Partial Evaluation and Mixed Computation
D.Bjørner, A.P.Ershov, and N.D.Jones (Editors)
Elsevier Science Publishers B.V. (North-Holland)
©.IFIP, 1988

A COMPILER GENERATOR PRODUCED BY A SELF-APPLICABLE SPECIALIZER CAN HAVE A SURPRISINGLY NATURAL AND UNDERSTANDABLE STRUCTURE

Sergei A. ROMANENKO

Keldysh Institute of Applied Mathematics
Academy of Sciences of the USSR
Miusskaya Sq.4, SU-125047, Moscow, USSR

SUMMARY

This paper describes the structure of, and the ideas behind, a self-applicable specializer of programs, as well as the principles of operation of a compiler generator that has been produced automatically by specializing the specializer with respect to itself. It has been found that the structure of the compilers produced can be improved by making use of such devices as introducing different representations for the values of K- and U-parameters, splitting the subject program into K- and U-program, and automatically raising the arity of functions in the residual program.

KEY WORDS AND PHRASES: Compiler generator, partial evaluation, mixed computation, non-partial evaluation, self-applicability, specializer, unmixed computation.

0. INTRODUCTION

Let Spec be a two-argument function specified by a program and satisfying the equation F(X,Y) = Spec(F,X)(Y), where F is an arbitrary two-argument function specified by a program, X and Y being inputs for F. Such a function Spec will be referred to as a "specializer".

The idea that specializers can become a programming tool of practical value dates back to the late 1960s [Lom 67], while it seems that the term "specializer" appeared in the early 1970s [Dix 71], [ChL 73].

In 1971 it was found by Y.Futtamura that compiling may be carried out by specializing interpreters [Fut 71] in the following way. Let Int be an interpreter of a programming language, i.e. a two-argument function satisfying the equation Int(S,D) = S(D), where S(D) denotes the result of applying a program S to an input D. According to [Fut 71], Spec(Int,S) can be considered to be the result of compiling the program S into the target language of the specializer, for Spec(Int,S)(D) = Int(S,D) = S(D).

In the same paper [Fut 71] Y.Futtamura pointed out that interpreters can be automatically converted to compilers by specializing a specializer with respect to the interpreters, the reason being that Spec(Spec,Int)(S) = Spec(Int,S). Of course, this approach implies that the source language of the specializer and the language it is written in are identical, which makes the self-application of the specializer feasible.

Several years later it was realized [Bec 76], [Tur 77, 79, 80] that a compiler generator can be automatically produced by evaluating Spec(Spec,Spec). This compiler generator transforms interpreters into compilers, for Spec(Spec,Spec)(Int) = Spec(Spec,Int).

No matter how attractive this method of producing compiler generators may seem, for several years it remained a purely speculative possibility.

Although there are a few publications reporting success in compiling by specializing interpreters [Fut 71], [Tur 79], [TNT 82], [Tur 86], it is only recently that the group under the leadership by N.Jones succeeded in designing a non-trivial specializer that proved to be self-applicable not only in theory, but also in practice [JSS 85a], [JSS 85b], [Ses 86]. To the author's best knowledge, this specializer was the first to be used in practice to convert interpreters to compilers by Futtamura's method and to generate a non-trivial compiler generator Spec(Spec,Spec).

For brevity's sake, the specializer developed by N.Jones and coworkers will be, henceforth, referred to as the "Copenhagen" one.

It should be noted that the Copenhagen specializer is not a completely automatic one. It requires a hand-made annotation of the subject program. The user has to classify all function calls appearing in the subject program as "eliminable" or "residual". In the course of specialization, the eliminable calls are unfolded (i.e. replaced with the reduced equivalent of the called function's body), while the residual calls are suspended (i.e. replaced with a call to a residual variant of the called function). Thus, the actual execution of the residual calls is postponed up to the time when the residual program is run. Annotating the subject program is done by replacing the key word "call" with the key word "callr" for all residual calls.

Classifying all function calls as residual or eliminable proved to be a hard problem for the computer. This problem seems to be the main difficulty that impedes the self-application of completely automatic non-trivial specializers [Tur 86].

The paper [JSS 85a] describes the results obtained by means of the Copenhagen specializer as follows.

The compilers produced by evaluating Spec(Spec,Int) "have a surprisingly natural structure"; nevertheless, as far as the compiler generator Spec(Spec,Spec) is concerned, the situation is less satisfactory. Although, the compiler generator produced is "of reasonable size", "its logic is harder to follow" than the logic of the compilers generated. Furthermore, "it contains some unexpected constructions (like ''''nil!)".

In early 1986, by the courtesy of Neil Jones, the author received a detailed description of the structure and principles of the Copenhagen specializer. It inspired the author to make an attempt to reproduce the results obtained by the Copenhagen group. The original objectives of the project were:

* To modify the Copenhagen technique in order to make it applicable to programs written in Refal, rather than Lisp.

* To behold compilers generated this way, especially the compiler generator, and to examine their structure.

