RAPPORT
NR: 85/11
ISSN 0107-8283

The Structure of a Self-Applicable Partial Evaluator

Peter Sestoft

Datalogisk Institut, Københavns Universitet
Institute of Datalogy, University of Copenhagen
Sigurdsgade 41 DK-2200 København N

B

The Structure of a Self-Applicable Partial Evaluator

Peter Sestoft

DIKU
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
November 15th, 1985

Table of Contents

A preliminary version of this paper was presented at the workshop "Programs as Data Objects" at DIKU, Copenhagen, Denmark in October, 1985. A shorter version will appear in the proceedings of this workshop, published in Springer Lecture Notes in Computer Science.

The Structure of a Self-Applicable Partial Evaluator

Peter Sestoft

DIKU
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
November 15th, 1985

Introduction

The present paper describes the ideas behind a simple, self-applicable partial evaluator called Mix as well as its structure. This partial evaluator was developed at DIKU (by Neil D. Jones, Harald Søndergaard, and the author) during 1984 with the explicit goal in mind that it should be self-applicable and thus make possible the automatic construction of compilers from interpreters and even of a compiler generator. This work is already partly documented in (Jones, Sestoft, Søndergaard 85).

Outline

The structure of the present paper is as follows.

First, the concept of partial evaluation is defined and various pieces of notation are introduced. Second, the application of partial evaluation to compiling and compiler generation is explained, and the goals and problems of the project are discussed. Third, the solutions to these problems and the resulting structure of Mix are described in the central part of the paper. Finally, we sum up what has and what has not been done, and suggest further work.

1 Partial Evaluation - Concepts and Notation

In this chapter, we give a brief formal definition of partial evaluation and of some concepts related to compiling. These will be used extensively in the following. The definitions are the same as those given in (Jones, Sestoft, Søndergaard 85).

1.1 Programming Languages

Since we use programs as input to other programs and even to themselves, we assume that programs and data will be of the same nature, that is, members of a universal domain D of symbols (e.g. character strings, LISP lists, natural numbers, or the like). Below, D* is the set of finite sequences of elements from D; broken arrow $\dashrightarrow$ means partial function; equality "=" means that either both sides are undefined or they are both defined and equal.

Definition A *programming language* L, then, is a "semantics function" $L : D \dashrightarrow D^* \dashrightarrow D$ so that $L\ p$ is the function $L\ p : D^* \dashrightarrow D$ computed by program p, and $L\ p \langle d_1, ..., d_n \rangle$ is the result of running L-program p on data $\langle d_1, ..., d_n \rangle \in D^*$. The set of L programs is the subset of D to which L assigns a meaning, i.e. L-programs = domain(L) . □

1.2 Partial Evaluation

First, we will introduce the concept of residual program.

Definition Let L be a programming language, p ∈ L-programs a program. Then r ∈ L-programs is a *residual program* for p with respect to known input $\langle x_1, ..., x_m \rangle \in D^*$ iff

$$L\ p \langle x_1, ..., x_m, y_1, ..., y_n \rangle = L\ r \langle y_1, ..., y_n \rangle$$

for all sequences of remaining input $\langle y_1, ..., y_n \rangle \in D^*$. □

That is, the residual program r is the result of "running the original program p on partially known input" or "specializing p to fixed partial input" $x_1, ..., x_m$.

Now, a partial evaluator *mix* is defined as being a program that produces residual programs.

Definition An *L-partial evaluator mix* is an L-program such that for any L-program p, and partially known input $\langle x_1, ..., x_m \rangle \in D^*$, L *mix* $\langle p, x_1, ..., x_m \rangle$ is a residual program for p with respect to $\langle x_1, ..., x_m \rangle$, or in other words,

mix has
variable
arity m+1

$$L\ (L\ mix\ \langle p, x_1, ..., x_m \rangle) \langle y_1, ..., y_n \rangle = L\ p \langle x_1, ..., x_m, y_1, ..., y_n \rangle \qquad (1)$$

for all sequences of remaining input $\langle y_1, ..., y_n \rangle \in D^*$.

The L program p is called a *subject program* and accordingly, L is the *subject language* of the partial evaluator. The data $\langle x_1, ..., x_m \rangle$ are called the *known input*, and $\langle y_1, ..., y_n \rangle$ the *unknown* or *residual input*. A partial evaluator defined this way (it is itself written in its own subject language) is called an *autoprojector* by (Ershov 82). □

A partial evaluator is an implementation of the primitive recursive function S from Kleene's S-m-n Theorem of recursive function theory (Kleene 52). By that theorem, partial evaluators exist.

1.3 Interpreters and Compilers

Let L and S be programming languages.

Definition An *interpreter for S (written in L)* is an L-program int so that

$$\text{L int } \langle s, d_1, \ldots, d_n \rangle = \text{S s } \langle d_1, \ldots, d_n \rangle \tag{2}$$

for all S-programs s and input tuples $\langle d_1, \ldots, d_n \rangle \in D^*$. □

int has variable arity n+1, or it takes one sequence (arity 1) of any length n+1 (then length is variable).

Definition A *compiler from S to L (written in L)* is an L-program comp so that

$$\text{L (L comp } \langle s \rangle) \langle d_1, \ldots, d_n \rangle = \text{S s } \langle d_1, \ldots, d_n \rangle \tag{3}$$

for any S-program s and all input tuples $\langle d_1, \ldots, d_n \rangle \in D^*$. □

All the above concepts and definitions can be generalized in various obvious ways. For example, a partial evaluator may produce residual programs in a language different from the language in which it is written.

2 Goals, Motivations, and Problems

First we describe the goals of the Mix project and their background. Then we proceed to discuss some of the practical and theoretical problems of attaining these goals.

2.1 The Applications of Partial Evaluation to Compiling

This is a very brief account of the relations between partial evaluation and compiling. For a fuller treatment, see for example (Ershov 78, 82), (Futamura 83), (Jones, Sestoft, Søndergaard 85), or (Turchin 80).

In the following, let L be an implementation language, i.e. a language for which we have a processor (compiler, interpreter), so that L-programs, in fact, can be executed. In our case, L is a very small subset of LISP with some syntactic sugar (extensions) which will be described in Section 3.1.

Also, let *mix* be an L-partial evaluator (an autoprojector for L), and let S be a programming language. Then, in principle, the following is feasible.

Compiling

From an S-interpreter int and an S-program s to make an equivalent L-program target.

Compiler generation

From *mix* and an S-interpreter int to make a compiler comp from S to L written in L.

Compiler generator generation

From *mix* alone to make a compiler generator capable of transforming interpreters into compilers.

The formal reasons for this are

Compiling

With target = L *mix* <int, s> (4)

we have

$$\begin{array}{lll} \text{L target } \langle d_1, \ldots, d_n \rangle & = & \text{by (4)} \\ \text{L (L } \mathit{mix} \text{ } \langle \text{int, s} \rangle) \langle d_1, \ldots, d_n \rangle & = & \text{by (1)} \\ \text{L int } \langle \text{s}, d_1, \ldots, d_n \rangle & = & \text{by (2)} \\ \text{S s } \langle d_1, \ldots, d_n \rangle & & \end{array}$$

for all input $\langle d_1, \ldots, d_n \rangle \in D^*$. Therefore, the L-program target and the S-program s are equivalent, and target may be considered a target program for s.

Compiler generation

With comp = L *mix* <*mix*, int> (5)

we have

$$\begin{array}{lll} \text{L comp } \langle \text{s} \rangle & = & \text{by (5)} \\ \text{L (L } \mathit{mix} \text{ } \langle \mathit{mix}\text{, int} \rangle) \langle \text{s} \rangle & = & \text{by (1)} \\ \text{L } \mathit{mix} \text{ } \langle \text{int, s} \rangle & = & \text{by (4)} \\ \text{target} & & \end{array}$$

(from above) for all S-programs s . Therefore, comp is a compiler from S to L written in L.

Compiler generator generation

Finally, with

cocom = L *mix* <*mix*, *mix*> (6)

we have

$$\begin{array}{lll} \text{L cocom } \langle \text{int} \rangle & = & \text{by (6)} \\ \text{L (L } \mathit{mix} \text{ } \langle \mathit{mix}, \mathit{mix} \rangle) \langle \text{int} \rangle & = & \text{by (1)} \\ \text{L } \mathit{mix} \text{ } \langle \mathit{mix}\text{, int} \rangle & = & \text{by (5)} \\ \text{comp} & & \end{array}$$

(from above) for any S-interpreter int written in L. Therefore, cocom is a general compiler generator, transforming interpreters into compilers. (More generally, cocom implements the currying function on the representation of general recursive functions as programs).

Compiling and compiler generation along the lines sketched above were described for the first time in (Futamura 71), while it seems that (Turchin 79) contains the first reference to the idea of obtaining a compiler generator by partial evaluation (according to Prof. Turchin the idea dates back to 1975).

2.2 Practice Lagging Behind Theory

While the feasibility *in principle* of compiler resp. compiler generator generation has been known for more than a decade, apparently nobody has realized these in practice until fall 1984. Also, this seems to be the case in spite of the numerous attempts to do that (mainly in Japan, Sweden, and the USSR). Thus, *compiling* using partial evaluation was realized using a variety of formalisms and languages, see for example (Ershov 78), (Emanuelson, Haraldsson 80), (Kahn, Carlsson 84), and (Haraldsson 78). But as far as we know, no one has reported success in producing compilers or a compiler generator this way.

Inspired by this problem, we (initially Neil D. Jones and Harald Søndergaard) set out to produce a partial evaluator capable of producing compilers as well as a compiler generator. Also, some interest and insight into the problem stemmed from its relationship to the CERES compiler generator project, expounded in (Jones, Tofte 83).

As can be seen from equations (5) and (6) above, a partial evaluator has to be *self-applicable* in order to achieve the goal mentioned. Probably, this is the main source of problems, practical as well as theoretical, and the reason why the earlier efforts remained fruitless.

2.3 Obstacles to Self-Application

I think that the following problems with writing a self-applicable partial evaluator can be distinguished:

1. Partial evaluation is not well defined.
2. When reasoning about the process of self-application one tends to confuse the usually disparate levels of program and data.
3. The fact that the subject language of the partial evaluator is input language as well as meta-language for the partial evaluator makes the choice of an appropriate subject language hard as well as important.

I will discuss these problems in some greater detail.

1. The definition of a partial evaluator (and equation (1)) does not capture the natural expectation that the residual program produced by a partial evaluator should be "reasonable", i.e. neither unnecessarily large nor too inefficient. We would like the partial evaluator to be able to take the greatest possible advantage of the subject program's known input to make this into an efficient specialized residual program. But the definition of a partial evaluator, in fact, allows it to make trivial residual programs. That is, it may make from a subject program p consisting of one function f of two parameters,

```
        f(x,y) =  ...
```

and a known value a for x , a *trivial residual program* like this

```
        g(y)   =  f(a, y)
        f(x,y) =  ...
```

This residual program is, of course, correct but not interesting (except that it proves the existence of partial evaluators in the same way the S-m-n Theorem is proved). We would like to make the definition of partial evaluation more precise by stating some of its desired properties, e.g. always making the shortest possible (or fastest possible) residual program, or (much weaker) always producing a constant expression when the result of the subject program depends only on the known input. But this seems to make partial evaluation (of general recursive functions) an uncomputable problem. The paper (Heering 85) gives a precise meaning to the vague requirement "make maximal use of known input" and shows that, in general, this is not possible using a finite number of rewriting (reduction) rules. The consequence of this is that we have no *precise, useful requirements* for a partial evaluator that could help us in development process, or in proving that an alleged partial evaluator is not the trivial one producing trivial residual programs.

2. When running L *mix* <*mix*, *mix*> , that is, applying a partial evaluator to itself to produce a compiler generator, we see the text of *mix* in three very different roles. First, as a partial evaluator to be run, second, as a program to be partially evaluated (= as first input to a running partial evaluator), and third, as known input to a program to be partially evaluated (= as second input to partial evaluator). But the fact that the representations (program texts) are indistinguishable, makes it very hard to reason in cold blood about what takes place during the process of partial evaluation.

3. The subject language of the partial evaluator must be very carefully chosen to satisfy somewhat conflicting demands:

   On the one hand, as subject language to be processed by the partial evaluator, it should be as simple as possible to process. Therefore, it should:

   - have a simple syntax (few, uniform language constructs) so that programs can be easily represented and handled as data structures
   - have a simple semantics (in other words, be quite small and unsophisticated).

   On the other hand, as the language in which the partial evaluator is to be written, it should:

   - support straightforward representation and manipulation of programs (as trees/terms)
   - support structuring/abstraction/modularization in order to ease program construction
   - be (humanly) readable
   - have some reasonably efficient implementation
   - be expressive, convenient to work with.

   This is mainly a practical problem, of course, but a very important one. Developing a new algorithm of the complexity of a usable partial evaluator requires (has required!) much experimentation and repeated rewriting of major parts of the system. When one has to program in a very restricted language, forcing one to use lots of tricks and clever encodings (such as handling a recursion stack explicitly), it becomes unbearable and one tends to lose belief in the entire undertaking. In short, a wise choice of subject language is a prerequisite for success.

3 The Partial Evaluator Mix

In this chapter a quite detailed account of the algorithms of the partial evaluator Mix will be given.

First, the subject language we chose for Mix is presented. Second, the structure of Mix and some of Mix's algorithms are presented together with reasons for their being that way. Thus analysis is not clearly separated from presentation. Third, an extension to Mix called "variable splitting" is described, and some of our experience with producing compilers and a compiler generator using our partial evaluator Mix are described.

The paper (Sestoft 85) gives some directions for using the Mix system implementation (as of spring 1985). This is not attempted here.

3.1 The Subject Language L of Mix

Above, we used L for the subject language of a partial evaluator *mix*. Below we describe our partial evaluator Mix and its *particular* subject language called L.

