A PARTIAL EVALUATOR AND POST-OPTIMIZER FOR A FLOW CHART LANGUAGE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

LEVENT ERKÖK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

JULY 1997

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Fatoş Yarman Vural
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Halit
Oğuztüzün
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Adnan Yazıcı _____

Assist. Prof. Dr. Cem Bozşahin _____

Assist. Prof. Dr. İlyas Çiçekli _____

Assist. Prof. Dr. Halit Oğuztüzün _____

Assist. Prof. Dr. Göktürk Üçoluk _____

ABSTRACT

A PARTIAL EVALUATOR AND POST-OPTIMIZER FOR A FLOW CHART LANGUAGE

Erkök, Levent

MSc., Department of Computer Engineering

Supervisor: Assist. Prof. Dr. Halit Oğuztüzün

July 1997, 137 pages

In this thesis, the concept of partial evaluation and post optimization techniques has been studied and a system, called ILPOS, implementing the ideas on a flow chart language is developed. The effects of specialization and optimizations, including useless code removal and program minimization are discussed. A method for handling incomplete specifications is discussed in the context of partial evaluation. The termination problem of partial evaluation is studied and the ILPOS solution to the problem is described.

Keywords: Partial Evaluation, Program Specialization and Optimization, Program Termination.

ÖZ

AKIŞ ŞEMASI TİPİ BİR DİL İÇİN BİR KISMİ HESAPLAYICI VE EN İYİLEYİCİ

Erkök, Levent

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Yrd. Doç. Dr. Halit Oğuztüzün

Temmuz 1997, 137 sayfa

Bu tez çalışmasında, kısmi hesaplama ve en iyileme teknikleri incelenmiş ve bu fikirler akış şeması tipi bir dil üzerinde uygulanarak ILPOS isimli bir sistem ile ortaya konmuştur. Program özelleştirme ve en iyileme teknikleri, gereksiz program parçacığı yok edilmesi ve program ufaltılması konuları tartışılmıştır. Eksik tanımlamaların durumu ve bunların kısmi hesaplama içersindeki durumları incelenmiş ve ILPOS tarafından kullanılan metod açıklanmıştır. Kısmi hesaplamada program sonlanması problemi ve çözümü incelenmiştir.

Anahtar Kelimeler: Kısmi Hesaplama, Program Özelleştirme ve En İyileme, Program Sonlanması.

ACKNOWLEDGMENTS

I would like to thank to my supervisor Mr. Halit Oğuztüzün for his guidance and help throughout the study. Many thanks go to my colleague Mr. M. Uğur Yılmaz for his help on every sort of problem that I have met. Special thanks go to Ms. Şengül Vurgun and to my family for their invaluable encouragement and support. I also wish to express my gratitude to other colleagues and friends for their help and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

ILPOS, an acronym for **I**ntegrated **L** **P**artial evaluator and post **O**ptimizer **S**ystem, has been developed in this thesis to study the program specialization technique called partial evaluation and various post optimization techniques on the resulting residual programs. The system implements all these operations on a flow chart language called L.

1.1 Program Specialization and Optimization Techniques

Program specialization and optimization techniques have always been a challenging and active research area in computer science. This section briefly introduces the techniques used in ILPOS.

Partial Evaluation is a program specialization technique aiming the use of the whole static information about the program at the specialization time. As discussed later, a partial evaluator accepts some program written in a high level language and produces another program, again in the same language, as its output. This output is called the residual program. The intended transformation is such that the execution time of the residual program is less than the execution time of the original program, thus achieving a certain speed-up. The main theme of partial evaluation is the usage of the static arguments of the programs. The residual program will be free of such static parameters and it will only depend on the dynamic part of the input.

Post optimization techniques aim at the production of more efficient programs in terms of both time and space. Removal of useless code is one such technique which employs the idea of no nonsense computation. The main idea is to avoid any computation that has no effect on the output of the program. Such optimizations rely on the flow analysis of the programs. ILPOS employs a flow analyzer for the language L which collects the information needed by the useless code remover. The main usage of useless code removal is for gaining time efficiency through the removal of useless code portions.

Program minimization is another technique aiming at space efficiency. As it will become clear later on, the language L is very suitable for such a minimization. The idea of minimization has been adopted from the formal language theory where it is applied to the reduction of finite state machines.

Briefly, ILPOS is an experimental system for studying these program specialization and optimization techniques. The language under consideration is a flow chart language called L. This thesis describes the operation of ILPOS and describes the functions of its modules as they are needed. The structure of the thesis is as follows: First recent literature is summarized and an overview of ILPOS is given. This is followed by the description of the L language and its properties. Then the concept of partial evaluation is given and the technique of partial evaluation for L is described. The thesis will continue with the post optimizations that are implemented by ILPOS. After this, termination problem and its solution is discussed. The discussion will continue with the description of the symbolic gain analyzer system and a case study on definite integrals. Throughout the discussion several examples will be given to demonstrate the ideas.

1.2 Recent Literature on Partial Evaluation

The earliest work on partial evaluation dates back to 1967. The term *partial evaluation* has been used in those days for the discussion of computation with incomplete information. Since then, a huge literature has been formed. A detailed guide to the literature can be found in chapter 18 of [14]. Here some pointers to the literature not listed there is given.

For the principals of partial evaluation, a very recent work in binding time analysis techniques is given in [12] by L. Hornof et.al. Correctness proofs for on-line and off-line partial evaluators by C. Consel et. al. is also an important work in the foundations of the area, see [8]. Another recent study for the program adaptation techniques has been done by C. Consel in [5].

The idea of program specialization has also been applied to operating systems. A work that applies the ideas to a commercial system by G. Muller et. al. appeared recently in [19]. Another paper describing the applications to operating systems is by E. N. Volanschi et. al, see [22] for details. A paper by C. Consel et. al. describes the incremental specialization techniques as they are applied to operating systems, see [9] for details.

Functional languages has always been a first choice for partial evaluation research. A partial evaluator for Scheme, called *Schism*, see [6], and binding time analysis techniques for such languages, see [7], is among the research areas. The message dispatching problems of object oriented programming languages has also been addressed. A work by Jeffrey Dean et. al, see [10], analyses the topic.

Another important point of work is that of code generation on the fly. In this approach, the code is generated at the run time whenever partial evaluation is useful. This direction of research has become very popular in recent years. Works describing compile time and run time specialization techniques can be found in [3, 20]. An interesting programming language work has been done by Dawson R. Engler with the *tick-c* language, see [11]. This language, a descendant of C, has special constructs for run time code generation.

1.3 ILPOS Overview

ILPOS has been entirely implemented in the Scheme language and works under MIT Scheme, Release 7.3.0 (beta) and is freely available. See appendix D for the details of getting it. It is R4RS (see reference [4]) compliant. ILPOS is composed of the following modules:

- L interpreter system

  - L lexical analyzer
  - L parser and unparser
  - L interpreter
    * L library manager
    * Expression Evaluator
- Partial Evaluator system
  - Binding Time Analyzer
  - Partial Evaluator
    * Incomplete Specification Handler
    * Code Generator
    * Expression Reducer
    * Termination Handler
- Post Optimizer System
  - Useless Code Remover
    * L flow analyzer
  - Program Minimizer
  - Linearizer and Canonicalizer
- Gain Analysis System
  - Symbolic Analysis Runner
  - Statistics Collector
- Documentation system, the log file generator

All the modules of ILPOS will be explained together with the ideas that they implement throughout this thesis. The source code for all these modules are supplied in the appendix. Figure 1.1 shows the modules and their relations in a graphical format.

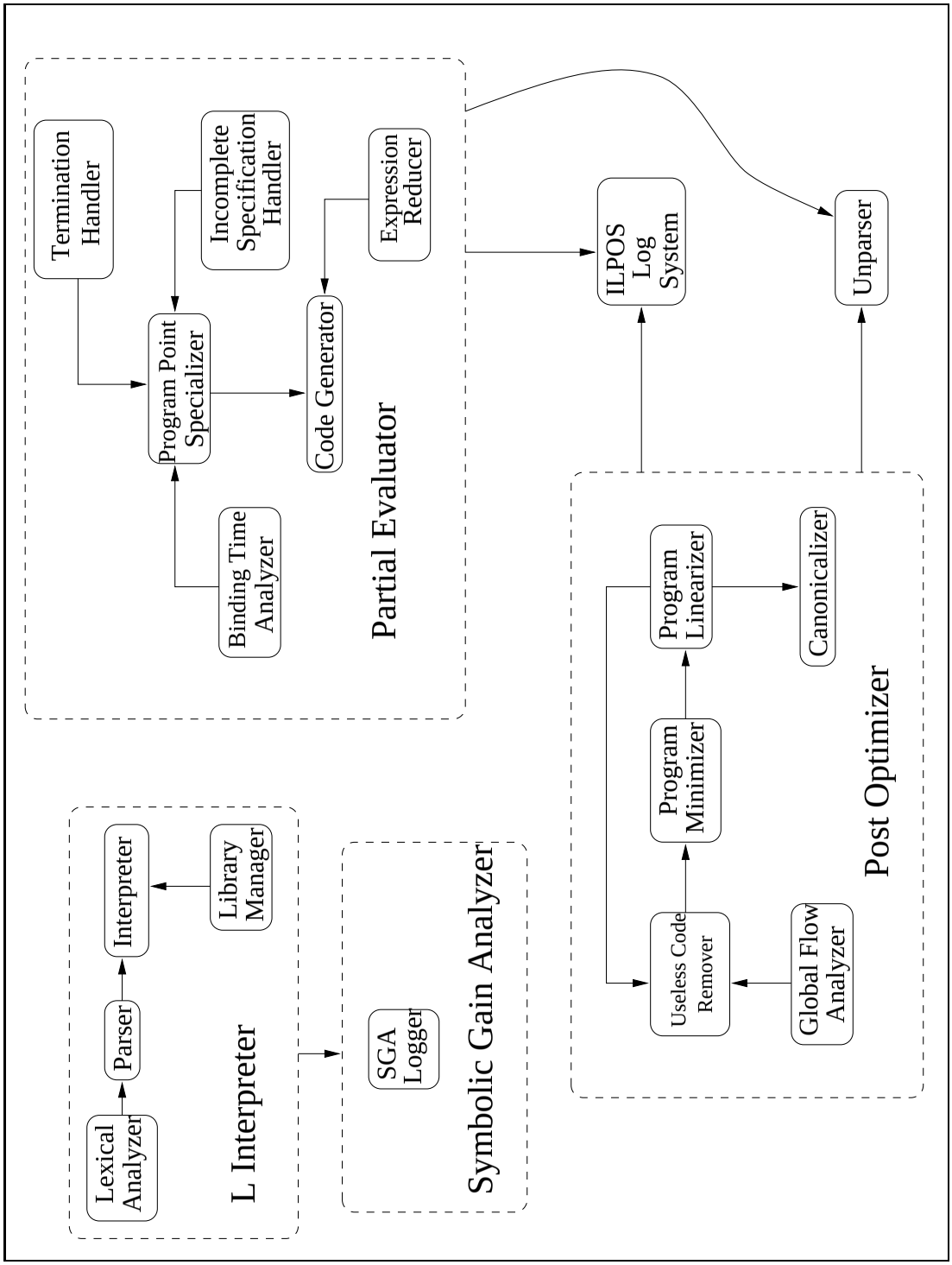

Figure 1.1: The ILPOS module chart

CHAPTER 2

THE LANGUAGE L

ILPOS, being an experimental system, demonstrates the ideas on a toy flow chart language called L. Although very simple, it proves to be a real programming language since any Turing machine can be simulated by an L program, up to the allowed memory limits of the underlying computer.

2.1 Syntax of L

The syntax of L has been given in [14]. ILPOS uses the same syntax with some little modifications. The syntax of L in extended BNF notation, as employed by ILPOS is given in Figure 2.1.

Syntactically, L is a free indentation language in the sense that any white space is welcome unless it breaks up some intended token. The user is free to format the input file in any way s/he likes. Tokens are separated by white space (blank, tab or newline). Comments are also welcome, any string of characters that follow a `#` character up to the next newline is assumed to be a comment and ignored by the lexical analyzer. There is no predefined limit on the length of identifiers or the range of the numbers that are written as literals.

Notice that L does not have the concept of user defined functions. The built-in functions are listed under the <Op> category in the grammar. On the other hand, one can easily add new functions to the library, i.e. to the built-in collection, using

the underlying implementation language Scheme. To make the lexical analyzer and the parser recognize this new function, it is not necessary to modify them. The lexical analyzer and the parser has been prepared in such a way that the addition of the name of the new function to an existing functions list handles the rest automatically (see appendix D.7 for a description of how to do this). In this way, a new library function can be added to the system without altering the internals of the lexical analysis and the parsing systems of ILPOS.

```
<Program>      ::= read ( <VarList> ); <BasicBlock>+
<VarList>      ::= λ | <Var> <VarListRest>
<VarListRest>  ::= , <Var> <VarListRest> | λ
<BasicBlock>   ::= <Label> <Assignment>* <Jump>
<Assignment>   ::= <Var> := <Expr> ;
<Jump>         ::= goto <Label> ;
                 | if <Expr> goto <Label> else <Label> ;
                 | return <Expr> ;
<Expr>         ::= <Constant>
                 | <Var>
                 | <Op> ( <ExprList> )
<ExprList>     ::= λ | <Expr> <ExprListRest>
<ExprListRest> ::= , <Expr> <ExprListRest> | λ
<Constant>     ::= ' <Val>
<Op>           ::= hd | rest | cons  | list | member | append | eq
                 | add | sub | mul | div | odd | even | informSGA
                 | gte | gt  | lte | lt | sqrt | exp
<Label>        ::= <Id> | <Number>
<Val>          ::= <Number> | <Id> | <List>
<List>         ::= ( <Val>* )
<Id>           ::= <L> (<L>|<D>)*
<Number>       ::= <D>+ (. <D>+)?
<D>            ::= 0 | 1 | ... | 9
<L>            ::= the ascii character set
```

Figure 2.1: Syntax of the language L

2.2 Semantics of L

Being a flow chart language, the semantics of L is quite straightforward. Any L program has a single entry point which is the `read` block that must appear at the top. The termination of an L program must necessarily be caused by some `return` statement, no other sort of termination is provided. This means that any program in L computes some value and returns it to the caller. This feature nicely fits in our interpretation of programs as functions.

An L program is composed of a sequence of blocks each of which is capable of making any number of assignments. A block is always terminated either by a conditional or an unconditional jump to some other block, or by a return statement which causes the program to cease the execution. Upon the start of the execution, the read block is activated and the initial values of the variables listed in the read list is read from the standard input. Then the first block takes the action, performing some assignments and then jumping to some other block. Notice that the read block does not have any jumps, control passes to the textually following block automatically. Execution goes in this fashion until it reaches some return statement or the program crashes for some reason[1]. It is also possible that the program may loop forever without ever returning a value.

All variables are assumed to be global and a reference to some variable that has not been initialized before crashes the program. There is no variable declaration, the first assignment to some variable name automatically creates a storage location for it. An L variable is dynamically typed. The same variable can assume any valid L data type, i.e. numbers, constants or Scheme like lists at different points in the execution of the program. As each assignment is executed, the store is updated to reflect the new values of the variables.

An example L program is depicted in Figure 2.2. As usual, *hd* and *rest* are the main list processing functions. This program searches the *namelist* for a specific name and returns the value corresponding to it in the *valuelist*. This operation is typical of a symbol list search in a compiler. Notice that the case when *name*

[1] For example, some library routine may crash or program may jump to some block that is not present.

is not in *namelist* is ignored by the program. This is done deliberately and the consequences are investigated in Section 4.6.

```
# simple L program, searching two parallel lists.
read(namelist, valuelist, name);

search: if eq(name, hd(namelist)) goto found else cont;
cont:   valuelist := rest(valuelist);
        namelist := rest(namelist);
        goto search;
found: return hd(valuelist);
```

Figure 2.2: An example L program: search

2.3 Other features of L

There are no side effects in L, meaning that any library routine is evaluated solely for the value it produces, not for any other purpose[2]. This feature is particularly important as it allows the partial evaluator to regard library calls as ordinary values, rather than commands that may affect the internal store in some undetectable manner. The store of an executing L program can only be altered through the assignment statements present in the basic blocks.

As indicated before, L supports the list data structure. A list can be of any depth and may contain constants or numbers in it. Also, L supports arbitrarily large integer numbers and arithmetic operations on them.

The L interpreter supplied within ILPOS has been designed to accommodate easy extension of the L library. Any user can add new functions to the library by adding its definition to the *llibrary.s* file (See Section D.7). Note that the definition must be given in the Scheme language and it must be free of any side effects. For a reference of Scheme see [1, 4].

[2] This explains the reason for having no output functions in L.

CHAPTER 3

PARTIAL EVALUATION

Consider some mathematical function f having two arguments. It is obvious that one can obtain another function, say g, which has only one argument and obtained by freezing one of the arguments of f to some known fixed value. For example let $f(x, y) = x! + xy$ and set x to 4. Now, g is a function of y only and its definition is $g(y) = 24 + 4y$. Note that for all computations of the function f where the value of x is 4, one can use the g function instead of it and obtain the same result much more easily. Technically we say that g is the projection (or restriction) of f with respect to $x = 4$. In logic, the same idea follows with currying.

Now consider what would happen when the same idea is carried over to programs written in some language. In essence, any program can be thought of as a function that maps its arguments to the corresponding output value. This is a trivial consequence of the Church-Turing thesis indicating that any computable function can be computed by a Turing machine and a Turing machine is nothing but a transformer of its initial tape contents (i.e. arguments) to a final tape (i.e. the output).

In fact this idea has been investigated by Kleene in 1952 with his famous *s-m-n theorem* [15]. Although Kleene's formulation had nothing to do with the efficiency of such a specialization process, his work signals the existence of such specialized Turing machines. For a recent survey of the topic, see [13].

To illustrate the concept of partial evaluation, consider the programs in Figure 3.1 written in a hypothetical programming language.

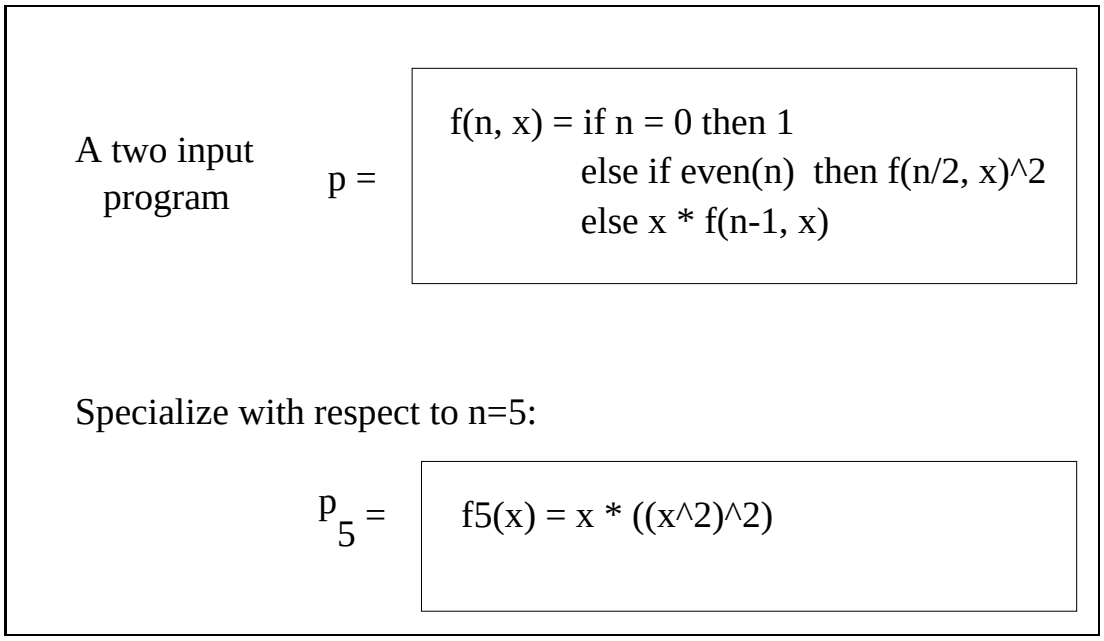

Figure 3.1: A specialization example

The program p simply raises its argument x to the power n. The program p_5 is a specialized form of the original program raising its argument x to the power 5. Clearly one can use p_5 instead of p whenever the second argument is known to be 5 and this would allow an efficient computation since there would be no tests etc. that must be performed at the run time. The merit of partial evaluation lies in the generation of such residual programs from the original ones automatically.

3.1 Semantic formulation of Partial Evaluation

Consider some program p written in some language L. Then, the notation $[\![p]\!]_L$ is used to denote the meaning of the program p. This can be considered as the transfer function (or input/output function) corresponding to the program p. Now assume that p has n input arguments and consider that p is run with arguments $a_1, a_2, \ldots a_n$. Then the notation $[\![p]\!]_L[a_1, a_2, \ldots, a_n]$ denotes the output

of the program with respect to these inputs, i.e.:

$$result = [\![p]\!]_L[a_1, a_2, \ldots, a_n]$$

The program p may go into an infinite loop, which is, in general, undetectable due to the halting problem, and in such a case $result$ is simply undefined.

Using these notational conventions one can describe the partial evaluation process in a more concise way. Historically, the partial evaluator programs has been given the name *mix*, a short for mixed computations. Suppose that p is an n argument program written in some language L. Furthermore assume that the k of its arguments are *static*. By static we mean that its value is known at the specialization time, i.e. the program will be specialized with that fixed specific value. By this discussion the remaining $n - k$ arguments are assumed to be dynamic, i.e. their initial values will not be available until run time. Without any loss of generality and for the sake of notational convenience, we can assume that these k static arguments are the first k of the program arguments[1]. Let's denote $a_1, a_2, \ldots, a_k$ by $s_1, s_2, \ldots s_k$ (reminding that they are static) and denote $a_{k+1}, a_{k+2}, \ldots, a_n$ by $d_1, d_2, \ldots, d_{n-k}$ (reminding that they are dynamic). Now the entire computation of program p with these inputs can be given as:

$$result = [\![p]\!]_L[s_1, s_2, \ldots, s_k, d_1, d_2, \ldots, d_{n-k}]$$

Now assume that we partially evaluate program p with respect to its static inputs and obtain the program p_{res}, denoting residual p. This computation is expressed as (assuming mix is written in language M):

$$p_{res} = [\![mix]\!]_M[p, s_1, s_2, \ldots, s_k]$$

Note that p_{res} is again a program in the language L. Now this residual program is run with the dynamic input only, i.e.:

$$result = [\![p_{res}]\!]_L[d_1, d_2, \ldots, d_{n-k}]$$

[1] Note that this does not impose any restriction on the process since one can always reorder the arguments in this way.

It is clear from the definition of partial evaluation that this result is the same as that obtained by running the original program on all of the inputs, i.e. the following equation, known as the *mix equation*, holds:

$$[\![p]\!]_L[S,D] = \Big[\Big[\underbrace{[\![mix]\!]_M[p,\ S]}_{p_{res}}\Big]\Big]_L[D]$$

3.2 The motivation for using Partial Evaluation

Clearly, performing partial evaluation on some source program has some cost associated with it. The required transformation is not straightforward in the sense that it requires extensive analysis of the source program. This will become clear when the technique for partial evaluation is explained later on. If the static part of the input does not change frequently and the program is run for the changing values of the dynamic variables a lot of times, then specialization will give us good results.

More precisely, let t_p denote the time required for running the program p. Assume that p has a set of static inputs and the residual program obtained by partially evaluating p with respect to them is p_{res}. Let the specialized program be run α times. Then the ratio:

$$\frac{\alpha \times t_p(\text{all inputs})}{t_{mix}(p, \text{static inputs}) + \alpha \times t_{p_{res}}(\text{dynamic inputs})}$$

gives us the gain that we obtain by partial evaluation. For partial evaluation to be useful, this ratio must be greater than 1. Note that as α gets larger the significance of the specialization time gets smaller. So one should think of specialization when α is a large value and the produced residual code runs faster than the original one. The second condition is much easier to satisfy as it will be shown later by examples. When α is low, the specialization time must also be considered.

Another motivation for partial evaluation is that of the trade off between efficiency versus generality. People tend to write programs more and more general, i.e. using as much parameters as possible. This tendency results in highly readable and modular codes. Although this is desirable, the resulting programs are often 'more than needed' and run slower than their equivalent, although performing a more specific task, programs. Partial evaluation bridges this gap in the sense

that more general programs are automatically transformed into specialized, and thus more efficient, programs. This achieves efficiency without loosing generality and modularity.

A typical example of this fact can be given as follows: Function calls, in general, are bottlenecks for efficiency, yet they make programs much structured. One can stick to function calls and let the partial evaluator unfold them, and in fact specialize them, to get efficient versions without sacrificing any modularity in the programs.

Partial evaluation has found place in many fields of computer science. These applications range from computer graphics to neural networks, from scientific computing to database query optimizations. A guide to such applications can be found in Chapter 13 of [14].

CHAPTER 4

PARTIAL EVALUATION FOR THE L LANGUAGE

Consider the search program presented in Figure 2.2. We can think of that L program either as a stand alone program or as a module of a larger one. In any case assume that we know, at the specialization time, that the value of *namelist* is *'(a b c d)* and the value of *name* is *'c*. Also assume that we do not know the value of the *valuelist*. This scenario is not artificial, in the senses that it describes exactly what happens when some interpreter refers to the value of some variable in the symbol list. It is clear that the program, at this stage, is equivalent to the program in Figure 4.1.

```
read(valuelist);

lpe0: valuelist := rest(valuelist);
      valuelist := rest(valuelist);
      return hd(valuelist);
```

Figure 4.1: Specialized search program

It is clear that the specialized version of the program is free of any tests that the original should make. Also the resulting program is a linear one, in the sense

that it does not jump to any other blocks, so there is no jump cost. As the *namelist* gets longer and *name* is found towards the end of the list, the gain would be much more apparent.

The aim of partial evaluation for L language is to obtain these programs automatically. ILPOS handles this specialization using a technique called *program point specialization* as explained in the subsequent section.

4.1 Program point specialization for L

Consider the interpretation of some program p in L. A program that is being executed is called a process. Each point in the lifetime of a process can be described with a tuple $(pp, store)$. Here pp denotes the program block that the process is currently in and $store$ denotes the values of the program variables. Clearly, a program that has just executed its `read` block and is at the beginning of its first basic block can be described as $(pp_0, store_0)$. Similarly, a process that executes its ith block with the values of variables denoted as $store_j$ will be represented by $(pp_i, store_j)$.

Observe that, within each basic block of an L program, the execution is necessarily linear. This means that the control can not pass to some other block without performing all the assignments of the current block. Using this notation, an L program in execution can be represented as:

$$(pp_0, store_0) \rightarrow (pp_1, store_1) \rightarrow \ldots \rightarrow (pp_n, store_n)$$

where the block corresponding to pp_n terminates with a return statement. In this notation, each pp_i is an element of the basic blocks of the current process for $0 \leq i \leq n$. Note that pp_i may be the same as pp_j, corresponding to the same basic block in the process. This sequence of transitions is called the *trace* of the process.

Notice that we can obtain a unique trace of an L program if we know all the input arguments. In such a case the L program is equivalent to a program that has a single block returning the overall result. Now assume that we know only a part of the input arguments. In this case, we can define a *partial trace*

representing the moves of the program. It is clear that in a partial trace the items in the sequence need not be unique. This is due to the fact that some block may end up with a conditional if statement resulting in two different execution paths. For instance, assume that l_0 is a block that ends with a conditional jump statement such as:

`if eq(a, '()) goto l1 else l2;` Then:

$$(l_0, store_i) \longrightarrow \begin{cases} (l_1, store_q) & \text{if } \texttt{eq(a, '())} \text{ evaluates to true,} \\ (l_2, store_r) & \text{otherwise.} \end{cases}$$

This assumes that the conditional expression can not be evaluated at the specialization time because the value of a is not known[1]. It is clear that the set of all such traces correspond to all possible execution paths of the program.

Now consider that some program L is given and we have a program to produce all partial traces corresponding to that program. If some of the arguments to the program are known in advance then this information can be used to cut down some of the traces of the original program into a set of traces which are possible with respect to the known data values. By this means, we have a way of simulating the original program without knowing all the values of the variables.

Exactly in this point does the concept of program point specialization (PPS) is introduced. The main idea is to produce a new program whose set of all possible traces is exactly equivalent to the partial traces of the original program with respect to the values of the known arguments at the specialization time. The name PPS follows from the fact that the residual program consists of a set of blocks which are the specialized versions of the blocks in the original program.

4.2 How to perform PPS

In this section the example program given in Figure 2.2 will be used to demonstrate how one can apply PPS to obtain a residual program.

Again assume that we want to specialize the search program with respect to the values $namelist \mapsto$ *'(a b c d)* and $name \mapsto' c$. We construct the partial trace

[1] In fact $store_q$ is exactly equivalent to $store_r$ since L does not allow any side effects.

of the program with respect to this known arguments step by step.