The availability of detailed information on the Copenhagen specializer enabled these objectives to be achieved without much difficulty. However, in the author's opinion, the compilers produced turned out to be unsatisfactory from the aesthitic point of view. Whereas the compilers were still within the human comprehension, the ugliness of the compiler generator prevented any attempt at reading it.

Nevertheless, a strong desire to fathom the mysterious principles of operation of the compiler generator made the author seek a way to improve the structure of residual programs produced by the specializer. As a result, the original specializer had been revised. For brevity's sake, the modified version of the specializer will be, henceforth, referred to as "the Moscow specializer".

The following sections discuss the differences and similarities of the two specializers. Finally, a description is given of the structure and principles of operation of the automatically generated compiler generator.

1. THE PRINCIPLES THE TWO SPECIALIZERS ARE BASED UPON

The main feature of the Copenhagen specializer that distinguishes it from those previously presented in the literature, and based on partial evaluation is that the specialization is done in several steps.

At the first step, the program is flow analyzed by a simple abstract interpretation over a domain consisting of the two symbols "K" and "U". "K" represents a value known at partial evaluation time, and "U" represents a value that may be unknown at partial evaluation time.

Partial evaluation is meta-evaluation with respect to ordinary evaluation, and KU-evaluation is meta-evaluation with respect to partial evaluation. Thus, KU-evaluation is meta-meta-evaluation.

The existence of a meta-meta-level is a peculiarity of the Copenhagen specializer, for the earlier specializers have only dealt with the basic level and the meta-level.

The correspondence between the three levels may be characterized as follows.

The basic level involves values, evaluation, interpreter.

The meta-level involves meta-values (terms of the semantic metalanguage), meta-evaluation (partial evaluation), meta-interpreter.

The meta-meta-level involves meta-meta-values ("K" and "U"), meta-meta-evaluation (meta-specialization), meta-meta-interpreter.

The first phase computes for each function in the subject program a description classifying the function's parameters as "known" K-parameters (eliminable) or "unknown" U-parameters (residual).

The information obtained by KU-interpretation is then used at the second step, when the subject program is transformed, i.e. annotated . The essence of annotation is in representing the global information obtained by meta-meta-interpretation locally. In the course of annotation the program is supplemented with additional directions meant for the meta-interpreter.

Thus, instead of being placed under the command of a supervisory device (as, for example, has been done in [Tur 80, 86]), the meta-interpreter is made to follow the directions inserted into the annotated program. This is the second principle the Copenhagen specializer is based upon. This principle might be referred to as "the principle of self-surveillence".

The Moscow specializer is fully faithful to the two above principles. It adheres to these principles even more thoroughly and consistently than the Copenhagen specializer does. For this reason, the Moscow specializer may be considered, in a sense, to be a more "Copenhagen" one than the Copenhagen specializer is.

2. THE SEMANTIC METALANGUAGE RL

As has been pointed out by Y.Futtamura [Fut 71], a system capable of producing compilers by self-application of a specializer has to be based on a "semantic metalanguage" satisfying a number of requirements. The specializer as well as all interpreters to be transformed into compilers have to be written in this language. As far as the Copenhagen specializer is concerned, use is made of a dialect of pure Lisp, the language L.

The author's goal was to design a specializer capable of accepting programs in the language Refal. Since Refal was developed to serve as an "algorithmic metalanguage" [Tur 66], [Tur 68], it may well be used as a semantic metalanguage. As a matter of fact, it has been used this way [Tur 79], [Tur 80], [Tur 86].

Refal provides high-level facilities (such as pattern matching), which enables symbolic manipulation algorithms to be written in the form that is clear and simple from the human point of view. However, as far as a specializer like the Copenhagen one is concerned, these facilities tends to complicate the operation of the specializer. This induced the author to choose an alternative approach.

The semantic metalanguage used by the Moscow specializer is the language RL, which has been tailored by the author for this purpose. For this reason, a Refal-program subject to specialization has to be compiled into RL.

RL is an intermediate language, which provides data structures identical to those of Refal, but, as compared with Refal, is a low-level language.

The general structure of programs and the control constructs provided by RL are similar to those of Lisp. For this reason, the name "RL" may be interpreted as "Refal-Lisp".

RL is less convenient for the human programmer than Refal, though it is not low-level enough to prevent RL programs from being written by hand. However, RL may well be, in some cases, more suitable than Refal for automatic program generation and transformation.

RL is easier to implement than Refal, so an RL-program automatically generated are hardly worth recompiling into Refal.

An RL-program is a non-empty list of function definitions. The first function of the program is the goal function. Input to the program is through the parameters of this function, and output is the value returned by it.

All functions used in RL-program may be classified as "primitive" or "defined". A primitive function (or "operator") can be called directly, whereas a defined function can only be called by means of the operator "call".

A defined function has a fixed arity, i.e. a certain number of arguments (which is allowed to be equal to zero). This is a peculiarity of RL in which it differs from Refal, as all functions defined in a Refal-program are formally one-argument. Any object expression (i.e. an arbitrary sequence of symbols and parentheses in which the parentheses are properly paired) can be taken as value by any parameter.