First we list some useful characteristics of L, then we give its syntax and an informal semantics. A syntactic extension called LETL is also described, and an L-interpreter is given as an example of the use of LETL.

Characteristics of the Language L

We chose L to be a first-order, statically scoped subset of pure applicative LISP without special treatment of numbers. Therefore L has the following characteristics:

- programs are easily represented as data structures (LISP lists), and a program is its own abstract syntax tree; hence, programs are easily analyzed, decomposed and composed, and therefore, easily transformed.
- manipulation of (syntax) trees is naturally expressed by recursion in L.
- L has a very simple and regular syntax (all operators have fixed arities in contrast to "real" LISP where **cond** and **list** violate this requirement) as well as semantics.
- there exist reasonably efficient implementations of L.

The main drawback of this language is that it is very tedious to program in because of all the parentheses needed to express structure and because of the need to use **car/cdr** sequences to select branches in a tree. This problem, however, is alleviated by the extension LETL described below.

Syntax and Informal Semantics of L

The only data type is LISP lists.

1. A program is a non-empty list of function definitions

<program> ::= (<fcn-def> <fcn-def> *)

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.

2. A function definition consists of a function name, a list of parameters, and a function body.

<fcn-def> ::= (<fname> <parlist> <body>)

The scope of the parameters is the body
of the function.

3. A function name is a symbol (a LISP atom), a parameter list is a list of symbols (LISP atoms), and a function body is an expression

<fname> ::= <atom>
<parlist> ::= (<atom> *)
<body> ::= <exp>

4. An expression is either a constant, a variable, or an operator **car**, **cdr**, **atom**, **cons**, **equal**, or **if**, applied to expressions, or a function call

<exp> ::= (**quote** <LISP-list>) -- shorthand: '<LISP-list>
| <variable>
| (**car** <exp>)
| (**cdr** <exp>)
| (**atom** <exp>)
| (**cons** <exp> <exp>)
| (**equal** <exp> <exp>)
| (**if** <exp> <exp> <exp>)
| (**call** <fname> <argexps>)

A variable refers to the value of a parameter of the function in whose body it appears. All operators are strict and call-by-value except **if** and **quote**, and they have their usual (LISP-) semantics.

5. "Argument expressions" is a sequence of expressions

<argexps> ::= <exp> *

Extension LETL of L

In order to facilitate programming in L, we define an extension LETL adding much to the practical usability of L. Also, we have written a LETL to L compiler automatically transforming the LETL constructs into basic L constructs.

LETL extends L with

- **let** and **where** decomposition patterns, e.g. (**let** (op exp1 exp2) = exp **in** ...). This eliminates the need for **car/cdr** expressions to decompose trees, as well as a lot of parentheses.
- an **if-then-elsf-else** (syntactically sugared McCarthy) conditional
- an infix, right associative **cons** operator "::". (**cons** a (**cons** b c)) = (a :: b :: c)
- logical connectives: **null, not, and, or**
- a **list** builder

The paper (Sestoft 85) describes these languages in more detail.

self-interpreter

An L Interpreter Written in LETL

Here we give a (metacircular) definition of L in the form of an interpreter for L , also serving as an example of a LETL program.

```
(
(L-int (program input) =
      (let ((fname1 parlist1 body1) . rest)  =  program  in
            (call Exp body1 parlist1 input program)))

(Exp (exp vnames vvalues program) =
      (let    (op exp1 exp2 exp3)          =  exp
              (call? fname . argexps)      =  exp  in
      (if     (atom exp)            then
              (call Lookupv exp vnames vvalues)
      elsf    (equal op 'quote)     then
              exp1
      elsf    (equal op 'call)      then
              (call Call     (call Lookupf fname program)
                             (call Pars argexps vnames vvalues program)
                             program)
      else    (let v1 = (call Exp exp1 vnames vvalues program)  in
      (if     (equal op 'car)       then  (car v1)
      elsf    (equal op 'cdr)       then  (cdr v1)
      elsf    (equal op 'atom)      then  (atom v1)
      elsf    (equal op 'if)        then
              (if  v1  then (call Exp exp2 vnames vvalues program)
              else          (call Exp exp3 vnames vvalues program))
      else    (let v2 = (call Exp exp2 vnames vvalues program)  in
      (if     (equal op 'equal)     then  (equal v1 v2)
      elsf    (equal op 'cons)      then  (cons v1 v2)
      else    (list 'SYNTAX  'ERROR:  exp))))))))

(Call (fcn-def vvalues program) =
      (let  (fname parlist body)  =  fcn-def  in
            (call Exp body parlist vvalues program)))

(Pars (explist vnames vvalues program) =
      (let    (exp1  .  exprest)  =  explist  in
      (if     (null explist)        then  'nil
      else    (cons (call Exp exp1 vnames vvalues program)
                    (call Pars exprest vnames vvalues program)))))

(Lookupv (var vnames vvalues) =
      (let    (vn1  .  vnr)  =      vnames
              (vv1  .  vvr)  =      vvalues      in
      (if     (null vnames)         then  (list  'UNKNOWN  'VARIABLE: var)
      elsf    (equal var vn1)       then  vv1
      else                                (call Lookupv var vnr vvr))))

(Lookupf (fname program) =
      (let    ( (fcn-def1  :  (f1 pars1 body1))  .  rest)  =  program          in
      (if     (null program)      then  (list  'UNKNOWN  'FUNCTION: fname)
      elsf    (equal fname f1)    then  fcn-def1
      else                              (call Lookupf fname rest))))
)
```

3.2 Structure of the Partial Evaluator

In this section we will describe the structure of the partial evaluator. First, we give a presentation of the general ideas and an overview of the phase structure of the partial evaluator, then a more detailed discussion is attempted. Section 3.3 below describes the individual phases and the actual algorithms of the partial evaluator.

3.2.1 Ideas Behind and Structure of Mix

An Example of Partial Evaluation

Consider the following LETL-program (in which we have left out some parentheses) with two parameters, an atom x and a linear list y. Output is a list of the same length as the list y, each element of which is the atom x. However, if x is nil, it will be a list of "a"s, preceded by the atom "EXCEPTION".

```
h(x,y) =    if (null x) then (cons 'EXCEPTION (call h 'a y))
            else if (null y) then 'nil
            else (cons x (call h x (cdr y)))
```

We would like to partially evaluate this program for y unknown and x known to be nil.

Now we can proceed to evaluate (**call** h 'nil y) symbolically by *unfolding* h, i.e. replacing the call by the function definition. The conditional (**null** x) is known to be true, therefore

```
h('nil,y) =  (cons 'EXCEPTION (call h 'a y))                          (r1)
```

Evaluating (**call** h 'a y) symbolically we get

```
h('a,y) =  if (null y) then 'nil else (cons 'a (call h 'a (cdr y)))   (r2)
```

This we could further unfold to

```
h('a,y) =  if (null y) then 'nil
           else if (null (cdr y)) then '(a)
           else (cons 'a (cons 'a (call h 'a (cdr (cdr y)))))          (r2')
```

but it would not lead to much improvement, and such further unfolding could never eliminate the need for recursion, since we have no bound on the length of y, and so we stick to the first version (r2) above.

Since we cannot do more useful transformations by symbolic evaluation alone, we will make the above equations (r1) and (r2) into a residual program with two functions. The first being the goal function, which is h specialized to x='nil, and the other a variant of h specialized to x='a, thus

$h_{['nil]}$ (y) = (**cons** 'EXCEPTION (**call** $h_{['a]}$ y))

$h_{['a]}$ (y) = **if** (**null** y) **then** 'nil **else** (**cons** 'a (**call** $h_{['a]}$ (**cdr** y)))

Summary: This residual program was constructed by evaluating expressions symbolically, unfolding function definitions, and suspending function calls (deciding *not* to unfold), and finally, by making function variants specialized to certain values of the known parameter. In principle our partial evaluator Mix uses exactly these transformations.

We will introduce a little terminology. Suppose the subject program has goal function

$$f_1 (x_1, ..., x_m, y_1, ..., y_n) = \text{exp}_1$$

and that the subject program's known input parameters (those available during partial evaluation) are $x_1, ..., x_m$. Then a parameter x_{ij} of some function f_i is said to be *Known* during partial evaluation if the value of x_{ij} can only depend on the values of the parameters $x_1, ..., x_m$ that are available, not on $y_1, ..., y_n$ that are not available. Correspondingly, x_{ij} is said to be *Unknown* if it may depend on $y_1, ..., y_n$.

Mix Principles

a. The residual program corresponding to a subject program and its known input consists of a collection of function definitions, each resulting from *specializing* (the body of) some function definition in the given subject program to known values of some of its parameters. These are called residual functions.

b. Intuitively, partial evaluation proceeds as *symbolic evaluation* of the subject program. Instead of parameters being bound to their actual values, they are bound to L-expressions denoting their possible values. Symbolic evaluation of expressions which do not contain function calls is straightforward reduction/rewriting of the expressions. Evaluating a function call symbolically, we can do one of two things: *Unfold* the call (i.e. replace it with the reduced equivalent of the called function's body) or *suspend* the call (i.e. replace it with a call to a residual variant of the called function).

c. We require the user of the partial evaluator to decide (before applying it) which function calls in the subject program should be *unfolded* (eliminable call) and which should be *suspended* (residual call). This is done by *annotating* the function call with an "r" (for residual, yielding **callr**) if the user wants it to be suspended.

d. The partial evaluation process is *divided into phases*.

First, the (call annotated) subject program is abstractly interpreted over a value domain only distinguishing known and unknown values. This results in information on which parameters of each function will be known at partial evaluation time, and which will possibly be unknown. The information obtained is used in the second phase for annotating the subject program, dividing the parameter list of each function into two: the *eliminable parameters* (known values at partial evaluation time) and the *residual parameters* (values possibly unknown). This is required for the later specialization of each function into its (zero or more) residual variants in the residual program, cf. a. above. Also, each *operator* **car, cdr**, ... is annotated either as *eliminable* (**care, cdre**, ...) or as *residual* (**carr, cdrr**, ...), yielding a heavily annotated version of the subject program. The third phase then takes as input the subject program annotated with respect to calls, parameters, and operators, together with the actual values of the subject program's known input. In this phase, the residual program is constructed as a number of variants of the subject program's functions, specialized to various values of their eliminable parameters.

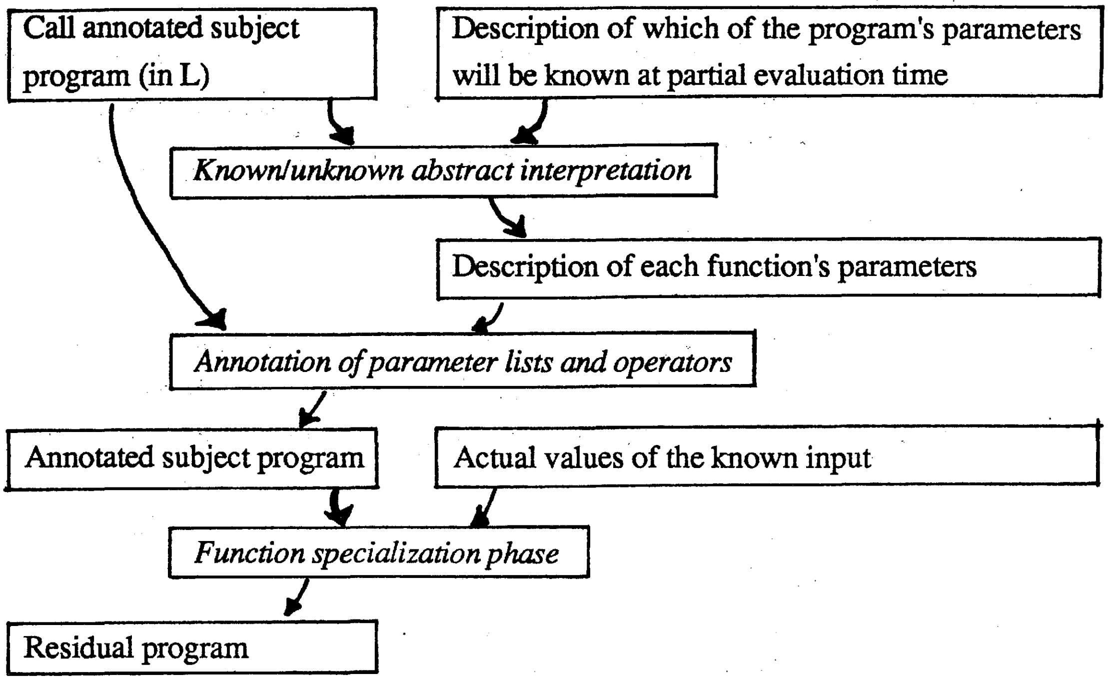

Fig. 1: Phase Division of Partial Evaluation

Italics denote phases in the process, whereas plain text denotes objects handled by the phases.