At the start of the program, we are in block `search`. This program point is represented in our tuple notation as: (search, {(a b c d), c}). Note that the second element of the tuple is the store of the known values, *namelist* and *name* respectively. Now we run our partial evaluator (the program that obtains all possible partial traces) starting with this program point. The method is to generate code for the current block with respect to the known store and then obtain the successors of this block. Looking at the program, the partial evaluator determines that it can execute the conditional jump with no difficulty since it involves only the known values. Since the condition turns out to be false, the successor node is `cont`. Up to now, the partial evaluator found out that the execution path that leads to the `found` block is impossible with the given start values so it cuts down that branch obtaining the following start of the chain:

$$(\text{search}, \{(\text{a b c d}), \text{c}\}) \rightarrow (\text{cont}, \{(\text{a b c d}), \text{c}\})$$

At the same time the following code is generated for the initial block:

```
(search, {(a b c d), c}) : goto (cont, {(a b c d), c});
```

It is important to note that the label in the goto statement is annotated with the store associated with the program at that point. At this point the partial evaluator looks for the code in block `cont`. It sees that there are two assignments and the second one can be executed at the specialization time. It updates its store by executing that assignment and assigns *namelist* the new value *'(b c d)*. The partial evaluator notices that it can not handle the first assignment at this time. Upon examining the jump statement associated with the block, the partial evaluator generates the following code for this program point:

```
(cont, {(a b c d), c}) : valuelist := rest(valuelist);
                         goto (search, {(b c d), c});
```

At this point partial evaluator knows that it needs the code for the program point (search, {(b c d), c}) and looks if it had already generated it before. Checking

the partial trace that it had generated so far, it determines that this is an entirely new program point, so it extends the trace with this new item to obtain:

$$\text{(search, \{(a b c d), c\})} \rightarrow \text{(cont, \{(a b c d), c\})} \rightarrow \text{(search, \{(b c d), c\})}$$

The code generated for (search, {(b c d), c}) follows the same lines as described. The result is:

```
(search, {(b c d), c}) : goto (cont, {(b c d), c});
```

Again looking at the partial trace, (cont, {(b c d), c}) is found to be a new program point, giving the chain:

$$\text{(search, \{(a b c d), c\})} \rightarrow \text{(cont, \{(a b c d), c\})} \rightarrow$$
$$\text{(search, \{(b c d), c\})} \rightarrow \text{(cont, \{(b c d), c\})}$$

Similarly the following code is generated:

```
(cont, {(b c d), c}) : valuelist := rest(valuelist);
                       goto (search, {(c d), c});
```

The generation of this code extends the partial trace to:

$$\text{(search, \{(a b c d), c\})} \rightarrow \text{(cont, \{(a b c d), c\})} \rightarrow \text{(search, \{(b c d), c\})} \rightarrow$$
$$\text{(cont, \{(b c d), c\})} \rightarrow \text{(search, \{(c d), c\})}$$

Proceeding as before, the partial evaluator finds out that the program point (search, {(c d), c}) generates the following code:

```
(search, {(c d), c}) : goto (found, {(c d), c});
```

producing the partial trace:

$$\text{(search, \{(a b c d), c\})} \rightarrow \text{(cont, \{(a b c d), c\})} \rightarrow \text{(search, \{(b c d), c\})} \rightarrow$$
$$\text{(cont, \{(b c d), c\})} \rightarrow \text{(search, \{(c d), c\})} \rightarrow \text{(found, \{(c d), c\})}$$

Finally the code for (found, {(c d), c}) is produced as:

```
(found, {(c d), c}) : return hd(valuelist);
```

This describes the end of the computation since the partial evaluator has come to a `return` statement and it has no items in the partial chain that remains to be elaborated. So the whole partial chain corresponds to the actions of the residual program. Combining all the codes that the partial evaluator produced one obtains the program in Figure 4.2.

```
(search, {(a b c d), c}) : goto (cont, {(a b c d), c});
(cont,   {(a b c d), c}) : valuelist := rest(valuelist);
                           goto (search, {(b c d), c});
(search, {(b c d), c})   : goto (cont, {(b c d), c});
(cont,   {(b c d), c})   : valuelist := rest(valuelist);
                           goto (search, {(c d), c});
(search, {(c d), c})     : goto (found, {(c d), c});
(found,  {(c d), c})     : return hd(valuelist);
```

Figure 4.2: Uncompressed residual code for search

This completes the actions of the partial evaluator. It successfully produced the residual code that depends only on the value of the *valuelist* argument.

It should be noted that the illustrative example chosen here was a very simple one. It never produced choices in the continuation of the partial trace. However, as indicated before, this is not always the case. In case there exists more than one continuation from a program point, the partial evaluator takes notes of each of them and expands in both branches. This will allow the correct generation of all the possible program paths thus ensuring the correctness of the produced residual code.

4.3 Transition Compression

The last section described how a PPS partial evaluator can produce residual programs. The example gave us the residual in Figure 4.2. Previously it has been indicated that a specialized form of the same program would appear as in Figure 4.1. Inspecting the program given in Figure 4.2, one notices that each

block is terminated with a `goto` statement. It is clear that one can always copy the designated block after the `goto` statement (and by removing the `goto`) and still obtain an equivalent program. This procedure is called transition compression. Applying the technique to program in Figure 4.2 one gets the program in Figure 4.3.

```
(search, {(a b c d), c}) : valuelist := rest(valuelist);
                           valuelist := rest(valuelist);
                           return hd(valuelist);
(cont,   {(a b c d), c}) : valuelist := rest(valuelist);
                           valuelist := rest(valuelist);
                           return hd(valuelist);
(search, {(b c d), c})   : valuelist := rest(valuelist);
                           return hd(valuelist);
(cont,   {(b c d), c})   : valuelist := rest(valuelist);
                           goto (search, {(c d), c});
(search, {(c d), c})     : return hd(valuelist);
(found,  {(c d), c})     : return hd(valuelist);
```

Figure 4.3: Initial form of compressed search program

A simple reachability analysis on this program reveals that starting at (search, {(a b c d), c}) no other block is reachable. So we can delete the remaining blocks. Finally by adding the initial read block, reading only the value of the *valuelist* one obtains the final code as in Figure 4.4.

```
read(valuelist);

(search, {(a b c d), c}) : valuelist := rest(valuelist);
                           valuelist := rest(valuelist);
                           return hd(valuelist);
```

Figure 4.4: Compressed form of specialized search program

Notice that the program in Figure 4.4 is isomorphic to the program in Figure 4.1 up to a relabeling of the blocks.

4.4 Binding Time Analysis

It is clear from the previous discussion that PPS technique involves the generation of specialized versions of the program points that are present in the subject program. In doing this, it sometimes generates code for the corresponding statement in the original program or it updates the internal store to reflect the actions of the program. For this reason, the decision concerning whether a statement should produce code is an important issue in PPS.

Naturally, the partial evaluator attempts to generate as few code as possible. In order to achieve this goal, it must know which statements can be evaluated at the specialization time and which must be transferred to the residual code. It is apparent that some statement can be evaluated at the specialization time if it depends only on the static variables of the program. Then the question is to determine which variables are static given the list of the known arguments. This analysis is called the binding time analysis (BTA).

In the previous section, the only program variables were those that are read through the `read` block of the associated program. Of course this is not always the case, the program may contain other variables. The aim of BTA is to construct the so called *division* which gives all the static variables of the subject program.

Simple minded[2] definitions for a static variable and a static expression can be given as follows:

Definition 4.1: A *dynamic* variable of an L program can be defined as follows:

1. All the variables that are in the `read` list of the program but not specified as known are dynamic.

2. Let $v := \alpha$ be an assignment in the program. If α is not a static expression then v is dynamic.

[2] The term *simple minded* will become clear when termination of partial evaluation is discussed.

3. No other variable is dynamic unless 1 or 2 is satisfied.

Definition 4.2: A variable v is called *static* if it is not *dynamic*.

Definition 4.3: An expression α of an L program is called a *static expression* if and only if the followings hold:

- α is a constant,
- α is a static variable, or
- α is of form $op(\alpha_1, \alpha_2, \ldots \alpha_n)$ where each expression $\alpha_i, 1 \leq i \leq n$ is static.

Definition 4.4: An expression α of an L program is called a *dynamic expression* if it is not static.

The main idea in BTA algorithm is to apply these definitions on the subject program to come up with the division. Note that the BTA described here assumes that a variable is either static or dynamic in all parts of the program under consideration. The division constructed by these algorithms, and hence by ILPOS, are *monovariant* divisions in this sense. Of course, this is not necessarily the case and a *polyvariant* division can be computed which will exhibit the change in the division for each block separately. For the details of polyvariant divisions see [14].

As mentioned before, the analysis performed by the binding time analyzer gives the specializer the required information for making the decision of generating code for a program statement. The partial evaluator attempts to generate code for a statement if it determines that the expression it involves is dynamic (by definition 4.4). If an expression is static, it is computed at the specialization time and the store is updated.

The last definition concerning the division constructed by BTA is that of finiteness:

Definition 4.5: A division is said to be *finite* if it allows the partial evaluator to generate a finite residual program.

As it will be explained later, unfortunately, finiteness is not guaranteed with these definitions.

4.5 PPS algorithm and ILPOS

Equipped with the previous discussions, a naive description of the PPS algorithm will be given in this section. The PPS algorithm can be thought in 3 parts: the initialization part, the trace construction part and the code generator part.

The initialization part of the PPS algorithm, as depicted in Figure 4.5 includes the binding time analysis of the source program.

{*Given the list of input arguments v_{spec} and their starting values which are known at the specialization time, construct the residual program by the program point specialization technique, initialization part*}

1. Let v_{dyn} be the set of variables which appear in the `read` list but not in v_{spec}.
2. Apply definition 4.1 with the initial set v_{dyn} to compute the set of all dynamic variables that appear in the program.
3. Let v_{stat} be the set of all program variables which are not marked as dynamic by step 2.
4. Let $store_0$ be the store containing the variables in v_{stat} together with their initial values. If some variable in this set is not an input argument then let its value be ERR indicating that it is a not yet initialized variable.
5. Let $Trace = \{(pp_0, store_0)\}$ where pp_0 is the name of the first block in the program. Let $Residual = \phi$.

Figure 4.5: Algorithm for preparing a program for PPS

The initialization algorithm aims to find the division of the program variables. To do this, it examines every assignment in the program to find out the variables which are dynamic. Initially the set of input variables (i.e. those variables in the `read` list) which are not specified to be static are included in the list of dynamic variables. Then, for all assignments, whenever the right hand side contains a dynamic variable, the left hand side is also included in the list. This closure algorithm is applied until the set of dynamic variables cease to change. The set difference of all program variables and these dynamic variables gives us the

set of all static variables of the source program. After this analysis the trace construction part is activated whose algorithm is given in Figure 4.6.

{*PPS algorithm, trace construction part.* }

1. Select an element $\{(pp_i, store_i)\}$ in $Trace$ and mark it as *processed*. If no unmarked element exists goto step 5.
2. Let $curCode =$ "$(pp_i, store_i) :$"
3. Let $curBB$ be the basic block corresponding to pp_i. Apply the *codeGeneration* algorithm of Figure 4.7 to $curBB$.
4. Let $Residual = Residual \cup curCode$, Go to step 1.
5. Relabel all the blocks in *Residual* to identifiers.
6. Initialize *ResidualCode* to a `read` statement containing only those input arguments which are dynamic.
7. Append each element of *Residual* to *ResidualCode*.
8. Apply transition compression of section 4.3 to *ResidualCode*.
9. Output *ResidualCode* as the partially evaluated code.

Figure 4.6: Algorithm for Trace Construction

The trace construction algorithm explores all possible program points emanating from the starting program point. The algorithm tries to find out all possible continuations from a given configuration and elaborates them as code segments in the residual program. The main engine of the algorithm goes with the code generation part as depicted in Figure 4.7.

ILPOS employs these algorithms with a change in step 8 of the trace construction part (Figure 4.6) to implement PPS for language L. ILPOS employs *transition compression on the fly* instead of a separate phase for compression. Remember that a `goto` is said to be compressed if the upcoming block is put in place of it. This has the advantage of both saving an extra phase and eliminating the need for a reachability analysis as done in section 4.3. This is due to the fact

that, a new residual block is generated only if it is reachable.

{*PPS algorithm, code generation part.* }

1. For each assignment $v := \alpha$ in $curBB$ do the followings:
   - if v is static by division then compute α with respect to $store_i$. Update $store_i$ to reflect the change in v.
   - if v is dynamic then reduce the expression α to obtain β with respect to $store_i$. Let $curCode = curCode \circ "v := \beta;"$ where $\circ$ is the concatenation operator.
2. According to the jump statement of $curCode$ do the following:
   - If the jump is a `goto` then let $curCode = curCode \circ "goto(pp_j, store_i)"$ where pp_j is the label of the `goto`. Let $Trace = Trace \cup \{(pp_j, store_i)\}$.
   - If the jump is a `return` then let its expression be α. If α is static then compute its value into β otherwise reduce it to expression β. Let $curCode = curCode \circ "return\ \beta;"$.
   - if the jump is a conditional jump of the form: if α goto l_1 else l_2, then proceed as follows:
     - if α is static then compute its value. If it evaluates to `true` then let $curCode = curCode \circ goto(pp_k, store_i)$ otherwise let $curCode = curCode \circ goto(pp_j, store_i)$ where pp_k and pp_j are the program points corresponding to l_1 and l_2. In each case let $Trace = Trace \cup \{(pp_l, store_i)\}$ where pp_l is pp_k if $\alpha = true$ and pp_j otherwise.
     - if α is dynamic then reduce it to β. Let $curCode = curCode \circ "if\ \beta\ goto\ (pp_k, store_i)\ else\ (pp_j, store_i);"$ where pp_k and pp_j are the program points corresponding to l_1 and l_2 respectively. Let $Trace = Trace \cup \{(pp_k, store_i), (pp_j, store_i)\}$.

Figure 4.7: Algorithm for code generation

The algorithm for PPS refers to the concept of reducing an expression with respect to some store. The definition of a reduced expression follows:

Definition 4.6: An expression is *reduced* if it is,

- a constant

- a variable v marked as dynamic by division
- an application $op(e_1, e_2, \ldots, e_n)$ and each of $e_i, 1 \leq i \leq n$ is reduced and at least one of $e_i, 1 \leq i \leq n$ is a dynamic expression.

Notice that the reduced form of a static expression is equivalent to a constant. Using this definition, ILPOS employs an expression reducer which, given a dynamic expression, returns the equivalent reduced expression with respect to the current store associated with the block. The reduced expression is either a constant or an application with dynamic variables in it.

4.6 Handling Incomplete Specifications

```
# an incompletely specified L program:
read(names, name);

look:  if eq(name, hd(names)) goto ok else cont;
cont:  names := rest(names);
       goto look;
ok:    return names;
```

Figure 4.8: An example of an incomplete specification

Consider the L program given in Figure 4.8. The intended meaning of the program is to find the first occurrence of *name* in the *names* list and return the rest of the list starting with it. However, there is an assumption on the contents of *names*. The programmer assumes that it necessarily contains *name*[3]. This is a completely acceptable program in the sense that the programmer might have made sure that *name* would occur in *names* by some other means. Since it is guaranteed to be there, the programmer saved a test of emptiness. Programming in this way has nothing wrong with it as far as the programmer is aware of it[4].

[3] In case *name* does not exist, the program will crash at run time, since it will attempt to take the `rest` of an empty list.

[4] Notice that it may also be the case that the test for emptiness might have been forgotten by mistake.

Although nothing is wrong with this style, such programs cause problems when they are partially evaluated. Assume that this program is partially evaluated with respect to the known input argument *names = '(a b)*. The partial traces obtained are as follows:

- Trace corresponding to $name = a$:

$$(\text{look}, \{(\text{a b})\}) \rightarrow (\text{ok}, \{(\text{a b})\})$$

- Trace corresponding to $name = b$:

$$(\text{look}, \{(\text{a b})\}) \rightarrow (\text{cont}, \{(\text{a b})\}) \rightarrow (\text{look}, \{(\text{b})\}) \rightarrow (\text{ok}, \{(\text{b})\})$$

- Trace corresponding to neither:

$$(\text{look}, \{(\text{a b})\}) \rightarrow (\text{cont}, \{(\text{a b})\}) \rightarrow (\text{look}, \{(\text{b})\})$$
$$\rightarrow (\text{cont}, \{(\text{b})\}) \rightarrow (\text{look}, \{()\}) \rightarrow \textbf{CRASH}$$

In terms of the graph of partial traces this situation is depicted in Figure 4.9.

Since the last path in trace results in a crash the entire partial evaluation process fails. Of course this is not acceptable since the program and its static data is well formed. Unfortunately PPS can not handle this situation. To remedy the problem ILPOS uses *guards*.

4.6.1 Guards

The problem specified above is the result of applying a library routine to some value for which that routine is not defined. In the above example the library function *hd* was applied to $'()$ resulting in crash.

In general, all library functions implement some partial function. It is this partiality that causes the problems[5]. If the partial evaluator knows for every library function the values at which they are undefined then it can handle such incomplete specifications correctly. The idea is to check whether some application is safe for its arguments. To do this we define guards:

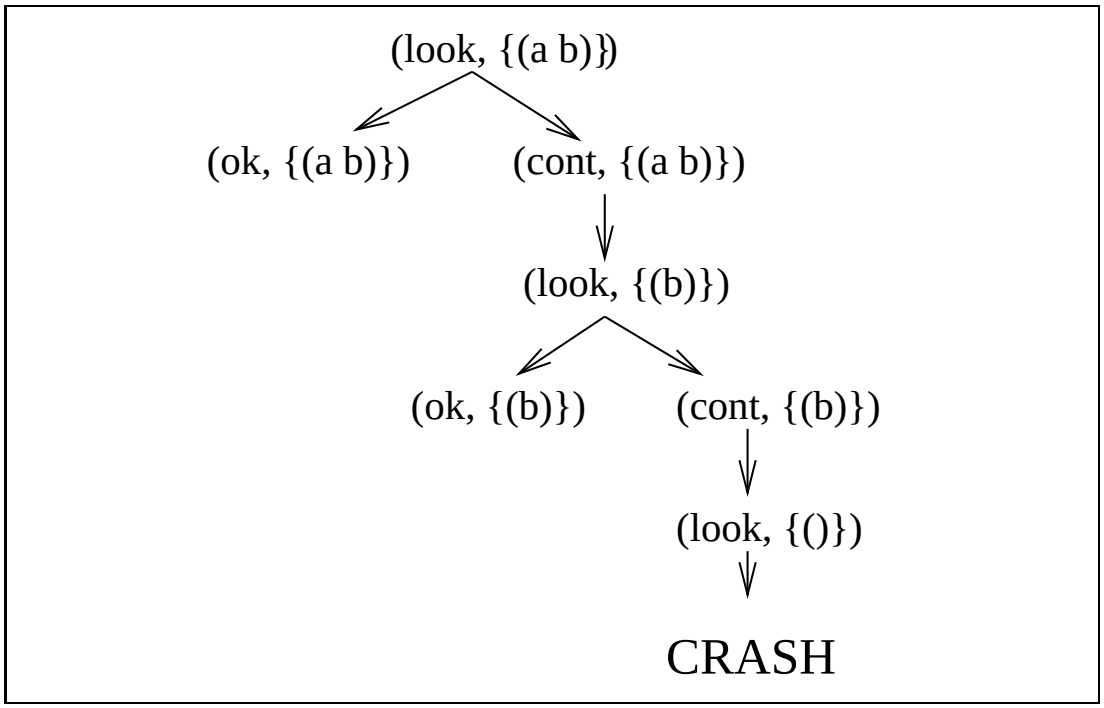

Figure 4.9: Traces of an incomplete specification

Definition 4.7: A *guard* for a library function is another function which returns `true` if the corresponding function is defined at some argument, `false` otherwise.

For instance a guard for `hd` function is a function returning `false` if the argument is an empty list or a non-list and `true` otherwise.

4.6.2 Guards in ILPOS

The library supported with L contains many functions each of which is equipped with their guards. As mentioned before, ILPOS has an extendible library and the user is supposed to supply guards with the new functions all defined in the Scheme language.

Whenever the ILPOS partial evaluator attempts to evaluate some application that is static, it first calls the associated guard to check whether the application is safe. If the operation is determined to be safe then PPS goes accordingly. Otherwise an incomplete specification is signaled and the code in *curCode* (see algorithm in Figure 4.7) is set to `"return run_time_crash;"`. This guarantees

[5] For example, *hd* is a partial function since it is not defined for an empty list.

that if the residual code reaches to this block, it will indicate the crash to the user as the original would do. Using this method, ILPOS generates the residual program in Figure 4.10 for the above example.

```
read(name);

lpe0: if eq(name, 'a) then lpe1 else lpe2;
lpe1: return '(a b);
lpe2: if eq(name, 'b) then lpe3 else lpe4;
lpe3: return '(b);
lpe4: return run_time_crash;
```

Figure 4.10: Residual program for an incompletely specified program

Noticing that this may be something intended or something that is forgotten by mistake, the ILPOS log generator adds to its output log a record for its action. The corresponding function call and the static argument is recorded with a suitable warning message. The log generator is described later in section 7.1. In this case, it will contain the a warning message as in Figure 4.11. A similar message is issued on the screen to notify the online user.

```
** Warning the call <hd> has problems.
** Partial Evaluator Warning: The static expression:
        hd(names)
** has been evaluated with respect to the static environment:
  names --> ()
** contains an unsafe operation.
** Signaling operation is indicated above.
```

Figure 4.11: Sample warning of guards in the log file

4.7 Computational Complexity of Partial Evaluation

As described in the previous sections of this chapter, partial evaluation is composed of a sequence of separate phases. In this section, the computational complexity of these algorithms will be discussed.

Any source program that is subject to partial evaluation must first be read from the disk and parsed into an abstract syntax tree. Once represented in this form, all operations are carried out as transformations of the parse tree corresponding to the program.

First of all, binding time analysis is performed on the program. This analysis starts with a set of dynamic variables as found by the set difference of the program arguments and the static parameters. The BTA algorithm iterates over all the assignments to find out if it can add the name of a new variable into the dynamic variables list (see Figure 4.5). Assuming there are n_{var} variables in the program, this iteration can be made at most n_{var} times. Notice that this is the worst case since it assumes that only one variable is marked as dynamic through each iteration. In each iteration, a number of assignment statements are checked for analysis. We can safely assume that the number of assignments is at most a constant times the number of blocks in the program. That is, we measure the size of the program as a function of its number of blocks. Then, the number of iterations for BTA is $c_1 \cdot n$ where n is the number of blocks in the program. This altogether counts for $c_1 \cdot n \cdot n_{var}$ operations, when counted for all variables that appear in the program. Roughly speaking, the BTA algorithms amounts to a nested loop structure and assuming n_{var} is a linear function of n (as in the number of assignments), the worst case complexity of BTA turns out to be $O(n^2)$.

After BTA is performed, the trace construction algorithm is activated (see Figure 4.6). It is not possible to give a complexity measure at this point since the algorithm simulates the actions of the source program on the static arguments. It can be said that the complexity is at least the complexity of the source program as all of its actions are imitated. Certainly, more operations are performed by the code generator (see Figure 4.7) and other book keeping parts. A rough analysis of the algorithms reveal that the complexity of the PPS algorithm need not be more

than a constant times the complexity of the source code, since for each possible configuration either the computation is imitated or a code portion is generated. Code generation is merely a copying of the original instructions where all the static parts are evaluated and substituted in function calls whenever they are a part of a dynamic call. Again, roughly speaking, the complexity of the whole PPS algorithm is a factor of the complexity of the source program.

As noted above, giving precise analysis of the complexity of the partial evaluation is not an easy one. The main complication arises from the fact that the input to the partial evaluator is not an ordinary data (as a list, number, structure etc.) but just another executable program. Also notice that, as it will be indicated in chapter 6, the partial evaluator may even not terminate in certain cases. The analysis given here assumes that this is not the case.

CHAPTER 5

POST OPTIMIZATIONS

ILPOS has two main components: the partial evaluator and the post optimizer. The previous sections investigated PPS for L. In this chapter the post optimizer system employed by ILPOS will be discussed.

The post optimizations performed by ILPOS has three components: useless code remover, program minimizer and program linearizer.

5.1 Useless Code Removal

Useless code removal (UCR) is a classical optimization technique which aims at "no nonsense computation". This section describes the need and the method for UCR in ILPOS.

5.1.1 Notion of useless code and useless variables

The notion of useless code and useless variables have been widely studied in the literature. In the context of partial evaluation, useless code and useless variables arise as the result of specialization of a large and modular program which performs many tasks. The specialization gets rid of the irrelevant parts of a huge program but it may leave some code portions which, in general, arrange the relationships between the modules of the original program. Although these code portions are a part of the residual code, they have no effect on the semantics of the final code.

It is very difficult, if not impossible, to give a real example to demonstrate the ideas since it would be a very big program. To illustrate the ideas consider the L program given in Figure 5.1.

```
read(a, b);

start:  c := add(a, b);
        if eq(a, '0) goto first else second;
first:  return b;
second: return c;
```

Figure 5.1: An L program demonstrating useless codes

Assume that a is static with initial value 0. The PPS algorithm produces the residual code in Figure 5.2.

```
read(b);

(start, {0}): c := add('0, b);
              return b;
```

Figure 5.2: PPS producing useless code

It is clear that the computation `add('0, b)` and assignment to c are useless. Technically, c is called a useless variable and the assignment `c := add(0, b)` is called a useless code. Following definitions formalize the problem:

Definition 5.1: Let b be a block in an L program. Then the set of all reachable blocks from b is said to be the *successors* of b. This set is called $succ_b$. By definition $b \in succ_b$.

Definition 5.2: Let v and w be two variables. We say that v *directly depends* on w if there exists an assignment $v := \alpha$ where α contains w. By definition

v depends on itself. The relation *depends* is defined as the reflexive-transitive closure of the directly depends relation.

Definition 5.3: Let v be a variable occurring in some block b. Let the set of all nodes which are reachable from b be $succ_b$ and let $contribute_b$ be the set of all return statements occurring in $succ_b$. Then v is called a *live variable* in a block b if some variable contained in at least one of the expressions in $contribute_b$ depends on v. A variable is said to be *useless* in some block if it is not live.

Definition 5.4: An assignment is said to be a useless code in some block if its left hand side is a useless variable in that block.

It is clear that performing the computation of a useless code does not do anything sensible. Deleting all such code from a program is the process of *useless code removal*. This analysis has been deeply investigated in the literature, see [2, 18] for details.

These ideas have been integrated into ILPOS through a data flow analyzer as explained in the following section.

5.1.2 Global Data Flow Analysis for L

Global data flow analysis (GDFA) is the process of collecting useful information from the text of a program without actually running it. As described in the previous section, determination of useless variables is an example of such an analysis.

Notice that the syntax of L is very clear in the sense that the blocks of an L program directly correspond to the definition of a basic block[1] in flow analysis literature. ILPOS employs GDFA to determine the useless variables occurring in blocks. This section explains these ideas more concretely. To begin with, an algorithm for computing the successors of a block is given in Figure 5.3. This algorithm has been referenced in the definition of a live variable and widely used in GDFA.

[1] A basic block is a sequence of instructions which are executed sequentially with no jumps.

{*Given an L program and a block b that appears in the program, return the set of all blocks that are reachable from b* }

1. Let $succ = \{b\}$.
2. Select an element q of *succ* that is not marked as *processed*. If no such block exists goto step 5. Mark q as *processed*.
3. According to the jump statement of q do one of the followings:
   - If the jump is a `goto l` then let $succ = succ \cup \{l\}$.
   - If the jump is a `if` α `goto l1 else l2` then let $succ = succ \cup \{l1, l2\}$
   - If the jump is a return do nothing
4. Go to step 2
5. *succ* is the set of all successors of b.

Figure 5.3: Algorithm for computing successors of a block

To compute the set of live variables for each block, ILPOS uses the following definition:

Definition 5.5: Let b be some block in an L program. Define in_b to be the set of variables live at the "point immediately before" block b. Define out_b to be the set of variables live at the "point immediately after" block b. Let def_b be the set of variables which are assigned values in b "before they are used" and let use_b be the set of those which are "used before assigned". Using set notation these definitions amount to the equations:

For any block b,

$$in_b = use_b \cup (out_b - def_b)$$

$$out_b = \bigcup_{p \in succ_b} in_p$$

These equations simply say the following: The variables which must be live (i.e. has some effect on the computations on block b) are either those that are referenced without being defined (use_b), or those that must be live at the exit of the block and not defined in the block. Similarly the variables which must be live at the exit of b are those variables which are useful in any of the successors of b.

When these two equations are solved simultaneously for all the blocks in the program the in_b sets denote the live variables of each block. To solve these equations ILPOS uses several algorithms. The algorithm computing def and use sets is a trivial one which explore all the assignments and variables involved in the block. The algorithm in Figure 5.4 is used to compute in and out sets given def and use sets. After computing the in sets by the algorithm in Figure 5.4, ILPOS activates the useless code remover to remove any assignment to a variable that does not occur in the in set of the block that the assignment resides in.

{*Given an L program together with def and use sets for each block, compute the in and out sets for all blocks*}

1. For each block b let $in_b = \phi$.
2. Let $changed = false$
3. For each block b compute out_b using the equation following definition 5.4. Then compute in_b and compare it with its old value. If in_b changes from the previous iteration then let $changed = true$.
4. If $changed = true$ then goto 2 else goto 5
5. The current value of in_b is the set of live variables at block b.

Figure 5.4: Algorithm for computing live variables

Note that in doing so one must be careful to delete only those assignments to useless variables which does not have any contribution to a live variable in the block. For example, let some program has a block as in Figure 5.5

```
a := add(k, d);
c := mul(a, k);
a := sub(c, 5);
return c;
```

Figure 5.5: An example of a used useless variable

In this block c is a live variable while a is useless. It is clear that it is safe to remove the second assignment to a while the UCR must save the first assignment. This is due to the fact that the value of a is used in the definition of the live variable c. ILPOS takes care of such situations.

When applied to program in Figure 5.2, these algorithms produce the following results: $use_{start} = in_{start} = out_{start} = \{b\}, def_{start} = \{c\}$. Upon determining that c is a useless variable the assignment is removed yielding the program in Figure 5.6.

```
read(b);

(start, {0}): return b;
```

Figure 5.6: UCR applied to the specialized code

5.2 Program minimization

Consider the text of an L program. One may view any L program as a finite graph. We have the following definition for the graph of an L program:
Definition 5.6: Let p be an L program and b be a block in it. Then *immediate successors* of b is defined according to its jump statement j, as follows:

- if j is a goto statement in the form `goto l` then the immediate successor of b is the set $\{l\}$.
- if j is an if statement in the form `if` α `goto l1 else l2` then the immediate successors of b is the set $\{l1, l2\}$.
- if j is a return statement then immediate successors of b is ϕ.

