Using Partial Evaluation in Support of Portability, Reusability, and Maintainability

Daniel J. Salomon

Dept. of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2, E-mail: salomon@cs.UManitoba.CA

Abstract. Partial evaluation is ordinarily intended to be used to increase program efficiency. This paper shows how partial evaluation can be used in place of a preprocessor phase and of source-code templates (e.g. C++ templates or Ada generics). In this way it can be used to support portability features provided by a preprocessor, and the reusability provided by code templates, but with higher maintainability due to the simpler syntax required. The important mechanisms needed are: annotating variables and functions with an evaluation time, treating declarations as translation-time "executable" statements, treating user-defined types as translation-time variables, giving programmers control over the scope of symbols, and providing translation-time name binding. The effects of these changes on the size and complexity of a compiler are estimated. A translator for a language called "Safer_C" which supports these techniques has been implemented. Important existing C software is analyzed to evaluate the applicability of these techniques in replacing the preprocessor.

1 Introduction

Partial evaluation [2, 5] is a program transformation technique whereby program constants and input whose values are known in advance of execution are used to specialize the program for those specific values and thus obtain a faster and/or smaller program. In this paper, we show that partial evaluation can also be used in support of portability, reusability, and maintainability of programs.

One of the reasons that the languages C & C++ have been popular for **portable** programming is that their preprocessors allows the automatic tailoring of programs to particular platforms. We show how the techniques of partial evaluation can be used to eliminate the need for a preprocessor phase by providing much of the same functionality.

A motivation for the development of templates in C++ and generic procedures in Ada has been to facilitate the coding of **reusable** procedures. We also show how partial evaluation can be used to duplicate the functionality of C++ templates.

The functionality of these two features is provided with minimal changes to the syntax and semantics of the original language, and certainly with less additional syntax than is ordinarily needed to support preprocessors and templates.

The simplicity and orthogonality with which partial evaluation can replace preprocessing and templates, makes the programs more readable, and easier to manipulate with program processors such as pretty printers, structured editors, and language version updaters, thus making programs more **maintainable.** At the same time the programmer is gaining the usual efficiency benefits of partial evaluation.

This work does not attempt to present new functionality for programming languages, but rather shows how partial evaluation can provide well recognized functionality in a simpler and more consistent way.

1.1 Replacing Preprocessors

The standardization and availability of the C preprocessor, cpp, is one of the principal reasons that the C language has become so popular for the programming of portable systems. With a preprocessor, the same source program can be customized down to fine details for different target architectures and operating systems.

Despite this advantage, preprocessors have fallen into disrepute. The principal complaints against them are that:

1. They add another level to the syntax and semantics of a programming language. This causes difficulties in the description, implementation, and use of the language.
2. They usually have a different syntax and semantics from the languages they are modifying.
3. Preprocessor statements can be misused to violate the structuredness of programs by overlapping instead of nesting control constructs.

The preprocessors for the C & C++ languages have all of the above faults, but because of their advantages and widespread use, they cannot simply be discarded.

In this paper, the author proposes that the principal functionality of preprocessors can be replaced by compile-time processing. Thus the number of program processing phases will be reduced by one. This equivalent functionality is provided within a single level of syntax, and with minimal changes to the syntax of the existing language.

Although we have focused on the C & C++ languages, our belief is that other languages too would benefit from these techniques by gaining the advantages of a preprocessor phase and of templates without the disadvantages.

1.2 Replacing Templates & Generic Packages

Templates in C++ and generic packages in Ada allow a generalized version of an algorithm to be coded, and translation-time parameters to be provided that customize the source code to a specific problem. The principal reason for introducing these language features was to increase the reusability of code. The translation-time specialization of functions, however, is the trademark of partial evaluation.

By treating user-defined types as translation-time variables, partial evaluation can be extended to provide the functionality of templates and generics, while hardly increasing the size of the grammar of a carefully designed language.