3.2.2 Discussion

Here, a more detailed yet brief treatment of the above is given.

a. **Building the residual program from specializations of the functions** in the subject program is the main principle. In principle, those specializations which have to appear in the residual program are determined as follows: If we consider the space of possible inputs to the subject program with its eliminable parameters restricted to their given, known values $\langle x_1, ..., x_m \rangle$ we have a subspace $\{\langle x_1, ..., x_m \rangle\} \times D^n$ of possible inputs, obtained by varying the remaining input $\langle y_1, ..., y_n \rangle$. Now the residual program has to have a variant $f[\langle ev_1, ..., ev_i \rangle]$ of a function f specialized to known values $\langle ev_1, ..., ev_i \rangle$ if in the course of running the subject program on any input from the subspace mentioned, f is called by a residual call with parameter values $\langle ev_1, ..., ev_i, rv_1, ..., rv_j \rangle$ for some values $\langle rv_1, ..., rv_j \rangle$ of the residual parameters. This is of course equivalent to stating that the residual program is complete in the sense that there has to be a residual variant of each function for every possible value of the eliminable parameters with which the original function can be called (because the eliminable parameters will not appear in the residual program). The variants in the residual program of a function from the subject program thus make up a kind of *tabulation* of the possible values of that function's eliminable parameters for any computation on the mentioned subspace of inputs. Our partial evaluation technique in this respect thus resembles those described in (Bulyonkov 84) for a simple imperative language, and in (Futamura 83) for an applicative language. Clearly, for partial evaluation to terminate this tabulation has to be finite. For "syntax directed" naturally recursive programs such as interpreters this is usually the case, but for programs handling a recursion stack of known values, for example, this is often not the case. (This might indicate that partial evaluation of imperative programs requires more sophisticated methods than partial evaluation of applicative programs).

b. **Symbolic evaluation** is the most operational, intuitive conception of partial evaluation. Symbolic evaluation takes place in a "symbolic environment" binding each variable to an expression instead of some concrete value. For each operator **car**, **cdr**, ... we have an evaluation (reduction) procedure that reduces, say, (**car** exp) as much as possible based on the form of the residual expression exp-r for exp, according to this table:

form of exp-r	(car exp)
(quote (t_1 . t_2)**)**	**(quote** t_1**)**
(**cons** exp_1-r exp_2-r)	exp_1-r
otherwise	(**car** exp-r)

c. **By requiring the user to make the call annotations**, we also put much of the responsibility for a reasonable structure of the residual programs on him.

Here we list various anomalous behaviours and explain their relation to call annotations.

1. Partial evaluation may loop infinitely. One reason for this may be too few residual calls, so that it is attempted to unfold a loop whose termination (test) essentially depends on the unknown input. Either, this is an infinite loop that would have occurred in total (usual) evaluation also, or it corresponds to an attempt to build an infinite residual expression, for instance, to try to unfold

f(x) = **if** c(x) **then** $e_1(x)$ **else** (**call** f $e_2(x)$)

(where c(x), $e_1(x)$, and $e_2(x)$ are expressions possibly containing x) to its infinite equivalent

f(x) = **if** c(x) **then** $e_1(x)$
else if c($e_2(x)$) **then** $e_1(e_2(x))$
else if c($e_2(e_2(x))$) **then** ...

(An attempt to produce at partial evaluation time infinitely many specializations of a function is another source of non-termination in partial evaluation, and this is independent of call annotations).

2. Partial evaluation may produce extremely slow residual programs. This can be the consequence of call duplication, that is, in the residual program the same subexpression containing a call is evaluated more than once. In the case that a function calls itself twice on the same substructure of one of its parameters, its run time may well shift from linear to exponential because of call unfolding. Witness the linear time program

f(n) = **if** (**null** n) **then** '1 **else** (**call** g (**call** f (**cdr** n)))
g(y) = (**cons** y y)

(with n unknown) which should not be unfolded to the exponential-time program

f(n) = **if** (**null** n) **then** '1 **else** (**cons** (**call** f (**cdr** n)) (**call** f (**cdr** n)))

Such call duplication usually can be avoided by inserting more residual calls.

3. Partial evaluation may produce extremely large residual programs. This is a "size" counterpart of the above exponential run time anomaly. Consider the program

f(n,x) = **if** (**null** n) **then** x **else** (**call** g (**call** f (**cdr** n) x))
g(y) = (**cons** y y)

with n known, x unknown. When n has length 1, unfolding f('(1) x) yields (**cons** x x), and when n has length 2, unfolding f('(1 1) x) yields (**cons** (**cons** x x) (**cons** x x)). For an n with length 10, the residual expression has $2^{10} = 1024$ x's and 1023 **cons**-operators, and it is equivalent to a program with 12 functions containing a total of 20 **calls** and one **cons**-operator, namely

$f_{10}(x) =$ (**call** g (**call** f_9 x))

...

$f_1(x) =$ (**call** g (**call** f_0 x))
$f_0(x) =$ x
$g(y) =$ (**cons** y y)

None of these problems are contrived; we have experienced all of them, only in more complicated settings. Note that, in general, it may be impossible to make call annotations for a subject program in a way ensuring reasonable residual programs. However, call annotation of syntax directed programs usually is not hard and can be semi-automated (by finding *unsafe* cycles in the call graph of the subject program without a descending known parameter). We have not done that yet, but it is currently being investigated.

d. **Dividing the partial evaluation process into phases,** a *statically* determined partitioning of each function's parameters into eliminable resp. residual parameters is obtained, as well as a statically determined classification of all operators in the subject program as either (definitely) eliminable or (possibly) residual.

The ideas are that known/unknown abstract interpretation yields *global* information on the subject program's possible run-time behaviour, and that the annotations represent this static information *locally*. *In principle*, static classification of parameters and operators is not necessary since the classification can be done *dynamically* (during symbolic evaluation/function specialization). That is, it can be determined dynamically whether an operator is doable independently of the unknown input, namely if its operands evaluate (symbolically) to constant expressions (**quote** ...). However, it turns out to be a prerequisite for successful self-application of the partial evaluator (and a distinguishing feature of ours) that the classification *is* made statically based on a description of which of the subject program's input parameters are known. We will try to give an operational explanation of this rather subtle problem.

We attempted to produce a compiler comp (from some S-interpreter int) by running comp = L mix_1<mix_2, int>, with *dynamic* operator classification, i.e. without operator annotations. (Here, $mix_1 = mix_2 =$ Mix, the indices are for reference only). This resulted in compilers of monstrous size, far too big to be printed out.

The reason turned out to be this: mix_1 as well as mix_2 contain some procedure for simplifying expressions such as (**car** exp) as much as possible at partial evaluation time. This depends on the residual (reduced) form exp-r of exp, which in turn depends on the form of exp and the values of the subject program int's known input. The operators occurring in mix_2 are of course nicely reduced by mix_1 but consider mix_2 being partially evaluated on int as above. Now focus on the application of mix_2's reduction procedure for **car** on an expression (**car** exp) in int. Let us assume that in int, this **car** expression's operand is int's first parameter (an S source program s). During *compilation* one applies *mix* to int and a source program, target = L *mix* <int,s>. Thus the source program s is present, and the **car** operator of int can be evaluated by *mix*. But during *compiler generation*, running comp = L mix_1<mix_2, int>, the source program s is not available and therefore even the *form* of the residual expression exp-r for exp in int is unknown. Therefore, the reduction procedure (in mix_2) for **car** cannot be executed by mix_1, and the compiler produced (i.e. the residual program for mix_2) will contain the entire reduction procedure for **car** for this single occurrence of **car** in int.

This procedure will be entirely superfluous since when running the produced compiler on an S source program, that program will be known, and a single **car** operator could replace the reduction procedure comprising several lines of L text. In fact, the problem is worse yet, because (**car** (**cdr** exp)) in the interpreter int will be "reduced" to the reduction procedure for **car** with the entire reduction procedure for **cdr** instantiated in several places. Thus the size of residual expressions in the compiler depends in an exponential way on the complexity of expressions in the interpreter, and this is clearly not acceptable.

If, on the other hand, operator annotations (static classifications into eliminable resp. residual operators) are used, a **car** operator in int working on int's eliminable input (the S source program) will be annotated eliminable (**care**), and partial evaluation of mix_2 on int will produce a single **car** operator in the compiler instead of a copy of the complicated reduction procedure. Note that it is the annotation of int that matters. Hence, this problem really is one of self-application.

Now, could not mix_1 (dynamically) infer that the operand of the discussed **car** operator in int that mix_2 is about to reduce depends only on int's first parameter, the S source program? Then mix_1 could avoid duplicating the entire reduction procedure for **car** (in mix_2) in the compiler, since it knew that when running the compiler on an S source program, that program would be known, and hence a single **car** would suffice in the compiler. That would require a *global flow analysis* of int at partial evaluation time to determine that the argument of this **car** operator only depends on the first parameter of int as is done during the first phase, the known/unknown abstract interpretation.

This should suffice to justify the need for dividing the partial evaluator into at least two phases. Notice that the known/unknown abstract interpretation introduces another binding time: The annotation of a subject program not only requires the subject program but also a description of *which* of the subject program's parameters will be known at partial evaluation time, so the subject program is, in fact, annotated for a *particular* use.

This concludes the discussion of the distinguishing principles of our partial evaluator.

3.3 Description of the Phases

In this section, the individual phases of the partial evaluation process and some of the algorithms involved are described in the order they are used.

With reference to the sketch of the structure (Figure 1), the phases are: Known/Unknown abstract interpretation described in subsection 3.1.1, the process of partitioning parameter lists and annotating operators, described in subsection 3.3.2, and the proper function specialization process, described in subsection 3.3.3. Some further post-transformations are described in subsection 3.3.4 that closes this section.

3.3.1 Known/Unknown Abstract Interpretation

The **purpose** of this phase is to compute for every function in the subject program a safe description of its parameters, whether they are definitely known or possibly unknown at partial evaluation time.

Inputs to this phase are 1) the call annotated subject program, and 2) a description of which of the subject program's (i.e. which of the goal function's) parameters are known and which are unknown at partial evaluation time. That is, this phase does not use the actual values of the known input, just a description telling *which* of the input parameters are known. (Equivalent to providing a value for m in Kleene's S-m-n Theorem).

Output is a *description*, i.e. a mapping that associates with every function a *parameter description*, classifying each of its parameters as Known resp. Unknown at partial evaluation time. Here Known means "definitely known for all possible values of the subject program's known input", and Unknown means "possibly unknown for some (or all) values of the known input".

For the following exposition we will assume this L subject program given

$$
\begin{array}{l}
((f_1\ (x_{11} \cdots x_{1k[1]})\ \ exp_1) \\
\qquad\qquad \ldots \\
\ (f_n\ (x_{n1} \cdots x_{nk[n]})\ \ exp_n)\ )
\end{array}
$$

Figure 2: An L Subject Program

consisting of $n \geq 1$ functions f_i each having $k_i \geq 0$ parameters, $i=1,\ldots,n$. Then the program's input is a k_1-tuple $\in D^{k_1}$, where D is the domain of LISP lists.

Algorithm 3.3.1

The phase works by an abstract interpretation of the subject program over a domain with two values for expressions, **D** = {Known, Unknown}. During this abstract interpretation, for every function a parameter description is maintained, telling for every parameter of the function whether it can be called with an unknown value. (Note that a parameter description may be considered an "abstract environment", associating with every parameter of a function an abstract value). Initially, all parameters except the goal function's are considered Known, and the parameter description for the goal function is the initial description given for the subject program's input parameters.

The abstract interpretation proceeds as follows: The body of the goal function is evaluated (using the parameter description) to see which functions it may call, giving them Unknown parameter values. The parameter descriptions for these functions are modified according to these findings to tell which of their parameters may be Unknown. Then the bodies of these functions are evaluated using the new parameter descriptions to see which functions they in turn may call with Unknown parameter values and so on. Each time a parameter description of a function becomes more unknown than the previous one, its body is re-interpreted using the new parameter description, possibly implying further re-interpretations of other functions. The process stops when no more parameters of any function f_i can become Unknown as a consequence of a call of f_i from some other function. Then the description computed is safe in the sense that any parameter described as Known will have values only depending on the program's known input at partial evaluation time.

More precisely, the abstract interpretation of the body of a function f_i proceeds in this way: For every call (**call** $f_c\ e_1 \ldots e_{k[c]}$) appearing in the body, the actual parameter expressions $e_1, \ldots, e_{k[c]}$ are abstractly interpreted using f_i's current parameter description (as sketched below) yielding an abstract value (Known or Unknown) for every parameter $x_{c1}, \ldots, x_{ck[c]}$ of the called function f_c. If any parameter x_{cj} described as Known in f_c's parameter description becomes Unknown, that parameter description is changed to Unknown for x_{cj}, and the body of the called function f_c is re-interpreted to check if any more parameters of (other) functions become Unknown as a consequence of this.

Abstract interpretation of parameter expressions is straightforward: A variable has the abstract value given in the current parameter description for the function in which it occurs, and any composite expression has value Known iff it does not contain any variables described as Unknown, otherwise, Unknown.

A More Formal Description of the Algorithm

In order to describe this process more formally, we put the ordering Known < Unknown on the domain **D**. In the sequel, Known and Unknown will be abbreviated K and U, respectively. A description of the parameters of a function f_i is a tuple in $\mathbf{D}^{k_i}$, and a description of all the parameters in the entire program above is a tuple in **Descr** $= \mathbf{D}^{k_1} \times \ldots \times \mathbf{D}^{k_n}$. This domain is partially ordered by using the above ordering "<" componentwise, and it is a complete lattice of finite height, with bottom element $\bot = \langle K^{k_1}, \ldots, K^{k_n} \rangle$, the "most known" description. Notice that the least upper bound $\delta_1 \sqcup \delta_2$ of any two descriptions $\delta_1, \delta_2 \in$ **Descr** exists, and is the most known description safely approximating δ_1 as well as δ_2.

Domains and Elements

D = {Known, Unknown}

δ	: **Descr** $= \mathbf{D}^{k_1} \times \ldots \times \mathbf{D}^{k_n}$	a description for the entire program.
π	: **D***	a parameter description for a function.

By $\text{only}_i(<v_1, \ldots, v_{k[i]}>)$ we denote the element δ of **Descr** with

$$\delta[j] = <K, \ldots, K> = K^{k_j} \qquad \text{for } j \neq i \text{, and}$$

$$\delta[i] = <v_1, \ldots, v_{k[i]}>,$$

i.e. it is K everywhere except at i, where it is $<v_1, \ldots, v_{k[i]}>$.