Definition 5.7: The graph of an L program is a directed graph $G = (V, E)$, with edge labels where V and E are defined as follows:

- Let V be the set of all blocks in the program. Note that V is always finite since any L program has a finite number of blocks.

- Let E contain an edge from some node v_1 to v_2 if and only if v_2 is an immediate successor of v_1 in the corresponding L program.

The labels are either *goto* (corresponding to a `goto` block), or T or F corresponding to the possible outcomes of the expression of the conditional jump. Notice that if the block is terminated with a `return` statement then it does not have any successors.

When viewed as a graph, an L program is much like a finite state automaton. In fact, it is natural to think an L program as a Moore type finite automaton where the node corresponding to its first block is the start state, see [16] for details of such machines.

Based on this analogy, it turns out that we can apply the results from finite automata theory to L programs. ILPOS uses the concept of a minimum state finite automata to minimize the number of states in an L program. The minimization process for a finite automaton is described in [17, 16].

Definition 5.8: An L program is said to be in its minimal form if there are no two blocks which are equivalent.

Definition 5.8 makes use of the concept of equivalent blocks. We define this equivalence as follows:

Definition 5.9: Two blocks of an L program are *equivalent* if they are code equivalent and their immediate successors are equivalent.

And code equivalence is defined as:

Definition 5.10: Two blocks of an L program are *code equivalent* if they have the same sequence of assignments, assigning same expressions to same variables, and they both terminate with the same kind of jump[2].

It is clear that definition 5.10 defines an equivalence relation on the blocks of an L program. Furthermore, by the definition of an equivalence relation, it

[2] In fact this definition is more restricted then needed. The assignments need not be in the same order but they must be permutations of each other resulting in the same results. The more restricted version is used in ILPOS for the sake of implementation efficiency.

induces a partition on the set of nodes. Apparently this partition gives us the equivalent blocks which may be replaced with a single representative block.

Keeping the analogy of an L program and the properties of its graph, the graph partitioning algorithm in Figure 5.7 may be used to find out a minimal set of blocks which is equivalent to the original program.

{*Given the graph of an L program, find the partitions of the nodes to construct the minimal program*}

1. Construct the initial partition π_0 corresponding to the graph. In π_0 two nodes are in the same block if and only if they are code equivalent.
2. Let $i = 0$
3. Let $i = i + 1$
4. Construct π_i as follows. Two nodes are in the same block of π_i if they are in the same block of the partition π_{i-1} and their immediate successors are in the same block of π_{i-1}
5. If $\pi_i \neq \pi_{i-1}$ goto 3 else goto 6
6. π_i is the final partition.

Figure 5.7: Algorithm for program minimization

Using the algorithm in Figure 5.7, ILPOS constructs the minimal graph corresponding to a given L program. After finding the partition it converts this graph back to a plain L program. In doing so, a representative node is chosen for each block in the partition. All the jumps are arranged to reflect correct label names.

To illustrate the effects of UCR and program minimization, consider the program in Figure 5.8.

The program constructs the list, called result, of those elements of a and b which are at the same index. Notice that the list is reversed. For instance, if a is *'(2 8 7 19 2 32)* and b is *'(1 8 123 19 8 2)* then *(19 8)* is returned. The program has been specialized with respect to the static argument $a \mapsto$ *'(2 3)* . The resulting residual code given in Figure 5.9.

```
read(a, b);

init   : result:='();
         goto start;
start  : if eq(a, '()) goto finish else cont1;
cont1  : if eq(b, '()) goto finish else cont2;
cont2  : if eq(hd(a), hd(b)) goto same else notsame;
same   : result := cons(hd(a), result);
         goto notsame;
notsame: a:=rest(a);
         b:=rest(b);
         goto start;
finish: return result;
```

Figure 5.8: An L program matching two lists

When UCR is applied to this program the assignments in blocks `lpe8, lpe9, lpe5 and lpe12` are removed.

After UCR, minimization is run on the program. The program graph corresponding to the program in Figure 5.9, as constructed by ILPOS, is depicted in Figure 5.10. Notice that the nodes are labeled as i for `lpei`. The nodes having two children are those terminated with a conditional jump, those that are leaf are terminated with returns. There is no node having a single child since all such nodes, which correspond to a direct goto jump, are eliminated by the transition compression algorithm. The edge labels are marked as `T` and `F` in Figures 5.10 and 5.11, they correspond to the `true` and `false` cases of the conditional expression respectively.

The minimizer constructs the following partitions (Using i for `lpe`i):

$$\pi_0 = \{\overline{1,4,5}, \overline{2,10,13}, \overline{3}, \overline{6,9}, \overline{7,11}, \overline{8}, \overline{12}\}$$

$$\pi = \pi_1 = \{\overline{1}, \overline{4}, \overline{5}, \overline{2,10,13}, \overline{3}, \overline{6,9}, \overline{7}, \overline{11}, \overline{8}, \overline{12}\}$$

This identifies that blocks `lpe2, lpe10, lpe13` and `lpe6, lpe9` are identical. Replacing them with a single block and relabeling one gets a residual code having only 10 blocks. The resulting graph is depicted in Figure 5.11.

```
read(b);

lpe1: if eq(b, '()) goto lpe2 else lpe3;
lpe2: return '();
lpe3: if eq('2, hd(b)) goto lpe4 else lpe5;
lpe4: b := rest(b);
      if eq(b, '()) goto lpe6 else lpe7;
lpe6: return '(2);
lpe7: if eq('3, hd(b)) goto lpe8 else lpe9;
lpe8: b := rest(b);
      return '(3 2);
lpe9: b := rest(b);
      return '(2);
lpe5: b := rest(b);
      if eq(b, '()) goto lpe10 else lpe11;
lpe10: return '();
lpe11: if eq('3, hd(b)) goto lpe12 else lpe13;
lpe12: b := rest(b);
       return '(3);
lpe13: b := rest(b);
       return '();
```

Figure 5.9: PPS applied to matcher program

Notice that the nodes 9, 10 and 13 disappear and all links to them are replaced by links to nodes 6, 2 and 2 respectively. Blocks 6-9 and 2-10-13 have merely the same effect on the program. The resulting L program is presented in Figure 5.12. Labels in this program has been rearranged by the canonicalizer described in the subsequent section.

5.3 Linearization and Canonicalization

The last phase that is applied in ILPOS is the linearization and the canonicalization phase. They are not intended to achieve great optimizations but they complete the whole cycle.

5.3.1 Linearization

After minimization is applied to some L code, it may turn out that the labels of a conditional jump statement come out to be the same. This is the case whenever

the upcoming blocks turn out to be equivalent. In such a case, that conditional jump simply resolves into a goto statement, since whatever the decision is, the jump will be to the same block in both cases. The aim of linearization is to check for such jumps and convert them to `goto`'s.

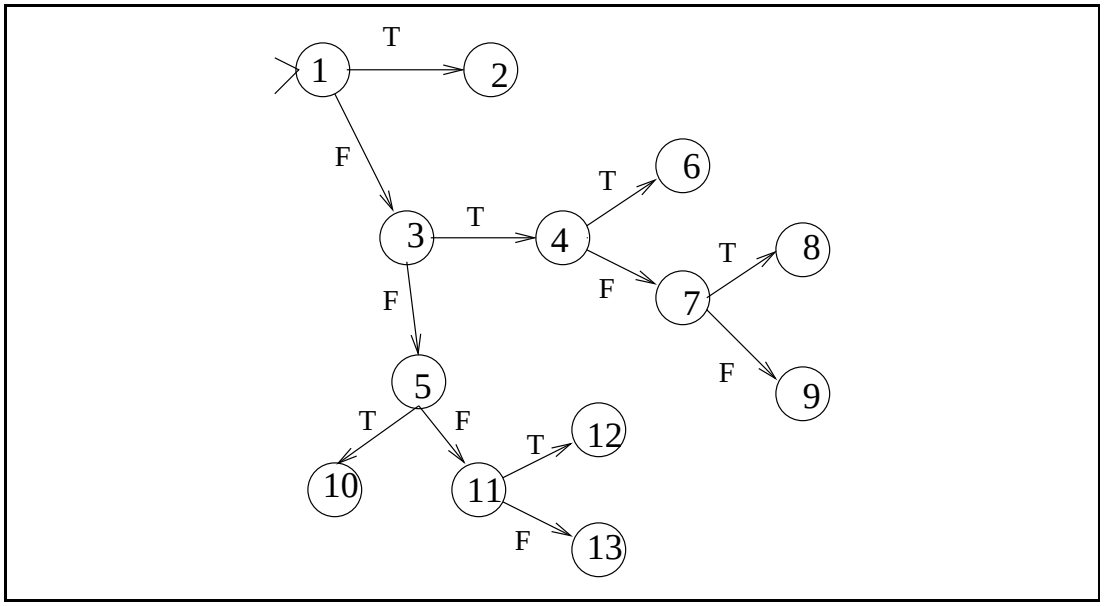

Figure 5.10: Program graph for the specialized program

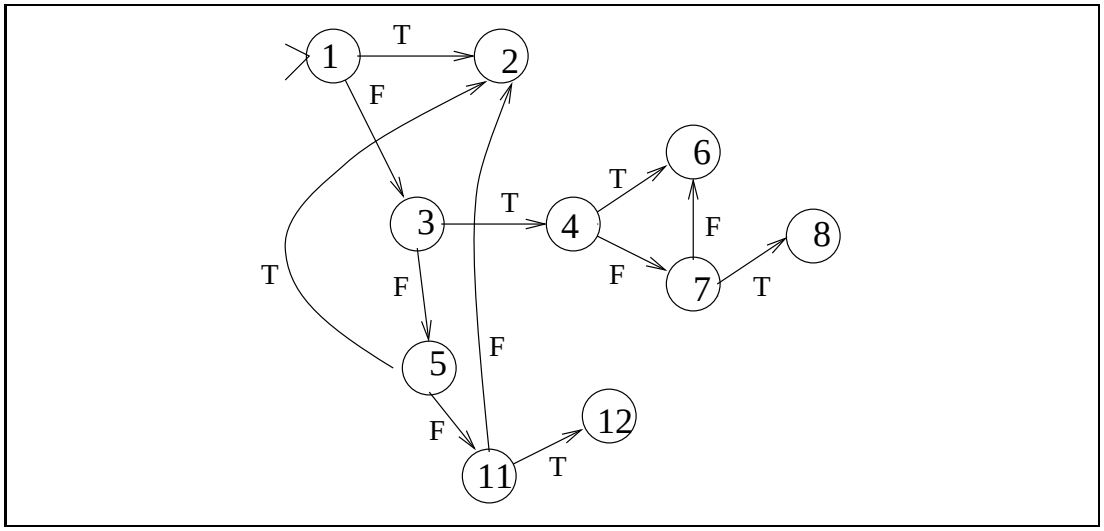

Figure 5.11: Program graph for the minimized program

```
read(b);

lpe0: if eq(b, '()) goto lpe1 else lpe2;
lpe1: return '();
lpe2: if eq('2, hd(b)) goto lpe3 else lpe7;
lpe3: b := rest(b); if eq(b, '()) goto lpe4 else lpe5;
lpe4: return '(2);
lpe5: if eq('3, hd(b)) goto lpe6 else lpe4;
lpe6: return '(3 2);
lpe7: b := rest(b); if eq(b, '()) goto lpe1 else lpe8;
lpe8: if eq('3, hd(b)) goto lpe9 else lpe1;
lpe9: return '(3);
```

Figure 5.12: Final form of matcher program

The algorithm for linearization consists of a simple scan of the blocks checking for such linear jumps. Whenever one is found, it is directly converted to a `goto`. After this is done, a need for transition compression arises as described before. ILPOS applies a reachability analysis which constructs the equivalent code with no `goto`'s in the resulting program.

Notice that, when a code is linearized there is a chance of removing more useless code. This is the case since when a jump is removed, the test expression also gets deleted. This has the effect of changing the *out* set (see section 5.1.2) of a block which changes the set of live variables associated with it. In order to handle this case, ILPOS sends the code back to the UCR routine if linearization achieves any compression. Note that the loop is completed with minimization and linearization again. ILPOS terminates this loop whenever the program linearizer can not find any such conditional statements.

5.3.2 Canonicalization

After ILPOS completes all the phases described above, the time comes to print out the final tailored code to the disk. Before doing so, ILPOS rearranges all the labels that appear in the program to start from zero and increase as the text of the program goes on. This phase is just for formatting purposes and is not

intended to achieve any optimization.

5.4 The need for post optimizations

The examples given in the previous sections for useless code removal and minimization may suggest that they are not too much useful. Although the effectiveness of these optimizations depend highly on the program that has been specialized, we can say that such optimizations are far away from being useless.

It is well known that, PPS can produce many useless code in the specialization of most programs which are beyond some certain level of complexity. Consider applying PPS to some program that has been written in a parametric way, such that the program performs many different tasks with respect to a set of parameters. When specialized with respect to a subset of these parameters, PPS would produce many computations related to other parameters. Although this directly corresponds to what the original program would do in an ordinary run, such computations do not do anything useful on the result of the program. Removing these computations becomes, then, a very useful optimization.

Program minimization generally results in space gains. Clearly a minimal program would occupy less space both in the disk and in the memory. When the language is an interpreted one, as is the case for L, this means small programs to be processed by the interpreters.

As an example on the gains from post optimizations consider the following test performed by ILPOS: An L program for simulating a finite state automaton has been prepared[3]. The program had two inputs, first: the machine to be simulated, second: the input list to be processed. When specialized with PPS technique with respect to a machine having 15 states, the program produced a program having 135 basic blocks. The original program had 19 basic blocks. When the code has been minimized it resulted in a 24 block program saving 111 blocks.

The same test had been repeated with a machine whose states were all accepting. The machine was a 2 state automaton. PPS resulted in a 17 block program which has been reduced to 6 blocks by the minimizer. Then linearization has

[3] The source code of this program can be seen in appendix B.

been applied, which removed some conditional if statement. The useless code remover and minimizer has been rerun automatically by ILPOS on the program giving a 3 blocks program as the final product. The resulting program is given in Figure 5.13.

```
read(input);
lpe0: if eq(input, '()) goto lpe1 else lpe2;
lpe1: return '1;
lpe2: input := rest(input);
      if eq(input, '()) goto lpe1 else lpe2;
```

Figure 5.13: An all states accepting minimal program

As seen the residual just scans through the input string and returns a 1 indicating the acceptance. Notice that, no matter how large is the original machine, the residual program would be same as the one given above, as far as all states are accepting (or rejecting). Although such a machine is practically not useful it helps to demonstrate the need and the power of the post optimizations after the program point specialization on the source programs.

5.5 Computational Complexity of Post Optimizations

In this section, the computational complexities of the algorithms used in the post optimizer will be discussed.

One fundamental operation used in the post optimization procedures is the computation of the successors of a certain block in a program (see Figure 5.3). This algorithm is a simple transitive closure algorithm whose complexity is clearly $O(n)$ where n is the number of blocks in the program. This corresponds to the worst case when all other blocks are reachable from the node under consideration.

The live variable analysis algorithm of Figure 5.4 depends on the generation of in, out, def and use sets of all blocks. The def and use sets are computed just by the analysis of the assignments of the related block while in and out depend

on the corresponding sets of other blocks. It is clear that these sets will contain the names of the variables in the program, so at the worst case each variable name will be included by one of these sets, requiring the analysis of assignments throughout the program. This suggests an algorithm of complexity $O(n^2)$, where n is the number of blocks in the program, as all blocks are searched for. As before, the number of variables has been assumed to be a linear function of the number of blocks in a program.

Once the live variable analysis is completed, all that remains to be done for useless code removal is to scan through the program to remove assignments to useless variables in their corresponding blocks, amounting to an algorithm of complexity $O(n \cdot m)$ where m is the average number of assignments in the program for each block.

The other stage in post optimizations is that of program minimization. Program minimization algorithm of Figure 5.7 uses the concept of code equivalence among two blocks. The definition of code equivalence has been supplied before. According to that usage, the code equivalence can be checked in $O(m)$ time where m is the number of assignments in the blocks to be compared. In each iteration of the minimization algorithm the partition is either refined or stays the same, upon which it terminates. So after at most $n - 1$ iterations the final partition would be found. Incorporating all these ideas, it turns out that the minimization algorithm is essentially a $O(n^2)$ algorithm.

The linearization and canonicalization algorithms consist of scanning the whole program for various information and can be thought of as $O(n)$ algorithms.

CHAPTER 6

TERMINATION OF PARTIAL EVALUATION

The definition of an algorithm includes the constraint that, for all inputs, it should terminate after a finite number of steps. It is known that the partial evaluation, in its pure form as defined here, may not terminate. In this chapter the causes of the problem and the solution implemented in ILPOS is described.

6.1 Infinite partial traces

In describing the PPS technique the concept of the partial traces had been introduced. The examples were chosen such that the graph of the partial traces is always finite for a given program and its static data. However this is not necessarily the case, it may turn out that the graph is infinite.

To clarify the topic, consider the Russian Peasant's Algorithm (RPA) for the multiplication of two numbers, given in Figure 6.1. The algorithm proceeds by halving the value of a and doubling the value of b each time through the loop until a becomes 1. Notice that the value of b is changed under the control of the variable a.

Consider the specialization of the RPA with respect to a static value of a where b is taken to be dynamic. Clearly the residual code will consist of a single block which multiplies b by 2 several times and adds as required in the algorithm. The result turns out to be exactly as expected and depicted when a is 10 in Figure 6.2.

```
# Russian peasant's algorithm for multiplication
read(a,b);

init:    result := '0; goto start;
start:   if odd(a) goto oddCase else process;
oddCase: result := add(result, b);
         goto process;
process: if eq(a, '1) goto finish else goOn;
goOn:    a := intdiv(a, '2); b := mul(b, '2);
         goto start;
finish:  return result;
```

Figure 6.1: Russian Peasant's algorithm

```
read(b);

lpe0:  result := '0;
       b := mul(b, '2);
       result := add(result, b);
       b := mul(b, '2);
       b := mul(b, '2);
       result := add(result, b);
       return result;
```

Figure 6.2: RPA specialized for a

The specialization for RPA works fine for this case. On the other hand consider the specialization of the same algorithm where b is static but a is dynamic. The tree of the partial traces are given in Figure 6.3. As seen in the tree of the partial traces the tree has an infinite number of nodes. The PPS algorithm constructs this graph on the fly and generates code for each block. Since the graph never terminates the process of generating code for the residual will never stop, thus causing the non-termination of partial evaluation.

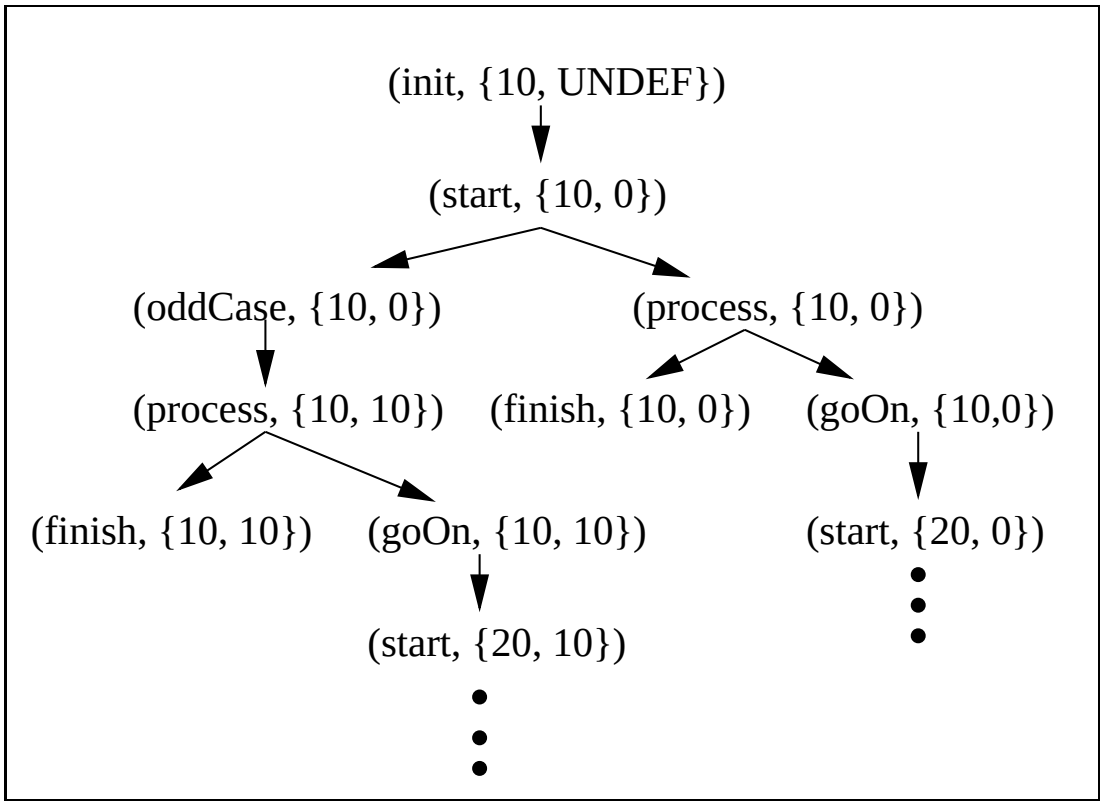

Figure 6.3: Infinite partial traces

In summary, it can be said that partial evaluation will not terminate when the graph of the partial traces is infinite. From the programmers point of view, the problem is caused by the changes in a static variable under the control of a dynamic variable. As seen in the Figure, there will be an infinite number of specialized versions of the same node with respect to the values of the static arguments. Notice the nodes with label `start` in the Figure, they are the roots of the subtrees which will never terminate.

6.2 Coping with non-termination

As mentioned before, to be a real algorithm, the partial evaluator must always terminate. Termination is known to be a difficult problem in the literature of computer science, specifically the problem of halting is known to be undecidable for the general setting of Turing machines. The language L and the paradigm it presents is of no exception and we can not expect to cope with the problem in

its entirety.

There can be two ways of viewing the problem of non-termination: first caused by the nontermination of the original program, second caused by the problems in the binding time analysis.

The first sort of nontermination corresponds exactly to the halting problem. If the original program run with its static data will not terminate, then the partial evaluator that tries to specialize it with respect to those arguments will also not terminate. Notice that, in this case, the non-termination problem of the underlying program causes the non-termination of the partial evaluator. Since the question is undecidable there is not too much to be done and ILPOS does (and can) not do anything for this case. It simply tries to construct the residual without ever stopping[1].

The second sort of problem is the one that the RPA algorithm has. Although there is nothing wrong in the algorithm, the process of partial evaluation does not terminate. There are two main approaches for the handling of this sort of non-termination: improving the binding time analyzer or limiting the residual code size.

Notice that, in the RPA algorithm, the partial evaluator might have detected that the choice of b as a static parameter, while a is dynamic, would cause the termination problem and might reject b as static. The definition for dynamic and static variables employed by ILPOS (given in section 4.4) can not handle these constraints. The implementation of a BTA that can handle such situations requires analysis of the loops involved with respect to the data changes. Such an approach is described in chapter 14 of [14]. This analysis is not easy to perform and depends on the concepts of monotonically *decreasing* and *increasing* properties of the functions called in the program being specialized. The main point is to mark enough number of variables as dynamic in order to ensure the finiteness of the resulting partial trace graph. It is noted that the computation of the minimum set that must be marked as dynamic in order to ensure finiteness is not computable. The approach yields such a set, although not necessarily the

[1] In fact this warns the user about the problem and the partial evaluator works as a "meta-debugger".

minimal, which would ensure the finiteness of the resulting partial trace graph. This method has the disadvantage of not specializing the subject program at all or poor specialization. For instance, the RPA algorithm when only b is dynamic would not be specialized at all. The residual would contain an initial block assigning the initial static value of b to the variable b, and then the original program itself. Due to these reasons and the high cost of implementation and run-time inefficiency, ILPOS uses the second style for handling non-termination: limiting the code size.

6.3 ILPOS termination handler

The approach employed by ILPOS is a more practical one, limiting the code size of the residual to a maximum number of elements that is determined by the number of blocks in the original program. Whenever the specialization starts, the ILPOS PPS module keeps track of the number of blocks that is generated for the residual program. The ILPOS termination handler guarantees that the generation of the residual will eventually stop after this number exceeds the predefined limit[2]. This section explains how this works.

Consider the operation of the PPS module. It constructs the graph of the partial traces on the fly and generates code for each block. At any time during the process of specialization, there are a number of blocks that has been processed and put into the residual code, and a number of blocks that still remains to be specialized. The ILPOS termination handler monitors the number of blocks that has been already generated and gets activated when this number exceeds the current limit allowed. From this point on, the remaining blocks which must be specialized should be processed without producing any new blocks.

Since the generation of new blocks are forbidden from this point on, the termination handler marks all variables as dynamic. This must be done carefully since after this marking, the assignments to the variables that are newly promoted to the dynamic class would generate code in the residual program. In order to ensure the correctness, each such block should contain an initial assignment

[2] Currently this is 20 times the original program size; the factor can be altered at will.

list, assigning the values of the static values to their variables. Notice that, since ILPOS uses transition compression on the fly, the upcoming blocks will be concatenated to the current one with no problems.

By this method, the generation of the new blocks is eventually stopped and the PPS algorithm is guaranteed to terminate, unless there is a termination problem of the first sort described above.

```
read(a);

lpe0:  if odd(a) goto lpe1 else lpe12;
lpe1:  if eq(a, '1) goto lpe2 else lpe3;
lpe2:  return '10;
lpe3:  a := intdiv(a, '2);
       if odd(a) goto lpe4 else lpe11;
lpe4:  if eq(a, '1) goto lpe5 else lpe6;
lpe5:  return '30;
lpe6:  b := '20;
       result := '30;
       a := intdiv(a, '2);
       b := mul(b, '2);
       if odd(a) goto lpe7 else lpe10;
lpe7:  result := add(result, b);
       if eq(a, '1) goto lpe8 else lpe9;
lpe8:  return result;
lpe9:  a := intdiv(a, '2);
       b := mul(b, '2);
       if odd(a) goto lpe7 else lpe10;
lpe10: if eq(a, '1) goto lpe8 else lpe9;
lpe11: b := '20;
       result := '10;
       if eq(a, '1) goto lpe8 else lpe9;
lpe12: b := '10;
       result := '0;
       if eq(a, '1) goto lpe8 else lpe9;
```

Figure 6.4: RPA specialized for b

In order to illustrate the idea, the RPA algorithm has been specialized when b is static. The value of b has been taken to be 10. The resulting residual is given

in Figure 6.4. The maximum code size factor has been set to 1, i.e. activating the termination handler just after 6 blocks (the size of the original program) are generated. This is just for the sake of demonstration.

There are a number of points to be pondered on the residual program. First of all, the number of blocks is 12. This means that when the termination handler is activated there had been 6 blocks already generated and 6 or less blocks waiting to be specialized. The termination process created another 6 originated from the remaining ones. It is not possible to say how many were there and how many were newly generated by the inspection of the residual but, for this specific example, all was in the list waiting to be processed. The second point is that the blocks `lpe6, lpe7, lpe11` and `lpe12` are the blocks where the variables `result` and `b` are switched from static to dynamic. The assignments indicate these transitions clearly. Another point is the general structure of the residual code, it contains some `return` statements, returning numbers, corresponding to the parts that are totally specialized and it has some blocks simulating the computation of the product of the numbers according to the RPA algorithm.

The original specialization, where the code size factor was 20, activated the termination handler after the generation of 120 blocks and the residual contained 165 blocks. The residual had the same properties as the prototype given above.

CHAPTER 7

LOGGING AND GAIN ANALYZER PARTS OF ILPOS

ILPOS is an interactive system for performing partial evaluation and post optimization experiments on the flow-chart language L. The previous discussions described the internals of the ILPOS. In this section the log system and the symbolic gain analyzer is described.

7.1 Logging system of ILPOS

ILPOS has two sorts of reporting systems. The first is the messages that are printed on the screen throughout the stages of computation. At each stage the user is given information on what the system is currently doing and on the status of the partial evaluation and the post optimization process. Along with this data, a log file is produced (with the suffix `peo.log`) containing the information regarding the whole process. It first lists the environment, file names, date etc. Then the statistical information regarding the input program (i.e. number of arguments, blocks, assignments, variables etc.) are given. This information is followed by the static arguments and their initial values. After this comes the messages of the guard system and the termination system, if any. The statistics for the residual code, before the post optimizations are activated, is also presented. The residual code which is not post optimized is dumped to the disk with the

suffix `pe.lp`. Then the post optimizer is activated and the messages related to the useless code remover, minimizer, linearizer and canonicalizer routines are given in the log. At each stage, the operations performed are summarized in the log file. When all the operations are completed, the final residual code is dumped to the disk with the suffix `peo.lp` and the statistics related to this last file is given in the log.

The main purpose of the log file is to keep track of the operations that are performed by ILPOS, it also serves as a notifier of the achievements of the different phases of the entire system on the subject program.

7.2 Symbolic Gain Analysis

The main motivation for partial evaluation and the post optimizations has been defined to be the speed-up that is obtained by these processes. The usual way of measuring the speed-up is to measure the running times of the original and the residual code and then dividing them. The ILPOS L interpreter can be used for this purpose[1]. Another kind of efficiency analysis is defined as the symbolic gain analysis. Rather than measuring the real time spent on executing the programs, one may collect information on the operations that are performed by the program. This approach is more useful when one needs information on the specific achievements. The ILPOS SGA is a utility that automates such analysis.

The main idea is to count the number of jumps, assignments, variable references, decisions and library calls that are made for a particular execution. Notice that these are the basic operations that the interpreter performs for executing the program. The library calls can also be divided into classes, calculating the count for each such call. Once all these counts are available, one can associate a cost to the program easily. This cost is computed as the dot product of a cost vector and the vector formed by these counts. The cost vector is a simple vector of weights that associate importance factors to each operation. The weights in this vector can be altered at will.