1.3 The Method

In this section, we summarize the changes to the syntax and processing of a C-like language that can give the compilation phase enough computational power to allow the elimination of the preprocessor phase and provide the functionality of templates. The changes are:

1. Program entities such as variables, functions and formal parameters, can be annotated at their declaration with a designation of their evaluation time.
2. The evaluation time of program entities is propagated through a program to determine the evaluation time of expressions, and where possible, control structures.
3. Predefined types are translation-time type constants, and user-defined type names are treated as translation-time variables.
4. Control structures can be annotated with an evaluation time to assist the compiler, or to clarify the programmer's intentions.
5. Declarations are treated as compile-time "executable" statements. Thus a programmer can gain control over not only what computations are to be performed, but also over what variables and functions are to be declared. This is an essential feature if partial evaluation is to be used to replace preprocessors and templates.
6. The programmer is provided with additional control over the scope of program entities.
7. Control structures are allowed outside of any function, where only declarations are normally allowed, provided that those control structures can be evaluated at compile time and that only declarations are left as residual code.
8. A boolean operator **decl** can be used to test whether an identifier has been declared. This operator is used to provide the functionality of the C preprocessor statements `#ifdef` and `#ifndef`.
9. Translation-time name binding is provided, so that the programmer can gain control over the final names of exported program entities. Normally names for program entities are bound at coding time.

These changes are described in greater detail in subsequent sections.

1.4 Origins of This Project

The work described in this paper originated from a project to design a language called Safer_C [10], which is a modern descendant of the C language. The primary objective of the design is to produce a language that is more error-resistant than C without sacrificing any expressiveness or computational power. The plan is to continue enhancing Safer_C until it matches C++ in expressive power, but with less of the tattered syntactic baggage that C++ inherited from C.

Due mostly to the fact that it was designed over 20 years ago, the C language has many syntactic deficiencies that lead to common programming errors. Some of the best known of these errors are: using the operator = for comparison instead of ==, forgetting the closing delimiters on comments, forgetting break statements at the end of switch cases, missing or adding erroneous semicolons, erroneous type declarations, and erroneous preprocessor statements. Some of these errors can persist in a program until run time, even though they originate strictly from deficient syntax. Koenig [6] gives a readable and valuable description of most of the syntactic deficiencies of C, along with tips on how to avoid them, and they are also summarized by Salomon [10].

Except for its added compile-time functionality, Safer_C is semantically identical to C, but has most of the syntactic deficiencies eliminated by using modern conventions. The C language was chosen for modernization because, despite its flaws, it is a popular language, and is used in many important systems that would benefit substantially from greater reliability. The intention was to design Safer_C such that all existing C programs could be machine translated to Safer_C, and further maintenance of those programs could be carried out in the more-error-resistant, modern syntax.

The greatest obstacle to modernizing C that was encountered is its preprocessor phase. Since preprocessors are used to change source text, the machine translation of C programs into a new version or a different form can be blocked by even tame preprocessor statements. Sometimes the actual C program that is being manipulated cannot be known until specific values are assumed for some of the preprocessor variables, and then only the program generated by those specific values can be manipulated, not the general form of the program.

Since the existence of a preprocessor phase impedes even simple source-to-source code manipulation, it was decided that the preprocessor should be replaced early in the evolution of Safer_C so that even though the transformation of existing C programs to Safer_C would still be hard, further transformation to later versions of Safer_C would be considerably simplified.

2 Related Work

Stroustrup [11], in the design of C++, has tried to eliminate the preprocessor phase as much as possible by allowing the compiler access to the values of `const` variables, and by providing in-line functions. These mechanisms, however, do not provide any support for conditional declarations, and `#include` preprocessor statements are still acknowledged as necessary.

The PL/I language [4] provides a preprocessor language with syntax and power that is quite similar to the run-time language. The many restrictions on the preprocessor language, a few extensions over the run-time language, and scope-rule differences, however, mean that the level of evaluation-time independence is actually quite limited. Since run-time code is always treated as text by the preprocessor, the readability and maintainability of programs is impaired. PL/I also provides a separate and powerful translation-time computation ability, which results in the duplication of implementation effort for preprocessing time and translation time.

The templates of C++ and the generic packages and procedures of Ada do provide reusability of code, but do so by providing an added level of function invocation with its own fully separate syntax and semantics.