Any primitive function (i.e. operator) has a fixed arity. Nevertheless, in order to reduce the length of RL-program, a call of any unary operator is allowed to be given an arbitrary number of arguments, in which case the values of the arguments are concatenated to form an expression. This expression is then taken as value by the parameter of the operator.

An operator is either a function call, a conditional "if", a constant "quote", a constructor, a selector, or a predicate.

A constructor is a function that builds object expressions from object expressions. There are the following constructors: "br", which encloses its input in parentheses, and "expr", whose result is equal to its input. Since "expr" is a unary operator, it can be used to concatenate expressions.

A selector is a function that extracts a component from its input. There are the following selectors: "first", which takes the first term of an input expression, "last", which takes the last term of an expression, "bf", which takes all terms of an expression but the first one, "bl", which takes all terms of an expression but the last one, "cont", which takes the contents of a pair of parentheses.

A predicate is a function that tests the truth of a condition and produces either the symbol "true" or the symbol "false". There are the following predicates: "symbol", which tests whether its input expression is a symbol, and "equal", which tests whether its two inputs are equal.

The syntax or RL-programs may be described as follows (where <XXX>* is an abbreviation for the construction <XXX> repeated zero or more times).

```
     <program> ::=
        <function-definition> <function-definition>*
     <function-definition> ::=
        ( <function-name> ( <parameter>* ) <RL-term> )
     <function-name> ::= <object-term>
     <parameter> ::= <RL-variable>
     <RL-variable> ::= <symbol>
     <RL-term> ::=
        <RL-variable> |
        (abort) |
        (quote  <object-expression> ) |
        (br  <RL-expression> ) |
        (expr  <RL-expression> ) |
        (first  <RL-expression> ) |
        (bf  <RL-expression> ) |
        (last  <RL-expression> ) |
        (bl  <RL-expression> ) |
        (cont  <RL-expression> ) |
        (symbol  <RL-expression> ) |
        (equal  <RL-term> <RL-term> ) |
        (call  <function-name> <RL-term>* ) |
        (if  <RL-term> <RL-term> <RL-term> )
     <RL-expression> ::= <RL-term>*
     <object-expression> ::= <object-term>*
     <object-term> ::=
        <symbol> | ( <object-expression> )
```

Here is given an RL-interpreter written in RL.

```
(RL-Int (Program Args)
  (call Call (cont (first Program)) Args Program)
)

(Call (FnDef Vals Program)
  (callr Term
    (last FnDef)
    (cont (first (bf FnDef)))
  Vals Program
  )
)

(Term (Term Pars Vals Program)
  (if (symbol Term)
    (call LookUpV Term Pars Vals)
    (call Term-
      (first (cont Term))
      (bf (cont Term))
      Pars Vals Program
    )
  )
)

(Term- (Key Info Pars Vals Program)
  (if (equal Key (quote quote))
    Info
  (if (equal Key (quote abort))
    (abort)
  (if (equal Key (quote call))
    (call Call
      (call LookUpF (first Info) Program)
      (call Pars (bf Info) Pars Vals Program)
      Program
    )
  (if (equal Key (quote callr))
    (call Call
      (call LookUpF (first Info) Program)
      (call Pars (bf Info) Pars Vals Program)
      Program
    )
  (if (equal Key (quote br))
    (br    (call Expr Info Pars Vals Program))
  (if (equal Key (quote expr))
           (call Expr Info Pars Vals Program)
  (if (equal Key (quote first))
    (first (call Expr Info Pars Vals Program))
  (if (equal Key (quote bf))
    (bf    (call Expr Info Pars Vals Program))
  (if (equal Key (quote last))
    (last  (call Expr Info Pars Vals Program))
  (if (equal Key (quote bl))
    (bl    (call Expr Info Pars Vals Program))
  (if (equal Key (quote cont))
    (cont  (call Expr Info Pars Vals Program))
  (if (equal Key (quote symbol))
    (symbol (call Expr Info Pars Vals Program))
  (if (equal Key (quote equal))
    (equal
      (call Term  (first Info)      Pars Vals Program)
      (call Term  (first (bf Info)) Pars Vals Program)
    )
  (if (equal Key (quote if))
    (if
      (call Term  (first Info)           Pars Vals Program)
      (call Term  (first (bf Info))      Pars Vals Program)
      (call Term  (first (bf (bf Info))) Pars Vals Program)
    )
  (abort)
  )))))))))))))
)

(Pars (Terms Pars Vals Program)
  (if (equal Terms (quote))
    (quote)
    (expr
      (br (call Term (first Terms) Pars Vals Program) )
          (call Pars (bf Terms)    Pars Vals Program)
    )
  )
)

(Expr (Terms Pars Vals Program)
  (if (equal Terms (quote))
    (quote)
    (expr
      (call Term (first Terms) Pars Vals Program)
      (call Expr (bf Terms)    Pars Vals Program)
    )
  )
)

 (LookUpV (Var Pars Vals)
  (if (equal Pars (quote))
    (abort)
  (if (equal Var (first Pars))
    (cont (first Vals))
    (call LookUpV Var (bf Pars) (bf Vals))
  ))
)

(LookUpF (FnName Program)
  (if (equal FnName (first (cont (first Program))))
    (cont (first Program))
    (call LookUpF FnName (bf Program))
  )
)
```

3. THE GENERAL STRUCTURE OF THE MOSCOW SPECIALIZER

In attempting to solve a problem, one should discern the difference between the end in view and the means to be made use of. In the context of the present work, specialization of programs is the end in view, whereas partial evaluation is merely one of the techniques applicable. There is thus no good reason to believe that partial evaluation is the only conceivable method of specializing programs. Other means may prove to be of value (e.g. the traditional optimization techniques).

