., C program specialization (revised version). DIKU, Department of Computer Science, University of Copenhagen. DIKU Report No. 92/14, 1992.
2. Andersen L. O., Partial evaluation of C and automatic compiler generation. In: Kastens U., Pfahler P. (eds.), *Compiler Construction. 4th International Conference*. (Paderborn, Germany). Lecture Notes in Computer Science, Vol. 641, 251-257, Springer-Verlag 1992.
3. Andersen L. O., Self-applicable C program specialization. In: *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. (San Francisco, California). 54-61, Yale University, Dept. of Computer Science 1992.
4. Andersen L. O., Binding-time analysis and the taming of C pointers. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (Copenhagen, Denmark). 47-58, ACM Press 1993.
5. ANSI, *Programming Language Fortran. American National Standard X3.9-1978.* American National Standards Institute: New York 1978.
6. Baier R., Glück R., Zöchling R., Partial evaluation of numerical programs in Fortran. In: *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. (Orlando, Florida). 119-132, University of Melbourne, Australia 1994.
7. Berlin A., Partial evaluation applied to numerical computation. In: *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. (Nice, France). 139-150, ACM Press 1990.
8. Berlin A., Weise D., Compiling scientific code using partial evaluation. In: *IEEE Computer*, 23(12): 25-37, 1990.
9. Bjørner D., Ershov A. P., Jones N. D. (eds.), *Partial Evaluation and Mixed Computation*. North-Holland: Amsterdam 1988.
10. Blazy S., Facon P., Partial evaluation for the understanding of Fortran programs. In: *Software Engineering and Knowledge Engineering*. (San Francisco, California). 517-525, 1993.
11. Bulyonkov M. A., Polyvariant mixed computation for analyzer programs. In: *Acta Informatica*, 21: 473-484, 1984.
12. Coen-Porisini A., De Paoli F., Ghezzi C., Mandrioli D., Software specialization via symbolic execution. In: *IEEE Transactions on Software Engineering*, 17(9): 884-899, 1991.
13. Consel C., Danvy O., Tutorial notes on partial evaluation. In: *Conference Record of the Twentieth Symposium on Principles of Programming Languages*. (Charleston, South Carolina). 493-501, ACM Press 1993.
14. Ellis T. M. R., *Fortran 77 Programming: with an Introduction to the Fortran 90 Standard (2nd Edition).* International Computer Science Series. Addison-Wesley: Wokingham, England 1990.
15. Ershov A. P., On the essence of compilation. In: Neuhold E. J. (ed.), *Formal Description of Programming Concepts*. 391-420, North-Holland 1978.
16. Ershov A. P., Mixed computation: potential applications and problems for study. In: *Theoretical Computer Science*, 18: 41-67, 1982.
17. Gomard C. K., Jones N. D., Compiler generation by partial evaluation: a case study. In: *Structured Programming*, 12: 123-144, 1991.
18. Jones N. D., Gomard C. K., Sestoft P., *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice Hall: New York, London, Toronto 1993.
19. Meyer U., Techniques for partial evaluation of imperative languages. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (New Haven, Connecticut). 94-105, ACM Press 1991.
20. Mossin C., Partial evaluation of general parsers. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (Copenhagen, Denmark). 13-21, ACM Press 1993.
21. Nirkhe V., Pugh W., Partial evaluation and high-level imperative programming languages with applications in hard real-time systems. In: *Conference Record of the Nighteenth Symposium on Principles of Programming Languages*. (Albuquerque, New Mexico). 269-280, ACM Press 1992.
22. Pagan F. G., Converting interpreters into compilers. In: *Software - Practice and Experience*, 18: 509-527, 1988.
23. Pagan F. G., Comparative efficiency of general and residual parsers. In: *SIGPLAN Notices*, 25(4): 59-65, 1990.
24. Pagan F. G., *Partial Computation and the Construction of Language Processors.* Prentice Hall Software Series. Prentice Hall: Englewood Cliffs, NJ 1991.
25. Sestoft P., The structure of a self-applicable partial evaluator. In: Ganzinger H., Jones N. D. (eds.), *Programs as Data Objects*. (Copenhagen, Denmark). Lecture Notes in Computer Science, Vol. 217, 236-256, Springer-Verlag 1986.