The ILPOS Symbolic Gain Analyzer System (SGA) has been designed with

[1] The entry point to the interpreter is the `li` function.

these points in mind. It keeps counts on the above mentioned operations. Function calls are counted specifically for designated functions[2] while all other calls are listed under the heading `other calls`. The SGA system has a feature of turning off and on the statistics collecting features. The special library function `inform_sga` switches these activities on and off. This call is provided in order to collect realistic information on the simulation of data structures not supported by L[3]. In addition to these counts, the system reports the number of blocks, assignments etc. that are present in the residual code produced.

When the ILPOS SGA is activated, it writes its output log to a file with suffix `peo.sga`. The measurements are done on three programs, first the original program with no specialization, second the partially evaluated code, third the code that is both partially evaluated and post optimized. Together with the log file generated, these two files constitute a source of information for the process of specialization and its consequences.

To illustrate the output of the SGA, the RPA algorithm (Figure 6.1) has been specialized with respect to $a = 9987654321123456789$. The variable b has been chosen as dynamic. Then the SGA has been run with the value of $b = 123456789987654321$. The resulting SGA file is given in Figure 7.1.

The numbers in the parenthesis show the index of the entry in the cost vector given in the figure. The first three columns indicate the counts for each file. The `Gain1` column is formed by dividing the values in column 1 by column 2, and `Gain2` column is formed by those of column 1 by column 3. The last column is the difference of the `Gain2` and `Gain1` columns, this is intended to indicate the gains by post optimizations only.

Notice that the post optimizations gain us nothing here, since the residual program has only a single block (i.e. no minimization is possible) and it has only shift and add actions (i.e. no useless code).

---

[2] Designated functions can be user specified, see appendix D.17 for details.

[3] For instance, L does not support random access arrays. Such a data structure may be simulated by a list which enforces sequential search. Thus, when an application uses a list to simulate an array and wants to perform a direct access, it first informs SGA to stop keeping statistics and when it is done (i.e. when linear search is complete) it reactivates the SGA to go on keeping counts. It is also possible to keep count of such calls, they are reported under the heading `intern calls`.

```
Welcome to ILPOS-SGA, Static Gain Analyzer System for ILPOS. (v1.0)

Activated on: Wednesday June 5, 1996, 3:20:57 PM
By user     : erkok
Working dir : /home1/erkok/tez/imp/ilpos/

Symbolic analysis performed on:

Input    : data/RPA/rpa.lp
PE file  : data/RPA/rpa.pe.lp
Residual : data/RPA/rpa.peo.lp

Symbolic Analysis Results: (only the relevant entries are printed.)

                Original   PE Only  Post-Opts     Gain1     Gain2  Opt Gain
                --------   -------  ---------    ------    ------  --------
( 0)    jumps:       225         0          0       inf       inf         0
( 1)  assigns:       160        97         97     1.649     1.649         0
( 2) var refs:       321       130        130     2.469     2.469         0
( 3)decisions:       128         0          0       inf       inf         0
( 4)       eq:         64         0          0       inf       inf         0
(11)      add:         33        33         33         1         1         0
(13)      mul:         63        63         63         1         1         0
(14)      div:         63         0          0       inf       inf         0
(15)      odd:         64         0          0       inf       inf         0
      # BBlks:          6         1          1         -         -         1
      # Asgns:          4        97         97         -         -         1

Cost vector is   : (2 2 1 2 3 2 2 3 3 3 0 3 3 3 3 2 2 2)

Cost of Original : 2144
Cost of PE Only  : 612
Cost of Residual : 612

Gain by PE Only  : 3.503
Gain by Post Opts: 0

Overall Gain is  : 3.503
The improvement  : 71.455 %

SGA completed successfully.
```

Figure 7.1: Result of symbolic gain analysis on RPA

CHAPTER 8

A CASE STUDY: DEFINITE INTEGRALS

This chapter is devoted for a case study on definite integrals. First, the Simpson's technique for evaluating definite integrals is described and an L program for computing them is given. The program is specialized to get the program for computing the erf function. Different combinations of static arguments are tried and the effects are discussed.

8.1 Definite Integration

A *definite integral* is one whose lower and upper bounds are numeric values. For instance,

$$\int_a^b f(x)dx$$

is the integral of the function $f(x)$ between the points a and b, where $a, b \in \mathtt{R}$. By definition, the value of the integral is the area of the region enclosed by the curve $f(x)$ and the lines $y = 0$, $x = a$ and $x = b$ in the cartesian space[1].

A definite integral can be evaluated mainly in two ways. The first one is the *analytical* method, which tries to find the function $F(x)$ whose derivative yields $f(x)$ and then computes the integral by the formula $F(b) - F(a)$. Unfortunately, this is not always possible; there are functions whose integral can not be found analytically. The erf function, defined later in this chapter, is one such example.

[1] The theory of integration is well developed and can be found in any standard undergraduate text, see [21] for instance.

Sometimes the derivation of $F(x)$ may be possible but very expensive. In such cases one considers numerical approximation techniques. A very well known method is that of Simpson's integration method as described in the next section.

8.2 Simpson's Formula

A very standard method for approximating the value of definite integrals is the Simpson's composite rule, which computes the area under the curve by a series of simple area calculations. The main idea is to divide the region into small intervals and sum up the areas of these regions to approximate the area of the entire region. The details of the method can be found in [21]. The formula that is going to be employed in this case study is:

$$\int_a^b f(x)dx \quad \approx \quad \frac{b-a}{6n}\sum_{i=1}^{n}(y_{2i-2} + 4y_{2i-1} + y_{2i})$$

where,

$$y_i = f(x_i), \qquad x_i = a + \frac{b-a}{2n}\ i$$

.

Here n is half of the number of intervals used for the approximation. The accuracy of the result increases as n gets larger. In open form, the formula can be written as:

$$\int_a^b f(x)dx \quad \approx \quad \frac{b-a}{6n}[(y_0+y_{2n})+2(y_2+y_4+\ldots+y_{2n-2})+4(y_1+y_3+\ldots+y_{2n-1})]$$

where y_i and x_i is the same as before. This form is programmed in L as described in the subsequent section.

8.3 Programming the Simpson's Composite Rule in L

The program to compute definite integrals has four different arguments: the lower bound (lb), the upper bound (ub), number of intervals (n) and the function to be integrated (f). The first three arguments are read from the outer world while the function argument is directly coded in the program for the sake of simplicity

and due to the limitations of L. This implies that each new function should be coded into the L program supplied for integration[2].

The program for computing definite integrals using Simpson's rule is given in appendix C.1. The function integrated in that program is $f(x) = 2x$.

8.4 Obtaining the Erf function Integrator Automatically

In this section we want to use program in appendix C.1 to automatically generate a program for computing the erf function defined as:

$$erf(x) \quad = \quad \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

.

The error function (erf) is used in statistics and other sciences, see [21] for details. An interesting property of erf is that it is not elementary, which implies that it can not be written with a finite number of operators and other ordinary functions (i.e. without using the integration sign).

First, we modify the program in appendix C.1 to compute the integral we desire. Recall that the function should be coded directly into the program. The changes to the program are minor. The `computeF` block starts with:

```
computeF: f := exp(sub('0, mul(curX, curX)));
```

This is the only place that needs to be altered for a new function. Apart from the integration, the error function has another constant factor $(2/\sqrt{\pi})$. To take this factor into account we add the following line to the final block:

```
finish: res := mul(res, div('2, sqrt('3.141592653589793));
```

Notice that this last change is only particular to the erf function.

If we inspect the erf function, we see that the lower bound is fixed at 0. Therefore lb is the natural candidate for specialization. We also fix n (half of the number of intervals) at 10 and we obtain the program in appendix C.2. When we look at the residual program we notice that the output simply computes the

[2] Another approach might be coding the function in Scheme and putting it into the library and then calling it.

increment value h and starting from 0 (the lower bound) computes the function at exactly 21 ($2n+1$ with $n = 10$) points. In the meantime it keeps on adding the values to the result in accordance with the Simpson's method. Finally the factor $2/\sqrt{\pi}$ is multiplied and the result is returned. Notice that the numeric value of this factor is computed and placed into the program.

Once we have specialized the integrator program for error function computation, we want to look at the gain we have obtained automatically by the system. The results of the symbolic gain analysis is given in appendix C.3. The upper bound (ub) used for performing the analysis was 100. The static gain analyzer gives an improvement over 60%.

8.5 Other Specializations

The previous choice of the static variables (i.e. lb and n) gave us good results. Remember that the residual code turned out to be linear. Other variations of static variables is possible. One option is to keep lb dynamic while ub and n are static. The resulting residual code is very similar to the previous specialization, we again get a single block program.

Another interesting set of static data is where lb and ub are static and n is dynamic. This residual will be helpful if somebody wants to analyze the precision of the Simpson's method with respect to the number of intervals. When specialized with respect to $lb = 0$, $ub = 10$ and n dynamic, we meet the termination problem. ILPOS generates the program in appendix C.4. The residual code has 226 blocks so not all of it is presented in the appendix. Also, in this case, post-optimizations help us. The useless code remover removed 146 assignments while the code minimizer got rid of 74 blocks. The log file is presented in appendix C.5.

The residual behaves as follows: first it computes the increment, i.e. length of each interval (see block `lpe0`). Although it knows lb and ub, notice that, n determines the length of intervals. After the increment is computed the code for computing the value of the function at the next point is emitted. Once this is done the residual code checks whether it is done, i.e. it compares the current index to the number of points to be computed. If they match, a return statement

is executed (see the jumps of blocks `lpe1`, `lpe2` and so on). Otherwise similar operations take place for the following point. Unfortunately, there is no point that the specialization can stop as n can be arbitrarily high. The code is goes similarly until the previous threshold for code size is reached. Once this point is crossed the termination handler is activated and all the variables are marked dynamic. This stops further specialization successfully. See blocks `lpe154`, `lpe155` and so on for the effect of this process.

The output of the SGA for $n = 20$ is presented in appendix C.6. The reported improvement is about 9% in this case.

8.6 Remarks on the case study

Simpson's rule for evaluating definite integrals is a very well known technique for numeric integration. This case study showed us how we can obtain efficient versions of the programs using this rule provided we know the number of intervals and one of the upper or lower bounds. It has also provided us with a case where termination problem occurs within partial evaluation. The error function has been used to obtain a specialized integrator. Since the formula for erf specifies the lower bound to be 0, it appears as a natural candidate for specialization. Also, when n is dynamic, we have seen that post-optimizations helped us to remove many useless assignments and blocks.

CHAPTER 9

FINAL THOUGHTS AND CONCLUSIONS

Previous chapters described the entire system and the ideas employed in it. This chapter first summarizes the work, including future directions and then concludes the thesis.

9.1 Final remarks and future work

The residual code generated by ILPOS has several characteristics. First of all it does not have any unconditional jump statements, all blocks terminating with such jumps are compressed. There are no static variables remaining in the residual, all static variables are embedded into the specialized blocks and all labels are rearranged and put in order.

With these points in mind, it is possible to think specialization in its two extremes: when there are no static arguments and when all arguments are static.

In the first case, the specialization is meaningful in the sense that all unconditional goto's would be removed from the code. Also, any static computation would be done in the specialization time. This is like filling up a list with values, computing some number which does not depend on the input arguments etc. Also the labels will get arranged.

The other extreme, where all arguments are static, the residual code would resolve into a single block program returning that particular value computed by the whole process, provided that the original program terminates with these input

arguments.

As far as the L language defined here is concerned, ILPOS seems to be mature with its all facilities for partial evaluation using the program point specialization technique, useless code removal using the data flow analysis technique and program minimization based on the theory of sequential machines. However, it is possible to study the partial evaluation with a stronger language. Most importantly the addition of partially static structures to the L language seems to be a fruitful direction for future work. A partially static structure is a composite data structure such as an array or a record. Notice that, for that case, some part of the data would be known at the specialization time and some part would be dynamic, hence the name partially static. It would also be of interest to add sub-flow charts to the language, which is a rather easy extension since every L language program can be thought of as a sub-flow chart that can be accessed from a main one.

Another point of extension is that of using polyvariant divisions. ILPOS uses a monovariant division algorithm assuming that every variable is either static or dynamic in its entire life time. Another approach would be to make the division for each block in the original program so that the dynamic and static properties belong to the blocks rather than the entire program. This idea is based on the assumption that the same variable can be used for different purposes in the program. Although theoretically this is feasible, practically it may not have much significance since this is not considered to be a good programming style.

9.2 Conclusions

In this work, a partial evaluator and a post optimizer system for a flow chart language has been defined and its implementation has been described. The appendix contains the code for the whole system implemented in the Scheme language.

The language L, being a flow chart language, has been chosen as the object language for the operations. Although very simple in its syntax and semantics, it proves to be a real programming language since it can simulate any Turing machine ignoring the memory limitations of the underlying computer. The simplicity

of the language allowed us to study both partial evaluation and post optimizations on the programs in a compact way.

The program point specialization technique with a monovariant division algorithm has been used to construct the partial evaluator part of ILPOS. It has been seen that the success of partial evaluation depends on the nature of the program being specialized and the arguments that are static.

It has been shown that incompletely specified programs cause problems for the partial evaluation process. Such programs are code portions that are prepared with several assumptions in mind, ignoring the generality of the algorithm. Although such codes are perfectly acceptable, the partial evaluation of such programs cause problems in the program point specialization technique. In order to remedy these problems, the concept of guards have been developed and integrated into the system. It has been pointed out that these guards both act as a meta-debugger for the cases when the incomplete specification is by the mistake of the programmer and as a tool for enabling the partial evaluation of such programs.

Global data flow analysis techniques have been implemented for the live variable analysis of the programs. This has been used in the removal of the unnecessary computations that are present in the partially evaluated code. It has also been shown that the machine minimization algorithms of the classical theory of sequential machines can be used to minimize the number of blocks that are present in an L program. This idea has been elaborated and applied in ILPOS. Another sort of optimization, called linearization, which aims at converting conditional jumps into unconditional ones has been described and implemented. Notice that linearization is likely after minimization takes place. All these three techniques constitute the post-optimizer part of ILPOS.

The termination problem of the partial evaluation has been studied and the approaches to the problem has been described. The idea of limiting the code size for the residual programs has been described and implemented as a part of ILPOS. It has been reminded that the problem of termination is undecidable for L and the solution employed by ILPOS uses a practical approximation idea by

limiting the output size. The problems associated with this technique and the solutions has also been described.

The computation of definite integrals using Simpson's rule has been investigated. Partial evaluation has been used successfully to generate a program to compute the error function automatically from the generic program. Together with this, various combinations of static and dynamic arguments have been considered. It has been observed that different combinations of static arguments yield significantly different behaviour of the partial evaluator.

Apart from these, ILPOS has a lexical analyzer, a parser and an interpreter for the L language. A logging system has been implemented to accompany the report generation for the whole process. Also supplied is a symbolic gain analyzer system that can be used to measure the gain in efficiency that is obtained by the partial evaluation and post optimization processes.

The contributions of this thesis are as follows: First, an experimental partial evaluation and post optimization environment with all of its supporting utilities is implemented from scratch. This work combined the concepts of partial evaluation and post optimization techniques into a single package. To the best of the authors knowledge, this is the first work that implements these ideas together. The concept of incomplete specifications and the method for handling them through the use of guards is also new. Another contribution is the application of finite automata minimization algorithm to L programs. Termination handling techniques, although quite straightforward, are also developed within this study. The symbolic gain analysis and the related utilities are also supporting ideas for the entire system.

As a conclusion, it can be said that, program specialization is a promising area for run time improvement of programs. The post optimization techniques can also improve the residual obtained by specialization by removing unnecessary computations and equivalent blocks. The system presented in this work aims to be an experimental system to study these techniques.

REFERENCES

[1] S. Adams et. al., *MIT Scheme User's Manual*, Massachusetts Institute of Technology, 1995.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Reading, MA: Addison-Wesley, 1986.

[3] D. Castle, *A Uniform Approach for Compile-Time and Run-Time Specialization*, LNCS, Lecture Notes in Computer Science, No. 1110, Springer-Verlag, pp. 54-72.

[4] W. Clinger et. al., *Revised*^{4} *Report on the Algorithmic Language Scheme*, 1991.

[5] C. Consel, *Program Adaptation based on Program Transformation*, ACM Computing Surveys, 28A(4), December 1996.

[6] C. Consel, *A tour of Schism: a partial evaluation system for higher-order applicative languages*, ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 66-77, 1993.

[7] C. Consel, *Polyvariant binding-time analysis for higher-order, applicative languages*, In ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 145-154, 1993.

[8] C. Consel and S. C. Khoo, *On-line and Off-line Partial Evaluation: Semantic Specifications and Correctness Proofs*, Journal of Functional Programming, 5(4), pp. 461-500.

[9] C. Consel, C. Pu, and J. Walpole, *Incremental specialization: the key to high performance, modularity and portability in operating systems*, in ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 44-46, 1993.

[10] J. Dean, C. Chambers and D. Grove, *Identifying Profitable Specialization in Object-Oriented Languages*, in PEPM 1994, Partial Evaluation and Program Manipulation Conference, pp. 85-96.

[11] D. R. Engler, W. C. Hsieh and M. F. Kaashoek, *'C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation*, POPL 1996, Conference on the Principles of Programming Languages.

[12] L. Hornof, J. Noyé, *Accurate Binding-Time Analysis for Imperative Languages: Flow, Context, and Return Sensitivity*, ACM SIGPLAN Conference in Partial Evaluation and Semantics-Based Program Manipulation, June 1997.

[13] N. D. Jones, *An Introduction to Partial Evaluation*, ACM Computing Surveys, Vol. 28, No. 3, September 1996, pp. 480-503.

[14] N. D. Jones, C. K. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.

[15] S. C. Kleene, *Introduction to Metamathematics*, Princeton, NJ: D. van Nostrand, 1952.

[16] Z. Kohavi, *Switching and Finite Automata Theory*, 2nd. ed., McGraw-Hill, 1978.

[17] C. L. Liu, *Elements of Discrete Mathematics*, Reading, MA: McGraw-Hill, 1985.

[18] S. S. Muchnick and N. D. Jones (editors), *Program Flow Analysis*, Reading, Prentice Hall, 1981.

[19] G. Muller, E. N. Volanschi and R. Marlet, *Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol*, ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation, June 1997.

[20] F. Noél, L. Hornof, *Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Study*, Research Report 1065. IRISA, November 1996.

[21] R. A. Silverman, *Calculus with Analytic Geometry*, Prentice Hall, 1985.

[22] E. N. Volanschi, G. Muller, and C. Consel, *Safe Operating System Specialization: The RPC Case Study*, in First Annual Workshop on Compiler Support for System Software, Tucson, Arizona, February 1996.

APPENDIX A

ILPOS USER MANUAL

ILPOS can run on any computer that has an R4RS (see [4]) compliant Scheme interpreter installed. The system has been written and tested on a UNIX system with MIT Scheme, Release 7.3.0 (beta) (see [1]).

To start ILPOS, one must start the Scheme interpreter and load the file *ilpos.s* (see appendix D.2). It will automatically load all other files required for the whole system. Alternatively one can run the shell script given in appendix D.1, prepared for Unix systems. The system will automatically be loaded by this script. When the Scheme prompt appears, one can interact with ILPOS through the following functions:

lpe: (*lpe* *inputFileName* *outputFileName* *commentFileName*)

This function is the entry point to the partial evaluator. The first argument, *inputFileName* is the object L program, the suffix `.lp` is appended to this name. Second and the third arguments to the function *lpe* are optional. If they are not present their names are formed by appending `.peo.lp` and `.peo.log` respectively. The second argument *outputFileName* is the name of the file where the final result will be written and *commentFileName* is the name of the file where all the log messages go. Although not specified by any arguments a file with suffix `pe.lp` is created containing the residual code that is not post optimized.

li: (li $inputFile$)

This function is the entry point to the interpreter. The program in $inputFile$ is interpreted by the L interpreter.

sga: (sga $inputFile$)

This function is the entry point to the symblic gain analyzer. It performs the symbolic gain analysis on the original, residual and post optimized residual programs. The output is written to the file with suffix `.peo.sga`. It assumes that the residual and the post optimized residual programs are stored in the same place as the original program and they all begin with the name $inputFile$. The suffixes of these programs must be `.lp`, `.pe.lp` and `.peo.lp` as created by the lpe function.

Apart from these main ones, ILPOS has many other functions as can be seen in appendix D. Some of the functions that might be of interest to the user are listed below:

readPgm: Read an L program and lexically analyze it. Returns a list of (token, lexeme) pairs to the caller. Technically it is the lexical analyzer. See appendix D.3.

parseL: Parse an L program represented as a (token, lexeme) pair (as returned by $readPgm$). It returns the abstract syntax tree (AST) of corresponding to the program. See appendix D.4.

unparseL: Reverse of the $parseL$. Given an AST of an L program dump it to the disk as a plain text file. See appendix D.5.

run: AST counterpart of the li function. Runs a given AST corresponding to an L program. See appendix D.6.

ucr: Perform useless code removal on an AST. See appendix D.13.

minAst: Perform program minimization on an AST. See appendix D.15.

linAst: Perform program linearization on an AST. See appendix D.16.

bta: Perform binding time analysis on an AST. See appendix D.11.

pe: Perform partial evaluation on an AST. See appendix D.12.

Notice that all the operations are carried on the related AST and each fundamental operation returns the modified AST. This enables one to use them in cascade. For example one may prefer to run the useless code remover and the minimizer on some program and then dump it back to the disk. Assuming the name of the program is `test` one may achieve this operation by typing:

```
(unparseL (minAst (ucr (parseL (readPgm "test")))) "test.out")
```

The output will be placed in file `test.out`.

APPENDIX B

FINITE AUTOMATON SIMULATOR IN L

This appendix contains the source code of the finite automaton simulator in L that has been discussed in section 5.4.

```
# Finite state machine interpreter.
# some data:
# static args: ("machine" "finals" "startState")
# machine:
#   ((empty a id) (empty b id) (empty 1 num)
#    (empty 2 num) (empty 3 num) (empty + sign) (empty - sign)
#    (num 1 num) (num 2 num) (num 3 num)
#    (id a id) (id b id) (id 1 id) (id 2 id) (id 3 id))
# finals:
#   (id num sign)
# start:
#   empty
# some inputs:
#   (a b 1 2 + 2 3 2 - b a a b 3 2 + 1 2)
#   (a b a b 1 2 a b 1 2 a b 1 3 2)
read(machine, finals, startState, input);

start: curState   := startState;
       results    := '();
       matchSoFar := '();
       matchPoint := 'NOMATCH;
       goto analyze0;

analyze0: finalsIter := finals;
          goto analyze1;

analyze1: if eq(finalsIter, '()) goto proceed else lookMore;

lookMore: if eq(curState, hd(finalsIter)) goto recordMatch else look2;

look2: finalsIter := rest(finalsIter);
       goto analyze1;

recordMatch: matchPoint := list(curState, matchSoFar, input);
```

```
            goto proceed;

proceed: if eq(input, '()) goto finish else move;

finish: if eq(matchPoint, 'NOMATCH) goto finishNoAdd else finishWithLast;

finishWithLast: matchSoFar := hd(rest(matchPoint));
                newComers  := list(hd(matchPoint), matchSoFar);
                results    := append(results, list(newComers));
                goto finishNoAdd;

finishNoAdd: return results;

move: curSymbol   := hd(input);
      input       := rest(input);
      matchSoFar  := append(matchSoFar, list(curSymbol));
      machineIter := machine;
      goto transition;

transition: if eq(machineIter, '()) goto notFound else tr1;

notFound: curState := 'NOTRANSITION;
          goto backtrack;

tr1: if eq(hd(hd(machineIter)), curState) goto checkMore else tr2;

checkMore: if eq(hd(rest(hd(machineIter))), curSymbol) goto found else tr2;

tr2: machineIter := rest(machineIter);
     goto transition;

found: curState := hd(rest(rest(hd(machineIter))));
       goto analyze0;

backtrack: if eq(matchPoint, 'NOMATCH) goto finish else back2;

back2    : matchSoFar := hd(rest(matchPoint));
           input      := hd(rest(rest(matchPoint)));
           newComers  := list(hd(matchPoint), matchSoFar);
           results    := append(results, list(newComers));
           matchSoFar := '();
           matchPoint := 'NOMATCH;
           curState   := startState;
           goto analyze0;
```

APPENDIX C

SIMPSON'S RULE IN L

This appendix contains the source code of the definite integral evaluator programs in L.

C.1 The Definite Integrator

The first file is the integrator for $f(x) = 2x$ function.

```
# Evaluation of definite integrals using Simpson's Composite Rule
# This program evaluates the integral:
#           ub
#           /
#           | f(x) dx,    where f(x) is integrated into the code
#           /
#          lb
# lb and ub are the bounds of integration.
# n is half of the number of intervals.
# Static inputs: lb, n  Dynamic input: ub

read(lb, ub, n);

# Compute the initial values of integration parameters.
init: h := div(sub(ub, lb), mul('2, n));
      i := '0;
      res := '0;           # res will hold the result
      k := div(h, '3);
      last1 := mul('2, n);
      last := add(last1, '1);
      goto integrate;

integrate: if eq(i, last) goto finish else goOn;

goOn: curX := add(mul(h, i), lb);
      goto cc1;
```

```
cc1: if eq(i, '0) goto coef1 else cc2;

cc2: if eq(i, last1) goto coef1 else cc3;

cc3: if odd(i) goto coef4 else coef2;

coef1: curCoef := '1;
       goto computeF;

coef2: curCoef := '2;
       goto computeF;

coef4: curCoef := '4;
       goto computeF;

# now compute f at curX: currently f(x) is 2x
computeF: f := mul('2, curX);
# don't alter the rest of this block
          res := add(mul(curCoef, f), res);
          i := add(i, '1);
          goto integrate;

# return the computed result:
finish: return mul(k, res);
```

C.2 Specialization for the Error Function

The following program is automatically generated by ILPOS using the previous section's program for the error function. The static arguments were $lb = 0$ and $n = 10$.

```
read(ub);

lpe0:
        h := div(sub(ub, '0), '20);
        res := '0;
        k := div(h, '3);
        curX := add(mul(h, '0), '0);
        f := exp(sub('0, mul(curX, curX)));
        res := add(mul('1, f), res);
        curX := add(mul(h, '1), '0);
        f := exp(sub('0, mul(curX, curX)));
        res := add(mul('4, f), res);
        curX := add(mul(h, '2), '0);
        f := exp(sub('0, mul(curX, curX)));
        res := add(mul('2, f), res);
        curX := add(mul(h, '3), '0);
        f := exp(sub('0, mul(curX, curX)));
        res := add(mul('4, f), res);
        curX := add(mul(h, '4), '0);
        f := exp(sub('0, mul(curX, curX)));
        res := add(mul('2, f), res);
        curX := add(mul(h, '5), '0);
        f := exp(sub('0, mul(curX, curX)));
        res := add(mul('4, f), res);
```

```
curX := add(mul(h, '6), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '7), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '8), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '9), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '10), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '11), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '12), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '13), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '14), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '15), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '16), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '17), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '18), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('2, f), res);
curX := add(mul(h, '19), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('4, f), res);
curX := add(mul(h, '20), '0);
f := exp(sub('0, mul(curX, curX)));
res := add(mul('1, f), res);
res := mul(res, '1.1283791670955126);
return mul(k, res);
```

C.3 SGA of the Error Function

The following is the output of the ILPOS-SGA routine. The upper bound was selected to be 100.

```
Welcome to ILPOS-SGA, Static Gain Analyzer System for ILPOS. (v1.0)

Activated on: Monday June 16, 1997, 5:47:17 PM
By user     : erkok
Working dir : /home/others/staff/erkok/tez/sonson/odtu/ilpos/SRC/

Symbolic analysis performed on:

Input    : data/SIMPSON/serrf.lp
PE file  : data/SIMPSON/serrf.pe.lp
Residual : data/SIMPSON/serrf.peo.lp

Symbolic Analysis Results: (only the relevant entries are printed.)

                Original   PE Only  Post-Opts     Gain1     Gain2  Opt Gain
                --------   -------  ---------    ------    ------  --------
( 0)    jumps:       146         0          0       inf       inf         0
( 1)  assigns:       112        67         67     1.672     1.672         0
( 2) var refs:       322       110        110     2.927     2.927         0
( 3)decisions:        82         0          0       inf       inf         0
( 4)       eq:        63         0          0       inf       inf         0
(11)      add:        64        42         42     1.524     1.524         0
(12)      sub:        22        22         22         1         1         0
(13)      mul:        67        65         65     1.031     1.031         0
(14)      div:         3         2          2       1.5       1.5         0
(15)      odd:        19         0          0       inf       inf         0
(17)   others:        22        21         21     1.048     1.048         0
      # BBlks:        11         1          1         -         -         1
      # Asgns:        14        67         67         -         -         1

Cost vector is   : (2 2 1 2 3 2 2 3 3 3 0 3 3 3 3 2 2 2)

Cost of Original : 1741
Cost of PE Only  : 679
Cost of Residual : 679

Gain by PE Only  : 2.564
Gain by Post Opts: 0

Overall Gain is  : 2.564
The improvement  : 60.999 %

SGA completed successfully.
```

C.4 A Non-terminating specialization

This appendix contains the specialization of the definite integral evaluator with respect to $lb = 0$, $ub = 10$. The resulting residual has 226 blocks hence not all of it is represented here. The following portion of it should give an idea about the structure of the residual code.

```
read(n);

lpe0:
        h := div('10, mul('2, n));
        res := '0;
        k := div(h, '3);
        last1 := mul('2, n);
        last := add(last1, '1);
        if eq('0, last) goto lpe1 else lpe2;

lpe1:
        return mul(k, res);

lpe2:
        curX := add(mul(h, '0), '0);
        f := mul('2, curX);
        res := add(mul('1, f), res);
        if eq('1, last) goto lpe1 else lpe3;

lpe3:
        curX := add(mul(h, '1), '0);
        if eq('1, last1) goto lpe4 else lpe225;
```