In fact, the two major parts of the Moscow specializer are <u>the arity reducer</u> and <u>the arity raiser</u>, only the arity reducer being based on partial evaluation (or, as we prefer to say for reasons to be given below, on meta-evaluation).

The arity reducer is the part that roughly corresponds to the Copenhagen specializer. The peculiarity of the specialization technique based on partial evaluation is that any function appearing in a residual program has at most the same number of parameters as the function from which it has been produced. This accounts for the term "arity reducer".

A program produced by the arity reducer is then passed on to the arity raiser, which is based on optimization techniques that is fairly traditional.

The specializer is called as follows:

```
      (call Spec Prog Pars-Cl K-Vals U-Types)
```

where the the arguments have the following meanings:

Prog is a source RL-program to be specialized;
Pars-Cl is a sequence of symbols "K" and "U", which describes whether the corresponding parameter of Prog will be known (K) or unknown (U) in the course of meta-evaluation;
K-Vals is a list of object expressions to be used as values of the K-parameters;
U-Types is a description of types of the U-parameters.

The function Spec calls other two functions: the arity reducer Reduce-Ar and the arity raiser Raise-Ar.

```
(Spec (Prog Pars-Cl K-Vals U-Types)
  (call Raise-Ar
    (call Reduce-Ar Prog Pars-Cl K-Vals)
    U-Types
  )
)
```

4. THE ARITY RAISER

The operation of the arity raiser proceeds in three steps.

* First, the types of the parameters and results of all functions are determined. The information obtained is then used at the following steps.

* Secondly, the parameters of functions are splitted.

* Finally, local optimization is done.

For example, suppose that, according to the information obtained by the type analyzer, any value of the parameter X is bound to have the structure

```
      (e1) e2 t3
```

where e1 and e2 are object expressions, and t3 is an object term. In this case X is splitted into three parameters X1, X2, and X2, which are to contain the expressions e1, e2, and t3. Then all occurrences of X in the function definition are replaced with the RL-term

```
      (expr (br X1) X2 X3)
```

Thus the principles upon which the arity raiser is based are widely known. For this reason, they will not be discussed in more detail in the present paper. However, there remains a few remarks to be made.

* Being complementary in their purposes, the arity raiser and the arity reducer cooperate in a natural manner.

* Although the techniqes used by the arity raiser are straightforward, they considerably improve the readability and efficiency of programs generated by the specializer.

* The arity raiser is completely automatic, which, as far as the variable splitting is concerned [JSS 85b], [Ses 86], releaves the user of a hand-made annotation of the subject program.

From now on, when discussing the differences between the Moscow specializer and the Copenhagen one, by the Moscow specializer we, for the most part, shall understand the arity reducer, because the Copenhagen specializer does not include any automatic arity raiser.

5. THE ARITY REDUCER

The arity reducer removes the K-parameters of each function appearing in the subject program to produce specialized versions of the function. Its operation proceeds in several steps.

First, the meta-meta-interpreter is called, which takes the call annotated subject program and a description telling which of the program's input parameters are known. Then the meta-meta-interpreter computes for each function a safe description of its parameters classifying them as K- or U-parameters.

Furthermore, the subject program's functions are classified as K- or U-functions. In the course of meta-interpretation, K-functions produce K-values, while U-functions produce U-values.

> This is a feature of the Moscow specializer that distinguishes it from the Copenhagen one, since the latter classifies parameters, but does not classifies the functions themselves.

Then the information is gathered about which of the subject program's functions are <u>residual</u>, i.e. called by the operator "callr" at least at one place in the program.

Secondly, on the basis of the information obtained, the U-functions are annotated, i.e. transformed in order to facilitate the operation of the meta-interpreter. A detailed description of the way in which the program is annotated will be given later.

The meta-meta-interpreter produces a three-term expression, which is then decomposed and passed on to the meta-interpreter in the form of three separate arguments. The meaning of these arguments will be described later.

Thirdly, the meta-interpreter builds a specialized version of the subject program, the actual values of the input K-parameters being given by K-Vals.

Finally, the program generated is passed on to the function Rename-Funcs, which invents new, shorter, names for the functions appearing in the residual program.