<u>Functions</u>

Function **A** : Program $\rightarrow \mathbf{D}^{k_1} \rightarrow$ **Descr**

This function returns the final description for the entire subject program, mapping every parameter of every function to either K or U.

$$\mathbf{A}\llbracket((f_1\ (x_{11} \ldots x_{1k[1]})\ exp_1) \ldots)\rrbracket <v_1, \ldots, v_{k[1]}> = h(\text{only}_1(<v_1, \ldots, v_{k[1]}>))$$

$$\underline{\text{whererec}}\ h(\delta) = \delta \sqcup h(\bigsqcup \mathbf{P}\llbracket exp_i \rrbracket\, \delta[i] \text{ for } i=1,\ldots,n)$$

Function **E** : Expression $\rightarrow \mathbf{D}^* \rightarrow \mathbf{D}$

This function computes the abstract value (K or U) of an expression in a given abstract environment.

$$\begin{aligned}
\mathbf{E}\llbracket(\textbf{quote}\ list)\rrbracket\,\pi &= K \\
\mathbf{E}\llbracket \text{variable } x_{ij}\rrbracket\,\pi &= \pi[j] \\
\mathbf{E}\llbracket(\textbf{car}\ exp)\rrbracket\,\pi &= \mathbf{E}\llbracket exp\rrbracket\,\pi
\end{aligned}$$

and similarly for **cdr**, **atom**.

$$\begin{aligned}
\mathbf{E}\llbracket(\textbf{cons}\ e_1\ e_2)\rrbracket\,\pi &= \mathbf{E}\llbracket e_1\rrbracket\,\pi \sqcup \mathbf{E}\llbracket e_2\rrbracket\,\pi \\
\mathbf{E}\llbracket(\textbf{equal}\ e_1\ e_2)\rrbracket\,\pi &= \mathbf{E}\llbracket e_1\rrbracket\,\pi \sqcup \mathbf{E}\llbracket e_2\rrbracket\,\pi \\
\mathbf{E}\llbracket(\textbf{if}\ e_1\ e_2\ e_3)\rrbracket\,\pi &= \mathbf{E}\llbracket e_1\rrbracket\,\pi \sqcup \mathbf{E}\llbracket e_2\rrbracket\,\pi \sqcup \mathbf{E}\llbracket e_3\rrbracket\,\pi \\
\mathbf{E}\llbracket(\textbf{call}\ f_i\ e_1 \ldots e_{k[i]})\rrbracket\pi &= (\bigsqcup \mathbf{E}\llbracket e_j\rrbracket\,\pi \text{ for } j=1,\ldots,k_i)
\end{aligned}$$

The last rule states that a function having at least one Unknown parameter may return an Unknown value, otherwise only Known values. The rule is the same for **callr**.

Function **P** : Expression $\rightarrow \mathbf{D}^* \rightarrow$ **Descr**

This function computes for a given description exp and a given abstract environment a "small" description that tells for the functions that may be called from exp, which of their parameters will be unknown as a consequence of these calls.

$$\begin{aligned}
\mathbf{P}\llbracket(\textbf{quote}\ list)\rrbracket\,\pi &= \bot \\
\mathbf{P}\llbracket \text{variable } x_{ij}\rrbracket\,\pi &= \bot \\
\mathbf{P}\llbracket(\textbf{car}\ exp)\rrbracket\,\pi &= \mathbf{P}\llbracket exp\rrbracket\,\pi
\end{aligned}$$

and similarly for **cdr**, **atom**.

$$\begin{aligned}
\mathbf{P}\llbracket(\textbf{cons}\ e_1\ e_2)\rrbracket\,\pi &= \mathbf{P}\llbracket e_1\rrbracket\,\pi \sqcup \mathbf{P}\llbracket e_2\rrbracket\,\pi \\
\mathbf{P}\llbracket(\textbf{equal}\ e_1\ e_2)\rrbracket\,\pi &= \mathbf{P}\llbracket e_1\rrbracket\,\pi \sqcup \mathbf{P}\llbracket e_2\rrbracket\,\pi \\
\mathbf{P}\llbracket(\textbf{if}\ e_1\ e_2\ e_3)\rrbracket\,\pi &= \mathbf{P}\llbracket e_1\rrbracket\,\pi \sqcup \mathbf{P}\llbracket e_2\rrbracket\,\pi \sqcup \mathbf{P}\llbracket e_3\rrbracket\,\pi \\
\mathbf{P}\llbracket(\textbf{call}\ f_i\ e_1 \ldots e_{k[i]})\rrbracket\pi &= \underline{\text{let}}\ v_j = \mathbf{E}\llbracket e_j\rrbracket\,\pi \text{ for } j=1,\ldots,k_i\ \underline{\text{in}} \\
&\qquad \text{only}_i(<v_1, \ldots, v_{k[i]}>) \sqcup (\bigsqcup \mathbf{P}\llbracket e_j\rrbracket\,\pi \text{ for } j=1,\ldots,k_i)
\end{aligned}$$

same for **callr**

The actual implementation of the algorithm closely resembles this scheme. It has two main data structures; namely, the partially computed description $\delta \in$ **Descr** as above, and a set Pending of pairs of a function name and a parameter description for that function, $(f_i, <v_1, ..., v_{k[i]}>)$. This set represents the function calls whose effects on the final value of **Descr** are not yet computed. A non-deterministic, imperative version of the algorithm is given below (in reality a deterministic, iterative applicative algorithm is used). In one iteration of the algorithm, an element of Pending (i.e. a call description) is chosen and removed from Pending, the effect on δ of this call is computed, and possibly the for statement adds new call descriptions to Pending in case an old description for any function has changed. The algorithm terminates when Pending becomes empty and is guaranteed to terminate (since the lattice **Descr** is of finite height so that the value of δ may only increase a finite number of times). This is a classical way of computing finite fixed points.

```
Set $<v_1, ..., v_{k[1]}>$ := the description of the subject program's input parameters;
Pending := { $(f_1, <v_1, ..., v_{k[1]}>)$ } ; $\delta := \bot$ ;
while Pending $\neq \emptyset$ do
    choose $(f_i, <v_1, ..., v_{k[i]}>) \in$ Pending, and remove it from Pending;
    $\delta' := \delta \sqcup \mathbf{P}[\![ exp_i ]\!] <v_1, ..., v_{k[i]}>$;
    for all i=1,...,n do
        if $\delta'[i] > \delta[i]$ then Pending := Pending $\cup$ { $(f_i, \delta'[i])$ };
    $\delta := \delta'$;
end;
```

This concludes the description of the Known/Unknown abstract interpretation algorithm.

3.3.2 Annotation of Parameter Lists and Operators

In this phase, the given subject program is transformed, i.e. annotated with respect to parameters and operators for use in the third phase, the function specialization phase.

Inputs to this phase are 1) the call annotated subject program, and 2) the description computed by the above phase, describing every parameter of every function in the program as either Known or Unknown.

Output is the subject program annotated with respect to parameters and operators. That is, the parameter list of each function is divided into a list of eliminable parameters (namely those described as Known) and a list of residual parameters (those described as Unknown). Of course, the argument list of every call to a function f_i is divided into two lists in exactly the same way as the formal parameters of f_i. Also, every operator **car, cdr, cons,** ... is annotated either as eliminable or as residual, becoming **care, cdre, conse,** ... or **carr, cdrr, consr,** ... respectively. An operator being eliminable implies that it is doable during the function specialization phase to follow, or in other words, its result depends only on the values of the known input supplied to the subject program at partial evaluation time, not the unknown. This is not quite true for the **if** operator, since its being eliminable means that the value of its conditional expression depends only on the known input, but then the **if** expression can be reduced to one of its branches during the function specialization phase.

Algorithm 3.3.2

This phase works like a recursive descent compiler, building the annotated subject program one function at a time as it goes through the given subject program. A parameter list (in a function definition) or an argument list (in a function call) is divided into two lists using the description (computed in the previous phase) in a straightforward way. Operators are annotated on the basis of an abstract interpretation of their argument expressions using the function E from subsection 3.3.1, associating with every expression an abstract value in {Known, Unknown}.

An annotated version of the subject program in Figure 2 may look like

$$\begin{array}{l} (\ (f_1\ (ex_{11}\ \dots\ ex_{1k[11]})\ (rx_{11}\ \dots\ rx_{1k[12]})\ exp_1^{ann}\ ) \\ \quad \dots \\ \quad (f_n\ (ex_{n1}\ \dots\ ex_{nk[n1]})\ (rx_{n1}\ \dots\ rx_{nk[n2]})\ exp_n^{ann}\ )\ ) \end{array}$$

Figure 3: An Annotated Subject Program

where $ex_{i1}, \dots, ex_{ik[i1]}$ are the eliminable parameters, $rx_{i1}, \dots, rx_{ik[i2]}$ the residual parameters of function f_i, and together they form a permutation of the original parameter list $x_{i1}, \dots, x_{ik[i]}$, so $k_{i1} + k_{i2} = k_i$, and exp_i^{ann} is the annotated version of exp_i. This annotated subject program will be used for reference below.

3.3.3 Function Specialization

This phase constructs the residual program by making a number of specialized variants of the annotated subject program's functions.

Inputs are 1) the annotated subject program produced by the previous (annotation) phase, and 2) the known input to the subject program, i.e. actual values for those of the goal function's parameters described as Known.

Output is the residual program that is constructed from variants of the annotated subject program's functions. They are specialized to various actual values of their eliminable parameters. The goal function of the residual program is the variant of the subject program's goal function that is specialized to the actual values for its eliminable parameters, i.e. the known input to the subject program. The (formal) parameters of a residual function corresponding to the original function f_i are the residual parameters $rx_{i1}, \dots, rx_{ik[i2]}$, cf. Figure 3. The residual function's name will be (the composite) $f_i[<v_1, \dots, v_{k[i]}>]$ when the function is called by a residual call with values $<v_1, \dots, v_{k[i]}>$ for the eliminable parameters $ex_{i1}, \dots, ex_{ik[i1]}$.

Algorithm 3.3.3

The construction of the residual program has two aspects: 1) Deciding which residual functions are needed for the given values of the known input (cf. subsection 3.2.2, first paragraph), and 2) Producing these residual functions. In principle, this can be done in separate stages, but in our partial evaluator and in the algorithm sketched here, these phases are intermixed. It is not clear whether this really is advantageous or whether it just obscures the algorithm. First, the algorithm will be described in words then a more formal algorithm like that of subsection 3.3.1 will be given. The reader is invited to keep the annotated subject program shown in Figure 3 in mind while reading this section.

Informal Description of the Algorithm

The algorithm resembles the fixed point computation of the Known/Unknown abstract interpretation (subsection 3.3.1) to a great extent. In fact, it can formally be considered an abstract interpretation over some suitable domain also, see (Jones, Mycroft 86) on "minimal function graphs", but here a less rigorous treatment is given. At any time the algorithm keeps a set Pending of function specializations which still have to be produced, and a list Out which contains the residual functions produced so far. The elements of Pending are pairs (f_i,<v_1, ..., $v_{k[i1]}$>) of a function name f_i and a tuple of values <v_1, ..., $v_{k[i1]}$> for f_i's eliminable parameters. A pair (f_i,<v_1, ..., $v_{k[i1]}$>) being in Pending indicates that a variant of f_i specialized to <v_1, ..., $v_{k[i1]}$> is required, but it may already be among the residual functions in Out.

Initially Out is the empty list, and Pending contains one element, namely the pair (f_1,<v_1, ..., $v_{k[11]}$>) consisting of the goal function's name and the known input to the subject program. Hence, there will always be a residual variant of the subject program's goal function, specialized to the subject program's known input, and this becomes the goal function of the residual program.

Now the algorithm works as follows:

1. If Pending is empty, the process is complete and Out is the residual program. Otherwise, choose some pair (f_i,<v_1, ..., $v_{k[i1]}$>) in Pending. If the corresponding residual function already is in Out, repeat this step.

2. Otherwise, produce a residual variant of f_i, called f_i[<v_1, ..., $v_{k[i1]}$>], with parameters rx_{i1}, ..., $rx_{ik[i2]}$ (the residual parameters of f_i), and a body exp_i-r, which is the result of evaluating the body exp_i^{ann} of f_i symbolically. This is done as described below by the function **E**, evaluating exp_i symbolically.

3. Collect the set of residual functions needed by the residual function just produced, i.e. those which it can call. This is represented as a set of pairs (f_j, <v_1, ...,$v_{k[j1]}$>) of a function name f_j and values for its eliminable parameters, and corresponds to the set of residual calls that are encountered when evaluating exp_i symbolically. It is collected by function **P** below. Add this set to Pending and continue with step 1.

Now we sketch the two main procedures **E** and **P** mentioned above: The procedure **E** constructing the residual equivalent of an expression by symbolic evaluation, and the procedure **P** collecting the residual functions called by the residual expression.

Symbolic Evaluation takes place in a "symbolic environment" binding the parameters of a function to expressions rather than values. Here, of course, the eliminable variables are bound to constant expressions (**quote** ...), and residual variables are bound to arbitrary expressions. Symbolic evaluation is quite straightforward. For instance, a variable evaluates to the expression to which it is bound, and symbolic evaluation of expressions which do not contain calls works by reduction. Symbolic evaluation *of calls* is the most interesting case.

An eliminable call (**call** f_i (e_1 ... $e_{k[i1]}$) (r_1 ... $r_{k[i2]}$)) is evaluated symbolically by evaluating the body exp_i of f_i symbolically in a symbolic environment constructed like this: The parameter expressions are evaluated symbolically, yielding residual expressions ev_{i1}, ..., $ev_{ik[i1]}$ resp. rv_{i1}, ..., $rv_{ik[i2]}$ for the eliminable resp. the residual parameter expressions. Now the eliminable parameters ex_{i1}, ..., $ex_{ik[i1]}$ of the called function are bound to ev_{i1}, ..., $ev_{ik[i1]}$, and the same is the case for the residual parameters. Thus symbolic evaluation of an eliminable call is usual call-by-value evaluation, except that the value domain consists of expressions. Note that non-termination is possible here (as in usual evaluation) if a function calls itself recursively by an eliminable call.