A number of other researchers have studied the applicability of partial evaluation to imperative programs. Some recent examples are: Meyer [7], Nirkhe & Pugh [8], and Baier, Glück, & Zöchling [1]. Their methods are more extensive than the ones described here in that they try to maximize the amount of computation performed before run time. The work described here relies more heavily on programmer annotations. In Safer_C, more emphasis is placed on the programmer being able to predict what computations will be done at compile time, and being able to control when computations will be performed. We concentrate on how to use partial evaluation to replace the functionality of a preprocessor and templates.

The work of Weise and Crew [12] on programmable syntax macros may seem to be similar to ours. Their method adds more programming power than text macros, but also adds a meta level of syntax, and still poses an obstacle to automatic source-to-source manipulations.

The idea of explicit control over the scope of program entities is due to Cormack [3], who also gives the benefits of such control for ordinary run-time variables.

3 Declaring the Evaluation Time of Program Entities

The mechanism used by Safer_C to replace the preprocessor phase is to provide program entities, such as types, variables, functions, and formal parameters, with an *evaluation-time* attribute. For brevity and readability in this discussion, the term "variable" will often be used to refer to all such program entities. A variable can be specified as having a translation-time evaluation by preceding its type specification with the keyword "`tran`," otherwise it is assumed to have run-time evaluation[1].

Sample declarations might be:

```
SIZE :: tran int := 10
table :: [0..SIZE-1] int
WIDTH :: tran float := SIZE/14.0
```

[1] Actually Safer_C allows the specification of one of five evaluation times: translation, linking, loading, frame allocation, and run time, and it attempts to provide the same computation power in each of those five phases. This corresponds to the principle of evaluation-time independence described by Salomon [9]. Since translation time and run time are the only phases currently operational in Safer_C, they are the only ones discussed in this paper.

In these declarations, `SIZE` and `WIDTH` are translation-time variables, and **`table`** is a run-time variable.

4 The Evaluation Time of Expressions and Statements

Methods for propagating evaluation time through a program appear in the literature (See for instance Meyer [7] or Nirkhe and Pugh [8]), so only the elements unique to Safer_C are emphasized here.

- Arithmetic Expressions (e_1 **op** e_2, **pre_op** e_1, or e_1 **post_op**) – The evaluation time of an arithmetic expression is the latest evaluation time of any of the subexpressions or variables that comprise it.
  Referencing is an exception to this rule. A reference operation takes the evaluation time of the address of the object referenced, not of the object.
  Short-circuit boolean operators are also treated specially. Consider the expression e_1 && e_2, where "&&" is the short-circuit **and** operator in C. If e_1 has an evaluation time t_1 that is earlier than t_2 the evaluation time of e_2 then if e_1 evaluates to **true** the expression takes the evaluation time and value of the second operand, whereas if e_1 evaluates to **false** then the expression takes the value and evaluation time of e_1. A corresponding calculation of evaluation time is used for the the short-circuit **or** operator.
- Assignment (Dest_Var := Source_Expr) – The expression Source_Expr must have an evaluation time earlier or equal to that of the variable Dest_Var, otherwise an error is reported. The assignment is carried out at the evaluation time of the variable Dest_Var.
- Boolean Selection (**if** (Bool_Expr) **then** Then_Stmt **else** Else_Stmt) – The selection is carried out at the evaluation time of the boolean expression Bool_expr. Only the selected statement is processed, the other statement is ignored. The ignored statement must be syntactically correct, but need not necessarily be semantically correct. In particular, if Bool_Expr is a translation-time expression, since declarations are "executed" at translation time, the effect is that no type checking is performed on the ignored statement, and any subprograms invoked need not be made available for any of the processing phases. The selected statement may have an evaluation time later than Bool_Expr, and then would be processed as residual code.
- Case Selection (**switch** (Ord_Expr) Case_List) – The execution of case selection statements is analogous to boolean selection. There is an added provision that, for efficiency reasons, the evaluation times of the case-label expressions must be translation-time.
- While Loops (**while** (Bool_Expr) Loop_Body) – If Bool_Expr evaluates to **false** at translation time, then the body of the loop is ignored. Otherwise, the loop is considered to be a run-time loop, unless it is explicitly annotated as a translation-time loop. To mark a **while** loop as a translation-time loop it is preceded by the marker **`#tran#`**. This method of determining evaluation times is used because unbounded run-time loops are a common mechanism