Thus the arity reducer is defined as follows:

```
(Reduce-Ar (Prog Pars-Cl K-Vals)
  (call Reduce-Ar-
    (call MM-Int Prog Pars-Cl)
    K-Vals
  )
)

 (Reduce-Ar- (Ann-Prog K-Vals)
  (call Rename-Funcs
    (call M-Int
      (cont (first Ann-Prog))
      (cont (first (bf Ann-Prog)))
      (cont (first (bf (bf Ann-Prog))))
      K-Vals
    )
  )
)
```

6. THE STRUCTURE OF AN ANNOTATED PROGRAM

An annotated program takes the form:

```
( R-Funcs ) ( K-Prog ) ( U-Prog )
```

where the expressions R-Funcs, K-Prog, and U-Prog have the following meanings.

R-Funcs is a list of the residual functions, i.e. the functions whose specialized versions can appear in a residual program.
K-Prog is a K-program, i.e. definitions of functions which have only K-parameters and produce K-values.
U-Prog is a U-program, i.e. definitions of functions which have at least one U-parameters and produce U-values.

The definitions appearing in the K-program are, without any change, inherited from the source program.

The definitions appearing in the U-program are taken from the source program and modified in the following way.

Let the source function definition be

```
(F (X1 X2 ... XL) T).
```

It is transformed into

```
(F (K1 K2 ... KM) (U1 U2 ... UN) AT),
```

where Ki are those Xi that belongs to the class K, and Ui are those Xi that belongs to the class U, M+N=L, and AT is an RL-term obtained by annotating T. Annotating an RL term is done as follows.

An RL-term involving neither U-parameters nor U-functions' calls will be referred to as a K-term. An RL-term that is not a K-term will be referred to as an U-term.

Let T be an RL-term. The result of annotating T will be denoted by AT.

AT is obtained from T by applying the following rules, with the precedence of the rules determined by the order in which they are listed.

If T=(quote C), then AT=(quote C).

If T=(abort), then AT=(abort).

If T is a K-term, then AT=(meta T).

If T is a variable, then AT=T.

If T=(call F T1 T2 ... TL), then AT=(call F (K1 K2 ... KM) (AU1 AU2 ... AUN)), where M+N=L, K1 K2 ... KM is the list of all Ti that correspond to the K-parameters of F, and AU1 AU2 ... AUN is the list of the terms that are obtained by annotating the terms Ti that correspond to the U-parameters of F.

If T=(callr F T1 T2 ... TL), then AT=(callr F (K1 K2 ... KM) (AU1 AU2 ... AUN)), where M+N=L, K1 K2 ... KM is the list of all Ti that correspond to the K-parameters of F, and AU1 AU2 ... AUN is the list of the terms that are obtained by annotating the terms Ti that correspond to the U-parameters of F.

If T=(if T0 T1 T2), where T0 is a K-term, then AT=(if-e T0 AT1 AT2).

If T=(if T0 T1 T2), where T0 is a U-term, then AT=(if-r AT0 AT1 AT2).

If T=(P T1 T2 ... TN), where P is one of the operators "br", "expr", "first", "bf", "last", "bl", "cont", "symbol", or "equal", then AT=(P AT1 AT2 ... ATN).

Consider the function

```
(Zipper (X Y)
  (if (equal X (quote))
    Y
  (if (equal Y (quote))
    X
    (expr
      (first X) (first Y)
      (call Zipper (bf X) (bf Y))
    )
  ))
)
```

On condition that X is a K-parameter, and Y is a U-parameter, the result of annotating Zipper is

```
 (Zipper (X) (Y)
  (if-e (equal X (quote))
    Y
  (if-r (equal Y (quote))
    (meta X)
    (expr
      (meta (first X)) (first Y)
      (call Zipper ((bf X)) ((bf Y)) )
    )
  ))
)
```

7. THE META-INTERPRETER

The meta-interpreter is called as follows:

```
      (call M-Int R-Funcs K-Prog U-Prog K-Vals)
```

The first three arguments contain an annotated program subject to specialization, and K-Vals contains values of the program's K-parameters. The meta-interpreter produces a residual program composed of functions, each function being a specialized version of a function appearing in U-Prog. More specifically, if the definition of a U-function is of the form

```
      (F (K1 ... KM) (U1 ... UN) T),
```

it can give rise to function definitions of the form

```
      ((F (C1) ... (CM)) (U1 ... UN) T')
```

where C1, ..., CM are object expressions which are values of the K-parameters K1, ..., KM, and T' is the result of meta-evaluating T, with the K-parameters' values being C1, ..., CM, and the U-parameters' values being these parameters U1, ..., UN themselves. ((F (C1) ... (CM)) is the name of the function generated. This name is to be replaced with a shorter one at the next stage of specialization.

Thus, the principal task of the meta-interpreter consists in evaluating RL-terms. The result of meta-evaluating an RL-term is a U-value, i.e. an RL-term.

