INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48105
PROGRAM INFERENCE FROM EXAMPLE COMPUTATIONS
REPRESENTED BY MEMORY SNAPSHOT TRACES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
Frederick Eugene Petry, B.S., M.S.

* * * * *

The Ohio State University
1974

Reading Committee:
Professor Alan Biermann
Professor B. Chandrasekaran
Professor Lee White

Approved By
Adviser
Department of Computer
and Information Sciences
ACKNOWLEDGEMENTS

I would like to thank Professor Alan Biermann for his invaluable guidance in this research. For his continuing help in this work and interest in my career, I am most grateful. I am also grateful to Professor B. Chandrasekaran and Professor Lee White for their interest and help in this work.

I especially thank my wife, Dottie, who has now patiently tolerated her second dissertation writer.

This work was supported by National Science Foundation Grant numbers GN-534.1 and GJ-34739X.
VITA

December 4, 1941 . . Born -- New Orleans, Louisiana

1963 . . . . . . B.S., Loyola University
               New Orleans, Louisiana

1963 - 1968 . . Teaching and Research Assistant
              Department of Physics and Astronomy
              Louisiana State University
              Baton Rouge, Louisiana

1968 . . . . . . M.S., Louisiana State University
               Baton Rouge, Louisiana

1968 - 1970 . . Assistant Professor
              Department of Physics and Mathematics
              Southern University
              New Orleans, Louisiana

1970 - 1974 . . Teaching and Research Associate
              Department of Computer and Information Science
              The Ohio State University
              Columbus, Ohio

PUBLICATIONS

"Speeding Up the Synthesis of Programs from Traces," A.W. Biermann,

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Artificial Intelligence -- Professor Alan Biermann and
Professor B. Chandrasekaran

Studies in Automata Theory and Computability -- Professor Larry Reeker

Studies in Computer Organization -- Professor Ming-Tsan Liu
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>VITA</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>7</td>
</tr>
<tr>
<td>3. COMPUTATIONAL ENVIRONMENT</td>
<td>11</td>
</tr>
<tr>
<td>Description of a Computational Environment</td>
<td></td>
</tr>
<tr>
<td>Model of a Computation in an Environment</td>
<td></td>
</tr>
<tr>
<td>Traces of Computations</td>
<td></td>
</tr>
<tr>
<td>4. DECOMPUTATION</td>
<td>27</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>Comparison of Decomputation of Traces</td>
<td></td>
</tr>
<tr>
<td>Decomputation Algorithm</td>
<td></td>
</tr>
<tr>
<td>Decomputation and Cell Names</td>
<td></td>
</tr>
<tr>
<td>Restrictions on the Environment</td>
<td></td>
</tr>
<tr>
<td>Number of Basic Operations Performed in the Decomputation Process</td>
<td></td>
</tr>
<tr>
<td>Automatic Indexing</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.</td>
<td>SYNTHESIS</td>
</tr>
<tr>
<td></td>
<td>Definitions for the Synthesis from I'C traces</td>
</tr>
<tr>
<td></td>
<td>Synthesis Outline</td>
</tr>
<tr>
<td></td>
<td>Annotated Synthesis Example</td>
</tr>
<tr>
<td></td>
<td>Trace Partitions</td>
</tr>
<tr>
<td></td>
<td>Pre-Processing</td>
</tr>
<tr>
<td></td>
<td>Re-Ordering</td>
</tr>
<tr>
<td></td>
<td>Elimination of Weight Zero Instructions</td>
</tr>
<tr>
<td></td>
<td>An Example of Unsuccessful Re-Ordering</td>
</tr>
<tr>
<td></td>
<td>Look-Ahead</td>
</tr>
<tr>
<td>6.</td>
<td>EXPERIMENTS</td>
</tr>
<tr>
<td></td>
<td>Effect of Re-Ordering and Look-Ahead</td>
</tr>
<tr>
<td>7.</td>
<td>CONCLUDING REMARKS</td>
</tr>
</tbody>
</table>

APPENDIX

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ENUMERATION OF PROGRAMS</td>
<td>111</td>
</tr>
<tr>
<td>B</td>
<td>AN ALTERNATE APPROACH TO SYNTHESIS</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>FROM I'C TRACES</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>DESCRIPTION OF EXPERIMENTS AND RESULTS</td>
<td>120</td>
</tr>
</tbody>
</table>