... code deleted in between ...

```
lpe33:
        curX := add(mul(h, '16), '0);
        if eq('16, last1) goto lpe34 else lpe210;

lpe34:
        f := mul('2, curX);
        res := add(mul('1, f), res);
        if eq('17, last) goto lpe1 else lpe35;

lpe35:
        curX := add(mul(h, '17), '0);
        if eq('17, last1) goto lpe36 else lpe209;
```

... code deleted in between ...

```
lpe154:
        lb := '0;
        i := '72;
        if odd(i) goto lpe152 else lpe153;

lpe155:
        lb := '0;
        i := '71;
        if odd(i) goto lpe152 else lpe153;

lpe156:
        lb := '0;
        i := '70;
        if odd(i) goto lpe152 else lpe153;
```

... code deleted in between ...

```
lpe224:
        lb := '0;
        i := '2;
        if odd(i) goto lpe152 else lpe153;
lpe225:
        lb := '0;
        i := '1;
        if odd(i) goto lpe152 else lpe153;
```

C.5 Log file for specialization

The following is the log file generated for the previous specialization.

```
Welcome to ILPOS, Integrated L Partial Evaluator and Optimizer System. (v1.0)

Activated on: Monday June 16, 1997, 7:45:35 PM
By user     : erkok
Working dir : /home/others/staff/erkok/tez/sonson/odtu/ilpos/SRC/
Input file  : data/SIMPSON/s2xN.lp
Output file : data/SIMPSON/s2xN.peo.lp
Pure PE file: data/SIMPSON/s2xN.pe.lp
Log file    : data/SIMPSON/s2xN.peo.log

Input has:
   3 formal arg(s)
   11 basic block(s)
   13 assignment(s)
   12 program var(s) : (ub n last h lb last1 curX curCoef f i k res)
   8 dynamic var(s) : (res f k last curX h last1 n)
   2 static arg(s)  : (lb ub)
   4 static var(s)  : (ub lb curCoef i)

Program is specialized with respect to:

   lb  <-  0
   ub  <-  10

** Termination Handler Notice:
   The residual code size exceeded 220 blocks
   Marking all variables as dynamic and stopping further partial evaluation.

Partial Evaluation Completed, residual code has:
   1 formal arg(s)
   300 basic block(s)

Dead Code Removal Completed, Number of Assignments Removed: 146

Minimization Completed:
Minimal code has      : 226 block(s)
Minimization saved us : 74 block(s)

Linearization completed, DCR+MIN+LIN loop terminates.


Totally 74 block(s) (out of 300) has/have been
```

```
saved by the optimizations after partial evaluation.

Totally 146 assignment(s) (out of 530) has/have been
removed by the dead code remover.

Final residual program has 226 block(s) and 384 assignment(s).

ILPOS completed successfully.
```

C.6 SGA for Non-terminating specialization

The following is the output of the SGA routine for $n = 20$.

```
Welcome to ILPOS-SGA, Static Gain Analyzer System for ILPOS. (v1.0)

Activated on: Monday June 16, 1997, 8:25:52 PM
By user     : erkok
Working dir : /home/others/staff/erkok/tez/sonson/odtu/ilpos/SRC/

Symbolic analysis performed on:

Input    : data/SIMPSON/s2xN.lp
PE file  : data/SIMPSON/s2xN.pe.lp
Residual : data/SIMPSON/s2xN.peo.lp

Symbolic Analysis Results: (only the relevant entries are printed.)
```

	Original	PE Only	Post-Opts	Gain1	Gain2	Opt Gain
(0) jumps:	286	160	160	1.788	1.788	0
(1) assigns:	211	212	210	0.995	1.005	0.009
(2) var refs:	580	567	567	1.023	1.023	0
(3)decisions:	162	160	160	1.012	1.012	0
(4) eq:	123	121	121	1.017	1.017	0
(11) add:	124	123	123	1.008	1.008	0
(12) sub:	1	0	0	inf	inf	0
(13) mul:	126	126	126	1	1	0
(14) div:	2	2	2	1	1	0
(15) odd:	39	39	39	1	1	0
# BBlks:	11	300	226	-	-	1.327
# Asgns:	13	530	384	-	-	1.38

```
Cost vector is   : (2 2 1 2 3 2 2 3 3 3 0 3 3 3 3 2 2 2)

Cost of Original : 3104
Cost of PE Only  : 2825
Cost of Residual : 2821

Gain by PE Only  : 1.099
Gain by Post Opts: 0.002

Overall Gain is  : 1.1
The improvement  : 9.117 %

SGA completed successfully.
```

APPENDIX D

SOURCE CODE OF ILPOS

This appendix includes descriptions and the Scheme code of the whole system. The ILPOS source has been divided into 17 files, each file is presented in a section.

The source code is electronically available upon request. For information please contact {erkok,oguztuzn}@ceng.metu.edu.tr. You can also visit the web page http://www.ceng.metu.edu.tr/~erkok for an online version of this thesis and the source code.

D.1 ILPOS loader: ilpos

```
# change the following variable to your MIT-Scheme interpreter
MITSCHEME=scheme
echo Starting ILPOS, Integrated L Partial Evaluator and Optimizer System, v1.0..
echo
$MITSCHEME -load ilpos.s
echo
echo "ILPOS terminated.."
```

D.2 ILPOS driver: ilpos.s

```
;;;----------------------------------------------------------------
;;;
;;; File ilpos: ILPOS: Integrated L Partial evaluator and
;;;                    Optimizer System
;;;
;;; ILPOS is an experimental system for studying partial evaluation and
;;; optimization issues. Partial evaluation is a program transformation
;;; technique that makes use of the abstract interpretation where all
;;; static information is used in the specialization time to produce an
;;; efficient but semantically equivalent residual code. The optimizations
```

```
;;; that are applied to the partially evaluated code includes dead code
;;; removal, block minimization and linearization. Dead code removal
;;; has to do with removing assignments that assign values to variables
;;; that are dead, i.e. whose values are not used after that point. Block
;;; minimization exploits the equivalent code concept by creating
;;; an equivalence relation among all basic blocks of the program. Clearly
;;; each partition defines a unique operational unit and can be represented
;;; by a single block. This results in the save of blocks thus achieving
;;; space optimization. Linearization is concerned with goto compression,
;;; an important property of the residual code: The produced code does
;;; not have any unconditional goto statements all of them are compressed
;;; through the translation process.
;;;
;;; ILPOS works on a language called L, which is a flow chart language
;;; that can express any algorithm in principle. The data structures
;;; allowes is constants and lists of any valid data structure. i.e.
;;; nested lists (to arbitrary depth) are allowed.
;;;
;;;----------------------------------------------------------------------

;;; Load the required files:
(load "lexer.s")
(load "parser.s")
(load "unparser.s")
(load "guards.s")
(load "llibrary.s")
(load "interpreter.s")
(load "symbSpeedUp.s")
(load "bta.s")
(load "lpeval.s")
(load "dcr.s")
(load "minimize.s")
(load "linearize.s")
(load "aux.s")
(load "util.s")
(load "setOperations.s")

;;; what we have:
;;; readPgm  : read disk file, display tokens (i.e. lexemes..)
;;; parseL   : parse what's read by readPgm, returns ast
;;; unparseL : unparse the ast and flush to disk.
;;; run      : run an ast
;;; dcr      : dead code remover
;;; minAst   : minimizer
;;; linAst   : linearize the code
;;; li       : interpret an L program in a file
;;; lCV      : interpret an L program in a file and construct the
;;;            cost vector associated with that particular run.
;;; bta      : binding time analyzer
;;; pe       : partial evaluator, work on ast
;;; lpe      : partial evaluator, work on files

;;; basic use of the partial evaluator:
;;;
;;;             (lpe inputFileName outputFileName commentFileName)
;;;
```

```
;;; which reads the program in file inputFileName and performs
;;; all operations and writes the residual code to outputFileName.
;;; commentFileName is used for producing statistics on the program
;;; and the translation process.

(define commentPort 'err)
;;; entry to whole system: ILPOS: integrated L partial evaluator and optimizer.
(define ilpos
  (lambda fnames
          (let ((noOfFiles (length fnames)))
               (case noOfFiles
                     (1 (lpe (string-append (car fnames) ".lp")
                             (string-append (car fnames) ".peo.lp")
                             (string-append (car fnames) ".pe.lp")
                             (string-append (car fnames) ".peo.log")))
                     (2 (lpe (string-append (car fnames) ".lp")
                             (string-append (cadr fnames) ".peo.lp")
                             (string-append (cadr fnames) ".pe.lp")
                             (string-append (cadr fnames) ".peo.log")))
                     (else (error "ILPOS: illegal count of arguments."))))))
```

D.3 Lexical Analyzer for L: lexer.s

```
;;;-------------------------------------------------------------------
;;;
;;; File lexer.s: contains functions for lexical analysis
;;;
;;; The lexical analyzer reads the disk file and performs lexical
;;; analysis. all whiteSpace and comments are removed.  note that
;;; we allow 333aaa as an identifier. All composite lists that
;;; may appear in the program is handled through these functions.
;;;
;;;-------------------------------------------------------------------
(define whiteSpace?
  (lambda (char)
          (member char '(#\tab #\space #\newline #\#))))

(define punctuation?
  (lambda (char)
          (member char '(#\( #\) #\; #\, #\:))))

(define quotedLit?
  (lambda (char)
          (equal? char #\')))

;;; Function readPgm: reads the named file, and sends it as a list:
(define readPgm
  (lambda (fileName)
          (let* ((port (open-input-file fileName))
                 (pgm  (readFile port)))
                (close-input-port port)
                pgm)))

;;; Function readFile: given the port no, reads the program into a list,
;;; effectively it calls the lexer to collect the symbols.
```

```
(define readFile
  (lambda (port)
          (let ((nextObj (lexer port)))
               (if (equal? nextObj 'done)
                   ()
                   (cons nextObj (readFile port))))))

;;; Function lexer: lexical analyzer for L-programs.
;;; it returns the next object from the port, or something
;;; that makes eof-object? true, in case file ends.
(define lexer
  (lambda (port)
          (if (eof-object? (discardWs port))
              'done
              (getNext port "" 'nothing))))

;;; Function discardWs: discard white space from the input:
(define discardWs
  (lambda (port)
          (let ((lookAhead (peek-char port)))
               (if (eof-object? lookAhead)
                   lookAhead
                   (if (whiteSpace? lookAhead)
                       (if (equal? lookAhead #\#)
                           (begin (read-char port)
                                  (discardComment port)
                                  (discardWs port))
                           (begin (read-char port) (discardWs port)))
                       'done)))))

;;; Function discardComment: skip comments: skip until the next newline
(define discardComment
  (lambda (port)
          (let ((lookAhead (peek-char port)))
               (if (eof-object? lookAhead)
                   lookAhead
                   (if (equal? lookAhead #\newline)
                       (begin (read-char port) 'done)
                       (begin (read-char port) (discardComment port)))))))

;;; Function getNext: collect the next lexeme and return it:
(define getNext
  (lambda (port prev what)
          (let ((lookAhead (peek-char port)))
               (cond ((eof-object? lookAhead) prev)
                     ((whiteSpace? lookAhead) prev)
                     ((quotedLit? lookAhead) (collectConst port "" 0))
                     ((punctuation? lookAhead)
                      (if (equal? what 'nothing)
                          (if (equal? lookAhead #\:)
                              (catchAssign (char->string (read-char port))
                                           port)
                              (char->string (read-char port)))
                          prev))
                     (else
                      (getNext port
```

```
                              (string-append prev
                                             (char->string (read-char port)))
                              'something))))))

;;; Function catchAssign: look if we have :=
(define catchAssign
  (lambda (prev port)
          (let ((lookAhead (peek-char port)))
               (if (equal? lookAhead #\=)
                   (string-append prev (char->string (read-char port)))
                   prev))))

;;; Function collectConst: grasp quoted literals:
(define collectConst
  (lambda (port prev balance)
          (let ((lookAhead (peek-char port)))
               (cond ((eof-object? lookAhead) prev)
                     ((or (whiteSpace? lookAhead)
                          (equal? lookAhead #\;)
                          (equal? lookAhead #\,))
                      (if (equal? balance 0)
                          prev
                          (collectConst
                           port (string-append
                                 prev (char->string (read-char port)))
                           balance)))
                     ((equal? lookAhead #\()
                          (collectConst
                           port (string-append
                                 prev (char->string (read-char port)))
                           (+ balance 1)))
                     ((equal? lookAhead #\))
                          (if (= balance 1)
                              (string-append prev
                                             (char->string (read-char port)))
                              (if (= balance 0)
                                  prev
                                  (collectConst
                                    port (string-append
                                      prev (char->string (read-char port)))
                                   (- balance 1)))))
                     (else (collectConst
                            port (string-append
                                  prev (char->string (read-char port)))
                            balance))))))
```

D.4 Parser for L: parser.s

```
;;;-------------------------------------------------------------------
;;;
;;; File parser.s: contains functions for the parser.
;;;
;;; The parser for L has been implemented as a variant of recursive
;;; descent parsing technique. The resulting parse tree is a mit-scheme
;;; structure (called pgm) and all the parse tree is represented as a
```

```
;;; structure of structures kind of data structure. These structures
;;; can be seen below.
;;;
;;;-----------------------------------------------------------------

;;; constitutients of the parse tree:
(define-structure pgm readBlk basicBlks)
(define-structure basicBlk lbl assigns jump)
(define-structure assign var expr)
(define-structure const type val) ; type: list | id (no numbers!)
(define-structure varRef var)
(define-structure app rator rands)
(define-structure goto lbl)
(define-structure condJump expr lbl1 lbl2)
(define-structure return exp)

;;; Parsing stuff:
(define parseL
  (lambda (pgm)
          (ps0 (instLevel pgm))))

;;; psK, where k is a number, are the states of the parser.
;;; ps0: entry to the program:
(define ps0
  (lambda (pgm)
          (cond ((null? pgm) (error " syntax: lambda is not in L. "))
                (else
                 (let ((inst (car pgm)))
                      (if (and (> (length inst) 2)
                               (equal? (car inst) "read")
                               (equal? (cadr inst) "("))  ; ) special comment
                               (make-pgm (collectReadVars (cddr inst))
                                         (map ps1
                                              (basicBlockLevel (cdr pgm) ())))
                          (error " syntax: read statement ill-formed."))))))))

;;; Fun: collectReadVars: construct the list of variable names:
(define collectReadVars
  (lambda (lst)                          ; ( special comment
          (if (equal? (last lst) ")")
              (if (equal? (length lst) 1)
                  ()
                  (if (> (length lst) 1)
                      (if (id? (car lst))
                          (cons (car lst) (getRestVars (exceptLast (cdr lst))))
                          (error " syntax: id expected." lst))
                      (getRestVars (exceptLast lst))))
              (error " syntax: ill-formed id list:" lst))))

;;; Fun: getRestVars: handles kleene star part:
(define getRestVars
  (lambda (lst)
          (cond ((null? lst) ())
                ((< (length lst) 2) (error " syntax: wrong id list:" lst))
                ((and (id? (cadr lst)) (equal? (car lst) ","))
                 (cons (cadr lst) (getRestVars (cddr lst))))
```

```
                ((not (id? (cadr lst)))
                 (error " syntax: not an id: " (cadr lst)))
                (else (error " syntax wrong id list:" lst)))))

;;; ps1: parse all basic blocks:
(define ps1
  (lambda (blk)
          (let* ((blkLen (length blk))
                 (blkJmp (if (equal? blkLen 1)
                             (cddar blk)    ; just a jump
                             (last blk)))   ; composite
                 (blkLab (caar blk))        ; always there!
                 (blkBod (if (equal? blkLen 1)
                             '()            ; empty body
                             (cons (cddar blk)
                                   (exceptLast (cdr blk))))))
                (make-basicBlk (verifyLbl blkLab)
                               (map parseAssign blkBod)
                               (parseJump blkJmp)))))

;;; Function parseAssign: parse assignments..
(define parseAssign
  (lambda (asgn)
          (if (not (>= (length asgn) 3))
              (error "syntax: invalid assignment statement: " asgn)
              (if (number? (string->number (car asgn)))
                  (error "syntax: invalid identifier: " (car asgn))
                  (if (not (equal? ":=" (cadr asgn)))
                      (error "syntax: invalid operator: " (cadr asgn) asgn)
                      (make-assign (car asgn)
                                   (parseExprs (cddr asgn))))))))

;;; Function parseJump: parse jumps..
(define parseJump
  (lambda (jmp)
          (let ((tag (car jmp)))
               (cond ((equal? tag "goto")
                      (if (not (equal? (length jmp) 2))
                          (error "syntax: invalid goto: " jmp)
                          (make-goto (verifyLbl (cadr jmp)))))
                     ((equal? tag "return")
                      (if (not (>= (length jmp) 2))
                          (error "syntax: invalid return: " jmp)
                          (make-return (parseExprs (cdr jmp)))))
                     ((equal? tag "if") (parseCondJump (cdr jmp)))
                     (else (error "syntax: unrecognized jump: " jmp)))))))

;;; Function parseCondJump: parse if's:
(define parseCondJump
  (lambda (cjmp)
          (let ((tmp (reverse cjmp)))
               (cond ((or (not (equal? (cadr tmp) "else"))
                          (not (equal? (cadddr tmp) "goto")))
                      (error (string-append " syntax: else or goto missing"
                                            " or misplaced in if ") cjmp))
                     (else (make-condJump (parseExprs (upto cjmp "goto"))
```

```
                                        (verifyLbl (caddr tmp))
                                        (verifyLbl (car tmp))))))))

;;; Function parseExprs: parse expressions..
(define parseExprs
  (lambda (expr)
          (cond ((and (equal? (length expr) 1)
                      (equal? (string-ref (car expr) 0) #\'))
                 (getConst (list->string (cdr (string->list (car expr))))))
                ((and (equal? (length expr) 1)
                      (not (number? (string->number (car expr)))))
                 (make-varRef (string->symbol (car expr))))
                (else (parseApp expr)))))    ; otherwise an application

;;; Function getConst: collect the constant value:
(define getConst
  (lambda (constVal)   ; represented as a string
          (let ((listForm (string->list constVal)))
               (cond ((equal? (car listForm) #\() ; ) special comment
                      (make-const 'listing
                                  (flatten (exceptLast (cdr listForm)))))
                     (else (make-const 'singleton constVal))))))

;;; Function parseApp: parse application..
(define parseApp
  (lambda (app)
          (if (or (< (length app) 3)        ;;; ( special comment
                  (not (equal? (last app) ")"))
                  (not (equal? (cadr app) "(")) ;;; ) special comment
                  (equal? (string-ref (car app) 0) #\'))
              (error " syntax: ill-formed application: " app)
              (if (not (isOperator? (car app)))
                  (error " problem: not a known operator: " (car app))
                  (make-app
                   (car app)
                   (map
                    parseExprs
                    (map reverse
                         (separateOps
                          (reverse
                           (cons ","
                                 (reverse (exceptLast (cddr app))))))))))))))

;;; Function separateOps: distinguish operands:
;;; input is of form:  q, r, s, t,   (comma is appended extra..)
(define separateOps
  (lambda (ops)
          (cond ((null? ops) ())
                (else (let ((fArg (reverse (uptoFarg ops 0))))
                           (cons fArg
                                 (separateOps (afterFArg fArg ops)))))))))

;;; Function afterFArg: get after the first Arg, don't include comma
(define afterFArg
  (lambda (firs ops)
          (letrec ((l1 (+ 1 (length firs)))
```

```
                  (drop (lambda (lis n)
                                (if (equal? n 0) lis
                                    (drop (cdr lis) (- n 1))))))
                 (if (equal? l1 1) '()
                     (drop ops l1)))))

;;; Function uptoFArg: get upto the first Arg, don't include the
;;; following comma
(define uptoFArg
  (lambda (ops npar)
          (cond ((null? ops) '())
                ((and (equal? (car ops) ",") (equal? npar 0)) '()) ;finished
                ((equal? (car ops) "(") ;;; ) spec. comment.
                 (cons (car ops) (uptoFArg (cdr ops) (+ 1 npar)))) ;( spec.com.
                ((equal? (car ops) ")")
                 (cons (car ops) (uptoFArg (cdr ops) (- npar 1))))
                (else (cons (car ops) (uptoFArg (cdr ops) npar))))))
```

D.5 Unparser for L: unparser.s

```
;;;-------------------------------------------------------------------
;;;
;;; File unparser.s: contains functions for unparsing an abstract
;;; syntax tree.
;;;
;;; Unparser maps an abstract syntax tree in to the equivalent
;;; concrete syntax.
;;;
;;; The unparser routines are used for outputting the final residual
;;; code to the disk, which is the overall aim after all. Also these
;;; functions are used for debugging purposes.
;;;
;;;-------------------------------------------------------------------
(define unparseL
  (lambda (ast fileName)
          (let ((port (open-output-file fileName)))
               (begin (unparse ast port)
                      (close-output-port port)
                      #t))))

;;; produce output on port:
(define unparse
  (lambda (ast port)
          (begin
           (upRead (pgm-readBlk ast) port) (newline port)
           (forEach (lambda (blk) (begin (upBlk blk port)
                                         (newline port)))
                    (pgm-basicBlks ast)))))

;;; upRead: output the read Block
(define upRead
  (lambda (vars port)
          (begin
           (display "read(" port) ;) special comment
           (outputVars vars port) ;( special comment
```

```
           (display ");" port))))

;;; outputVars: print out arguments:
(define outputVars
  (lambda (lst port)
          (cond ((null? lst) ())
                ((null? (cdr lst)) (display (car lst) port))
                (else (begin (display (car lst) port)
                             (display ", " port)
                             (outputVars (cdr lst) port))))))

;;; upBlk: unparse basic blocks:
(define upBlk
  (lambda (blk port)
          (begin (newline port)
                 (display (basicBlk-lbl blk) port)
                 (display ": " port) (newline port) (display "\t" port)
                 (forEach (lambda (asgn) (begin (upAsgn asgn port)
                                                (newline port)
                                                (display "\t" port)))
                          (basicBlk-assigns blk))
                 (upJmp (basicBlk-jump blk) port))))

;;; upAsgn: unparse an assignment:
(define upAsgn
  (lambda (asgn port)
          (begin
           (display (assign-var asgn) port) (display " := " port)
           (upExpr (assign-expr asgn) port)
           (display ";" port))))

;;; upJmp: unparse a jump
(define upJmp
  (lambda (jmp port)
          (cond ((condJump? jmp) (begin
                                   (display "if " port)
                                   (upExpr (condJump-expr jmp) port)
                                   (display " goto " port)
                                   (display (condJump-lbl1 jmp) port)
                                   (display " else " port)
                                   (display (condJump-lbl2 jmp) port)
                                   (display ";" port)))
                ((goto? jmp)     (begin
                                   (display "goto " port)
                                   (display (goto-lbl jmp) port)
                                   (display ";" port)))
                ((return? jmp)   (begin
                                   (display "return " port)
                                   (upExpr (return-exp jmp) port)
                                   (display ";" port)))
                (else            (error "something wrong with ast " jmp)))))

;;; upExpr: unparse an expression
(define upExpr
  (lambda (expr port)
          (cond ((const? expr) (upConst expr port))
```

```
                ((varRef? expr) (display (varRef-var expr) port))
                ((app? expr) (begin
                               (display (app-rator expr) port)
                               (display "(" port) ;) special comment
                               (if (not (null? (app-rands expr)))
                                   (begin
                                    (upExpr (car (app-rands expr)) port))
                                   ())
                               (forEach (lambda (ex) (begin (display ", " port)
                                                            (upExpr ex port)))
                                        (cdr (app-rands expr))) ;(spec. comment
                               (display ")" port)))
                (else (error "something wrong with ast " expr)))))

;;; upConst: unparse a constant
(define upConst
  (lambda (const port)
          (let ((type (const-type const))
                (val (const-val const)))
               (cond ((equal? type 'singleton) (begin (display "'" port)
                                                      (display val port)))
                     ((equal? type 'listing) (begin (display "'" port)
                                                    (printList val port)))
                     (else (error "unknown constant type " type val))))))

;;; printList: have to convert strings to symbols:
(define printList
  (lambda (lst port)
          (letrec ((beautify (lambda (elm)
                                     (cond ((list? elm) (map beautify elm))
                                           ((string? elm) (string->symbol elm))
                                           (else elm)))))
                  (display (map beautify lst) port))))
```

D.6 Interpreter for L: interpreter.s

```
;;;-------------------------------------------------------------------
;;;
;;; File interpreter.s: contains functions for interpreting L programs.
;;;
;;; The environment is kept as an associative list and all assignments
;;; are handled through the modifications of that list.
;;;
;;;-------------------------------------------------------------------

(define-structure interpResult val env)
;;; initial environment: there are no predefined values only the library
;;; functions exist in the initial environment.
(define initialEnv
  (list (list 'hd               hdL)
        (list 'tl               tlL)
        (list 'cons             consL)
        (list 'first_instruction firstInstL)
        (list 'rest             restL)
        (list 'firstsym         firstSymL)
```

```
        (list 'new_tail          newTailL)
        (list 'eq                equalL)
        (list 'list              listL)
        (list 'transition        transitionL)
        (list 'append            appendL)
        (list 'member            memberL)
        (list 'add               addL)
        (list 'sub               subL)
        (list 'mul               mulL)
        (list 'div               divL)
        (list 'odd               oddL)
        (list 'even              evenL)
        (list 'gt                gtL)
        (list 'lt                ltL)
        (list 'gte               gteL)
        (list 'lte               lteL)
        (list 'exp               expL)
        (list 'sqrt              sqrtL)
        (list 'intdiv            intdivL)
        (list 'inform_sga        informSGAL)))

;;; File interpreter:
(define li
  (lambda (fname) (run (parseL (readPgm fname)))))

;;; interpreter for L programs:
(define run
  (lambda (ast)
          (interpResult-val
           (initiate (pgm-basicBlks ast)
                     (interpResult-env
                      (loadVars (pgm-readBlk ast) initialEnv))))))

;;; extend initial env through initial reads
(define loadVars
  (lambda (readList env)
          (make-interpResult 'NOVALUE (gatherEnv readList env))))

;;; gatherEnv: collect variables:
(define gatherEnv
  (lambda (elms env)
          (cond ((null? elms) env)
                (else (gatherEnv (cdr elms)
                                 (begin (display (car elms)) (display "? ")
                                        (update env (car elms)
                                                (normalize (read)))))))))

;;; normalize: L does not support numbers, in case a number is read convert
;;; it to a symbol:
(define normalize
  (lambda (n)
          (if (number? n) (number->symbol n) n)))

;;; initiate: start execution:
(define initiate
  (lambda (bblks env)
```

```
          (cond ((null? bblks) (make-interpResult 'NOVALUE env))
                (else (execute bblks (car bblks) env)))))


;;; execute: execute the pgm:
(define execute
  (lambda (pgm curBlk env)
          (let* ((newEnv (performAssigns (basicBlk-assigns curBlk) env))
                 (nextBlkInfo (whereToGo (basicBlk-jump curBlk) newEnv)))
                (if (equal? (car nextBlkInfo) 'TERMINATE)
                    (make-interpResult (cdr nextBlkInfo) newEnv)
                    (execute pgm (getBlk pgm (car nextBlkInfo)) newEnv)))))

;;; getBlk: return the matching basic block:
(define getBlk
  (lambda (pgm label)
          (cond ((null? pgm) (error "No such label: " label))
                ((equal? (basicBlk-lbl (car pgm)) label) (car pgm))
                (else (getBlk (cdr pgm) label)))))


;;; performAssigns: evaluate and form the new assignments: return env
(define performAssigns
  (lambda (assigns env)
          (cond ((null? assigns) env)
                (else (performAssigns (cdr assigns)
                                      (doAssign (car assigns) env))))))

;;; doAssign: perform a single assignment: return the new environment.
(define doAssign
  (lambda (astmt env)
          (update env (assign-var astmt) (evalExp (assign-expr astmt) env))))

;;; whereToGo: execute and decide jumps:
(define whereToGo
  (lambda (jstmt env)
          (cond ((goto? jstmt) (cons (goto-lbl jstmt) 'NOVALUE))
                ((return? jstmt) (cons 'TERMINATE
                                       (evalExp (return-exp jstmt) env)))
                ((condJump? jstmt) (cons
                                    ((if (evalExp (condJump-expr jstmt) env)
                                         condJump-lbl1
                                         condJump-lbl2)
                                     jstmt) 'NOVALUE))
                (else (error "Invalid jump: " jstmt)))))

;;; evalExp: return value of the expression:
(define evalExp
  (lambda (exp env)
          (cond ((const? exp) (cond ((equal? (const-type exp) 'singleton)
                                     (beautify (const-val exp)))
                                    ((equal? (const-type exp) 'listing)
                                     (formList (beautify (const-val exp)) env))
                                    (else (error "Unknown const" exp))))
                ((varRef? exp) (lookUp env (symbol->string (varRef-var exp))))
                ((app? exp) (apply (lookUp env (string->symbol (app-rator exp)))
```

```
                                (map (lambda (e) (evalExp e env))
                                     (app-rands exp))))
                (else (error "Unknown exp type: " exp)))))

;;; beautifier, convert strings to symbols to make them appear more natural:
(define beautify
  (lambda (elm)
          (cond ((list? elm) (map beautify elm))
                ((string? elm) (string->symbol elm))
                (else elm))))

;;; formList: construct a list from the environment:
;;; this is unnecessary, just for compatibility:
(define formList (lambda (x y) x))
```