in C programs, and it is not always possible to determine whether or not they can be replaced by unrolled code at translation time. Rather than attempting to make such a prediction, Safer_C unrolls **while** loops only if explicitly requested to do so by the programmer.
If Bool_Expr is a *non-false translation-time expression*, and the programmer requests translation-time evaluation, then the loop is unrolled until Bool_Expr becomes false at or until a **break** statement is executed.
- For loops (**for** (Init_Stmt; Test_Expr; Inc_Stmt) Loop_Body) – Safer_C **for** loops have a standard mapping into while loops, and that mapping holds for determining the evaluation time of the **for** loop as well. The mapping is the same as for the C language: the statement Init_Stmt precedes the loop, the expression Test_Expr is used as the boolean control expression of the generated while loop, and the statement Inc_Stmt is inserted as the last statement of the loop body.
- Do Loops (**do** Loop_Body **while** (Bool_Expr)) – Safer_C **do** is a post-test loop that is handled in a fashion closely analogous to the pretest **while** loop.
- Goto statements are currently allowed only for run-time evaluation.

5 The Evaluation Time of Function Invocations

In Safer_C, as in C, functions returning void are equivalent to procedures, and hence this discussion of function invocation applies equally well to procedure invocation. The number of possible strategies for partial evaluation of function invocations is large. In keeping with the overall simple implementation philosophy of the C language, Safer_C also has a fairly simple strategy. Safer_C provides three kinds of partial evaluation of functions, each described in one of the following subsections. Function invocations not processed in one of these three ways are left as ordinary run-time function invocations.

5.1 Replacement by a Result

In some situations, it is possible to replace a function invocation by a function result even if the function is declared as having run-time evaluation. This kind of partial evaluation is done if:

1. The values of all of the actual arguments and external variables accessed by the function are known at translation time,
2. Either the source code for the function is available at translation time, or the object code for the function is available and a dynamic loader is provided to the translator, and
3. The function has no side effects.

Since a programmer may want to invoke the version of a function provided on the target machine, rather than the version provided on the compilation machine, a "time-warp" notation is available to delay the execution of functions until run-time.

5.2 In-Line Expansion

A function is expanded in-line, such that the function invocation is replaced by any computations specified by the function that cannot be carried out at translation time, if:

1. The source code for the function is available at translation time, and
2. The function is declared to have translation-time evaluation.

This is the kind of partial evaluation that is normally used to replace C-preprocessor macro invocations. Care must be taken to observe the normal scope rules of function variables expanded in-line. In source-to-source translators, scope rules can be enforced by variable renaming.

5.3 Function Specialization

A specialized version of a function that assumes specific values of its formal parameters is generated if:

1. The source code for the function is available at translation time,
2. The function is declared to have run-time evaluation, and
3. Some of the formal parameters of the function are declared to have translation-time evaluation.

An aggressively optimizing partial evaluator could apply this transformation even for functions with only run-time formal parameters.

This kind of partial evaluation can be used in place of C++ templates, or Ada generics. The translation-time formal parameters play the part of translation-time template parameters. Since, in Safer_C, user-defined types are treated as translation-time variables, functions can be specialized for specific type arguments.

The Implications for Reusability. When a function is coded to be as general as possible, in order to maximize the number of applications for which it can be reused, a common result is that some part of the function will be useless for some applications. The superfluous code is often marked by superfluous formal parameters that will always take the same value for any particular application. This kind of partial evaluation permits a specialized version of the function to be instantiated with the superfluous code eliminated.

A common indicator of superfluous parameters is default values for formal parameters, which are allowed by some languages. The default values are the ones unlikely to be changed by the programmer, and hence usually add redundant generality.

Many programmers resist reusing general code written for other projects because they fear it will slow down their application. For the same reason they may resist writing reusable code themselves. Translation-time specialization can increase efficiency, thus increasing the likelihood of coding and reusing more-general code.

Implementation Strategy. When a function with translation-time formal parameters is invoked, a record is kept of the particular values used for the corresponding actual arguments, and a new name is assigned to that forthcoming instantiation of the function. Repeated use of the same values for the same translation-time parameters does not increase the size of the list of specialized functions, since functions are instantiated only once for each combination of the specializing arguments. This method is called "polyvariant" specialization [2].