BIBLIOGRAPHY | 140
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Times for the Synthesis of a Program for Computing ( N! )</td>
</tr>
<tr>
<td>6.2</td>
<td>Times for the Synthesis of a Program for Computing ( F_N ), the ( N^{th} ) Fibonacci Number</td>
</tr>
<tr>
<td>6.3</td>
<td>Effects of Re-ordering and Look-ahead on the Synthesis Times for Fibonacci Programs</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary of Programs Synthesized</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Data Structures for a Calculation</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Trace of Memory Snapshots of a Calculation of the Sum of the Odd Numbers of an Array</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Program Inferred Using the Memory Trace of Figure 1.2</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>A Portion of a Program P</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>P Represented as a Composition of Instruction Vertex - Condition Vertex Pairs</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Environment and Flowchart for an Integer Square Root Calculation</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Traces of the Computation of $\sqrt{49}$</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Flowchart of Decomputation Process of an I'C Trace</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>D-Sets Resulting from Decomputation of the I'C Trace of the Computation of $\sqrt{49}$</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Snapshot of UPDATE at Step 11 of I'C Trace</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>HASH Table Entries</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>UPDATE Entry Changes</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>HASH Entry Changes</td>
<td>37</td>
</tr>
<tr>
<td>Section</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Example Program</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>IC Trace of Program of Figure 5.1</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Flowchart of the Synthesis Process</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>Decomputed Representation of an I'C Trace</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Synthesis Procedure for Trace of Figure 5.4</td>
<td>58</td>
</tr>
<tr>
<td>5.6</td>
<td>Program Produced by Synthesis</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>Partition P</td>
<td>64</td>
</tr>
<tr>
<td>5.8</td>
<td>Refinement P'</td>
<td>64</td>
</tr>
<tr>
<td>5.9</td>
<td>Decomposition R</td>
<td>64</td>
</tr>
<tr>
<td>5.10</td>
<td>I'C Trace After Decomputation</td>
<td>78</td>
</tr>
<tr>
<td>6.1</td>
<td>Inference System</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>Data Structures for the Calculation of the Average of a List</td>
<td>90</td>
</tr>
<tr>
<td>6.3</td>
<td>Initial Environment and Trace Input</td>
<td>91</td>
</tr>
<tr>
<td>6.4</td>
<td>Transitions Produced by Synthesis</td>
<td>93</td>
</tr>
<tr>
<td>6.5</td>
<td>Flowchart Corresponding to Transitions of Figure 6.4</td>
<td>93</td>
</tr>
<tr>
<td>6.6</td>
<td>Incorrect Factorial Program</td>
<td>96</td>
</tr>
<tr>
<td>6.7</td>
<td>Incorrect Factorial Program</td>
<td>96</td>
</tr>
<tr>
<td>6.8</td>
<td>Correct Factorial Program</td>
<td>96</td>
</tr>
<tr>
<td>6.9</td>
<td>Incorrect Fibonacci Program</td>
<td>98</td>
</tr>
<tr>
<td>6.10</td>
<td>Incorrect Fibonacci Program</td>
<td>98</td>
</tr>
<tr>
<td>6.11</td>
<td>Correct Fibonacci Program</td>
<td>99</td>
</tr>
<tr>
<td>B.1</td>
<td>Program Enumeration</td>
<td>118</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

Since the advent of computers, there has been considerable concern over the difficulties of man-machine communications. One of the major problem areas has been in programming, the communication of algorithms from man to machine. Developments in higher level languages have helped to alleviate some of these difficulties, but the task of obtaining the desired behavior from a machine is still often a formidable problem. Additional improvements in programming systems will be made when methods of expressing algorithms are incorporated which are more natural for the human programmer.

Examples are used quite naturally in person-to-person conversation to aid in clarifying meaning. An example of a concept or technique allows a recipient of such information to test his understanding of the abstract idea on something concrete. Because they are so common in human communications, examples should be permitted, if feasible, in man-machine communication. Thus this work is a study of some techniques for communicating algorithms using examples.

In a man-to-man exchange, rather sketchy examples may be given, relying on the listener to fill in details from some general knowledge he has about the problem. In order to focus precisely on the use of examples, we deal only with examples given in enough detail so as not to require such knowledge.
Let us illustrate an explanation by example. We will perform a sample calculation which finds the sum of the odd elements of a list of numbers. First some data structures should be declared as shown in Figure 1.1.

```
A | I | SUM | TMP | N
3 |   |     |     | 3
5 |   |     |     |
4 |

Figure 1.1 Data Structures for a Calculation.
N is the Size of List A.
```

Then a description of the calculation might be given as follows:

"First we'll set I to one and initialize SUM to zero. Then start to scan down the array to find the odd elements. The first one, A(1) = 3, when divided by 2 produces a non-zero remainder of 1 in TMP. So it is odd and is added to SUM giving 3. Continuing down with I = 2, we get TMP = 1 again and so add 5 to SUM yielding 8. Next with I = 3, TMP is zero since A(3) is even. Then incrementing I to 4, we see that since I > N, we have reached the end of the list and stop."