A residual call (**callr** f_i (e_1 ... $e_{k[i1]}$) (r_1 ... $r_{k[i2]}$)) has to appear in the residual program, and thus the result of symbolic evaluation is a call (**call** $f_i[<ev_{i1}, ..., ev_{ik[i1]}>]$ rv_{i1} ... $rv_{ik[i2]}$) to a function with the composite name $f_i[<ev_{i1}, ..., ev_{ik[i1]}>]$ and residual argument expressions rv_{i1}, ..., $rv_{ik[i2]}$. Here, as above, ev_{i1}, ..., $ev_{ik[i1]}$ and rv_{i1}, ..., $rv_{ik[i2]}$ are the residual equivalents of the parameter expressions in the call that was symbolically evaluated.

Collecting the Residual Functions Needed for an expression exp to be evaluated symbolically in a certain "symbolic environment" resembles symbolic evaluation a great deal except that the value of an expression is a set of pairs, each representing a necessary residual function. This takes place in an environment where only the eliminable parameters are bound to (constant) expressions. In constant expressions, in variables, and in eliminable expressions **care**, **cdre**, ... (except **ife**), no (new) residual calls can appear. The residual calls of an eliminable **ife** expression are the residual calls of one of its branches; which branch is decided by the value of the conditional expression in the given symbolic environment. The set of residual calls of any expression other than a call is the union of the sets of residual calls of its subexpressions. The set of residual calls of an eliminable call (**call** f_i (e_1 ... $e_{k[i1]}$) (r_1 ... $r_{k[i2]}$)) is the union of those appearing in the expressions r_1, ..., $r_{k[i2]}$ for the residual parameters with those in the body exp_i of f_i. Similarly, the set of residual calls of a residual call (**callr** f_i (e_1 ... $e_{k[i1]}$) (r_1 ... $r_{k[i2]}$)) is the union of those appearing in the residual parameter expressions with the singleton $\{(f_i, <v_1, ..., v_{k[i1]}>)\}$, representing the call itself, where v_j is the residual equivalent of eliminable parameter expression e_j, $j=1,...,k_{i1}$.

A More Formal Presentation of the Algorithm

In the following, we are a bit careless concerning the domains. "Program" in the arity of **R** means "annotated L program", whereas "Program" in the co-arity means "L program extended with composite function names". This remark also concerns Expression. Also, the algorithm will be given in a mixture with its iterative main loop expressed as an imperative program, and the much nicer **P** and **E** expressed in near-denotational form.

Domains and Elements

$F = \{f_1, \ldots, f_n\}$	function names.
Pend $=$ set of $(F \times \mathbf{D}^*)$	set of pairs of a function name f_i and values for the eliminable parameters of f_i.
$\pi_e : \Pi_e =$ Expression*	values (constant expressions) for the eliminable parameters of a function.
$\pi_r : \Pi_r =$ Expression*	values (expressions) for the residual parameters.
$\pi = (\pi_e, \pi_r) : \Pi_e \times \Pi_r$	values for all parameters of a function.
Out : Program	

Functions

In the following, (**car** exp) on the right hand side of an equation will denote the *term* (construction) with operator **car** and the operand denoted by exp.

Function $\mathbf{R}$: Program × $\mathbf{D}^*$ → Program

$\mathbf{R}[\![$ program p $]\!] <v_1, \ldots, v_{k[11]}>$ = Out, computed by the following algorithm (if it terminates)

```
    Pending := { ($f_1$, <$v_1, \ldots, v_{k[11]}$>) }; Out := [];
    while Pending $\neq \emptyset$ do
        choose ($f_i$, <$v_1, \ldots, v_{k[i1]}$>) in Pending, and remove it from Pending;
        if $f_i$[<$v_1, \ldots, v_{k[i1]}$>] is not already defined in Out then
            fname := $f_i$[<$v_1, \ldots, v_{k[i1]}$>];
            body := $\mathbf{E}[\![exp_i^{ann}]\!]$ (<$v_1, \ldots, v_{k[i1]}$>,<$rx_{i1}, \ldots, rx_{ik[i2]}$>) p;          (**)
            fcn-def := ( fname ($rx_{i1} \ldots rx_{ik[i2]}$) body );
            add fcn-def to Out;
            Pending := Pending $\cup$ $\mathbf{P}[\![exp_i^{ann}]\!]$ <$v_1, \ldots, v_{k[i1]}$> ;
        endif;
    end;
```

(**) Note: In this line, $<rx_{i1}, \ldots, rx_{ik[i2]}>$ is a tuple of *variable expressions*, with the effect that residual variable rx_{ij} is bound to *itself* in **E** when symbolically evaluating exp_i^{ann}, $j=1,\ldots,k_{i2}$).

Function **E** : Expression $\rightarrow \Pi_e \times \Pi_r \rightarrow$ Program $\rightarrow$ Expression

This function does symbolic evaluation, i.e. given an expression exp and a symbolic environment, it builds the residual expression corresponding to exp for this environment.

$\mathbf{E}\llbracket(\textbf{quote}\ \text{list})\rrbracket\pi\ p \quad = (\textbf{quote}\ \text{list})$

$\mathbf{E}\llbracket\text{variable}\ ex_{ij}\rrbracket\pi\ p \quad = \pi_e[j]$

$\mathbf{E}\llbracket\text{variable}\ rx_{ij}\rrbracket\pi\ p \quad = \pi_r[j]$

$\mathbf{E}\llbracket(\textbf{care}\ \text{exp})\rrbracket\pi\ p \quad = (\textbf{quote}\ t_1)$ <u>where</u> $(\textbf{quote}\ (t_1\ .\ t_2)) = \mathbf{E}\llbracket\text{exp}\rrbracket\pi\ p$

and similarly for **cdre, atome, conse, equale.**

$\mathbf{E}\llbracket(\textbf{ife}\ e_1\ e_2\ e_3)\rrbracket\pi\ p \quad =$ <u>if</u> $\mathbf{E}\llbracket e_1\rrbracket\pi\ p = (\textbf{quote}\ \text{nil})$ <u>then</u> $\mathbf{E}\llbracket e_3\rrbracket\pi\ p$ <u>else</u> $\mathbf{E}\llbracket e_2\rrbracket\pi\ p$

```
E[[(carr exp)]]π p  =
    let exp-r = E[[exp]]π p in
    case form of exp-r of
        (quote (t_1 . t_2))         : (quote t_1)
        (cons exp_1-r exp_2-r)      : exp_1-r
        otherwise                   : (car exp-r)
    end
```

and a similar reduction procedure for each of **cdrr, atomr, consr, equalr.**

```
E[[(ifr e_1 e_2 e_3)]]π p  =
    let exp-r = E[[e_1]]π p in
    case form of exp-r of
        (quote nil)                 : E[[e_3]]π p
        (quote (t_1 . t_2))         : E[[e_2]]π p
        (cons exp_1-r exp_2-r)      : E[[e_2]]π p
        otherwise                   : (if exp-r E[[e_2]]π p E[[e_3]]π p)
    end
```

```
E[[(call f_i (e_1 ... e_k[i1]) (r_1 ... r_k[i2]) )]]π p =
    let   ev_j = E[[e_j]]π p  for j=1,...,k_i1  and  rv_j = E[[r_j]]π p  for j=1,...,k_i2   in
    let   p contain ... (f_i (ex_i1 ... ex_ik[i1]) (rx_i1 ... rx_ik[i2]) exp_i^ann) ...
    in
          E[[exp_i^ann]] (<ev_i1, ..., ev_ik[i1]>,<rv_i1, ..., rv_ik[i2]>) p
```

```
E[[(callr f_i (e_1 ... e_k[i1]) (r_1 ... r_k[i2]) )]]π p =
    let   ev_j = E[[e_j]]π p  for j=1,...,k_i1  and  rv_j = E[[r_j]]π p  for j=1,...,k_i2   in
    let   p contain ... (f_i (ex_i1 ... ex_ik[i1]) (rx_i1 ... rx_ik[i2]) exp_i^ann) ...
    in
          (call f_i[<ev_i1, ..., ev_ik[i1]>]  rv_i1 ... rv_ik[i2])
```

Function $\mathbf{P} : \text{Expression} \rightarrow \Pi_e \rightarrow \text{Program} \rightarrow \text{Pend}$

This function computes the set of residual functions needed by (the residual variant of) the given expression.

$\mathbf{P}[\![(\mathbf{quote}\ \text{list})]\!]\pi_e\, p \quad = \varnothing$

$\mathbf{P}[\![\text{variable } ex_{ij}]\!]\pi_e\, p \quad = \varnothing$

$\mathbf{P}[\![\text{variable } rx_{ij}]\!]\pi_e\, p \quad = \varnothing$

$\mathbf{P}[\![(\mathbf{care}\ \text{exp})]\!]\pi_e\, p \quad = \varnothing$

and similarly for **cdre, atome, conse, equale.**

$\mathbf{P}[\![(\mathbf{ife}\ e_1\ e_2\ e_3)]\!]\pi_e\, p \ =$

$\quad \underline{if}\ \mathbf{E}[\![e_1]\!]\ (\pi_e, \langle\rangle)\ p = (\mathbf{quote\ nil})\ \underline{then}\ \mathbf{P}[\![e_3]\!]\pi_e\, p\ \underline{else}\ \mathbf{P}[\![e_2]\!]\pi_e\, p$

$\mathbf{P}[\![(\mathbf{carr}\ \text{exp})]\!]\pi_e\, p \quad = \mathbf{P}[\![\text{exp}]\!]\pi_e\, p$

and similarly for **cdrr, atomr.**

$\mathbf{P}[\![(\mathbf{consr}\ e_1\ e_2)]\!]\pi_e\, p \quad = \mathbf{P}[\![e_1]\!]\pi_e\, p \cup \mathbf{P}[\![e_2]\!]\pi_e\, p$

and similarly for **equal**

$\mathbf{P}[\![(\mathbf{ifr}\ e_1\ e_2\ e_3)]\!]\pi_e\, p \quad = \mathbf{P}[\![e_1]\!]\pi_e\, p \cup \mathbf{P}[\![e_2]\!]\pi_e\, p \cup \mathbf{P}[\![e_3]\!]\pi_e\, p$

$\mathbf{P}[\![(\mathbf{call}\ f_i\ \ (e_1 \dots e_{k[i1]})\ (r_1 \dots r_{k[i2]})\ )]\!]\pi_e\, p =$

$\quad \underline{let} \quad ev_j = \mathbf{E}[\![e_j]\!]\ (\pi_e, \langle\rangle)\ p \quad \text{for } j=1,\dots,k_{i1}\ \ \underline{in}$

$\quad \underline{let} \quad p \text{ contain } \dots\ (f_i\ (ex_{i1} \dots ex_{ik[i1]})\ (rx_{i1} \dots rx_{ik[i2]})\ exp_i^{ann})\ \dots$

$\quad \underline{in}$

$\qquad \mathbf{P}[\![exp_i^{ann}]\!]\ \langle ev_{i1}, \dots, ev_{ik[i1]}\rangle\ p \cup (\cup\ \ \mathbf{P}[\![r_j]\!]\pi_e\, p \quad \text{for } j=1,\dots,k_{i2})$

$\mathbf{P}[\![(\mathbf{callr}\ f_i\ \ (e_1 \dots e_{k[i1]})\ (r_1 \dots r_{k[i2]})\ )]\!]\pi_e\, p =$

$\quad \underline{let} \quad ev_j = \mathbf{E}[\![e_j]\!]\ (\pi_e, \langle\rangle)\ p \quad \text{for } j=1,\dots,k_{i1}\ \ \underline{in}$

$\quad \underline{let} \quad p \text{ contain } \dots\ (f_i\ (ex_{i1} \dots ex_{ik[i1]})\ (rx_{i1} \dots rx_{ik[i2]})\ exp_i^{ann})\ \dots$

$\quad \underline{in}$

$\qquad \{\ (f_i, \langle ev_{i1} \dots ev_{ik[i1]}\rangle)\ \}\ \cup (\cup\ \ \mathbf{P}[\![r_j]\!]\pi_e\, p \quad \text{for } j=1,\dots,k_{i2})$

3.3.4 Postprocessing

In this section, some postprocessing of the residual program produced in the function specialization phase above is described.

The residual program produced by Mix in the function specialization phase can neither be read by humans nor executed by machines unless it is subjected to some postprocessing. The composite residual function names produced have to be replaced by suitable atomic names as a prerequisite for being able to run the residual program, and this also makes it possible to read the residual program produced. (The compiler generator produced by running L Mix<Mix,Mix> contains the entire program for the function specialization phase of Mix as part of almost all the residual function names and therefore shrinks by a factor 100 when these are replaced by atoms). Also, folding **(car (cdr (cdr** x))) into **(caddr** x), folding nested **if**'s into **if-then-elsf-else** and folding **(cons** x_1 **(cons** x_2 ... 'nil) ...) into **(list** x_1 x_2 ...) improves readability of the residual programs a lot. Since it is most interesting to study the residual programs, especially the compilers produced, we have implemented these transformations as a separate postprocessor phase.

3.4 Variable Splitting

In this section we describe an extension to Mix allowing the generation of better residual programs.

A Problem with Generality

As can be inferred from subsection 3.3.3 on function specialization, any residual variant of a function f_i has at most the same number of parameters as f_i, since the parameters of the variant are the residual parameters of f_i, i.e. a subset of f_i's parameters. This can sometimes be unfortunate.

Consider an S-interpreter int for a functional language like the one given for L in Section 3.1. This interpreter contains a parameter (say, "vnames") holding parameter *names* for a function in the source program of this interpreter, and another (say, "vvalues") holding *values* for these parameters.

When partially evaluating int with respect to some S source program, "vnames" is known and disappears during partial evaluation, whereas "vvalues" is unknown and is found in the target program. In the target program, this *one* variable holds the values of *all* the parameters in the source program's function's parameter list. This results in much packing and unpacking of values when the target program is run and is quite wasteful.