Consider, for example, the above function Zipper. It has the K-parameter X and the U-parameter Y. Suppose that the meta-interpreter has to evaluate Zipper, with the values of X and Y being

```
      X = "ONE TWO";  Y = "VAR".
```

(From here on, variable values will be put in double quotation marks for fear that they might be confused with variable names.) The above conditions being met, the meta-evaluation of Zipper yields the RL-term

```
 (if (equal VAR (quote))
  (quote ONE TWO)
  (expr
    (quote ONE) (first VAR)
    (if (equal (bf VAR) (quote))
      (quote TWO)
      (expr
        (quote TWO) (first (bf VAR))
        (bf (bf VAR))
      )
    )
  )
)
```

In the course of meta-evaluating the body of Zipper, the meta-interpreter has to carry out the recursive call of Zipper twice. When Zipper is called for the first time, the parameters take the values

X = "TWO"; Y = "(bf VAR)",

and when Zipper is called for the second time, they take the values

X = ""; Y = "(bf (bf VAR))".

Thus, the most complicated operation performed by the meta-interpreter consists in meta-evaluating RL-terms.

The basic principle followed in designing the Moscow specializer is that, in contrast to an ordinary interpreter, the meta-interpreter should deal with values of two kinds: K-values and U-values. These values are entirely different in nature, for they belong to different levels: K-values correspond to the basic level, whereas U-values correspond to the meta-level. U-values may, with respect to K-values, be regarded as meta-values, since they are RL-terms, which are to produce K-values only at the time the specialized program will be run.

Thus K-values and U-values have to be operated on in entirely different ways. For example, the application of the operator "bf" to the K-value "ONE TWO THREE" yields the K-value "TWO THREE", whereas the application of "bf" to the U-value "(expr X1 X2)" yields the U-value "(bf (expr X1 X2))", the "bf" being treated as a "meta-level" operator.

Hence, the meta-interpreter may take advantage of using entirely different representations for K-values and U-values. K-values, naturally, may be stored and operated on as "true" object expressions, the way they would be dealt with by an ordinary RL-interpreter, while U-values, naturally, may be stored as RL-terms.

Accordingly, K-values can be operated on "really", i.e. the way an RL-interpreter does, whereas U-values can be operated on "nominally", in a "meta" manner (which may be reduced to placing the operator applied before the RL-term).

Thus, the first principle the Moscow specializer is based upon is that, instead of being mixed, K-computation and U-computation ought to be soroughly separated. This principle may be referred to as <u>the principle of unmixed computation</u>.

The second basic principle is that any computation involving K- or U-values ought to be carried to completion. This is feasible since all K- or U-values dealt with by the meta-interpreter are completely known to the latter (although they are operated on differently). As there is no good reason to regard completed evaluation performed by the meta-interpreter as partial, the second principle may be referred to as the principle of non-partial evaluation.

The above considerations account for the author's preference for the term "meta-evaluation" over the term "partial evaluation" as far as the Moscow specializer is concerned.

The general structure of the meta-interpreter conforms to the above principles. The basis of the meta-interpreter is formed by two functions: Eval-Term and Spec-Term. Eval-Term computes the K-value of a K-term with respect to given K-parameters' values and a K-program. Spec-Term computes the U-value of a U-term with respect to given K-parameters' values, U-parameters' values, a K-program and a U-program.

Thus the meta-interpreter includes the K-interpreter, which is an ordinary RL-interpreter, and the U-interpreter, which is a meta-interpreter in the true sense of the word.

Having been called, the K-interpreter never calls the U-interpreter, whereas the U-interpreter calls the K-interpreter whenever the U-term under interpretation involves K-terms. This situation arises when the meta-interpreter runs into one of the following constructs:

```
(meta K)
(if-e K U1 U2)
(call  F (K1 ... KM) (U1 ... UN))
(callr F (K1 ... KM) (U1 ... UN))
```

As the class of any subterm is determinable from the context, there is no need for annotating operators appearing in the subject program by replacing each occurrence of an operator "p" either with "p-e" or "p-r" (as this is done by the Copenhagen specializer). The only exeption is the operator "if".

8. THE RESULTS OBTAINED BY APPLYING THE SPECIALIZER TO ITSELF

It is a peculiarity of both the Moscow specializer and the Copenhagen one that interpreters can be converted to compilers by evaluating Spec(M-Int,Ann-Int) instead of Spec(Spec,Int), where M-Int is the meta-interpreter, Int is an interpreter, and Ann-Int is the result of annotating Int. Similarly, a compiler generator can be produced by evaluating Spec(M-Int,Ann-M-Int), where Ann-M-Int is the result of annotating M-Int, the inputs of M-Int being classified as follows: R-Funcs, K-Prog, and U-Prog are K-parameters, and K-Vals is an U-parameter. This is feasible because the meta-interpreter is the only part of the specializer that has to know the value of the input parameter K-Vals (i.e. the values of the subject program's K-parameters) [JSS 85a], [JSS 85b], [Ses 86].

The autor succeeded in getting the specializer to produce several compilers from interpreters. Among the interpreters converted are a simple interpreter implementing an imperative two-register machine (which is essentially the same as the interpreter described in [JSS 85b]), an interpreter of finite automata, the RL-interpreter presented above, and an interpreter of the strict Refal [Tur 86].

The structure of all the compilers obtained turned out to be quite natural from the human point of view, the compilers being easy to read. The RL-compiler, as could be expected, proved to be an RL-optimizer, rather than a true compiler, its source and target languages being the language RL.