Ideally the compiler should have a way of communicating the particular values of formal parameters used for each instantiation of a function to the linker, so that the same instantiation could serve even separately compiled invocations of the function. This can be done by *name mangling:* coding the fixed parameters values into the name of the specialized function.

6 Providing Translation-Time Input and Output

Safer_C strives to achieve evaluation-time independence [9], which means that the same level of computational power should be provided for calculations performed at translation-time as is provided at run time. This principle includes I/O facilities. To meet this goal, Safer_C provides I/O streams that have translation-time evaluation. In particular, the input stream "tranin" is an input stream corresponding to "stdin" that is open for input at translation time, and "tranout" is the translation-time standard output stream corresponding to "stdout" at run-time.

By allowing explicit translation-time input, Safer_C source code can be customized at translation time for a particular application, without having to edit source code. The system tailoring parameters can be input interactively at translation time, or prepared in an independent specification file. Similarly, translation-time output can be used to prompt an installer for input, and report on the progress of compilation. If installers were instead required to edit system constants directly into source code then they would need a good understanding of the implementation language, and the possibilities for installation errors would be increased. Translation-time I/O is not a new idea. It has been used for decades in operating-system generation on minicomputer systems, such as RSX-11M for the DEC PDP-11 processor.

7 Translation-Time "Executable" Declarations

In C, declarations can be included or excluded from the source program under preprocessor control. This facility provides significant programmer control over the size of task-images, and the ability to perform substantial customization of programs for specific target systems. In order to duplicate this level of program control, Safer_C treats declarations as translation-time executable statements. The "execution" of a declaration consists of inserting a new entry into the symbol table. This treatment of declarations might seem peculiar, but it is actually not much different from the way that declarations are actually implemented in existing C compilers.

Safer_C's parent, the C language, provided an unexpected advantage in this respect. In C, there are no delimiters between declaration sections and executable sections of code, and Safer_C keeps this characteristic. As a result, declarations can be enclosed and controlled by control structures as easily as other statements.

The form of a Safer_C declaration is:

$$\textit{declaration} \rightarrow \textit{symbol\_list} :: \ [\textit{eval\_time}] \ \textit{type\_expr} \ [:= \ \textit{initial\_value}]$$

This notation is like Ada's declaration syntax except that Ada's colon has been replaced by a double colon, and an optional evaluation time can be specified.

As an example, a programmer may wish to select between two declarations, based on the operating system used by the target machine, by using statements similar to the following:

```
if  ( Target == System_1 )
    Field :: int
    process(&Field, 5124)
else
    Field :: float
    process(&Field, 5.124)
endif
```

8 Replacing #include Statements

One of the important uses of the C preprocessor is to insert declaration header files into a source program as requested by an `#include` statement. For instance, a common preprocessor statement in C programs is:

```
#include <stdio.h>
```

which inserts standard I/O declarations, macros, and function headers into the program. The concept of the `#include` statement—escaping to separately specified code—is similar to that of procedure invocation, and in Safer_C translation-time procedure invocation is used to take its place. The above include statement would be coded as:

```
stdio_h()
```

Both declarations and executable statements can be placed in a translation-time procedure, and it can be invoked at translation time to have the effect of inserting those statements into a program at the invocation site.

This mechanism may seem to be an extreme way of handling standard declarations as compared to the methods used by other programming languages, such as Ada or Modula-3, but few other languages allow executable statements in their declaration files.

9 Additional Control over the Scope of Program Entities

One problem with using compile-time procedures to replace `#include` statements is that normally each procedure starts a new scope, and the usual implicit scope rules will result in all the declarations within the procedure becoming inaccessible outside of the procedure. Normally the body of a function in Safer_C is enclosed between the keywords `block` and **end**, which correspond to braces {} in the C language. If, instead, the body of a function is enclosed by the keywords **body** and **end** then no new scope will be started for the body of the function. Thus any declarations made in the function will effectively be made at the point where the function is invoked. A new distinct lexical scope still exists for the formal parameters of the function.

10 Translation-Time Name Binding

An important use of the C preprocessor is to create new distinct identifiers for external symbols. In code being ported to different operating systems, different architectures, or different peripheral-device environments, the names of external symbols may need to change to reflect the particular environment.