In the interpreter, this generality is necessary: We *have* to represent the parameter values as a list of values packed into one variable, since we do not know in advance the length of the parameter list in the S source program to be interpreted. But in the target program, this length is known and fixed, and thus the list could be replaced by a number of variables each corresponding to one parameter from the S source program (or by an array, if our language allowed this). That the problem is not contrived, is indicated by the fact that the compiler generator cocom generated by an earlier version of our Mix spent approximately 75% of its run time doing garbage collection.

A Solution: Variable Splitting

We would like that for a function of a specific S source program s for which the parameter names are vnames = (z_1 z_2 ... z_k), there should be k variables representing the source program's k parameters in the target program produced. To obtain this, we have extended the function specialization phase of Mix and introduced a new kind of annotation. Using the annotations one can express, for example, that the value of residual parameter "vvalues" will always be a list of the same form as the value of eliminable parameter "vnames". Then in the residual (target) program, the simple variable "vvalues" is replaced by as many variables as there are elements in "vnames". In the above case, where vnames = (z_1 z_2 ... z_k) at compile time, the target program will contain k variables called "z_1", "z_2", ..., "z_k" instead of the single residual variable "vvalues".

This improvement of Mix works well in practice, generating more efficient and more readable residual programs.

4 Experience with using Mix

First we describe the way in which we apply Mix to itself to generate compilers and a compiler generator, and we then describe the resulting structure of these programs and other experiments with Mix.

4.1 Self-Application of Mix

When partially evaluating an S-interpreter int with respect to an S source program s, we proceed as follows.

1. Make call annotations for int.
2. Annotate int with respect to parameters and operators (by using the first and second phases of Mix), describing its first parameter (the S source program) as known, its second (the input to the S source program) as unknown, obtaining int^{ann}.
3. Produce the target (residual) program by applying the function specialization phase (here called Mix3) to int^{ann} and some S source program s,

   target := L Mix3<int^{ann},s>
4. Postprocess this to get a runnable target program.

Now, since only Step 3 above requires the S source program s, in self-application of Mix we need only apply Mix to Mix3, the function specialization phase. Mix self-application, therefore, can be sketched thus, analogously to the above:

1. Call-annotate Mix3.
2. Annotate Mix3: First parameter (the subject program) known, the second parameter (the subject program's known input) unknown, obtaining Mix3^{ann}.
3. Produce a compiler by applying Mix3 to itself with the interpreter as know input

   comp := L Mix3<Mix3^{ann},int^{ann}>.
4. Postprocess comp to get a runnable compiler.

Notice that the interpreter still has to be annotated.

4.2 Compilers Generated by Self-Application of Mix

Structure of the Compilers

As can be seen from the above, a compiler generated by self-application of Mix is a residual program for Mix3, and it may therefore inherit some of Mix3's structure and components.

In general the characteristics of a Mix-generated compiler are these.

a. Its main recursion structure is that of Mix3 for generating a set of residual functions.

b. It contains the reduction procedures (for residual operators **carr**, **cdrr**, ...), working as optimizing code generation functions as well as auxiliary functions, which are all inherited from Mix3.

c. It contains a number of compiling functions (and auxiliary functions) obtained by transforming interpreting functions (and auxiliary functions) from the interpreter int.

All in all, a Mix generated compiler usually has a reasonable structure. This structure resembles that of a recursive descent compiler, except that Mix carries out constant folding and some symbolic reduction while constructing the target program, not in a separate pass.

Size and Efficiency

The size (in lines) of a compiler seems to be a constant plus something dependent on the complexity of the interpreter it was generated from. The constant part is because of the machinery inherited from Mix3, whereas the rest depends highly on reasonable call annotations in the interpreter. It may therefore require some experimentation to get a compiler of a reasonable size.

Below we give program sizes and run times for some experiments.

Size (LETLISP versions of target, int, comp, Mix3 and cocom, not counting comments)

program	# functions	# lines
source	–	30
target	13	46
interpreter int	9	105
compiler comp	29	381
Mix3 (function specialization)	34	591
cocom	86	1736

Run Times (in seconds, VAX/785)

doing	run time	+ garbage coll.	total	speed-up
res = L int<src, data>	5.50	+ 1.16	6.66	
res = L target<data>	0.34	+ 0.64	0.98	6.8
tar = L Mix3<int,src>	3.18	+ 0.00	3.18	
tar = L comp<src>	0.16	+ 0.00	0.16	19.9
comp = L Mix3<Mix3,int>	63.56	+ 4.54	68.10	
comp = L cocom<int>	2.08	+ 2.18	4.26	16.0
cocom = L Mix3<Mix3,Mix3>	455.94	+ 22.64	478.58	
cocom = L cocom<Mix3>	14.48	+ 12.40	26.88	17.8

The figures only account for the time spent on function specialization (Mix3), which is 90 percent of Mix's run time, and not for the known/unknown abstract interpretation or annotation. The figures are for the variable splitting version of Mix.

A typical interpreter int (resembling a direct semantics) for a very small imperative language MP with a list data type, comprising 105 lines, gave a compiler of 381 lines (pretty-printed LETLISP text).

As a more complex example of a compiler, we may take the compiler generator cocom (which is a compiler for a "meta-compiling language" with the syntax of annotated L programs and a weird semantics (Jones, Tofte 83), produced from the "interpreter" Mix3). Whereas Mix3 comprises 591 lines, cocom is 1736 lines.

This indicates that the compilers are of a usable size (in fact, not much larger than equivalent hand-written compilers would be), although they may contain code that is obviously superfluous.

The compilers are also quite efficient. For the small imperative language MP mentioned above and a 30 line MP source program "source" to compute integer exponentiation, compile time plus target program run time is almost 6 times smaller than interpreted source program run time! This, by the way, should prove that our partial evaluator is not a trivial one.

4.3 Partially evaluating a Self-Interpreter

Another interesting experiment is partial evaluation of a (self-) interpreter for L written in L. Call such a program "sint" for self-interpreter. It has the property

$$\text{L sint} <p,d_1, ..., d_n> \;=\; \text{L p} <d_1, ..., d_n>$$

for any L-program p and input $<d_1, ..., d_n>$ in D*. Now by equation (1), for any L-program p and input $<d_1, ..., d_n>$,

$$\text{L (L Mix} <\text{sint},p>) <d_1, ..., d_n> \;=\; \text{L sint} <p,d_1, ..., d_n> \;=\; \text{L p} <d_1, ..., d_n>$$

so L Mix <sint,p> is an L-program equivalent to p. Furthermore, with

$$\text{transf} = \text{L Mix} <\text{Mix},\text{sint}>$$

the program "transf" is an equivalence preserving L program transformer, i.e.

$$\text{L(L transf} <p>) <d_1, ..., d_n> \;=\; \text{L p} <d_1, ..., d_n> .$$

Since the transformed program L transf <p> = L Mix<sint,p> will have some of the properties of the self-interpreter, we may obtain different kinds of transformations. For the "natural" self-interpreter given in Section 3.1, the transformed program produced is not only semantically equivalent to the original program, but also textually equivalent (modulo renaming of functions). Although this might not seem interesting, it establishes another kind of non-triviality of our partial evaluator, since, as can be readily seen, the most trivial partial evaluator (cf. Section 2.3) would not be able to reproduce a program verbatim by partial evaluation of a self-interpreter.

4.4 Conclusion

Other experiments with Mix concern parser generation and parser generator generation from a general parsing algorithm (taking as inputs a grammar and a subject string to be parsed). A series of such experiments is rather completely documented in (Dybkjær 85), reporting on successes, problems and pitfalls in applying a version of Mix to this. Although reasonable parser generators etc. could be generated, this required some experimentation and a certain programming style. This indicates that partial evaluation in general may prove an important program transformation technique, that Mix implements fairly powerful transformations by simple means, and finally, that much work has to be done before Mix can be considered a practically useful tool.

5 Summary

We have described an experimental, self-applicable partial evaluator Mix capable of generating compilers and a compiler generator of reasonable size and efficiency. To our knowledge this is not done before. The partial evaluator has a multiphase structure which seems to be a prerequisite for successful self-application and which has not been used for partial evaluators before. This structure and the algorithms of Mix have been described in much detail.

One of the main deficiencies of our partial evaluator is that the decision whether to unfold or suspend a function call is not automated. We require the user of the partial evaluator to make this decision in advance. Also, the partial evaluator is not a powerful general purpose tool: The goal of the project was to construct a self-applicable partial evaluator, and here modesty seems essential.

Future Work

Much work remains to be done before compilers and compiler generators produced by partial evaluation can be used in practice. Partial evaluation of imperative languages requires more sophisticated techniques than the ones described here and deserves investigation.

The most promising next step (in a practical direction) probably would be to build a more powerful partial evaluator along the lines drawn here for some other language having the same characteristics, e.g. a Prolog subset or a higher order functional language.

Also, there is a pressing need for a more well-founded "theory of partial evaluation". For example, it might be possible to prove (or disprove) that the *static* classification of variables described in this paper is essential for self-application of a partial evaluator.

Acknowledgement

All of this is joint work with Neil D. Jones and Harald Søndergaard (at DIKU). I would like to thank them for a most fruitful collaboration without which this paper would not have been.

References

(Bulyonkov 84)
Bulyonkov, M. A. Polyvariant mixed computation for analyzer programs. *Acta Informatica 21*, (1984), pp. 473-484.

(Dybkjær 85)
Dybkjær, Hans. Parsers and partial evaluation: An experiment. DIKU Student Report 85-7-15 (July 1985). 128 pp.

(Emanuelson, Haraldsson 80)
Emanuelson, Pär & Anders Haraldsson. On compiling embedded languages in LISP. In *Conf. Rec. of the 1980 LISP Conference, Stanford, California*, pp. 208-215.

(Ershov 78)
Ershov, Andrei P. On the essence of compilation. In Neuhold, E. J. (ed.): *Formal Description of Programming Concepts*, North-Holland, 1978, pp. 391-420.

(Ershov 82)
Ershov, Andrei P. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science 18* (1982), pp. 41-67.

(Futamura 71)
Futamura, Yoshihiko. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls* 2, no. 5 (1971), pp. 45-50.

(Futamura 83)
Futamura, Yoshihiko. Partial computation of programs. *Proc. RIMS Symp. Software Science and Engineering, Kyoto, Japan, 1982. Springer LNCS 147* (1983), pp. 1-35.

(Haraldsson 78)
Haraldsson, Anders. A partial evaluator and its use for compiling iterative statements in LISP. In *Conf. Rec. of the 5th ACM POPL, Tucson, Arizona, 1978*, pp.195-203.

(Heering 85)
Heering, Jan. Partial evaluation and ω-completeness of algebraic specifications. Report CS-8501, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands.

(Jones, Mycroft 86)
Jones, Neil D. & Alan Mycroft. Data flow analysis using minimal function graphs. In *Conf. Rec. of the 13th ACM POPL, St. Petersburg, Florida, 1986.* (To appear).

(Jones, Sestoft, Søndergaard 85)
Jones, Neil D., Peter Sestoft & Harald Søndergaard. 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. (A preliminary version appeared as DIKU Report 85/1, January 1985).

(Jones, Tofte 83)
Jones, Neil D. & Mads Tofte. Some principles and notations for the construction of compiler generators. Unpublished working paper, DIKU, July 29, 1983. 15 pp.

(Kahn, Carlsson 84)
Kahn, Kenneth M. & Mats Carlsson. The compilation of Prolog programs without the use of a Prolog compiler. In *Proc. of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, 1984*, ICOT, 1984, pp. 348-355.

(Kleene 52)
Kleene, S. C. *Introduction to Metamathematics.* Van Nostrand, 1952.

(Sestoft 85)
[Sestoft, Peter]. The Mix system: User manual and short description. DIKU, April 26th, 1985. 14 pp.

(Turchin 79)
Turchin, Valentin F. A supercompiler system based on the language REFAL. *SIGPLAN Notices 14*, no. 2 (February 1979), pp. 46-54.

(Turchin 80)
Turchin, Valentin F. Semantic definitions in REFAL and the automatic construction of compilers. In Jones, Neil D. (ed.): *Semantics Directed Compiler Generation. Springer LNCS 94*, (1980), pp. 441-474.

Appendix: Some Listings

This appendix contains a number of listings showing what kind of programs the Mix system may produce. We use the interpreter for the simple imperative language MP mentioned in Section 4.2 as an example together with some related programs, namely,

1. The interpreter for MP, called int.
2. A compiler for the MP language generated from this interpreter.
3. A source program in the MP language, namely, the exponentiation program mentioned in Section 4.2.
4. A target program (in LETLISP) for this source program.

Comments on the Listings:

1. The interpreter is given in LETL, and it is *not* annotated for variable splitting (Section 3.4). It should be quite straightforwardly understandable, only the environment (that associates values with variables) is split into two: A name list "vn" and a value list called "vv" or "vv0".

2. The compiler from MP to L was generated by running

comp = L Mix3<Mix3$^{\text{ann}}$,int$^{\text{ann}}$>

or comp = L cocom<int$^{\text{ann}}$>

as demonstrated in Section 2.1.

The functions constituting the compiler can be classified as follows:

Main recursion structure (inherited from Mix3): MP-int-1, Geteqn-1, Mix1-1, Lookupout-1.

Compiling functions (from the interpreter):

Code generating: Exp-1, While-1, Block-1, Cmd-1, Initvars2-1, MP-int-2, Update-1.

Controlling target program generation: While-2, Block-2, Cmd-2, MP-int-3.

Optimizing code building functions (from Mix3): Carrl-1, Cdrrl-1, Atomrl-1, Consrl-1, Equalrl-1, Ifrl-1.

Auxiliary functions: U-e-1, Lookupvar-1.

Trivial or superfluous functions (ashaming!): Initvars2-2, Initvars1-2, Initvars1-1, Update-2, Lookupvar-2, Exp-2.