Then, by specializing the meta-interpreter with respect to the meta-interpreter, a compiler generator was generated. As could be expected, the names of functions appearing in the compiler generator proved to be, in many cases, rather insipid (for instance, the names Spec-Term-1, Spec-Term-2, ..., Spec-Term-45). Nevertheless, these names having been replaced by the author's hand with more suggestive ones, the compiler generator turned out to be quite readable.

A close examination of the compiler generator enabled its principles of operation to be fully understood. As a result, the way in which the interpreters mentioned above had been converted to compilers became apparent. Moreover, the compiler generator itself having been produced from the meta-interpreter in conformity with the same principles, the correspondence between the meta-interpreter and the compiler generator became clear.

9.THE WAY IN WHICH INTERPRETERS ARE CONVERTED TO COMPILERS

A compiler produced from an interpreter comprises two parts: the administrator and the generator. The administrator puts the compiler as a whole into operation, whereas the generator builds the residual program. The compilers produced by the Copenhagen specializer have a similar structure [JSS 85b], [Ses 86].

Being merely a slightly specialized version of the meta-interpreter's administrative part, the administrator has the structure that is almost independent of the source interpreter.

The structure of the generator is, on the contrary, completely dependent on the structure of the source interpreter, being entirely different for different interpreters.

As has been said above, an annotated interpreter takes the form

```
( R-Funcs ) ( K-Prog ) ( U-Prog )
```

The functions from the K-Prog are transferred to the compiler with insignificant alterations (such as local optimizations and renaming of functions and parameters).

The functions from U-Prog are transformed and then transferred to the compiler. The result of the transformation may be obtained by applying the following rules (modulo local optimizations and renaming of functions and parameters).

The functions from U-Prog will be referred to as "interpreting", and the corresponding functions from the compiler will be referred to as "compiling".

Let T be an RL-term appearing in the definition of an interpreting function. The result of transforming T, which corresponds to T in the compiling function, will be denoted by CT.

Let the definition of an interpreting function be

```
(F (K1 ... KM) (U1 ... UN) T).
```

It is transformed into the compiling function

```
(F (K1 ... KM U1 ... UN) CT),
```

where CT is obtained from T by applying the following rules, with the precedence of the rules determined by the order in which they are listed.

If T=(quote C), then CT=(quote (quote C)).
If T=(abort), then CT=(quote (abort)).
If T=(meta K), then CT=(br (quote quote) K).
If T is a variable, then CT=T.
If T=(call F (K1 ... KM) (U1 ... UN)), then CT=(call F K1 ... KM CU1 ... CUN)).
If T=(callr F (K1 ... KM) (U1 ... UN)), then CT=(br (quote call) (br (quote F) (br K1) ... (br KM)) CU1 ... CUN).
If T=(if-e K U1 U2), then CT=(if K CU1 CU2).
If T=(if-r U0 U1 U2), then CT=(br (quote if) CU0 CU1 CU2).
If T=(P U1 ... UN), where P is one of the operators "br", "expr", "first", "bf", "last", "bl", "cont", "symbol" or "equal", then CT=(br (quote P) CU1 ... CUN).

For example, the above function Zipper is transformed into the compiling function

```
(Zipper (X Y)
  (if (equal X (quote))
    Y
  (br (quote if)
    (br (quote equal) Y (quote (quote)))
    (br (quote quote) X)
    (br (quote expr)
      (br (quote quote) (first X))
      (br (quote first) Y)
      (call Zipper (bf X) (br (quote bf) Y)
    )
  ))
)
```

The above principles of transforming interpreting functions into compiling ones appear to be quite natural. They, in all probability, have been used in the hand-written compiler generator reported in [Bec 76]. However, in our case, of particular interest is the fact that these principles have been automatically "discovered" by the computer in the course of specializing a specializer.

It should be noted that the above principles of producing compiling functions are valid provided that the compiler generator is dealt with in its integrity, with the inclusion of the automatic arity raiser. Had the arity raiser been excluded from the compiler generator, the compiling functions produced would have had two parameters exactly. The first parameter, K-Vals, would have contained K-values, whereas the second parameter, U-Vals, would have contained U-values.

CONCLUSION

The main feature of the Moscow specializer that distinguishes it from the Copenhagen one is more strict and static differentiation between K-values and U-values. It has been achieved by the following means.

* In addition to the separation of K-parameters from U-parameters, the separation of K-functions from U-functions has been introduced.

* Subject programs are annotated in a different way, so that a program is divided into K-program and U-program. Thus, there is no need for replacing each occurrence of an operator "P" with either "P-e" or "P-r".

* The new method of annotation allowed the meta-interpreter to be divided into the K-interpreter and U-interpreter. Being an ordinary RL-interpreter, the K-interpreter deals only with K-values.

* The separation of the K-interpreter from the U-interpreter has made it possible to choose different representations for values of K- and U-parameters, such that a K-value is an object expression, whereas a U-value is an RL-term.