D.7 The L library: llibrary.s

```
;;;-------------------------------------------------------------------
;;;
;;; File llibrary.s: contains functions for the run time library
;;; of the L language. Adding a new function to the library is easy,
;;; the steps to be followed:
;;;      1. In the interpreter.s file, add the name of the function
;;;        and the name of the corresponding one in the library
;;;      2. In the llibrary.s file (this one) define that function
;;;        using scheme.
;;;      3. register the name of the L function in the util.s file
;;;        by adding its name to the list in function isOperator?
;;;      4. in the guards.s file, define the corresponding guard.
;;;      5. in the symbSpeedUp.s file, update weight costs for them.
;;;      (if no special care is needed, it will automatically go into
;;;      the others section, so nothing is needed to be done..)
;;;
;;; that's all folks for adding a new function.
;;;
;;;-------------------------------------------------------------------

;;; initial environment:
(define hdL car)              ;bound to hd
(define consL cons)           ;bound to cons
(define firstInstL car)       ;bound to first_instruction
(define restL cdr)            ;bound to rest
(define listL list)           ;bound to list
(define memberL member)       ;bound to member
(define appendL append)       ;bound to append
(define newTailL              ;bound to newTail
  (lambda (lab pgm)
          (cond ((null? pgm) (error "(runtime) No such label: " lab))
                ((equal? (caar pgm)
                         (string->symbol
                          (string-append (number->string lab) ":")))
                 pgm)
                (else (newTailL lab (cdr pgm))))))

(define numberL
```

```
  (lambda (i)
          (number? (symbol->number i))))

(define equalL                ;bound to eq
 (lambda (i1 i2)
         (or (equal? i1 i2)
             (equal? (convString i1) (convString i2)))))

;;; convString: auxilary to equalL above, used to perform a typeless
;;; comparison.
(define convString
  (lambda (elm)
          (cond ((string? elm) (toLowerString elm))
                ((number? elm) (toLowerString (number->string elm)))
                ((symbol? elm) (toLowerString (symbol->string elm)))
                (else elm))))

;;; toLowerString: convert all chars to lower case:
(define toLowerString
  (lambda (s)
          (list->string (map char-downcase (string->list s)))))

(define transitionL       ;bound to transition
  (lambda (machine state symbol)
          (cond ((null? machine) 'NOTRANSITION)
                ((and (equalL state (caar machine))
                      (equalL symbol (cadar machine)))
                 (caddar machine))
                (else (transitionL (cdr machine) state symbol)))))

(define firstSymL         ;bound to firstsym
  (lambda (lst)
          (cond ((null? lst) 'B)    ; return blanks
                (else (car lst)))))

(define tlL               ;bound to tl
  (lambda (lst)
          (cond ((null? lst) ())    ; return empty list
                (else (cdr lst)))))

;;; arithmetic routines:
(define (symbol->number x) (string->number (symbol->string x)))
(define (number->symbol x) (string->symbol (number->string x)))
(define (addL x y) (number->symbol (+ (symbol->number x) (symbol->number y))))
(define (subL x y) (number->symbol (- (symbol->number x) (symbol->number y))))
(define (divL x y) (number->symbol (/ (symbol->number x)
                                      (symbol->number y))))
(define (mulL x y) (number->symbol (* (symbol->number x) (symbol->number y))))
(define (oddL x) (myodd? (symbol->number x)))
(define (evenL x) (not (oddL? x)))
(define (gtL x y) (> (symbol->number x) (symbol->number y)))
(define (ltL x y) (< (symbol->number x) (symbol->number y)))
(define (gteL x y) (>= (symbol->number x) (symbol->number y)))
(define (lteL x y) (<= (symbol->number x) (symbol->number y)))
(define (sqrtL x) (number->symbol (sqrt (symbol->number x))))
(define (expL x) (number->symbol (exp (symbol->number x))))
```

```
(define (intdivL x y) (number->symbol
                       (quotient (symbol->number x) (symbol->number y))))

(define (myodd? x)
  (let ((y (exact->inexact x)))
       (if (integer? y) (odd? y) #f)))

(define SGAstopCount 'err) ; allocate space for flag
;;; informSGAL: the interface from the program to the static gain analyzer:
;;; parameter flag:
;;;      if 0: stop counting
;;;      if 1: start counting
;;;      if 2: increment flag
(define informSGAL        ; bound to inform_sga
  (lambda (flag)
          (cond ((equalL flag 0) (set! SGAstopCount #t))    ; stop counting
                ((equalL flag 1) (set! SGAstopCount #f))  ; restart counting..
                ((equalL flag 2) (begin (incrCV! searchCV)  ; record..
                                        (incrCV! internCV)))
                (else (error "inform_sga: incorrect flag: " flag)))))
```

D.8 Set operations package: setOperations.s

```
;;;-------------------------------------------------------------------
;;;
;;; File setOperations.s: contains functions that implement several
;;; set operations.
;;;
;;; supported functions:
;;;        setUnion (of arbitrary arity), setDifference, setEqual?,
;;;        isSubset?, setIntersection
;;;-------------------------------------------------------------------

;;; setUnion: return the union of any number of sets:
(define setUnion
 (lambda allSets
         (cond ((null? allSets) ())
               ((null? (cdr allSets)) (car allSets))
               (else (setUnionAux (car allSets) (apply
                                                 setUnion (cdr allSets)))))))

;;; setUnionAux: work on two sets:
(define setUnionAux
 (lambda (s1 s2)
         (cond ((null? s1) s2)
               ((member (car s1) s2) (setUnionAux (cdr s1) s2))
               (else (cons (car s1) (setUnionAux (cdr s1) s2))))))

;;; setDifference: return the difference of two sets:
(define setDifference
 (lambda (s1 s2)
         (cond ((null? s1) ())
               ((member (car s1) s2) (setDifference (cdr s1) s2))
               (else (cons (car s1) (setDifference (cdr s1) s2))))))
```

```
;;; setEqual?: return #t if two sets are equal:
(define setEqual?
 (lambda (s1 s2) (and (isSubset? s1 s2) (isSubset? s2 s1))))

;;; isSubset?: return #t if s1 is a subset of s2
(define isSubset?
 (lambda (s1 s2)
        (cond ((null? s1) #t)
              ((member (car s1) s2) (isSubset? (cdr s1) s2))
              (else #f))))

;;; setIntersection: return the intersection of two sets:
(define setIntersection
  (lambda (s1 s2)
         (cond ((null? s1) ())
               ((member (car s1) s2) (cons (car s1)
                                           (setIntersection (cdr s1) s2)))
               (else (setIntersection (cdr s1) s2)))))
```

D.9 Commenting and Debugging: aux.s

```
;;;-------------------------------------------------------------------
;;;
;;; File aux.s: contains functions for debugging and commenting
;;;
;;;-------------------------------------------------------------------

;;; gor: see the concrete syntax of a given abstract syntax:
(define gor
  (lambda (expr)
         (let ((ee (open-output-file "zz")))
              (begin
               ((cond ((assign? expr) upAsgn)
                      ((condJump? expr) upJmp)
                      ((goto? expr) upJmp)
                      ((return? expr) upJmp)
                      ((basicBlk? expr) upBlk)
                      ((attr? expr) seeAttr)
                      (else upExpr)) expr ee)
               (close-output-port ee)
               (system "cat zz")
               (system "rm zz")))))

;;; seeAttr: see the attribute records associated with each basic block,
;;; mainly used for debugging in the dead code removal algorithm:
(define seeAttr
  (lambda (attr dummy)
         (display "Attr: ") (display attr) (newline)
         (display "  Label: ") (display (attr-lbl attr)) (newline)
         (display "  Flag : ") (display (attr-flag attr)) (newline)
         (display "  In   : ") (display (attr-in attr)) (newline)
         (display "  Out  : ") (display (attr-out attr)) (newline)
         (display "  Def  : ") (display (attr-def attr)) (newline)
         (display "  Use  : ") (display (attr-use attr)) (newline)))
```

```
;;; mgor: gor for a list of abstract syntax records
(define mgor
  (lambda (lst)
          (forEach (lambda (elm) (gor elm) (newline)) lst)))

;;; mdisplay: easier display, for commenting on the screen.
(define mdisplay
  (lambda lst
          (forEach (lambda (elm) (display elm)) lst)))

;;; comment: write out the arguments to the comment file:
;;; mainly used for producing statistics..
(define comment
    (lambda lst
            (forEach (lambda (elm) (display elm commentPort)) lst)))

;;; symbolic gain analyzer comment interface:
(define sgaComment
  (lambda lst
          (forEach (lambda (elm) (display elm (car lst))) (cdr lst))))

;;; introduce: introduce yourself:
(define introduce
  (lambda ()
          (comment
           "Welcome to ILPOS, Integrated L Partial Evaluator"
           " and Optimizer System. (v1.0)\n\n"
           "Activated on: " (decoded-time/date-string (get-decoded-time))
           ", " (decoded-time/time-string  (get-decoded-time))
           "\nBy user     : "
           (unix/current-user-name) "\n")))

;;; sgaIntroduce: introduce yourself to the symbolic gain analyzer:
(define sgaIntroduce
  (lambda (prt)
          (sgaComment prt
           "Welcome to ILPOS-SGA, Static Gain Analyzer System for"
           " ILPOS. (v1.0)\n\n"
           "Activated on: " (decoded-time/date-string (get-decoded-time))
           ", " (decoded-time/time-string  (get-decoded-time))
           "\nBy user     : "
           (unix/current-user-name) "\n")))

;;; commentForSpecArgs: indicate the specialization arguments and
;;; their values:
(define commentForSpecArgs
  (lambda (args env)
          (forEach (lambda (elm)
                           (comment "   " elm
                                    "  <-  "
                                    (lookUp env elm) "\n"))
                   args) (comment "\n")))

;;; several comment functions:
;;; comment for environment:
(define comment1
```

```
  (lambda (f1 f2 f3 f4 noForm noBlks noAssigns)
          (comment "Working dir : "
                   (directory-namestring (working-directory-pathname))
                   "\nInput file  : " f1 "\nOutput file : " f2
                   "\nPure PE file: " f4
                   "\nLog file    : " f3 "\n\nInput has: \n   "
                   noForm " formal arg(s)\n   " noBlks
                   " basic block(s)\n   " noAssigns " assignment(s)\n")))

;;; comment after completing partial evaluation:
(define comment2
  (lambda (l1 l2)
          (comment "Partial Evaluation Completed, residual code has: \n   "
                   l1 " formal arg(s)\n   " l2 " basic block(s)\n\n")))

;;; comment after completing dead code removal:
(define comment3
  (lambda (n)
          (comment
           "Dead Code Removal Completed, Number of Assignments Removed: "
           n "\n\n")))

;;; comment after minimization of the code:
(define comment4
  (lambda (l1 l2)
          (comment "Minimization Completed:\nMinimal code has      : "
                   l2 " block(s)\nMinimization saved us : "
                   (- l1 l2) " block(s)\n\n")))

;;; comment after linearization and canonicalization of the code:
(define comment5
  (lambda (l2 l4 l5 h2 h5)
          (comment "Linearization completed, DCR+MIN+LIN loop terminates.\n"
                   "\n\nTotally " (- l2 l5) " block(s) (out of "
                   l2 ") has/have been\nsaved by the optimizations after "
                   "partial evaluation.\n\nTotally " (- h2 h5)
                   " assignment(s) (out of " h2 ") has/have been\n"
                   "removed by the dead code remover.\n\n")))

;;; final comments on the process:
(define comment6
  (lambda (l5 h5)
          (comment "Final residual program has " l5
                   " block(s) and " h5 " assignment(s).\n\nILPOS completed "
                   "successfully.\n")))

;;; howManyAssigns: given an ast return the number of assignments it has:
(define howManyAssigns
  (lambda (ast)
          (letrec ((howManyAssignsAux
                    (lambda (bblks)
                            (cond ((null? bblks) 0)
                                  (else (+ (length
                                            (basicBlk-assigns (car bblks)))
                                           (howManyAssignsAux (cdr bblks))))))))
                  (howManyAssignsAux (pgm-basicBlks ast)))))
```

```
;;; phasor: apply some phase or phases to some file..
(define phasor
  (lambda ops
          (unparseL
           (phasorAux (parseL (readPgm (car ops))) (cdr ops))
           (string-append (car ops) ".phs"))))

;;; phasorAux; apply each operator in turn:
(define phasorAux
  (lambda (ast opList)
          (cond ((null? oplist) ast)
                (else (phasorAux ((car opList) ast) (cdr opList))))))

;;; warning: issue a warning regarding incomplete specifications:
(define warning
  (lambda (expr divVs)
          (comment "** Partial Evaluator Warning: The static expression:\n\t")
          (upExpr expr commentPort)
          (comment "\n** has been evaluated with respect to the static "
                   "environment:\n")
          (forEach (lambda (elm) (comment "  " (car elm) " --> "
                                          (cadr elm) "\n"))
                   divVs)
          (comment "** contains an unsafe operation.\n"
                   "** Signalling operation is indicated above.\n\n")))

;;; terminationHandlerReport: if limits are reached, give a report
;;; in the log file
(define terminationHandlerReport
  (lambda ()
          (if terminationReported ()
              (begin
               (mdisplay "Termination problem detected, termination handler"
                         " is activated.\n")
               (comment "** Termination Handler Notice:\n"
                        "   The residual code size exceeded "
                        maximumCodeSize " blocks\n"
                        "   Marking all variables as dynamic and stopping "
                        "further partial evaluation.\n\n")
               (set! terminationReported #t)))))
```

D.10 Utility functions: util.s

```
;;;-------------------------------------------------------------------
;;;
;;; File util.s: contains functions for general usage, a collection
;;; of tools. Mainly used by lexer.s in performing the lexical analysis.
;;;
;;;-------------------------------------------------------------------

;;; Fun: id? checks if it is an id: (not keyword.)
(define id?
  (lambda (lexeme)
          (and (not (punctuation? lexeme))
```

```
              (not (keyword? lexeme))
              (not (equal? (string-ref lexeme 0) #\')))))

;;; Fun: keyword? checks if keyword
(define keyword?
  (lambda (lexeme)
          (member lexeme '("read" "goto" "if" "else" "return" "quote"))))

;;; Fun: isOperator?: is it a known function?
(define isOperator?
  (lambda (op)
          (member op '("hd" "tl" "cons" "new_tail" "firstsym"
                       "rest" "first_instruction" "list" "eq"
                       "append" "member" "transition" "inform_sga"
                       "sub" "add" "mul" "div" "odd" "even" "lt" "gt"
                       "lte" "gte" "exp" "sqrt" "intdiv"))))

;;; Fun: last: return the last element.
(define last
  (lambda (lst)
          (cond ((null? lst) ())
                ((equal? (length lst) 1) (car lst))
                (else (last (cdr lst))))))

;;; Fun: exceptLast: return all but last:
(define exceptLast
  (lambda (lst)
          (cond ((null? lst) ())
                ((equal? (length lst) 1) ())
                (else (cons (car lst) (exceptLast (cdr lst)))))))

;;; Function instLevel: divide the program into instructions:
(define instLevel
  (lambda (pgm)
          (cond ((null? pgm) ())
                (else (cons (getFirstInst pgm)
                            (instLevel (skipFirstInst pgm)))))))

;;; Function getFirstInst: return first instruction:
(define getFirstInst
  (lambda (pgm)
          (cond ((null? pgm) ())
                ((equal? (car pgm) (char->string #\;)) ())
                (else (cons (car pgm) (getFirstInst (cdr pgm)))))))

;;; Function skipFirstInst: return the rest:
(define skipFirstInst
  (lambda (pgm)
          (cond ((null? pgm) ())
                ((equal? (car pgm) ";") (cdr pgm))
                (else (skipFirstInst (cdr pgm))))))

;;; Fun: basicBlockLevel: extract basic blocks:
(define basicBlockLevel
  (lambda (pgm prev)
          (cond ((null? pgm) ())
```

```
                ((or (member "goto" (car pgm))
                     (member "if"   (car pgm))
                     (member "return" (car pgm)))
                 (cons (append prev (list (car pgm)))
                       (basicBlockLevel (cdr pgm) ())))
                (else (basicBlockLevel (cdr pgm)
                                       (append prev (list (car pgm))))))))

;;; Function upto: gets a list and a key, returns list upto key, not
;;; including key:
(define upto
  (lambda (lst key)
          (cond ((null? lst) '())
                ((equal? (car lst) key) '())
                (else (cons (car lst) (upto (cdr lst) key))))))

;;; Function after: gets a list and a key, returns list after key, not
;;; including key:
(define after
  (lambda (lst key)
          (cond ((null? lst) '())
                ((equal? (car lst) key) (cdr lst))
                (else (after (cdr lst) key)))))

;;; Fun: verifyLbl: check if given thing is a valid label, if so
;;; return it, otherwise raise an error:
(define verifyLbl
  (lambda (lbl)                 ; essentially it shouldn't be quoted..
          (if (equal? (string-ref lbl 0) #\')
              (error " syntax: invalid label: " lbl)
              lbl)))

;;; Function flatten: we have a string representing a list,
;;; turn it into a real list:
(define flatten
    (lambda (org)     ; org is a list
            (let ((inp (trim org)))
                 (cond ((null? inp) ())
                       ((equal? (car inp) '#\()  ;) special comment
                        (cons (flatten (exceptLast (toClosing (cdr inp) 1)))
                              (flatten (trim (afterClosing (cdr inp) 1)))))
                       (else (cons (list->string (getWord org))
                                   (flatten (afterWord org))))))))

;;; Function toClosing: return with last closing paranthesis
(define toClosing
  (lambda (strList cnt)
          (cond ((equal? cnt 0) ())
                ((equal? (car strList) '#\() ;) special comment
                 (cons '#\( ; ) special comment
                       (toClosing (cdr strList) (+ cnt 1))))  ;( special comment
                ((equal? (car strList) '#\))  ;( special comment
                 (cons '#\)
                       (toClosing (cdr strList) (- cnt 1))))
                (else (cons (car strList) (toClosing (cdr strList) cnt))))))
```

```
;;; Function afterClosing: return after closed
(define afterClosing
  (lambda (strList cnt)
          (cond ((equal? cnt 0) strList)
                ((equal? (car strList) '#\()
                 (afterClosing (cdr strList) (+ cnt 1)))
                ((equal? (car strList) '#\))
                 (afterClosing (cdr strList) (- cnt 1)))
                (else (afterClosing (cdr strList) cnt)))))

;;; Function getWord: return the first lexeme..
(define getWord
  (lambda (lst)
          (cond ((null? lst) ())
                ((whiteSpace? (car lst)) ())
                (else (cons (car lst) (getWord (cdr lst)))))))

;;; Function afterWord: return except the first lexeme.
(define afterWord
  (lambda (lst)
          (cond ((null? lst) ())
                ((whiteSpace? (car lst)) (trim (cdr lst)))
                (else (afterWord (cdr lst))))))

;;; trim: skip the initial whitespaces:
(define trim
  (lambda (lst)
          (cond ((null? lst) ())
                ((whiteSpace? (car lst)) (trim (cdr lst)))
                (else lst))))

;;; forEach: order dependent map style function
(define forEach
  (lambda (f lst)
          (cond ((null? lst) ())
                (else (begin (f (car lst)) (forEach f (cdr lst)))))))

;;; lookUp: look up a var in the associative list:
(define lookUp
  (lambda (env var)
          (cond ((null? env) (error "uninitialized variable: " var))
                ((equal? (caar env) var) (cadar env))
                (else (lookUp (cdr env) var)))))

;;; update: update and return the new environment:
(define update
  (lambda (env var val)
          (cond ((null? env) (list (list var val)))
                ((equal? (caar env) var) (cons (list var val) (cdr env)))
                (else (cons (car env) (update (cdr env) var val))))))

;;; listDiff: return the difference of lists:
(define listDiff
  (lambda (l1 l2)
          (cond ((null? l1) '())
                ((member (car l1) l2) (listDiff (cdr l1) l2))
```

```
                (else (cons (car l1) (listDiff (cdr l1) l2))))))

;;; newPgmPoint: return a new label:
(define newPgmPoint
  (lambda ()
          (symbol-append 'L (generate-uninterned-symbol 'PE))))
```

D.11 Binding Time Analyzer for L: bta.s

```
;;;-----------------------------------------------------------------
;;;
;;; File bta.s: contains functions for binding time analysis
;;; Binding time analysis refers to the analysis of all the
;;; program variables to determine whether they can be marked
;;; as static or dynamic. Essentially all the program arguments
;;; are dynamic except for those for which the specialization
;;; is done. For the rest of the variables the rules are as
;;; follows: if a variable appears in the left hand side of an
;;; assignment and the right hand side of that assignment contains
;;; a variable that is marked as dynamic, that variable becomes
;;; dynamic. Note that this is a closure algorithm and the least
;;; set that satisfies these properties is the set we are
;;; looking for. Once all dynamics are determined, the division
;;; is easily found as the difference of all variables from the
;;; list of dynamic variables.
;;;
;;;-----------------------------------------------------------------

;;; bta: perform binding time analysis
;;; return a btaRec structure with first field as a list of all
;;; program variables which are static, second is the environment
;;; containing the initial values of these static variables..
(define bta
  (lambda (prg)
          (mdisplay "Performing Binding Time Analysis..\n")
          (let* ((staticArgs (getStaticArgs prg))
                 (division (getDivision prg staticArgs))
                 (ongoingEnv (gatherEnv staticArgs '())))
                (comment "   " (length staticArgs)
                         " static arg(s)  : " staticArgs "\n   "
                          (length division)
                         " static var(s)  : "
                         division "\n\n"
                         "Program is specialized with respect to:\n\n")
                (commentForSpecArgs staticArgs ongoingEnv)
                 (make-btaRec division
                              (pushErrors
                               (setDifference division staticArgs)
                               ongoingEnv)))))

;;; pushErrors: for each val push an error.. This is used for filling
;;; up the initial environment for the partial evaluation phase. Pushing
;;; err does not mean anything since these variables are, logically,
;;; never used before being defined. In case something like that occurs,
;;; this means that the user is referencing to a variable that has not been
```

```
;;; initialized before. The result is that he/she will get a garbage value.
;;; Note that if this can be detected at specialization time then a
;;; warning will be issued in the log file.
(define pushErrors
  (lambda (lst env)
          (cond ((null? lst) env)
         (else (pushErrors (cdr lst) (update env (car lst) 'UNINITIALIZED))))))

;;; getStaticArgs: from the read list which are static, learn from user
(define getStaticArgs
  (lambda (prg)
          (begin
           (display "Arguments   : ")
           (forEach (lambda (elm) (display elm) (display " "))
                    (pgm-readBlk prg))
           (display "\nStatics Args? ")
           (read))))

;;; getDivision: return the division of the whole program variables..
;;; i.e. a list containing the names of all static variables.
(define getDivision
  (lambda (prg init)
          (let ((allVarsInPgm (allVariables prg))
                (allDyns (dynamicClosure (apply append
                                                (map basicBlk-assigns
                                                     (pgm-basicBlks prg)))
                                         (setDifference (pgm-readBlk prg)
                                                        init))))
               (comment "   " (length allVarsInPgm)
                        " program var(s) : "  allVarsInPgm "\n"
                        "   " (length allDyns) " dynamic var(s) : "
                         allDyns "\n")
               (setDifference allVarsInPgm allDyns))))

;;; allVariables: return the set of all variables of the program:
;;; construct the union of all variables in all blocks:
(define allVariables
  (lambda (prg)
          (setUnion (pgm-readBlk prg)
                    (apply setUnion
                           (map allVariablesInBlk (pgm-basicBlks prg))))))

;;; allVariablesInBlk: return the set of variables in some block:
;;; i.e. the set of variables in assignments and the variables in the
;;; expression of the jump.
(define allVariablesInBlk
  (lambda (blk)
          (setUnion
           (apply setUnion (map varsInAssign (basicBlk-assigns blk)))
           (varsInJump (basicBlk-jump blk)))))

;;; varsInAssign: return the variables in an assignment:
(define varsInAssign
  (lambda (assignment)
          (setUnion (list (assign-var assignment))
                    (map symbol->string
```

```
                        (dependents (assign-expr assignment))))))

;;; varsInJump: return the variables in the jump:
(define varsInJump
  (lambda (jmp)
          (cond ((return? jmp) (map symbol->string
                                    (dependents (return-exp jmp))))
                ((condJump? jmp) (map symbol->string
                                      (dependents (condJump-expr jmp))))
                (else ()))))) ; goto has no variables..

;;; dynamicClosure: starting from an initial dynamic list, expand through
;;; all variables.
(define dynamicClosure
  (lambda (assignsList prev)
          (computeDyns (map graspDeps assignsList) prev)))

;;; graspDeps: given an assignment, return a list with first element
;;; the assignment variable, second element a list of dependent variables:
(define graspDeps
  (lambda (asgn)
          (list (assign-var asgn) (map symbol->string
                                       (dependents (assign-expr asgn))))))

;;; computeDyns: given an associative list showing dependencies of variables
;;; and an initial set of dynamics, return all variables that are dynamic
(define computeDyns
  (lambda (depList prev)
          (let ((newDyns (dynPass depList prev)))
               (if (equal? (length newDyns) (length prev))
                   newDyns
                   (computeDyns depList newDyns)))))

;;; dynPass: analyze expressions to see which of them can be marked as
;;; dynamics, this is an iterative algorithm, easily converted to
;;; tail recursion.
(define dynPass
  (lambda (depList olds)
          (cond ((null? depList) olds)
                ((member (caar depList) olds) (dynPass (cdr depList) olds))
                ((null? (setIntersection (cadar depList) olds))
                 (dynPass (cdr depList) olds))
                (else (setUnion (list (caar depList))
                                (dynPass (cdr depList) olds))))))
```