3. The MP source program computes exponentiation x^y as the number of tuples of length y over a set with x members. It is, admittedly, *not* very readable.

4. The corresponding target program could be produced either as

target = L Mix3<int$^{\text{ann}}$,source>

or target = L comp<source>

as described in Section 2.1 and Section 4.2.

The listings are commented to a certain extent, especially the compiler generated from int.

As can be seen, the compiler could easily be improved by quite simple means (by identifying functions that only may return one of their parameters, e.g. Lookupvar-2 in the compiler). The target programs generated by the compiler, on the other hand, could not conceivably be more compact or efficient granted the rather primitive methods and the primitive target language we use.

Interpreter for MP (1st of 3 parts)

```
; MP-int - an interpreter for a simple
; imperative programming language with lists as data type.
; 1985 April 26

; Syntax of input programs:
; ⟨program⟩ ::= (program ⟨pars⟩ ⟨vars⟩ ⟨block⟩)
; ⟨pars⟩    ::= (pars ⟨vname⟩*)
; ⟨vars⟩    ::= (dec ⟨vname⟩*)
; ⟨block⟩   ::= (⟨cmd⟩*)
; ⟨cmd⟩     ::= (:= ⟨vname⟩ ⟨exp⟩)
;               !  (if ⟨exp⟩ ⟨block⟩ ⟨block⟩)     -- first branch
;                                                  iff ⟨exp⟩ not nil
;               !  (while ⟨exp⟩ ⟨block⟩)          -- iterate while
;                                                  ⟨exp⟩ not nil
; ⟨exp⟩     ::= (quote ⟨list⟩)                    -- constant
;               !  ⟨vname⟩
;               !  (car ⟨exp⟩)
;               !  (cdr ⟨exp⟩)
;               !  (cons ⟨exp⟩ ⟨exp⟩)
;               !  (atom ⟨exp⟩)                    -- nil iff not atom
;               !  (equal ⟨exp⟩ ⟨exp⟩)             -- nil iff unequal

; Semantics: Programs are given a fixed number of input values,
; which are bound to the variables named in the (pars ...) list.
; The other variables are initially all nil.
; The semantics resembles that of Pascal, with
; the exceptions that 1) the result is the entire store,
; 2) the if command takes its first branch if the expression
; is non-nil, 3) the while loops as long as the expression is
; non-nil.

; Main data structures in the interpreter:
; The variables program, block, cmd, exp, vars, and pars
; take values which are program fragments conforming to
; the syntax ⟨program⟩, ⟨block⟩, etc above.
; The variable vn is a list of variable names.
; The variables vv, vv0 are lists of variable values (states).

; Main functions in the interpreter:
; "MP-int" interprets entire MP programs, "Block", "Cmd", and
; "While" interpret blocks, commands and while statements and
; return the result new state.  "Exp" interprets expressions
; and return the value of an expression.


((MP-int (program input) =

; Main function: "program" is the program to be interpreted,
;                "input"   is input to that program.
; Output is a list of the final values of the interpreted program's variables.

 (let  (program? pars vars block) = program     in
 (let  (pars? . parlist)          = pars
       (dec?  . varlist)          = vars        in
 (let  vn      = (call Initvars1 varlist parlist)
       vv      = (call Initvars2 varlist input)      in
         (call Block  block vn vv)
))))

```

Interpreter for MP (2nd of 3 parts)

```

(Block (block vn vv0) =

; Interpretation of a sequence of statements in environment (vn,vv0).

 (let   (cmd1 . blockrest)      =  block        in
 (if  block  then
        (call Block blockrest vn (callr Cmd cmd1 vn vv0))
  else
        vv0
)))


(Cmd (cmd vn vv0 ) =

 (let (op e1 e2 e3)      =  cmd  in
 (if  (equal ':= op)     then
        (call Update vn vv0 e1 (call Exp e2 vn vv0))
  elsf (equal 'if op)    then
        (if (call Exp e1 vn vv0)  then
                (call Block e2 vn vv0)
         else
                (call Block e3 vn vv0)
        )
  elsf (equal 'while op)  then
        (callr While e1 e2 vn vv0)
  else
        (list 'ILLEGAL 'COMMAND: cmd)
)))


(While (condit block vn vv0 ) =

 (if  (call Exp condit vn vv0)  then
        (callr While condit block vn (call Block block vn vv0))
  else
        vv0
))


(Exp (exp vn vv0 ) =

 (let  (op e1 e2) =  exp         in
 (if  (atom exp)     then
        (call Lookupvar vn vv0 exp)
  elsf (equal 'quote op)         then
        e1
  elsf (equal 'car   op)         then
        (car (call Exp e1 vn vv0))
  elsf (equal 'cdr   op)         then
        (cdr (call Exp e1 vn vv0))
  elsf (equal 'cons  op)         then
        (cons (call Exp e1 vn vv0)
              (call Exp e2 vn vv0))
  elsf (equal 'atom  op)         then
        (atom (call Exp e1 vn vv0))
  elsf (equal 'equal op)         then
        (equal (call Exp e1 vn vv0)
               (call Exp e2 vn vv0)
        )
  else
        (list 'ILLEGAL 'EXPRESSION: exp)
)))

```

Interpreter for MP (3rd of 3 parts)

```

(Initvars1 (vars pars) =
; Make a list of names of declared variables and parameters.
 (let (v1  . restvars)   =  vars  in
 (if vars then
        (v1 :: (call Initvars1 restvars pars))
  else
        pars
)))


(Initvars2 (vars input) =
; Make a list of values of declared variables (which are initialized to
; nil) and parameters (which get their values from input).
 (let (v1 . restvars)    =  vars in
 (if vars then
        ('nil :: (call Initvars2 restvars input))
  else
        input
)))


(Update (vn vv var val) =
 (let  (vn1 . vnrest)   =  vn
       (vv1 . vvrest)   =  vv  in
 (if (equal 'nil vn)    then
        (list 'UNKNOWN 'VARIABLE: var)
  elsf (equal vn1 var)  then
        (val :: vvrest)
  else
        (vv1 :: (call Update vnrest vvrest var val))
)))


(Lookupvar (vn vv var) =
 (let (vn1 . vnrest)    =  vn
      (vv1 . vvrest)    =  vv   in
 (if (equal 'nil vn)    then
        (list 'UNKNOWN 'VARIABLE: var)
  elsf (equal vn1 var)  then
        vv1
  else
        (call Lookupvar vnrest vvrest var)
))))
```

A compiler from MP to L (1st of 6 parts)

```
(deflet 'MP-int-1
        '(program)
        '(Geteqn-1 'nil (list 'MP-int program) 'nil))
```

```
(deflet 'Geteqn-1
        '(out fname1 pending)
        '(if (equal (car fname1) 'MP-int)
             then
             (Mix1-1 (cons (list (cons (car fname1) fname1)
                                 '(input)
                                 (MP-int-2 (cadr fname1) 'input))
                           out)
                     (MP-int-3 (cadr fname1) pending))
             elsf
             (equal (car fname1) 'Cmd)
             then
             (Mix1-1 (cons (list (cons (car fname1) fname1)
                                 '(vv0)
                                 (Cmd-1 (cadr fname1)
                                        (caddr fname1)
                                        'vv0))
                           out)
                     (Cmd-2 (cadr fname1) (caddr fname1) pending))
             elsf
             (equal (car fname1) 'While)
             then
             (Mix1-1 (cons (list (cons (car fname1) fname1)
                                 '(vv0)
                                 (While-1 (cadr fname1)
                                          (caddr fname1)
                                          (cadddr fname1)
                                          'vv0))
                           out)
                     (While-2 (cadr fname1)
                              (caddr fname1)
                              (cadddr fname1)
                              pending))
             else
             (list 'UNDEFINED 'FUNCTION: (car fname1))))
```

Determines which target functions are necessary.

```
(deflet 'Lookupvar-1
        '(vn var vv)
        '(if (equal 'nil vn)
             then
             (list 'quote (list 'UNKNOWN 'VARIABLE: var))
             elsf
             (equal (car vn) var)
             then
             (Carr1-1 vv)
             else
             (Lookupvar-1 (cdr vn) var (Cdrr1-1 vv))))
```

Compiles variable reference (R-value)

```
(deflet 'Cdrr1-1
        '(uofe1)
        '(if (atom uofe1)
             then
             (list 'cdr uofe1)
             elsf
             (equal (car uofe1) 'quote)
             then
             (list 'quote (cdadr uofe1))
             elsf
             (equal (car uofe1) 'cons)
             then
             (caddr uofe1)
             else
             (list 'cdr uofe1)))
```

Produces a cdr expression (possibly reduced)

A compiler from MP to L (2nd of 6 parts)

```
(deflet 'Carr1-1
        '(uofel)
        '(if (atom uofel)
             then
             (list 'car uofel)
             elsf
             (equal (car uofel) 'quote)
             then
             (list 'quote (caadr uofel))
             elsf
             (equal (car uofel) 'cons)
             then
             (cadr uofel)
             else
             (list 'car uofel)))
```

Produces a car expression (possibly reduced)

```
~~(deflet 'Lookupvar-2
        '(vn var pending)
        '(if (equal 'nil vn)
             then
             pending
             elsf
             (equal (car vn) var)
             then
             pending
             else
             (Lookupvar-2 (cdr vn) var pending)))~~
```

Superfluous function

```
(deflet 'Mix1-1
        '(out pending)
        '(if pending
             then
             (if (Lookupout-1 out (car pending))
                 then
                 (Mix1-1 out (cdr pending))
                 else
                 (Geteqn-1 out (car pending) (cdr pending)))
             else
             out))
```

This and the following function control building the target program together with Geteqn-1.

```
(deflet 'Lookupout-1
        '(out fname)
        '(if out
             then
             (if (equal fname (cdaar out))
                 then
                 (car out)
                 else
                 (Lookupout-1 (cdr out) fname))
             else
             'nil))
```

```
(deflet 'Update-1
        '(vn var vv val)
        '(if (equal 'nil vn)
             then
             (list 'quote (list 'UNKNOWN 'VARIABLE: var))
             elsf
             (equal (car vn) var)
             then
             (Consr1-1 val (Cdrr1-1 vv))
             else
             (Consr1-1 (Carr1-1 vv) (Update-1 (cdr vn) var (Cdrr1-1 vv) val))))
```

Produces code for a store update.

```
(deflet 'Consr1-1
        '(uofe1 uofe2)
        '(if (atom uofe1)
             then
             (list 'cons uofe1 uofe2)
             elsf
             (atom uofe2)
             then
             (list 'cons uofe1 uofe2)
             elsf
             (equal '(quote . quote) (cons (car uofe1) (car uofe2)))
             then
             (list 'quote (cons (cadr uofe1) (cadr uofe2)))
             else
             (list 'cons uofe1 uofe2)))
```

Produces a cons expression (possibly reduced).

A compiler from MP to L (3rd of 6 parts)

```
~~(deflet 'Update-2~~
~~        '(vn var pending)~~
~~        '(if (equal 'nil vn)~~
~~             then~~
~~             pending~~
~~             elsf~~
~~             (equal (car vn) var)~~
~~             then~~
~~             pending~~
~~             else~~
~~             (Update-2 (cdr vn) var pending)))~~
```

Superfluous function (returns the parameter "pending").

```
(deflet 'Initvars2-1
        '(vars input)
        '(if vars
             then
             (Consr1-1 ''nil (Initvars2-1 (cdr vars) input))
             else
             input))
```

Generate code to initialize variables (to nil).

```
~~(deflet 'Initvars2-2~~
~~        '(vars pending)~~
~~        '(if vars then (Initvars2-2 (cdr vars) pending) else pending))~~
```

Superfluous function.

```
~~(deflet 'Initvars1-1~~
~~        '(vars pars)~~
~~        '(if vars~~
~~             then~~
~~             (list 'quote~~
~~                   (cons (car vars) (U-e-1 (list (cdr vars) pars))))~~
~~             else~~
~~             (list 'quote pars)))~~
```

Superfluous function.

```
(deflet 'U-e-1
        '(evv)
        '(if (car evv)
             then
             (cons (caar evv) (U-e-1 (list (cdar evv) (cadr evv))))
             else
             (cadr evv)))
```

Build compile time variable name list.

```
~~(deflet 'Initvars1-2 '(vars pars pending) 'pending)~~
```

Superfluous.

```
(deflet 'Exp-1
        '(exp vn vv0)
        '(if (atom exp)
             then
             (Lookupvar-1 vn exp vv0)
             elsf
             (equal 'quote (car exp))
             then
             (list 'quote (cadr exp))
             elsf
             (equal 'car (car exp))
             then
             (Carr1-1 (Exp-1 (cadr exp) vn vv0))
             elsf
             (equal 'cdr (car exp))
             then
             (Cdrr1-1 (Exp-1 (cadr exp) vn vv0))
             elsf
             (equal 'cons (car exp))
             then
             (Consr1-1 (Exp-1 (cadr exp) vn vv0) (Exp-1 (caddr exp) vn vv0))
             elsf
             (equal 'atom (car exp))
             then
             (Atomr1-1 (Exp-1 (cadr exp) vn vv0))
             elsf
             (equal 'equal (car exp))
             then
             (Equalr1-1 (Exp-1 (cadr exp) vn vv0) (Exp-1 (caddr exp) vn vv0))
             else
             (list 'quote
                   (list 'ILLEGAL 'EXPRESSION: exp))))
```

Compile an expression in environment (vn, vv0)

A compiler from MP to L (4th of 6 parts)

```
(deflet 'Equalr1-1
        '(uofe1 uofe2)
        '(if (atom uofe1)
             then
             (list 'equal uofe1 uofe2)
             elsf
             (atom uofe2)
             then
             (list 'equal uofe1 uofe2)
             elsf
             (equal 'quote (car uofe1))
             then
             (if (equal 'quote (car uofe2))
                 then
                 (list 'quote (equal (cadr uofe1) (cadr uofe2)))
                 else
                 (list 'equal uofe1 uofe2))
             else
             (list 'equal uofe1 uofe2)))
```