A considerable improvement in the structure of residual programs is, for the most part, due to the automatic arity raiser and the use of different representations for K- and U-values.

As far as the Copenhagen specializer is concerned, K- and U-values are treated in a different way: any value assigned to a K-parameter, instead of being an ordinary constant, is a representation of the constant in the form of a term of the semantic metalanguage. In other words, instead of a constant "C", use is made of the term "(quote C)". This has a disastrous effect on the size and readability of the compilers generated. Let, for instance, "(first X)" be a term appearing in a source interpreter, X being a K-parameter. Then, if K- and U-values had the same representation, this term would give rise to the term

```
(br (quote quote) (first (bf (cont X))))
```

in the compiler, whereas the compiler generator produced by the Moscow specializer transfers this term to the compiler without any change.

Additionally, the separation of the K-interpreter from the U-interpreter eliminated the necessity of performing immediate local optimizations of the U-values being produced in the course of meta-interpretation, since these optimizations, in any case, is to be performed by the arity raiser. Besides, these optimizations can be done better by the arity reducer, because it can make use of the global information on the types of functions and variables.

Thus, it can be easily seen from the above considerations that making use of different representations of K- and U-values results in the more clear structure of the compiler generator and compilers produced from interpreters.

ACKNOWLEDGMENTS

The author expresses his appreciation and gratitude to Sergei Abramov, Andrei Klimov, Arkady Klimov, Nikolay Kondratiev, Victor Kistlerov, Alexandr Romanenko and other members of the Refal working group. While in progress, the present work has been regularly discussed at the meetings of the group. The advice and critical help recieved have been a permanent encouragement to the author's efforts.

In addition, the author wants to use this occasion to express his gratitude to Vsevolod S.Shtarkman, whose friendly and permanent support much contributed to the success of the work.

REFERENCES

[Bec 76]
L.Beckman, A.Haraldson, O.Oskarsson, E.Sandewall. A partial evaluator, and its use as a programming tool. Artificial Intelligence, Vol.7, No.4, 1976, pp.319-357.

[ChL 73]
Ch.Chang, R.Lee. Symbolic logic and mechanical theorem proving. - Academic Press, 1973.

[Dix 71]
J.Dixon. The specializer, a method of automatically writing computer programs. - Division of Computer Research and Technology, National. Inst. of Health, Bethenda, Maryland, 1971.

[Fut 71]
Y.Futtamura. Partial evaluation of computation process - an approach to a compiler compiler. - Systems, Computers, Controls, Vol.2, No.5, 1971, pp.45-50.

[JSS 85a]
N.D.Jones, P.Sestoft, H.Sondergaard. An experiment in partial evaluation: The generation of a compiler generator. - SIGPLAN Notices, Vol.20, No.8, 1985, pp.82-87.

[JSS 85b]
N.D.Jones, P.Sestoft, H.Sondergaard. An experiment in partial evaluation: The generation of a compiler generator. - In Proc. 1st Intl. Conf. on Rewriting Techniques and Applications, Dijon, France, 1985. Springer LNCS 202 (1985), pp.124-140.

[Lom 67]
L.A.Lombardi. Incremental computation. - Advances in Computers, 8, Academic Press, New York, 1967.

[Rom 87]
S.A.Romanenko. A compiler generator produced by a self-applicable specializer can have a clear and natural structure. Preprint, the Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, 1987, No.26.

[Ses 86]
P.Sestoft. The structure of a self-applicable partial evaluator. - In H.Ganzinger and N.D.Jones (Eds.): Programs as Data Objects, Copenhagen, Denmark, 1985. Springer LNCS 217 (1986), pp.236-256.

[TNT 82]
V.F.Turchin, R.N.Nirenberg, D.V.Turchin. Experiments with the supercompiler. - Conference Record of the ACM Symposium on Lisp and Functional Programming, 1982, pp.47-55.

[Tur 66]
V.F.Turchin. A metalanguage for the formal description of algorithmic languages. - In Tsyfrovaya Vychislitelynaya Tekhnika i Programmirovaniye, Moscow, Sovetskoye Radio, 1966, pp.116-124 (in Russian).

[Tur 68]
V.F.Turchin. The metaalgorithmic language. - Kibernetika, No.4, 1968, pp.45-54 (in Russian).

[Tur 77]
Basic REFAL and its implementation on computers. (Bazisnyi REFAL i yego realizatsiya na vychislitelynykh mashinakh.) - GOSSTROY SSSR TSNIPIASS, Moscow, 1977, pp.92-95 (in Russian).

[Tur 79]
V.F.Turchin. A supercompiler system based on the language REFAL. - SIGPLAN Notices, Vol.14, No.2, 1979, pp.46-54.

[Tur 80]
V.F.Turchin. Semantic definitions in REFAL and the automatic production of compilers. - In : Semantic Directed Compiler Generation (N.D.Jones Ed.). Springer LNCS 94 (1980),pp.441-474.

[Tur 86]
V.F.Turchin. The concept of a supercompiler. - ACM Transactions on Computer Languages and Systems, Vol.8, No.3, July 1986, pp.292-325.