D.12 The L Partial Evaluator: lpeval.s

```
;;;-------------------------------------------------------------------
;;;
;;; File lpeval.s: contains functions for the partial evaluation of
;;; L programs.
;;;
;;; The main technique applied in partial evaluation is that of program
;;; point specialization. According to this technique each basic block
;;; of the source program gives us a program point. The idea is that
```

```
;;; when the program executes it passes through a sequence of program
;;; points. Of course it may be in the same program point later in time
;;; with a different environment. The whole idea is that, given a
;;; program point and an environment specifying the values of the
;;; static variables of the program we can generate a residual block
;;; that uniquely represents that point. Clearly this would remove any
;;; operations that are done only on static values, thus achieving the
;;; merit of partial evaluation.
;;;
;;; This idea is implemented as a graph traversal algorithm which considers
;;; the nodes of the graph as the program points with values of static
;;; variables. The program is interpreted in an abstract sense and the
;;; residual code is generated at the same time.
;;;
;;;----------------------------------------------------------------

;;; given pgm, partially evaluate it
;;; read division and the values of the static variables:

;;; bta results stored in btaRec:
(define-structure btaRec div vs)

;;; take care of incomplete specifications...
(define incompleteSpecDetected #f)
(define incompleteProgramDetected #f)

;;; take care of termination through limited code size:
(define maximumCodeSize 0)
(define residualCodeSize 0)
(define terminationReported #f)
(define maxCodeSizeFactor 20) ; allow at most N times large

;;; Partial Evaluator with all operations cascaded.. also produce
;;; comments on the translation process.
(define lpe
  (lambda (inFile outFile onlyPeFile comFile)
          (let ((cmF (open-output-file comFile)))
               (mdisplay "Reading the program from \"" inFile "\"..\n")
               (set! incompleteProgramDetected #f)
               (let ((ast0 (readPgm inFile)))
                    (mdisplay "Parsing the program..\n")
                    (let ((ast1 (parseL ast0)))
                         (set! commentPort cmF)
                         (set! residualCodeSize 0)
                         (set! terminationReported #f)
                         (set! maximumCodeSize     ; arrange for termination
                               (* maxCodeSizeFactor
                                  (length (pgm-basicBlks ast1))))
                         (introduce)
                         (comment1 inFile outFile comFile onlyPeFile
                                   (length (pgm-readBlk ast1))
                                   (length (pgm-basicBlks ast1))
                                   (howManyAssigns ast1))
                         (mdisplay "Partially Evaluating..\n")
                         (let ((ast2 (pe ast1)))
                              (comment2 (length (pgm-readBlk ast2))
```

```
                                    (length (pgm-basicBlks ast2)))
                                (if incompleteProgramDetected
                                    (mdisplay
                                     "Program incomplete, see log file.\n") #t)
                                (mdisplay "Writing non-optimized residual to \""
                                          onlyPeFile "\"..\n")
                                (unparseL ast2 onlyPeFile)
                                (mdisplay "Removing Dead Code..\n")
                                (let ((ast5 (dcrMinLinLoop
                                             ast2 #f
                                             (length (pgm-basicBlks ast2))
                                             (howManyAssigns ast2))))
                                     (mdisplay
                                      "Writing residual program to \""
                                      outFile "\"..\n")
                                     (unparseL ast5 outFile)
                                     (mdisplay "Partial Evaluator done, "
                                               "log file is: \"" comFile "\".")
                                     (comment6
                                      (length (pgm-basicBlks ast5))
                                      (howManyAssigns ast5))
                                     (close-output-port cmF))))))))


;;; dcrMinLinLoop: perform Dead code removal, minimization and Linearization.
;;; note that if Linearization does something then there is a further
;;; opportunity to remove more dead code and more minimization. so this
;;; loop is iterated untill we come out with a program where linearization
;;; does not do any good for us.
;;; the parameters st1 and st2 are statistic keeping parameters,
;;; representing the number of blocks and the number of assignments in
;;; the program before this loop is activated.
(define dcrMinLinLoop
  (lambda (ast2 reactivated st1 st2)
          (if reactivated
              (comment
               "Linearization worked, Reactivating DCR+MIN+LIN loop..\n\n") #t)
          (let ((ast3 (dcr ast2)))
               (comment3 (- (howManyAssigns ast2) (howManyAssigns ast3)))
               (mdisplay "Minimizing the program..\n")
               (let ((ast4 (minAst ast3)))
                    (comment4 (length (pgm-basicBlks ast3))
                              (length (pgm-basicBlks ast4)))
                    (mdisplay "Linearizing and canonicalizing..\n")
                    (let ((ast5 (linAst ast4))
                          (linWorked (isLinearizable? ast4)))
                         (if linWorked
                             (dcrMinLinLoop ast5 #t st1 st2)
                             (begin
                              (comment5
                               st1
                               (length (pgm-basicBlks ast4))
                               (length (pgm-basicBlks ast5))
                               st2
                               (howManyAssigns ast5))
                              ast5)))))))
```

```
;;; the partial evaluator:
(define pe
  (lambda (pgm)
          (let ((btaRes (bta pgm)))    ; perform BTA..
               (lpeval pgm (btaRec-div btaRes) (btaRec-vs btaRes)))))

;;; temporary data structure, this structure keeps track of the code
;;; generated for the current block that is under consideration.
;;; divVs is the division and the values of the static variables.
;;; pending gives the list of nodes that are already visited.
(define-structure peBB code divVs pending)

;;; lpeval: partial evaluator for L.
;;;         inputs:   pgm: ast of input
;;;                   div: divison
;;;                 divVs: division and the values of the static variables:
;;;         returns: ast of the residual pgm.
(define lpeval
  (lambda (prg div divVs)
          (make-pgm
           (listDiff (pgm-readBlk prg) div)
           (mixAux (pgm-basicBlks prg) divVs))))

;;; mixAux: prepare the pgm for mix:
(define mixAux
  (lambda (bblks divVs)
          (let ((pp0 (basicBlk-lbl (car bblks))))
               (mix bblks (list (list (newPgmPoint) pp0 divVs)) '()))))

;;; mix: the famous mix algorithm:
(define mix
  (lambda (bblks pending marked)
          (set! incompleteSpecDetected #f) ; indicate that we are fine
          (cond ((null? pending) ())    ; end of processing
                (else (set! residualCodeSize (+ residualCodeSize 1))
                      (let* ((nPending (cdr pending))
                             (nMarked (cons (car pending) marked))
                             (bb (getBlk bblks (cadar pending)))
                             (initCodeLbl (caar pending))
                             (bbResult
                              (mixBB (append (basicBlk-assigns bb)
                                             (list (basicBlk-jump bb)))
                                     nPending nMarked
                                     initCodeLbl
                                     (if (> residualCodeSize
                                            maximumCodeSize)
                                         (begin
                                          (terminationHandlerReport)
                                          ())  ; empty divVs
                                         (caddar pending)) ; following divVs
                                     (if (> residualCodeSize
                                            maximumCodeSize)
                                         ; initializing code
                                         (switchingCode (caddar pending))
                                         ())  ; empty code
```

```
                                    () bblks))
                             (newCode (if incompleteSpecDetected
                                          (createCrash bbResult)
                                          (peBB-code bbResult))))
                            (cons newCode
                                  (mix bblks (peBB-pending bbResult)
                                       nMarked)))))))

;;; mixBB: basic block partial evaluator, heart of the engine:
(define mixBB
  (lambda (bb pending marked codeLbl divVs code jump bblks)
          (cond ((null? bb) (make-peBB (make-basicBlk codeLbl
                                                      code
                                                      jump)
                                       divVs
                                       pending))
                ((null? (cdr bb))    ; then it is the jump:
                 (cond
                  ((goto? (car bb))  ; compress the transition
                   (let ((toGo (getBlk bblks (goto-lbl (car bb)))))
                        (mixBB (append (basicBlk-assigns toGo)
                                       (list (basicBlk-jump toGo)))
                               pending marked codeLbl divVs code jump bblks)))
                  ((return? (car bb))
                   (mixBB (cdr bb) pending marked codeLbl divVs code
                          (make-return
                           (if (isStatic? (return-exp (car bb)) divVs)
                               (evalS (return-exp (car bb)) divVs)
                               (reduce (return-exp (car bb)) divVs))) bblks))
                  ((condJump? (car bb))
                   (if (isStatic? (condJump-expr (car bb)) divVs)
                       (let ((toGo           ; static conditional, compress
                              (getBlk bblks
                                      ((if (const-val
                                            (evalS (condJump-expr (car bb))
                                                   divVs))
                                           condJump-lbl1
                                           condJump-lbl2) (car bb)))))
                            (mixBB (append (basicBlk-assigns toGo)
                                           (list (basicBlk-jump toGo)))
                                   pending marked codeLbl
                                   divVs code jump bblks))
                       ; dynamic conditional, go on
                       (if (equal? (condJump-lbl1 (car bb))
                                   (condJump-lbl2 (car bb)))
                           ; to same place, compress the transition
                           (let ((toGo (getBlk bblks (condJump-lbl1 (car bb)))))
                                (mixBB (append (basicBlk-assigns toGo)
                                               (list (basicBlk-jump toGo)))
                                       pending marked codeLbl
                                       divVs code jump bblks))
                           ; to different places..
                           (let* ((goLab1 (condJump-lbl1 (car bb)))
                                  (goLab2 (condJump-lbl2 (car bb)))
                                  (varmi1 (checkOutLabs goLab1 divVs
                                                        (append pending
```

```
                                                marked)))
                                    (varmi2 (checkOutLabs goLab2 divVs
                                                          (append pending
                                                                  marked)))
                                    (ne1 (if varmi1
                                             (getLabs goLab1 divVs
                                                      (append pending marked))
                                             (newPgmPoint)))
                                    (ne2 (if varmi2
                                             (getLabs goLab2 divVs
                                                      (append pending marked))
                                             (newPgmPoint))))
                                   (mixBB
                                    (cdr bb)
                                    (extendPending pending varmi1 varmi2
                                                   ne1 goLab1 ne2 goLab2 divVs)
                                    marked codeLbl divVs code
                                    (make-condJump
                                     (reduce (condJump-expr (car bb)) divVs)
                                     ne1 ne2) bblks)))))  ; dynamic ok.
                      ; jumps ok.
                      (else (error "not a jump: " (car bb)))))
                    (else  ; it is an assignment
                     (if (isStaticVar? (assign-var (car bb)) divVs)
                         (mixBB (cdr bb) pending marked codeLbl
                                (update divVs (assign-var (car bb))
                                        (const-val
                                         (evalS (assign-expr (car bb)) divVs)))
                                code jump bblks)
                         (mixBB (cdr bb) pending marked codeLbl divVs
                                (append code (list (make-assign
                                                    (assign-var (car bb))
                                                    (reduce (assign-expr (car bb))
                                                            divVs))))
                                jump bblks))))))

;;; switchingCode: returns the code that is needed to switch from the
;;; normal operation to the case where all variables are dynamic. This
;;; is accomplished by adding a series of assignments of variables to
;;; their values at the point of conversion:
(define switchingCode
  (lambda (divVs)
          (map (lambda (elm) (make-assign (car elm)
                                          (make-const
                                           (if (list? (cadr elm))
                                               'listing 'singleton)
                                           (cadr elm)))) divVs)))

;;; isStatic? returns true or false
;;; an expression is static if it is a constant, a var that is static by
;;; division or an application of all static arguments.
(define isStatic?
  (lambda (expr divVs)
          (cond ((const? expr)  #t)
                ((varRef? expr) (isStaticVar?
                                 (symbol->string (varRef-var expr)) divVs))
```

```
                ((app? expr) (andAll (map (lambda (elm) (isStatic? elm divVs))
                                          (app-rands expr))))
                (else (error "isStatic?: invalid expression: " expr)))))

;;; And a list, empty list is true..
(define andAll
  (lambda (lst)
          (cond ((null? lst) #t) (else (and (car lst) (andAll (cdr lst)))))))

;;; isStaticVar?: simply is it a member of the division that is determined
;;; by the binding time analyzer.
(define isStaticVar?
  (lambda (var divVs)
          (member var (map car divVs))))

;;; evalS: expression evaluator: note that we take care of incomplete
;;; specifications by enclosing the library functions with guards. If a
;;; guard notices that something goes wrong we don't generate a value
;;; but rather point out a warning. It is important to note that this
;;; may be something that the programmer did on purpose or something
;;; that the programmer forgot to handle.
(define evalS
  (lambda (expr divVs)
          (if (anyViolation? expr divVs)
              (begin
               (set! incompleteSpecDetected #t) ; indicate the problem!
               (set! incompleteProgramDetected #t) ; indicate the problem!
               ; return some erronaus value, just to continue.
               (make-const 'singleton 'ERR))
              (evalSAux expr divVs))))   ; otherwise evaluate..

;;; evalSAux: now we're safe to evaluate the expression..
(define evalSAux
  (lambda (expr divVs)
          (let ((val (evalExp expr (append divVs initialEnv))))
               (make-const (if (list? val) 'listing 'singleton) val))))

;;; reduce: reducer: This is a combination of an evaluator and a
;;; transformer. Mainly, it receives the abstract syntax tree of
;;; an expression and yields another one which does not contain
;;; any computation that depends only on static arguments. That is,
;;; if some subtree is solely static it is replaced by a constant value,
;;; but dynamic references are kept alive.
(define reduce
  (lambda (expr divVs)
          (cond ((const? expr) expr)
                ((and (varRef? expr) (isStaticVar?
                                      (symbol->string (varRef-var expr)) divVs))
                 (let ((item (lookUp (append divVs initialEnv)
                                     (symbol->string (varRef-var expr)))))
                      (make-const (if (list? item) 'listing 'singleton) item)))
                ((varRef? expr) expr) ; a dynamic variable
                ((and (app? expr) (isStatic? expr divVs)) (evalS expr divVs))
                ((app? expr) ; something dynamic inside..
                 (make-app (app-rator expr)
                           (map (lambda (elm) (reduce elm divVs))
```

```
                              (app-rands expr))))
                (else (error "invalid expression: " expr)))))

;;; checkOutLabs: look if marked contains such a label before:
(define checkOutLabs
  (lambda (lab divVs marked)
          (orAll (map (lambda (e) (equal? (list lab divVs) e))
                      (map cdr marked)))))

;;; getLabs: look if marked contains such a label before and return it:
(define getLabs
  (lambda (lab divVs marked)
          (cond ((null? marked) 'SPECERR)
                ((equal? (list lab divVs) (cdar marked)) (caar marked))
                (else (getLabs lab divVs (cdr marked))))))

;;; orAll: empty false
(define orAll
  (lambda (lst)
          (cond ((null? lst) #f)
                (else (or (car lst) (orAll (cdr lst)))))))

;;; extendPending: extend pending with new program points:
(define extendPending
  (lambda (pending v1 v2 l1 o1 l2 o2 divVs)
          (if incompleteSpecDetected   ; no need to consider rest..
              pending
              (epAux (epAux pending v2 l2 o2 divVs) v1 l1 o1 divVs))))

;;; epAux: auxilary to extendPending above.
(define epAux
  (lambda (pending exists lbl old divVs)
          (if exists pending
              (enlarge pending lbl old divVs))))

;;; enlarge: return union:
(define enlarge
  (lambda (pending lbl old divVs)
          (if (orAll (map (lambda (e) (equal? e (list lbl old divVs)))
                          (map cdr pending)))
              pending
              (cons (list lbl old divVs) pending))))

;;; anyViolation?: goven an expression, is there any problem with
;;; evaluation it statically at this time:
(define anyViolation?
  (lambda (expr divVs)
          (if (isSafe? expr divVs) #f
              (begin (warning expr divVs) #t))))

;;; isSafe?: given the expression, is it safe to evaluate it?
;;; being unsafe for an expression means the following:
;;;   if it is a constant: no such thing, a constant is always safe..
;;;              varRef  : it is not defined in the environment,
;;;                        i.e. it is used before being assigned any value
;;;              app     : one of the arguments is unsafe.
```

```
(define isSafe?
  (lambda (expr divVs)
          (cond ((const? expr) #t)
                ((varRef? expr) (let ((there (member (symbol->string
                                                      (varRef-var expr))
                                                     (map car
                                                          (append
                                                           divVs
                                                           initialEnv)))))
                                  (if (equal?
                                       (lookUp (append divVs initialEnv)
                                               (symbol->string
                                                (varRef-var expr)))
                                       'uninitialized)
                                      (begin
                                       (comment "** Warning: Unbound var "
                                                (varRef-var expr) "\n")
                                       #f))))
                ((app? expr) (and (andAll (map (lambda (elm)
                                                 (isSafe? elm divVs))
                                               (app-rands expr)))
                                  (safeApplication?
                                   (app-rator expr)
                                   (map
                                    (lambda (elm)
                                            (evalExp elm
                                                     (append divVs
                                                             initialEnv)))
                                    (app-rands expr)))))
                (else (error "invalid expression: " expr)))))

;;; createCrash: create a crash node corresponding to an incomplete
;;; specification result:
(define createCrash
  (lambda (bbResult)
          (let ((oldCode (peBB-code bbResult)))
               (make-basicBlk
                (basicBlk-lbl oldCode) ()
                (make-return (make-const 'singleton 'run_time_crash))))))
```

D.13 The Useless Code Remover: ucr.s

```
;;;-------------------------------------------------------------------
;;;
;;; File dcr.s: contains functions for dead code removal
;;;
;;; The algorithm for dead code removal consists of identifying live
;;; variables at each basic block and removing assignment statements
;;; that attach values to the dead variables. This is done via the
;;; computation of def (defines) and use (uses) sets at each basic
;;; block. def set corresponds to the variables that are defined
;;; before being used in that basic block. use is the set of
;;; variables that are used before being defined. In and Out sets
;;; serve our purpose in finding live variables. In set of a basic
;;; block is the set of variables that must be live when that block
```

```
;;; is being executed. Out set shows which variables should be
;;; evaluated, clearly it is the union of the in sets of the reachable
;;; blocks from the current block. The algorithm terminates by removing
;;; those assignments which are not necessary within the basic block
;;; under consideration.
;;;
;;;-----------------------------------------------------------------

;;; defSet: given a  basic block compute its def set
;;; which are used in the live variable analysis
(define defSet
 (lambda (bblk)
         (defSetAux (basicBlk-assigns bblk)  ())))

;;; defSetAux: compute def set:
(define defSetAux
 (lambda (assignList usedSet)
         (cond ((null? assignList) ())
               (else
                (let*
                 ((curVar (string->symbol (assign-var (car assignList))))
                  (tempList (dependents (assign-expr (car assignList))))
                  (newUsed  (setUnion tempList usedSet))
                  (rest     (defSetAux (cdr assignList) newUsed)))
                 (if (member curVar newUsed)   ; found before?
                     rest
                     (setUnion (list curVar) rest)))))))

;;; useSet: given a basic block compute its use set:
(define useSet
 (lambda (bblk)
         (useSetAux (append (basicBlk-assigns bblk)      ; consider both
                            (list (basicBlk-jump bblk))) ; assigns and jump
                    ())))

;;; useSetAux: compute use set:
(define useSetAux
 (lambda (instList defd)
         (cond ((null? (cdr instList))   ; it is the jump
                (if (return? (car instList))
                    (setDifference
                     (dependents (return-exp (car instList)))
                     defd)
                    (setDifference    ; condJump
                     (dependents (condJump-expr (car instList)))
                     defd)))
               (else (setUnion
                      (setDifference
                       (dependents (assign-expr (car instList))) defd)
                      (useSetAux (cdr instList)
                                 (setUnion
                                  (list (string->symbol
                                         (assign-var (car instList))))
                                  defd)))))))

;;; dependents: given an expression return the set of dependents:
```

```
(define dependents
 (lambda (expr)
         (cond ((const? expr) ())      ; a constant depends on nothing
               ((varRef? expr) (list (varRef-var expr)))
               ((app? expr) (apply setUnion (map dependents (app-rands expr))))
               (else (error "invalid expression: " expr)))))

;;; dcr: dead code remover, receive an ast return another
;;; ast which have no dead code:
(define dcr
 (lambda (prg)
         (make-pgm (pgm-readBlk prg)
                   (eliminateDeads
                    (pgm-basicBlks prg)
                    (dcrAux (pgm-basicBlks prg)
                            (prepareDefAndUse (pgm-basicBlks prg)))))))

;;; attr is associated with each basic block keeping the in out def and
;;; use sets:
(define-structure attr lbl flag in out def use)

;;; prepareDefAndUse: return a list of attr structures for basicBlocks
(define prepareDefAndUse
 (lambda (allBlocks)
         (map (lambda (blk)
                      (make-attr
                       (basicBlk-lbl blk) #f ; not changed
                       () 'ERR ; out not defined yet
                       (defSet blk)
                       (useSet blk)))
              allBlocks)))

;;; dcrAux: given program compute in and out sets
(define dcrAux
 (lambda (bblks attrList)
         (let ((newAttr (alterAttrList bblks attrList attrList)))
              (if (orAll (map attr-flag newAttr)) ; empty list must be false
                  (dcrAux bblks (map resetFlag newAttr))
                  (map attr-out newAttr)))))    ; return out lists

;;; resetFlag: reset flag to #f, the rest is copied along..
(define resetFlag
  (lambda (elm)
          (make-attr
           (attr-lbl elm) #f
           (attr-in elm) (attr-out elm)
           (attr-def elm) (attr-use elm))))

;;; alterAttrList: compute in and outs alter flag correspondingly
(define alterAttrList
 (lambda (bblks iterateAttr oldAttr)
         (cond ((null? bblks) oldAttr)
               (else (alterAttrList (cdr bblks) (cdr iterateAttr)
                                    (singleBlk (car bblks)
                                               (car iterateAttr)
                                               oldAttr))))))
```

```
;;; singleBlk: compute in and out list at this particular block:
(define singleBlk
 (lambda (blk curAttr oldAttrs)
         (let ((newOut (let ((nexts (successors (basicBlk-jump blk))))
                         (if (null? nexts)
                             (dependents (return-exp (basicBlk-jump blk)))
                             (setUnion (dependents (condJump-expr
                                                    (basicBlk-jump blk)))
                                       (computeOut nexts oldAttrs))))))
               (updateAttr oldAttrs (setUnion (attr-use curAttr)
                                              (setDifference
                                               newOut
                                               (attr-def curAttr)))
                           newOut (attr-lbl curAttr)))))

;;; successors: return the labels of the successor nodes:
(define successors
 (lambda (jmp)
         (cond ((return? jmp) ())   ; no successor of return
               ((condJump? jmp) (list (condJump-lbl1 jmp)
                                      (condJump-lbl2 jmp)))
               (else (error "invalid jump: " jmp)))))

;;; computeOut: out set of a node:
(define computeOut
 (lambda (succs attrList)
         (cond ((null? succs) ())
               (else (setUnion (getInSet (car succs) attrList)
                               (computeOut (cdr succs) attrList))))))

;;; getInSet: return the in set of a node in attrList
(define getInSet
 (lambda (name attrList)
         (cond ((null? attrList) (error "something wrong with attributes"))
               ((equal? (attr-lbl (car attrList)) name)
                (attr-in (car attrList)))
               (else (getInSet name (cdr attrList))))))

;;; updateAttr: update attr list, update flag if necessary
(define updateAttr
 (lambda (oldList newIn newOut which)
         (cond ((null? oldList) ())
               ((equal? (attr-lbl (car oldList)) which)
                (cons (updateThisAttr (car oldList) newIn newOut)
                      (cdr oldList)))
               (else (cons (car oldList)
                           (updateAttr (cdr oldList)
                                       newIn newOut which))))))

;;; updateThisAttr: update a particular attribute record:
(define updateThisAttr
 (lambda (old newIn newOut)
         (make-attr
          (attr-lbl old)
          (not (setEqual? newIn (attr-in old)))
```

```
          newIn newOut (attr-def old) (attr-use old))))

;;; eliminateDeads: using out sets remove unnecessary assignments:
(define eliminateDeads
 (lambda (bblks outLists)
         (map removeDeadAssigns bblks outLists)))

;;; removeDeadAssigns: return only the necessary part:
(define removeDeadAssigns
 (lambda (blk out)
         (let ((newBlk (make-basicBlk
                        (basicBlk-lbl blk)
                        (deadAssign (basicBlk-assigns blk) out
                                    (cdr (append (basicBlk-assigns blk)
                                                 (list (basicBlk-jump blk)))))
                        (basicBlk-jump blk))))
              (if (equal? (length (basicBlk-assigns blk))
                          (length (basicBlk-assigns newBlk)))
                  newBlk   ; iteration complete
                  (removeDeadAssigns newBlk out)))))  ; go on..

;;; deadAssign: remove or retain assignments:
;;; to remove an assignment it should assign to a variable
;;; that is not in the out set of that basic block, but that
;;; variable should also be not referenced within the same basic
;;; block, after that assignment takes place.
;;;  i.e. let the assignments be:
;;;           a := 3;
;;;           b := a+2;
;;; and say a is dead but b is live. removing the dead assignment
;;; a would not help in this case since b would take a wrong
;;; value in that case.
(define deadAssign
 (lambda (assigns out checkList)
         (cond ((null? assigns) ())
               ((or (member (string->symbol (assign-var (car assigns))) out)
                    (furtherUsed (string->symbol (assign-var (car assigns)))
                                 checkList))
                (cons (car assigns)
                      (deadAssign (cdr assigns) out (cdr checkList))))
               (else (deadAssign (cdr assigns) out (cdr checkList))))))

;;; furtherUsed: a var may not be in out, but may be used in some
;;; other point in the same basic block:
(define furtherUsed
  (lambda (var checkList)
          (member var (apply setUnion (map collectDepends checkList)))))

;;; collect depends: return dependents..
(define collectDepends
  (lambda (instr)
          (cond ((assign? instr) (dependents (assign-expr instr)))
                ((condJump? instr) (dependents (condJump-expr instr)))
                ((return? instr) (dependents (return-exp instr)))
                (else (error "invalid instruction " instr)))))
```

D.14 Guard system of ILPOS: guards.s

```
;;;-------------------------------------------------------------------
;;;
;;; File guards.s: this file contains guards with respect to the
;;; library functions, i.e. when they will go wrong..
;;;
;;;-------------------------------------------------------------------

;;; safeApplication?: is the function safe for these data?
(define safeApplication?
  (lambda (rator rands)
          (if ((selectGuard rator) rands) #t
              (begin (comment "** Warning the call <" rator "> has problems.\n")
                     #f))))

;;; define guards:

;;; head guard: single arity, argument must be non empty list.
(define headGuard
  (lambda (elm)
          (and (equal? (length elm) 1)
               (list? (car elm))
               (not (null? (car elm))))))

;;; consGuard: arity: 2, second must be list.
(define consGuard
  (lambda (elm)
          (and (equal? (length elm) 2)
               (list? (cadr elm)))))

;;; appendGuard: arity: n, all must be lists
(define appendGuard
  (lambda (elm)
          (andAll (map list? elm))))

;;; mustbeListGuard: single arity list.
(define mustbeListGuard
  (lambda (elm)
          (and (equal? (length elm) 1)
               (list? (car elm)))))

;;; eqGuard: arity: 2
(define eqGuard
  (lambda (elm)
          (equal? (length elm) 2)))

;;; noGuard: nothing imposed:
(define noGuard
  (lambda (elm) #t))

;;; twoArgNumsGuard: two arguments, both must be numbers:
(define twoArgNumsGuard
  (lambda (elm)
          (and (equal? (length elm) 2)
               (numberL (car elm))
               (numberL (cadr elm)))))
```

```
;;; divGuard: two arguments, both must be numbers, second non-zero:
(define divGuard
  (lambda (elm)
          (and (equal? (length elm) 2)
               (numberL (car elm))
               (numberL (cadr elm))
               (not (equalL (cadr elm) '0)))))

;;; oneArgNumGuard: two arguments, both must be numbers:
(define oneArgNumGuard
  (lambda (elm)
          (and (equal? (length elm) 1)
               (numberL (car elm)))))

;;; oneArgNumPosGuard: one arg, number, positive
(define oneArgNumPosGuard
  (lambda (elm)
          (and (equal? (length elm) 1)
               (numberL (car elm))
               (>= (symbol->number (car elm)) 0))))

;;; selectGuard: for each function specify the guard to be used:
(define selectGuard
  (lambda (func)
          (cond ((equal? func "hd")                headGuard)
                ((equal? func "cons")              consGuard)
                ((equal? func "first_instruction") headGuard)
                ((equal? func "rest")              headGuard)
                ((equal? func "list")              noGuard)
                ((equal? func "member")            consGuard)
                ((equal? func "append")            appendGuard)
                ((equal? func "new_tail")          noGuard)
                ((equal? func "eq")                eqGuard)
                ((equal? func "transition")        noGuard)
                ((equal? func "firstsym")          mustbeListGuard)
                ((equal? func "tl")                mustbeListGuard)
                ((equal? func "add")               twoArgNumsGuard)
                ((equal? func "mul")               twoArgNumsGuard)
                ((equal? func "sub")               twoArgNumsGuard)
                ((equal? func "div")               divGuard)
                ((equal? func "intdiv")            divGuard)
                ((equal? func "odd")               oneArgNumGuard)
                ((equal? func "even")              oneArgNumGuard)
                ((equal? func "lt")                twoArgNumsGuard)
                ((equal? func "gt")                twoArgNumsGuard)
                ((equal? func "lte")               twoArgNumsGuard)
                ((equal? func "gte")                twoArgNumsGuard)
                ((equal? func "inform_sga")        noGuard)
                ((equal? func "sqrt")              oneArgNumPosGuard)
                ((equal? func "exp")               oneArgNumGuard)
                (else (error "guard undefined for" func)))))
```