Produces an equal expression (possibly reduced).

```
(deflet 'Atomr1-1
        '(uofe1)
        '(if (atom uofe1)
             then
             (list 'atom uofe1)
             elsf
             (equal 'quote (car uofe1))
             then
             (list 'quote (atom (cadr uofe1)))
             elsf
             (equal 'cons (car uofe1))
             then
             ''nil
             else
             (list 'atom uofe1)))
```

Produces an atom expression (possibly reduced).

```
(deflet 'Exp-2
        '(exp vn pending)
        '(if (atom exp)
             then
             (Lookupvar-2 vn exp pending)
             elsf
             (equal 'quote (car exp))
             then
             pending
             elsf
             (equal 'car (car exp))
             then
             (Exp-2 (cadr exp) vn pending)
             elsf
             (equal 'cdr (car exp))
             then
             (Exp-2 (cadr exp) vn pending)
             elsf
             (equal 'cons (car exp))
             then
             (Exp-2 (cadr exp) vn (Exp-2 (caddr exp) vn pending))
             elsf
             (equal 'atom (car exp))
             then
             (Exp-2 (cadr exp) vn pending)
             elsf
             (equal 'equal (car exp))
             then
             (Exp-2 (cadr exp) vn (Exp-2 (caddr exp) vn pending))
             else
             pending))
```

Superfluous function (returns the parameter "pending")

```
(deflet 'Cmd-1
        '(cmd vn vv0)
        '(if (equal ':= (car cmd))
             then
             (Update-1 vn (cadr cmd) vv0 (Exp-1 (caddr cmd) vn vv0))
             elsf
             (equal 'if (car cmd))
             then
             (Ifr1-1 (Exp-1 (cadr cmd) vn vv0)
                     (Block-1 (caddr cmd) vn vv0)
                     (Block-1 (cadddr cmd) vn vv0))
             elsf
             (equal 'while (car cmd))
             then
             (list 'call
                   (list 'While
                         'While
                         (cadr cmd)
                         (caddr cmd)
                         vn)
                   vv0)
             else
             (list 'quote (list 'ILLEGAL 'COMMAND: cmd))))
```

Compile a command in environment (vn, vv0)

```
(deflet 'Cmd-2
        '(cmd vn pending)
        '(if (equal ':= (car cmd))
             then
             (Exp-2 (caddr cmd) vn (Update-2 vn (cadr cmd) pending))
             elsf
             (equal 'if (car cmd))
             then
             (Block-2 (cadddr cmd)
                      vn
                      (Block-2 (caddr cmd) vn (Exp-2 (cadr cmd) vn pending)))
             elsf
             (equal 'while (car cmd))
             then
             (cons (list 'While (cadr cmd) (caddr cmd) vn) pending)
             else
             pending))
```

Determine the target functions necessary for a command.

```
(deflet 'MP-int-2
        '(program input)
        '(Block-1 (cadddr program)
                  (U-e-1 (list (cdaddr program) (cdadr program)))
                  (Initvars2-1 (cdaddr program) input)))
```

First process declarations, then compile a block.

```
(deflet 'MP-int-3
        '(program pending)
        '(Initvars2-2 (cdaddr program)
                      (Block-2 (cadddr program)
                               (U-e-1
                                 (list (cdaddr program) (cdadr program)))
                               pending)))
```

Determine a part of the target functions necessary for the program.

A compiler from MP to L (6th of 6 parts)

```
(deflet 'Cmd-1
        '(cmd vn vv0)
        '(if (equal ':= (car cmd))
             then
             (Update-1 vn (cadr cmd) vv0 (Exp-1 (caddr cmd) vn vv0))
             elsf
             (equal 'if (car cmd))
             then
             (Ifrl-1 (Exp-1 (cadr cmd) vn vv0)
                     (Block-1 (caddr cmd) vn vv0)
                     (Block-1 (cadddr cmd) vn vv0))
             elsf
             (equal 'while (car cmd))
             then
             (list 'call
                   (list 'While
                         'While
                         (cadr cmd)
                         (caddr cmd)
                         vn)
                   vv0)
             else
             (list 'quote (list 'ILLEGAL 'COMMAND: cmd))))
```

Compile a command in environment (vn, vv0)

```
(deflet 'Cmd-2
        '(cmd vn pending)
        '(if (equal ':= (car cmd))
             then
             (Exp-2 (caddr cmd) vn (Update-2 vn (cadr cmd) pending))
             elsf
             (equal 'if (car cmd))
             then
             (Block-2 (cadddr cmd)
                      vn
                      (Block-2 (caddr cmd) vn (Exp-2 (cadr cmd) vn pending)))
             elsf
             (equal 'while (car cmd))
             then
             (cons (list 'While (cadr cmd) (caddr cmd) vn) pending)
             else
             pending))
```

Determine the target functions necessary for a command.

```
(deflet 'MP-int-2
        '(program input)
        '(Block-1 (cadddr program)
                  (U-e-1 (list (cdaddr program) (cdadr program)))
                  (Initvars2-1 (cdaddr program) input)))
```

First process declarations, then compile a block.

```
(deflet 'MP-int-3
        '(program pending)
        '(Initvars2-2 (cdaddr program)
                      (Block-2 (cadddr program)
                               (U-e-1
                                (list (cdaddr program) (cdadr program)))
                               pending)))
```

Determine a part of the target functions necessary for the program.

A source program in MP: Exponentiation

```
; An exponentiation program in MP.
;       The program computes  X  to the Y'th power as the number of
; tuples of length Y with elements from an X-element set.
;
; Notation:  We let #x denote the length of list x.
;
; Input:  Two lists x and y, the lengths of which are X = #x, Y = #y.
; Effect: The final value of variable out is a list all of Y-tuples
;         with elements from an X-element set, that is,
;           #out = X to the Y'th power = #x to the #y'th power.
; Output from the program is a list (a dump) of the variables' final
; values with out's value as its first element.
;


 (program (pars x y) (dec out next kn)
 ((:= kn y)
  (while kn
        ((:= next (cons x next))
         (:= kn   (cdr kn))
        )
  )
  (:= out (cons next out))              ; First combination
  ; Invariant: #next + #kn = #y
  (while next                           ; while more tuples
        ((if (cdr (car next))           ;  if next(1) can be increased
            ((:= next (cons (cdr (car next)) ;     do that
                            (cdr next)) )
             (while kn                  ; while #next < #y do
                ((:= next (cons x next));   put x in front of next
                 (:= kn   (cdr kn))     ;   preserving invariant
                )
             )
             (:= out (cons next out))
            )
         ;  else, backtrack, preserving invariant
            ((:= next (cdr next))
             (:= kn   (cons '1 kn))
            )
        ))
  )
 )
) ; end of program
```

A target program (in LETLISP) for the exponentiation program

```
(deflet 'MP-int-1
        '(input)        input is a list:  (x . (y . nil)) = input
        '(Cmd-6
          (Cmd-5
           (Cmd-2
            (Cmd-1
             (cons 'nil
                   (cons 'nil (cons 'nil input))))))))
(deflet 'Cmd-1
        '(vv0)
        '(cons (car vv0) (cons (cadr vv0) (cons (caddddr vv0) (cdddr vv0)))))
(deflet 'Cmd-2 '(vv0) '(While-1 vv0))
(deflet 'While-1
        '(vv0)
        '(if (caddr vv0) then (While-1 (Cmd-4 (Cmd-3 vv0))) else vv0))
(deflet 'Cmd-3
        '(vv0)
        '(cons (car vv0) (cons (cons (cadddr vv0) (cadr vv0)) (cddr vv0))))
(deflet 'Cmd-4
        '(vv0)
        '(cons (car vv0) (cons (cadr vv0) (cons (cdaddr vv0) (cdddr vv0)))))
(deflet 'Cmd-5
        '(vv0)
        '(cons (cons (cadr vv0) (car vv0)) (cdr vv0)))
(deflet 'Cmd-6 '(vv0) '(While-2 vv0))
(deflet 'While-2
        '(vv0)
        '(if (cadr vv0) then (While-2 (Cmd-7 vv0)) else vv0))
(deflet 'Cmd-7
        '(vv0)
        '(if (cdaadr vv0)
             then
             (Cmd-5 (Cmd-2 (Cmd-10 vv0)))
             else
             (Cmd-9 (Cmd-8 vv0))))
(deflet 'Cmd-8
        '(vv0)
        '(cons (car vv0) (cons (cdadr vv0) (cddr vv0))))
(deflet 'Cmd-9
        '(vv0)
        '(cons (car vv0)
               (cons (cadr vv0)
                     (cons (cons '1 (caddr vv0)) (cdddr vv0)))))
(deflet 'Cmd-10
        '(vv0)
        '(cons (car vv0) (cons (cons (cdaadr vv0) (cdadr vv0)) (cddr vv0))))
```

} environment not split

Fortegnelse over rapporter i 1983

83/1 Stepwise Development of Operational and Denotational Semantics for Prolog. Neil D. Jones and Alan Mycroft.

83/2 A Skeleton Interpreter for Specialized Languages. Jørgen Steensgaard-Madsen.

83/3 Naming Commands. An Analysis of Designers' Naming Behaviour. Anker Helms Jørgensen et al.

83/4 Gendannelse af forringede billeder ved invers - og Wienerfiltrering. Jørgen Bansler og Søren Olsen.

83/5 Stepwise Development of Logic Programmed Software Development Methods. Gregers Koch.

83/6 An Algorithm for the Steiner Problem in the Euclidean Plane. Pawel Winter.

83/7 Eksperimentelle teknikker i systemarbejdet. Jørgen Bansler og Keld Bødker.

83/8 Generering af en oversættergenerator. Mads Tofte.

83/9 Interval Arithmetic Implementations Using Floating Point Arithmetic. Michael Clemmesen.

83/10 Design practice and interface usability: evidence from interviews with designers. Anker Helms Jørgensen, N. Hammond, A. MacLean, P. Barnard, and J. Long.

83/11 En model for brugeres opfattelse af edb-baserede systemer. Jan Chr. Clausen.

83/12 Definition of the Programming Language MODEF. Jørgen Steensgaard-Madsen og Lars Møller Olsen.

83/13 The effect of task structure in interactive systems: a pilot experiment. Anker Helms Jørgensen, Phil Barnard, Nick Hammond, Allan MacLean.

83/14 The psychology of developing and using computer systems: five contributions. Anker Helms Jørgensen.

83/15 Systemudvikling som element i den kapitalistiske teknologiudvikling. Jørgen Bansler og Keld Bødker.

83/16 Oversætterteknik for programmeringssprog ved hjælp af PROLOG. Flemming Als, Carsten Hendriksen og Jens Johansen.

83/17 Generalized Steiner Problem in Outerplanar Networks. Pawel Winter.

Fortegnelse over rapporter i 1984

84/1 Production and Location on a Network under Demand Uncertainty. Francois Louveaux and Jacques-Francois Thisse.

84/2 Typed Representation of Objects by Functions. Jørgen Steensgaard-Madsen.

84/3 Steiner Problem in Halin Networks. Pawel Winter.

84/4 An Algorithm for the Enumeration of Spanning Trees. Pawel Winter.

84/5 Open Problems Presented at the Copenhagen Workshop on Computer Vision. Knud Henriksen, Peter Johansen, Søren Olsen.

84/6 Bufferingsmetoder. Bent Pedersen.

84/7 Datalogi 2 Notes: Functions, Expressions, Programming Languages, Computability. Neil D. Jones.

84/8 COMPILER GENERATORS - what they can do, what they might do, and what they will probably never do. Mads Tofte.

84/9 Forelæsningsnoter til administrativ databehandling. Ole Caprani, H.B. Hansen og Søren Lauesen.

84/10 Computer Vision in a Computer Science Framework. Peter Johansen and Edda Sveinsdottír.

84/11 Analyse af stereobilleder - rekonstruktion af tredimensionale flader. Søren Olsen.

85/12 Ingredients of Locational Analyses. Jakob Krarup & Peter Pruzan.

84/13 MODEF/1100 User's Guide. J. Steensgaard-Madsen.

84/14 A New Family of Exponential LP-problems. Jens Clausen.

84/15 Two Families of Bad LP-problems. Jens Clausen.

84/16 A Note on the Edmonds-Fukuda Pivoting Rule for Simplex. Jens Clausen.

84/17 Network Management. Brian E. Christiansen.

Fortegnelse over rapporter i 1985

85/1 An Experiment in Partial Evaluation: The Generation of a Compiler Generator. Neil D. Jones, Peter Sestoft, Harald SØndergaard.

85/2 FIFTH GENERATION PROGRAMMING Vol. 1. Logic Programming in Natural Language Analysis. Proceedings I from a workshop in Copenhagen December 1984. Edited by Gregers Koch.

85/3 A Survey of Systems Programming Languages: Concepts and Facilities. William F. Appelbe and Klaus Hansen.

85/4 Hjemmedatamaten - et brækjern til fremtiden? Leif Caspersen, Jacob NØrbjerg, Annelise Ravn, Thomas Stürup.

85/5 OSI modellens øvre lag. John Hunderup, Benny Pedersen, SØren Stockmarr, Kim Wagner, Michael Bundgaard, Kurt Pedersen, Carsten Bjernå, JØrgen Münster.

85/6 FIFTH GENERATION PROGRAMMING Vol. 2. Logic Programming in Natural Language Analysis. Proceedings II from a workshop in Copenhagen December 1984. Edited by Gregers Koch.

85/7 Inter-process Communication in Distributed Operating Systems. Erik Jul.

85/8

85/9

85/10 AfprØvning af systemudvikling med prototyper. Klaus Viby Mogensen.

85/11 The Structure of a Self-Applicable Partial Evaluator. Peter Sestoft.