Even mathematical functions can benefit from translation-time name binding. For instance, the computation of the trigonometric functions *sin* and *cos* could be generated from the same source with parameter rotation inserted at translation time, and renaming of the residual function.

Safer_C supports translation-time name binding by allowing the name of a variable or function to be taken indirectly from a translation-time string variable. Using the notation `#(tt_string)#` will cause the value of the variable tt_string to be used as an identifier. Without this feature, only coding-time name binding and code duplication would be available.

11 Effects on Compiler Size and Complexity

Providing Safer_C with evaluation-time annotations takes up 8% of the rules of the grammar. (The full grammar for Safer_C has only 91% of the number of rules in the grammar for C; the decrease being mostly due to a simplification of C's arcane declaration syntax and precedence rules.) Remember also that a grammar for the preprocessor (not included in this comparison) has been dispensed with.

Most of the code for evaluating expressions at compile time is already present in most C compilers, since the optimization phase usually makes provisions for some compile-time computation and constant folding. Code added to the compiler for boolean selection is similar in size and nature to the code that exists in the C preprocessor for the same purposes.

The translation-time evaluation of functions is not usually part of constant folding. It is similar in nature to macro expansion done by the preprocessor, but since the formal parameters can have types other than text strings, it is somewhat more complex.

New code must be added for compile-time case selection and loops, but it is slightly smaller than the code for expression evaluation needed by a preprocessor. All in all, the size of the translator grows little considering the added functionality of the method.

12 The Status of the Safer_C Project

The Safer_C language has been implemented as a translator to the C language. The translation-time computations are performed and the residual code is output in C. Since the translator is written in ANSI C and generates ANSI C output, its portability between systems should be good. All the run-time functionality of C is has been implemented. Translation-time loops and arrays are not yet fully supported, but, since these are not available in the C preprocessor either, enough of the Safer_C language has been implemented to allow the translation of all the ANSI C standard header files, such as `<stdio.h>`, `<math.h>`, and `<stdlib.h>`, into Safer_C translation-time functions. Since the dynamic loading of object modules at translation time is not yet supported, the source code for all translation-time function evaluations must be available at translation time.

13 Applicability of these Techniques

An attempt was made to judge the applicability of the techniques of this paper to real programs written by other programmers. For our analysis we chose code from three different projects:

1. Source code for the X11 library of the X Window System, release 5.
2. Source code for the kernel and SCSI device drivers of the LINUX operating system, version 1.1.59.
3. Source code for the Gnu C and C++ compilers (gcc and g++) version 2.5.8.

Because these freeware programs are designed to be portable across many operating systems and architectures, they make heavy use of preprocessor customization of source code. Since the sources were merely inspected for translatability to Safer_C, and not actually translated, some surprises may still arise. Nevertheless every attempt was made to do an accurate analysis.

By far the largest part of preprocessor usage in the code examined is plain and well behaved statements that can be easily and automatically translated into Safer_C: the simple definition of preprocessor constants with `#define`, the ordinary inclusion of header files with `#include`, and straightforward conditionally compiled source code using `#if` constructs.

Several instances of unusual code were found for which no direct translation would be advisable, since that would be duplicating the undesirable characteristic of preprocessors too. Recoding in a totally different style would be the best solution. In some cases, considerable thought had to be given as to how to use Safer_C constructs to provide the same effect as the preprocessor statements.

Usually the same problem recurs in the same form several times in the same source files, so once a translation scheme is found for one problem, it can be reapplied many times.

Ultimately, some of the translation problems encountered indicate that continued evolution of Safer_C is needed to provide for some of the uses of preprocessors. The two most important of these uses are:

1. Providing detailed control over declarations where control statements are currently not allowed in Safer_C, such as in formal parameter lists, and record structure declarations.
2. The stringizing of macro parameters, where the name of a formal parameter can be used as a string, has no equivalent in Safer_C, but is an especially useful feature for debugging and error reporting.