D.15 Code minimizer of ILPOS: minimize.s

```
;;;-------------------------------------------------------------------
;;;
;;; File minimize.s: contains functions for minimization of the program.
;;;
;;; The minimization algorithm is very much like the finite state
;;; machine minimization algorithms that appear in the literature.
;;; We define an equivalent code concept as follows: two basic
;;; blocks are operationally equivalent if they make the same
;;; assignments to same variables, i.e. some sort of structural
;;; equivalence. Then we say that two blocks are equivalent if
;;; they are code equivalent and their jumps are to those blocks which
;;; are also code equivalent. Note that this signals a closure algorithm.
;;;
;;; Clearly this definition defines an equivalence relation on the
;;; blocks of the program. Once those equivalence classes are found
;;; all the program is transformed into an equivalent code that only
;;; has N blocks where N is the number of equivalence classes. This
;;; clearly improves the efficiency.
;;;
;;; One improvement to the minimization of the blocks is the following:
;;; the analysis of codes to determine whether they are equivalent is
;;; not as good as it can be. We look at exact structural equivalence.
;;; This can be improved by allowing more flexible equivalence definitions.
;;; But note that this is a difficult problem since, essentially, you
;;; are trying to determine whether two functions (i.e. basic blocks)
;;; compute exactly the same function. This problem is an unsolvable problem
;;; for any arbitrary two Turing Machines. Of course the current problem is
;;; somewhat simpler but not at all tricky. Of course one may offer a
;;; better algorithm that applies some checks for enabling the
;;; identification of operationally equivalent blocks.
;;;
;;;-------------------------------------------------------------------

;;; minimize the ast using equivalent code concept:
(define minAst
  (lambda (ast)
          (let ((piFinal (computePi (findFlow (pgm-basicBlks ast))
                                    (pgm-basicBlks ast))))
               (make-pgm (pgm-readBlk ast)
                         (reducePgm (pgm-basicBlks ast) piFinal)))))

;;; findFlow: extract flow of the program, i.e. which blocks are
;;; followed by which.
(define findFlow
  (lambda (bblks)
          (map (lambda (blk)
                       (list (basicBlk-lbl blk)
                             (nextStates (basicBlk-jump blk))))
               bblks)))

;;; nextStates: which states follow this block?
(define nextStates
  (lambda (jmp)
          (cond ((condJump? jmp) (list (condJump-lbl1 jmp)
                                       (condJump-lbl2 jmp)))
```

```
               ((return? jmp) ())  ; no successor of return..
               (else (error "invalid jump: " jmp)))))

;;; computePi: return the partition of equivalent states:
(define computePi
  (lambda (flow bblks)
          (let ((pi0 (computePi0 (map car flow) bblks)))
               (computePiAux pi0 flow))))

;;; computePiAux: start from pi0 and find final partition
;;; the main algorithm is that two blocks are in the same
;;; block in the ith partition if they were in the same
;;; block in the (i-1)st partition and their next states are
;;; in the same block in the (i-1)st partition. The algorithm
;;; stops when ith and (i+1)st partitions are the same.
(define computePiAux
  (lambda (piN flow)
          (let ((piN+1 (nextPi piN flow)))
               (if (equal? (length piN+1) (length piN))
                   piN
                   (computePiAux piN+1 flow)))))

;;; computePi0: find initial partition:
;;; two nodes are in the same partition if they are code equivalent
(define computePi0
  (lambda (states bblks)
          (genPi0 (list (list (car states))) (cdr states) bblks)))

;;; genPi0: construct initial partition by inspecting the code..
(define genPi0
  (lambda (partSoFar rest bblks)
          (cond ((null? rest) partSoFar)
                (else (genPi0 (place partSoFar (car rest) bblks)
                              (cdr rest)
                              bblks)))))

;;; place: place a node in a partition
(define place
  (lambda (soFar elm bblks)
          (cond ((null? soFar) (list (list elm)))
                ((isEquivalent? (caar soFar) elm bblks)
                 (cons (cons elm (car soFar)) (cdr soFar)))
                (else (cons (car soFar) (place (cdr soFar) elm bblks))))))

;;; isEquivalent?:  are these two codes semantically equivalent?
;;; two blocks are semantically equivalent if they have the same
;;; assignments + they terminate with return with the same expression
;;; or they terminate with a conditional jump with the same expression
;;; no matter what their jumps are
(define isEquivalent?
  (lambda (l1 l2 bblks)
          (let ((blk1 (getBlk bblks l1))
                (blk2 (getBlk bblks l2)))
               (and (equal? (length (basicBlk-assigns blk1))
                            (length (basicBlk-assigns blk2)))
                    (sameJump? (basicBlk-jump blk1) (basicBlk-jump blk2))
```

```
                    (sameExpr? (getJumpExpr (basicBlk-jump blk1))
                               (getJumpExpr (basicBlk-jump blk2)))
                    (equal? (map assign-var (basicBlk-assigns blk1))
                            (map assign-var (basicBlk-assigns blk2)))
                    (andAll (map sameExpr?
                                 (map assign-expr (basicBlk-assigns blk1))
                                 (map assign-expr (basicBlk-assigns blk2))))))))

;;; sameJump?: both if or both return
(define sameJump?
  (lambda (j1 j2)
          (or (and (return? j1) (return? j2))
              (and (condJump? j1) (condJump? j2)))))

;;; getJumpExpr : return the expression associated with the jump
(define getJumpExpr
  (lambda (jmp)
          (cond ((return? jmp) (return-exp jmp))
                ((condJump? jmp) (condJump-expr jmp))
                (else (error "invalid jump: " jmp)))))

;;; sameExpr? are the expressions same?
(define sameExpr?
  (lambda (exp1 exp2)
          (cond ((and (const? exp1) (const? exp2))
                 (equal? (const-val exp1) (const-val exp2)))
                ((and (varRef? exp1) (varRef? exp2))
                 (equal? (varRef-var exp1) (varRef-var exp2)))
                ((and (app? exp1) (app? exp2))
                 (and
                  (equal? (app-rator exp1) (app-rator exp2))
                  (andAll (map sameExpr? (app-rands exp1) (app-rands exp2)))))
                (else #f))))

;;; nextPi: given a partition, refine it to find next..
(define nextPi
  (lambda (curPi flow)
          (apply append (map (lambda (block) (refine block flow curPi))
                             curPi))))

;;; refine: given a block in a partition, refine it..
(define refine
  (lambda (block flow prevPi)
          (refineAux (list (list (car block))) (cdr block) flow prevPi)))

;;; refineAux: place new nodes..
(define refineAux
  (lambda (prev rest flow prevPi)
          (cond ((null? rest) prev)
                (else (refineAux (refineInsert prev (car rest) flow prevPi)
                                 (cdr rest) flow prevPi))))))

;;; refineInsert: place or create new block:
(define refineInsert
  (lambda (prev elm flow prevPi)
          (cond ((null? prev) (list (list elm)))
```

```
                ((successorsInSameBlock? elm (caar prev) flow prevPi)
                 (cons (cons elm (car prev)) (cdr prev)))
                (else (cons (car prev) (refineInsert (cdr prev)
                                                     elm flow prevPi))))))

;;; successorsInSameBlock? check if the next states are in the same block..
(define successorsInSameBlock?
  (lambda (l1 l2 flow prevPi)
          (letrec ((search
                    (lambda (l f)
                            (cond ((null? f) (error "something wrong"))
                                  ((equal? l (caar f)) (cadar f))
                                  (else (search l (cdr f)))))))
                  (let ((next1 (search l1 flow))
                        (next2 (search l2 flow)))
                       (cond ((and (null? next1) (null? next2)) #t)
                             ((or  (null? next1) (null? next2)) #f)
                             (else (and
                                    (isTogether? (car next1)
                                                 (car next2) prevPi)
                                    (isTogether? (cadr next1)
                                                 (cadr next2) prevPi))))))))

;;; isTogether?: are these two in the same block of the partition
(define isTogether?
  (lambda (l1 l2 partition)
          (cond ((null? partition) #f)
                ((and (member l1 (car partition))
                      (member l2 (car partition))) #t)
                (else (isTogether? l1 l2 (cdr partition))))))

;;; reducePgm: reduce the program to have the minimum number of states:
(define reducePgm
  (lambda (bblks pi)
          (reducePgmAux bblks pi (allFalse (length pi)))))

;;; allFalse: return a list of length n with all true :
(define allFalse
  (lambda (n)
          (cond ((equal? n 0) ())
                (else (cons #f (allFalse (- n 1)))))))

;;; reducePgmAux: reduce the program by generating only the necessary code:
(define reducePgmAux
  (lambda (bblks pi used)
          (cond ((null? bblks) ())
                (else (let* ((blockNo (indexInPi (basicBlk-lbl (car bblks))
                                                 pi 0))
                             (done (isDone? blockNo used)))
                            (if done
                                (reducePgmAux (cdr bblks) pi used)
                                (cons (xformBlock (car bblks) blockNo pi)
                                      (reducePgmAux (cdr bblks) pi
                                                    (alterUsed
                                                     used blockNo)))))))))
```

```
;;; indexInPi: where does this label go in pi:
(define indexInPi
  (lambda (lbl pi cnt)
          (cond ((null? pi) (error "something wrong with pi"))
                ((member lbl (car pi)) cnt)
                (else (indexInPi lbl (cdr pi) (+ cnt 1))))))

;;; isDone?: is the nth entry true?
(define isDone?
  (lambda (n lst)
          (cond ((equal? n 0) (car lst))
                (else (isDone? (- n 1) (cdr lst))))))

;;; alterUsed: change n'th entry to #t
(define alterUsed
  (lambda (lst n)
          (cond ((equal? n 0) (cons #t (cdr lst)))
                (else (cons (car lst) (alterUsed (cdr lst) (- n 1)))))))

;;; xformBlock: transfer the block into the equivalent minimized form:
(define xformBlock
  (lambda (blk no pi)
          (make-basicBlk
           (string-append "lpe" (number->string no))
           (basicBlk-assigns blk)
           (xformJump (basicBlk-jump blk) pi))))

;;; xformJump: transfer jumps:
(define xformJump
  (lambda (jmp pi)
          (cond ((return? jmp) jmp)
                (else (make-condJump (condJump-expr jmp)
                                     (constNewLabel (condJump-lbl1 jmp) pi)
                                     (constNewLabel (condJump-lbl2 jmp) pi))))))

;;; constNewLabel: return new label name:
(define constNewLabel
  (lambda (old pi)
          (string-append "lpe" (number->string (indexInPi old pi 0)))))
```

D.16 Code linearizer of ILPOS: linearize.s

```
;;;--------------------------------------------------------------------
;;;
;;; File linearize.s: contains functions for linearization and
;;; canonicalization of the residual code.
;;;
;;; The minimization procedure gives us a minimum program (in the
;;; sense that it contains the minimum number of basic blocks).
;;; Due to the equivalent block concept it may turn around that
;;; a conditional jump statement may jump to the same (i.e
;;; semantically equivalent one) blocks. This is exploited by
;;; the minimizer by jumps to a node that represents the
;;; equivalence class of those two blocks. Clearly such a
;;; dynamic conditional jump is equivalent to a goto without
```

```
;;; any regard to the test expression. The linearization
;;; functions analyze the program for such conditional jumps and
;;; convert them to goto's. After this process there comes the
;;; idea of goto transition since no residual code need to have
;;; unconditional goto's (same idea as used in the partial evaluator).
;;; So another pass through the program compresses all these goto's.
;;;
;;; It is clear that this process will not be needed too much. This
;;; essentially requires the program to jump into equivalent states
;;; according to some dynamic condition. Nevertheless this analysis
;;; must be done because of two reasons: 1. to be complete, 2. strange
;;; static data may result in many dead code that is removed and
;;; thus making inequivalent states equivalent.
;;;
;;; After linearization one more pass is made for canonicalization. This
;;; pass ensures the labels to be from lpe0 to some lpeN by relabeling
;;; all the basic blocks. This phase also removes any block that becomes
;;; inaccessible by all the transformations that are done by the
;;; previous phases. So the resulting code becomes well-labeled and,
;;; in a sense, fully-connected with respect to the static data
;;; that it has been specialized for.
;;;
;;;----------------------------------------------------------------------

;;; linAst: given an ast first linearize then canonicalize and return
;;; a new ast for the resulting program:
(define linAst
  (lambda (ast)
          (make-pgm (pgm-readBlk ast)
                    (canonize
                     (onlyReachables (linearize (pgm-basicBlks ast)))))))

;;; linearize: check and linearize the basic blocks:
(define linearize
  (lambda (bblks)
          (if (orAll (map compressable? (map basicBlk-jump bblks)))
              (let ((allGotos (map convertToGoto bblks)))
                   (compressAll allGotos allGotos))
              bblks)))

;;; isLinearizable?: is the program linearizable?
(define isLinearizable?
  (lambda (ast)
          (orAll (map compressable?
                      (map basicBlk-jump (pgm-basicBlks ast))))))

;;; compressable?: is the block compressable? i.e. does it have a
;;; conditional jump that jumps to the same block?
(define compressable?
  (lambda (jmp)
          (cond ((return? jmp) #f)
                ((condJump? jmp) (equal? (condJump-lbl1 jmp)
                                         (condJump-lbl2 jmp)))
                (else (error "something wrong with jump: " jmp)))))

;;; convertToGoto: change the if to a jump if it is a compressible block
```

```
;;; this saves the time for computing the dynamic expression at run time,
;;; but creates goto's to be compressed later.
(define convertToGoto
  (lambda (blk)
          (if (compressable? (basicBlk-jump blk))
              (make-basicBlk
               (basicBlk-lbl blk)          ; copy along.
               (basicBlk-assigns blk)      ; copy along.
               (make-goto (condJump-lbl1 (basicBlk-jump blk))))
              blk)))

;;; compressAll: compress the goto transition
;;; attach the assignments of the blocks that are direct followers of
;;; this block, the classical goto compression stuff:
(define compressAll
  (lambda (bblks whole)
          (cond ((null? bblks) ())
                ((goto? (basicBlk-jump (car bblks)))
                 (cons (attachFollowers (car bblks) whole)
                       (compressAll (cdr bblks) whole)))
                (else (cons (car bblks) (compressAll (cdr bblks) whole))))))

;;; attachFollowers: construct a new block by attaching the followers:
(define attachFollowers
  (lambda (blk whole)
          (if (goto? (basicBlk-jump blk))
              (let ((follower (getBlk whole (goto-lbl (basicBlk-jump blk)))))
                   (attachFollowers
                    (make-basicBlk
                     (basicBlk-lbl blk)
                     (append (basicBlk-assigns blk)       ;attach assignments
                             (basicBlk-assigns follower)) ;to old ones..
                     (basicBlk-jump follower)) whole))
              blk))) ; return if no follower exists

;;; onlyReachables: the above process may lead to some states
;;; which become unreachable, remove them: apply a classical
;;; reachable set algorithm:
(define onlyReachables
  (lambda (bblks)
          (keepOrRemoveBlocks bblks
                              (computeReachables
                               (list (basicBlk-lbl (car bblks)))
                               bblks))))

;;; computeReachables: which nodes are reachable?
;;; starting from initial node, add, in each pass, the accessible
;;; nodes to our initial set until no new node is added.
(define computeReachables
  (lambda (initSet bblks)
          (let ((newSet (setUnion
                         initSet               ; add to initials
                         (apply setUnion
                                (map           ; whatever follows..
                                 (lambda (lbl) (whatFollows lbl bblks))
                                 initSet)))))
```

```
              (if (equal? (length initSet) (length newSet)) ; any change?
                  initSet                                   ; no: finished
                  (computeReachables newSet bblks)))))      ; yes: go on..

;;; whatFollows: what follows this block?
;;; inspect the labels in goto's and condJumps..
(define whatFollows
  (lambda (lbl bblks)
          (let ((theJump (basicBlk-jump (getBlk bblks lbl))))
               (cond ((return? theJump) ())
                     ((goto? theJump) (list (goto-lbl theJump)))
                     ((condJump? theJump) (list (condJump-lbl1 theJump)
                                                (condJump-lbl2 theJump)))
                     (else (error "something wrong with jump: " theJump))))))

;;; keepOrRemoveBlocks: if a block is reachable keep it otherwise remove:
(define keepOrRemoveBlocks
  (lambda (bblks reachables)
          (cond ((null? bblks) ())
                ((member (basicBlk-lbl (car bblks)) reachables)
                 (cons (car bblks) (keepOrRemoveBlocks (cdr bblks) reachables)))
                (else (keepOrRemoveBlocks (cdr bblks) reachables)))))

;;; canonize: make the labels appear numbered from 0 to N-1 where N is
;;; the number of blocks in the final residual program..
(define canonize
  (lambda (bblks)
          (let ((lblList (map basicBlk-lbl bblks)))
               (map (lambda (blk) (renameBlk blk lblList)) bblks))))

;;; renameBlk: rename the block canonically:
(define renameBlk
  (lambda (blk nameList)
          (make-basicBlk
           (xformName (basicBlk-lbl blk) nameList) ; change the name of block
           (basicBlk-assigns blk)                  ; no change in assignments
           (let ((jmp (basicBlk-jump blk)))
                (cond ((return? jmp) jmp)          ; take care of jumps.
                      ((goto? jmp) (make-goto (xformName (goto-lbl jmp)
                                                         nameList)))
                      ((condJump? jmp) (make-condJump
                                        (condJump-expr jmp)
                                        (xformName (condJump-lbl1 jmp)
                                                   nameList)
                                        (xformName (condJump-lbl2 jmp)
                                                   nameList)))
                      (else (error "illegal jump: " jmp)))))))

;;; xformName: return the canonical name: the new name is formed by attaching
;;; a number to the generic name lpe, the number corresponds to the index
;;; of that name in the reachable list..
(define xformName
  (lambda (old nameList)
          (string-append "lpe" (number->string (returnIndex old nameList 0)))))

;;; returnIndex: return the place of the elm:
```

```
(define returnIndex
  (lambda (old lst cnt)
          (cond ((null? lst) (error "something wrong in returnIndex"))
                ((equal? (car lst) old) cnt)    ; found..
                (else (returnIndex old (cdr lst) (+ cnt 1))))))
```

D.17 The Symbolic Gain Analyzer: symbSpeedUp.s

```
;;;-------------------------------------------------------------------
;;;
;;; File symbSpeedUp.s: contains functions for collecting statistics
;;; on the running times of the L programs.
;;;
;;; The symbolic speed-up will be determined by comparing the
;;; corresponding vectors for the original and specialized programs.
;;; The cost vector (CV) for an L program is a vector of 10 elements,
;;; each showing a different counts on that particular run. The elements
;;; are as follows:
;;;
;;;     jumps, assignments, variable references, decisions, eq call,
;;;     hd call, rest call, list call, append call, search, intern, other calls
;;;
;;; other calls entry is the total number of calls to the library
;;; functions other than eq, hd, rest, list and append.
;;;
;;; the search entry is slightly different. It is used to simulate
;;; no cost searches and only applied in the SGA analysis phase.
;;; intern shows the internal communication and has nothing to do with
;;; the actual analysis.
;;;
;;;-------------------------------------------------------------------

;;; indices into the CV vector:
;;; to add a new index:
;;;   1. define an index for it
;;;   2. alter the CV definitions 2 times below to include them, all 0's part
;;;     i.e. increment the index count by 1.
;;;   3. in recordLibCall, include this new function
;;;   4. in printAllResults, include the output line
;;;   5. alter weightsSGA vector to reflect the speed up
(define jumpCV      0)
(define assignCV    1)
(define varRefCV    2)
(define decisionCV  3)
(define eqCV        4)
(define hdCV        5)
(define restCV      6)
(define listCV      7)
(define appendCV    8)
(define searchCV    9)
(define internCV    10)
(define addCV       11)
(define subCV       12)
(define mulCV       13)
(define divCV       14)
```

```
(define oddCV       15)
(define evenCV      16)
(define othersCV    17)

;;; allZeros: produce a list of all zeros:
(define allZeros
  (lambda (n)
          (cond ((equal? n 0) ())
                (else (cons '0 (allZeros (- n 1)))))))

;;; the CV vector: the number of 0's must be: last index + 1.
(define CV (allZeros 18)) ; initial value..

;;; File analyzer:
(define lCV
  (lambda (fname)
          (lCVAst (parseL (readPgm fname)))))

;;; Ast analyzer:
(define lCVAst
  (lambda (ast)
          (begin (set! CV (allZeros 18))  ; reset
                 (let ((val (runCV ast)))
                      (list val CV)))))

;;; run the program and construct CV:
(define runCV
  (lambda (ast)
          (interpResult-val
           (initiateCV (pgm-basicBlks ast)
                       (interpResult-env
                        (loadVars (pgm-readBlk ast) initialEnv))))))

;;; initiate: start execution:
(define initiateCV
  (lambda (bblks env)
          (cond ((null? bblks) (make-interpResult 'NOVALUE env))
                (else (executeCV bblks (car bblks) env)))))

;;; executeCV: execute the pgm, record modifications into CV:
(define executeCV
  (lambda (pgm curBlk env)
          (let* ((newEnv (performAssignsCV (basicBlk-assigns curBlk) env))
                 (nextBlkInfo (whereToGoCV (basicBlk-jump curBlk) newEnv)))
                (if (equal? (car nextBlkInfo) 'TERMINATE)
                    (make-interpResult (cdr nextBlkInfo) newEnv)
                    (begin (incrCV! jumpCV)
                           (if (condJump? (basicBlk-jump curBlk))
                               (incrCV! decisionCV) #t)
                           (executeCV pgm (getBlk pgm (car nextBlkInfo))
                                      newEnv))))))

;;; whereToGoCV: execute and decide jumps:
(define whereToGoCV
  (lambda (jstmt env)
          (cond ((goto? jstmt) (cons (goto-lbl jstmt) 'NOVALUE))
```

```
                ((return? jstmt) (cons 'TERMINATE
                                       (evalExpCV (return-exp jstmt) env)))
                ((condJump? jstmt) (cons
                                    ((if (evalExpCV (condJump-expr jstmt) env)
                                         condJump-lbl1
                                         condJump-lbl2)
                                     jstmt) 'NOVALUE))
                (else (error "Invalid jump: " jstmt)))))

;;; performAssignsCV: evaluate and form the new assignments: return env
(define performAssignsCV
  (lambda (assigns env)
          (cond ((null? assigns) env)
                (else (performAssignsCV (cdr assigns)
                                        (doAssignCV (car assigns) env))))))

;;; doAssignCV: perform a single assignment: return the new environment.
(define doAssignCV
  (lambda (astmt env)
          (incrCV! assignCV)
          (update env (assign-var astmt) (evalExpCV (assign-expr astmt) env))))

;;; evalExpCV: return value of the expression, update CV accordingly..
(define evalExpCV
  (lambda (exp env)
          (cond ((const? exp) (cond ((equal? (const-type exp) 'singleton)
                                     (beautify (const-val exp)))
                                    ((equal? (const-type exp) 'listing)
                                     (formList (beautify (const-val exp)) env))
                                    (else (error "Unknown const" exp))))
                ((varRef? exp) (begin
                                 (incrCV! varRefCV)
                                 (lookUp env (symbol->string (varRef-var exp)))))
                ((app? exp) (applyCV
                             (app-rator exp)
                             (lookUp env (string->symbol (app-rator exp)))
                             (map (lambda (e) (evalExpCV e env))
                                  (app-rands exp))))
                (else (error "Unknown exp type: " exp)))))

;;; applyCV: first record the used function then apply it:
(define applyCV
  (lambda (fnName fn args)
          (recordLibCall fnName)
          (apply fn args)))

;;; recordLibCall: increment according to the function name:
(define recordLibCall
  (lambda (fnName)
          (incrCV! (cond ((equal? fnName "eq") eqCV)
                         ((equal? fnName "hd") hdCV)
                         ((equal? fnName "rest") restCV)
                         ((equal? fnName "list") listCV)
                         ((equal? fnName "append") appendCV)
                         ((equal? fnName "inform_sga") internCV)
                         ((equal? fnName "add") addCV)
```

```
                        ((equal? fnName "sub") subCV)
                        ((equal? fnName "mul") mulCV)
                        ((equal? fnName "div") divCV)
                        ((equal? fnName "odd") oddCV)
                        ((equal? fnName "even") evenCV)
                        (else othersCV)))))

;;; incrCV!: increment the entry by 1:
(define incrCV!
  (lambda (which)
          (if SGAstopCount #t  ; then do not do anything..
              (letRec ((alterCV (lambda
                                  (which what)
                                  (if (equal? which 0)
                                      (cons (+ 1 (car what)) (cdr what))
                                      (cons (car what)
                                            (alterCV (- which 1) (cdr what)))))))
                       (set! CV (alterCV which CV))))))

;;; document results as a list: sga: symbolic gain analyzer..
(define sga
  (lambda (fname)
          (let ((sgaPort (open-output-file (string-append fname ".peo.sga"))))
               (mdisplay "Preparing for gain analysis on \"" fname "\"..\n")
               (sgaIntroduce sgaPort)
               (set! SGAstopCount #f)
               (let* ((f1 (string-append fname ".lp"))
                      (f2 (string-append fname ".pe.lp"))
                      (f3 (string-append fname ".peo.lp"))
                      (ast1 (parseL (readPgm f1)))
                      (ast2 (parseL (readPgm f2)))
                      (ast3 (parseL (readPgm f3)))
                      (bblk1 (length (pgm-basicBlks ast1)))
                      (bblk2 (length (pgm-basicBlks ast2)))
                      (bblk3 (length (pgm-basicBlks ast3)))
                      (noas1 (howManyAssigns ast1))
                      (noas2 (howManyAssigns ast2))
                      (noas3 (howManyAssigns ast3)))
                     (sgaComment sgaPort
                                 "Working dir : "
                                 (directory-namestring
                                  (working-directory-pathname))
                                 "\n\nSymbolic analysis performed on:\n\n"
                                 "Input    : " f1 "\nPE file  : " f2
                                 "\nResidual : " f3 "\n\n"
                                 "Symbolic Analysis Results: "
                                 "(only the relevant entries are printed.)\n\n")
                     (mdisplay "Analyzing \"" f1 "\"..\n")
                     (let ((stat1 (cadr (lcvAst ast1))))
                          (mdisplay "Analyzing \"" f2 "\"..\n")
                          (let ((stat2 (cadr (lcvAst ast2))))
                               (mdisplay "Analyzing \"" f3 "\"..\n")
                               (let* ((stat3 (cadr (lcvAst ast3)))
                                      (gainList
                                       (determineGain
                                        (append stat1 (list bblk1 noas1))
```

```
                                        (append stat2 (list bblk2 noas2))
                                        (append stat3 (list bblk3 noas3))))
                                       (costs (map (lambda (elm)
                                                     (list (car elm)
                                                           (cadr elm)
                                                           (caddr elm)))
                                                   gainList))
                                       (onlyCosts (reverse
                                                   (cddr (reverse costs)))))
                                      (mdisplay "Writing SGA results to \""
                                                fname ".peo.sga\"..")
                                      (printAllResults gainList sgaPort)
                                      (sgaComment
                                       sgaPort
                                       "\nCost vector is   : " weightsSGA
                                       "\n\nCost of Original : "
                                       (findCost (map car onlyCosts) weightsSGA)
                                       "\nCost of PE Only  : "
                                       (findCost (map cadr onlyCosts)
                                                 weightsSGA)
                                       "\nCost of Residual : "
                                       (findCost (map caddr onlyCosts)
                                                 weightsSGA)
                                       "\n\nGain by PE Only  : "
                                       (findOverallGain
                                        (map car onlyCosts)
                                        (map cadr onlyCosts))
                                       "\nGain by Post Opts: "
                                       (diffCost
                                        (map car onlyCosts)
                                        (map cadr onlyCosts)
                                        (map caddr onlyCosts))
                                       "\n\nOverall Gain is  : "
                                       (findOverallGain
                                        (map car onlyCosts)
                                        (map caddr onlyCosts))
                                       "\nThe improvement  : "
                                       (percentImpro (map car onlyCosts)
                                                     (map caddr onlyCosts)))
                                      (sgaComment sgaPort " %\n\nSGA completed "
                                                  "successfully.\n")
                                      (close-output-port sgaPort))))))))

;;; determineGain: merge lists with analysis results..
(define determineGain
  (lambda (l1 l2 l3)
          (cond ((null? l1) ())
                (else (cons (singleGainList (car l1) (car l2) (car l3))
                            (determineGain (cdr l1) (cdr l2) (cdr l3)))))))

;;; singleGainList: perform operations on a single pair:
(define singleGainList
  (lambda (v1 v2 v3)
          (let* ((properDiv (lambda (a b)
                                    (if (and (equal? a 0) (equal? b 0)) "-"
                                        (if (equal? b 0) "inf"
```

```
                                        (exact->inexact (/ a b))))))
                 (r1 (properDiv v1 v2))
                 (r2 (properDiv v1 v3)))
                (list v1 v2 v3 r1 r2 (if (and (number? r1) (number? r2))
                                         (- r2 r1)
                                         (if (and
                                              (equal? r2 "inf")
                                              (equal? r1 "inf"))
                                             0 "-"))))))

;;; printAllResults: display in tabular form:
(define printAllResults
  (lambda (l prt)
          (sgaComment prt "                 Original   PE Only  Post-Opts"
                      "     Gain1     Gain2  Opt Gain\n")
          (sgaComment prt "                  --------   -------  ---------"
                      "    ------    ------  --------\n")
          (singleLinePrint prt "( 0)    jumps:" l 0)
          (singleLinePrint prt "( 1)  assigns:" l 1)
          (singleLinePrint prt "( 2) var refs:" l 2)
          (singleLinePrint prt "( 3)decisions:" l 3)
          (singleLinePrint prt "( 4)       eq:" l 4)
          (singleLinePrint prt "( 5)     head:" l 5)
          (singleLinePrint prt "( 6)     rest:" l 6)
          (singleLinePrint prt "( 7)     list:" l 7)
          (singleLinePrint prt "( 8)   append:" l 8)
          (internInfo      prt "( 9)   search:" l 9)
          (internInfo      prt "(10)   intern:" l 10)
          (singleLinePrint prt "(11)      add:" l 11)
          (singleLinePrint prt "(12)      sub:" l 12)
          (singleLinePrint prt "(13)      mul:" l 13)
          (singleLinePrint prt "(14)      div:" l 14)
          (singleLinePrint prt "(15)      odd:" l 15)
          (singleLinePrint prt "(16)     even:" l 16)
          (singleLinePrint prt "(17)   others:" l 17)
          (sgaComment prt "      # BBlks:") (printLineInfo prt (findLine l 18))
          (sgaComment prt "      # Asgns:")
          (printLineInfo prt (findLine l 19))))

(define properDiv
  (lambda (a b)
          (if (and (equal? a 0) (equal? b 0)) "-"
              (if (equal? b 0) "inf"
                  (exact->inexact (/ a b))))))

;;; printLineInfo: print info for that line:
(define printLineInfo
  (lambda (prt line)
          (sgaComment prt (goodNum (car line)) (goodNum (cadr line))
                      (goodNum (caddr line)) "         -         -"
                      (goodNum (properDiv (cadr line) (caddr line)))
                               "\n")))

;;; goodNum: format the number:
(define goodNum
  (lambda (n)
```

```
          (placeInStr
           (if (not (number? n))
               n   ; don't change
               (let ((rep (fluid-let ((flonum-unparser-cutoff '(absolute 3)))
                                     (number->string n))))
                    (if (equal? (string-ref rep (- (string-length rep) 1))
                                #\.)
                        (list->string (reverse (cdr (reverse
                                                     (string->list rep)))))
                        (if (equal? (string-ref rep 0) #\.)
                            (string-append "0" rep) rep)))))))

;;; placeInStr: put it in a string of length 10:
(define placeInStr
  (lambda (s)
          (letrec ((genEmpty (lambda (k)
                               (if (zero? k)
                                   "" (string-append
                                       " " (genEmpty (- k 1)))))))
                  (if (> (string-length s) 10) s
                      (string-append (genEmpty (- 10 (string-length s))) s)))))

;;; findLine: return that Line:
(define findLine
  (lambda (l c)
          (cond ((equal? c 0) (car l))
                (else (findLine (cdr l) (- c 1))))))

;;; internInfo: just single numbers:
(define internInfo
  (lambda (prt prompt l which)
          (let ((thatLine (findLine l which)))
               (if (not (zero? (car thatLine)))   ; if 0, then irrelevant.
                   (sgaComment prt prompt
                               (goodNum (car thatLine)) (goodNum 0) (goodNum 0)
                               "         -         -         -" "\n")
                   #t))))  ; return value not used..

;;; singleLinePrint: single Line statistics:
(define singleLinePrint
  (lambda (prt prompt l which)
          (let ((thatLine (findLine l which)))
               (if (not (zero? (car thatLine)))   ; if 0, then irrelevant.
                   (begin
                    (sgaComment prt prompt)
                    (forEach (lambda (elm) (sgaComment prt  (goodNum elm)))
                             thatLine)
                    (sgaComment prt "\n"))
                   #t)))) ; return value not used..


;;; dotProduct: compute dot product:
(define dotProduct
  (lambda (lst1 lst2)
          (cond ((null? lst1) 0)
                (else (+ (* (car lst1) (car lst2))
```

```
                        (dotProduct (cdr lst1)
                                    (cdr lst2)))))))

;;; findCost: return formatted dot product:
(define findCost
  (lambda (l1 l2)
          (string-trim (goodNum (dotProduct l1 l2)))))

;;; findOverallGain: compute and just divide:
(define findOverallGain
  (lambda (l1 l2)
          (string-trim (goodNum
                        (let ((val (properDiv (dotProduct l1 weightsSGA)
                                              (dotProduct l2 weightsSGA))))
                             (if (number? val) (exact->inexact val) val))))))

;;; diffCost: compute the difference of costs:
(define diffCost
  (lambda (l1 l2 l3)
          (let ((pr1 (dotProduct l1 weightsSGA))
                (pr2 (dotProduct l2 weightsSGA))
                (pr3 (dotProduct l3 weightsSGA)))
               (if (= pr3 0) (goodNum "inf")
                   (string-trim (goodNum (- (exact->inexact (/ pr1 pr3))
                                            (exact->inexact (/ pr1 pr2)))))))))

;;; percentImpro: percentage of the improvement:
(define percentImpro
  (lambda (c1 c2)
          (let ((pr1 (dotProduct c1 weightsSGA))
                (pr2 (dotProduct c2 weightsSGA)))
               (string-trim (goodNum
                             (- 100 (exact->inexact (/ (* 100 pr2) pr1))))))))

;;; define the weight vector:
(define weightsSGA '(2       ; jump
                     2       ; assign
                     1       ; varref
                     2       ; decision
                     3       ; eq
                     2       ; hd
                     2       ; rest
                     3       ; list
                     3       ; append
                     3       ; search
                     0       ; intern
                     3       ; add
                     3       ; sub
                     3       ; mul
                     3       ; div
                     2       ; odd
                     2       ; even
                     2))     ; others
```