14 Sample Code

The following short Safer_C program is included to give the flavour of the language. Comments are delimited by an exclamation mark and the end of the line. In a style similar to CLU, semicolons are not required to end statements. Declarations and definitions are identified by a double colon. The name of a function is enclosed in French quotation marks (i.e. between << and >>) when it is being defined, but not when it is merely being declared. Most other constructs and operators that are not explained by comments follow C conventions.

```
!! Sample Safer_C/1 program.
!! ------------------------

!! Explicitly specify source language version.
Safer_C version 1.5

stdio_h()          !! Instead of "#include <stdio.h>".
math_h()           !! Instead of "#include <math.h>".

!! Pi is declared as a translation-time constant and
!! SIZE as a translation-time variable.
Pi :: tran const float := 3.141592654
SIZE :: tran int   !! Size of problem.

work :: extern func (x :: float) float


!! This translation-time function will be unfolded (in-lined).
<<map>> :: tran func (x :: float) int
body
    return (::int) (x/SIZE)   !! Cast Result to type int.
end
```

```
!! This run-time function will be specialized.
!! The type expression "-> T" means "pointer to type T".
<<swap>> :: run func (T :: type; a, b :: -> T) void
block
    temp :: T

    temp := a@   !! Dereference a and copy to temp.
    a@ := b@
    b@ := temp
end


!! Define the function "main".
<<main>> :: func () int
block
    !! Declare some translation-time symbols.
    IDX  :: tran int   !! Loop index.
    PHI  :: tran int

    !! Declare some run-time symbols.
    point, result :: float
    i :: int

    !! Translation-time I/O.
    fprintf (tranout, "Maximum size of problem? ")
    fscanf (tranin, "%d", &SIZE)

    !! Allocate enough zones for specified size.
    zones :: static [0..(2*SIZE)] float
    PHI := Pi/SIZE

    !! Translation-time initialization loop.
    #tran# for (IDX:=1; IDX < 2*SIZE; IDX++)
        zones[IDX] := sin(IDX*PHI)
    endfor

    !! A run-time, bottom-test loop.
    do
        printf ("Enter problem point: ")
        scanf ("%f", &point)
        result := work (point)
        printf ("    Result is: %g\en", result)
        i := map(point)
        swap (float, &result, zones+i)
        swap (float, &point, zones+i+1)
    while (result > 0.0)
    enddo
    return 0
end
```

References

1. Baier, R., Glück, R., Zöchling, R.: Partial evaluation of numerical programs in FORTRAN. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'94, Orlando, Florida, June 25, 1994. Technical Report 94/9,* Dept. of Comp. Sci., Univ. of Melbourne. (1994) 119–132
2. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. *20-th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93, Charleston, SC.* (1993) 493–501
3. Cormack, G. V.: Extensions to static scoping. *ACM SIGPLAN Notices* **18**, 6 (1983) 187–191
4. IBM: *OS and DOS PL/I Language Reference Manual.* Reference No. GC26-3977-1, File No. S370-29, IBM Corp., San Jose, CA (1984)
5. Jones, N. D., Gomard, C. K., Sestoft, Peter: *Partial evaluation and automatic program generation.* Prentice Hall Int'l., Hemel Hempstead, UK (1993)
6. Koenig, A.: *C Traps and Pitfalls.* Addison-Wesley, Reading, MA (1989)
7. Meyer, U.: Techniques for partial evaluation of imperative languages. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'91. Yale Univ., New Haven, CT, June 17-19, 1991. ACM SIGPLAN Notices* **26**, 9 (Sept. 1991) 94-105
8. Nirkhe, V., Pugh, W.: A partial evaluator for the Maruti hard real-time system. *Twelfth Real-Time Systems Symposium. IEEE Computer Society Press, Los Alamitos, CA, USA* (1991) 64-73
9. Salomon, D. J.: Four dimensions of programming-language independence. *SIGPLAN Notices* **27**, 3 (March 1992) 35-53
10. Salomon, D. J.: Safer_C: syntactically improving the C language for error resistance. *Tech. Rep. 95/07,* Dept. of Comp. Sci., Univ. of Manitoba (1995)
11. Stroustrup, B.: *The C++ Programming Language, 2-nd Ed.* Addison Wesley, Reading, MA. (1991)
12. Weise, D., Crew, R.: Programmable syntax macros. *ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI'93. Albuquerque, NM, June 23-25, 1993. ACM SIGPLAN Notices* **28**, 6 (June 1993) 156-165