As the above description was made, the corresponding changes could be made in the data structures of Figure 1.1. Figure 1.2 then gives the sequence of variables affected and the changes made in their values as the calculation proceeded.

Now could such a description be used in lieu of writing a formal program? Removing the problem of understanding natural language, we see what is contained in this description is a trace of an execution of the algorithm. Such a trace could be communicated to the machine
I, 1
SUM, 0
TMP, 1
SUM, 3
I, 2
TMP, 1
SUM, 8
I, 3
TMP, 0
NOTE: TMP = 0
I = 4
NOTE: I > N
HALT

Figure 1.2 Trace of Memory Snapshots of a Calculation of the Sum of the Odd Numbers of an Array. The Sequence of Variables Affected by the Calculation is Shown with Their Associated Revised Values.
by setting up a computational environment in which the user can perform a calculation like the one described. Specifically, the user could be given a computer graphic display system, a graphic input system such as a light pen or touch sensitive screen, and an ability to declare data structures and execute calculations in a scratch pad manner.

Then as the programmer executes his example, the sequence of actions performed could be saved as in Figure 1.2. The question is whether such sparse information from the user could be sufficient to enable the machine to automatically infer the desired program. We will show that this type of computation trace is in general quite adequate for a correct program synthesis and, in fact, the correct program of Figure 1.3 was created from the trace of Figure 1.2 in just 12.5 seconds by our system. Other programs which have been created are various search and sort routines, an integer square root using Newton's method, multiplication of square matrices, and Hoare's algorithm FIND.

Basically, two steps are performed to infer a program from a trace of memory snapshots, decomputation and synthesis. Decomputation produces the set of instructions which could have caused the change in the memory snapshot for any entry in the trace. For example, in the fourth entry of the trace in Figure 1.2, the value of the variable SUM changed to 3. Some of the possible instructions which could explain this change are

\[
\begin{align*}
\text{SUM} &\rightarrow \text{SUM} + A(I) \\
\text{SUM} &\rightarrow A(I) \\
\text{SUM} &\rightarrow A(3) - I \\
\text{SUM} &\rightarrow A(I) \\
\text{SUM} &\rightarrow A(1)*I \\
\ldots
\end{align*}
\]
Figure 1.3 Program Inferred Using the Memory Trace of Figure 1.2
The synthesis process then creates, from the output of the decomputation, the smallest program which fits the data. This dissertation will study in detail the processes of decomputation and synthesis, describe an implementation of an inference system based on these processes, and give examples of its behavior.
CHAPTER 2. BACKGROUND

Much of the current activity in program synthesis or automatic programming has been spurred by the work of Waldinger [25] and Green [14] in the late 1960's. Their work showed that it was possible to employ theorem-proving systems to produce programs from proofs in the first-order predicate calculus. Generally, a formal statement can be given expressing the results desired of a program. Then within a given set of axioms, such a system performs a deduction using the resolution principle of Robinson [23]. A program is then extracted from the steps of the proof. The lack of power of current resolution-based theorem provers is one limitation to this approach. Another fundamental hinderance lies in the difficulty of using specific knowledge about the problem to guide the search for the proof.

More recently, work has been done utilizing a more sophisticated approach made feasible by current developments in very high level problem solving languages. The Planner language evolved by Hewitt [16] is a prototype of these. It represents a combination of programming and theorem proving methods, in particular employing a procedural representation of knowledge. Knowledge about programming in general and the specific problem domain being considered can then be embedded in the system to guide the search for the solution.

Sussman [24] has developed a system he calls HACKER which creates programs in the specific domain of the Blocks World (Winograd [27]).
The system maintains a library of procedures, and if an appropriate one cannot be called up for a particular problem, HACKER creates one using general knowledge of the problem domain and programming methods. If a bug is detected in such a routine, some generic knowledge of debugging is utilized to correct the specific error. In some cases the patches made to bugs can be generalized and similar mistakes thus avoided in future programs.

In Buchanan's work [8] a system is described which has generated programs in several problem domains. A formalism was evolved in which the programming environment can be defined thus permitting a formal statement of the problem to be solved. The basis for this formalism is an axiom system for programs developed by Hoare [17]. The construction of a program can then be considered as a search in this system for a proof that the generated program satisfies the given input-output assertions.

Some recent surveys of automatic programming can be found in Balzer [1], Feldman [10], Freeman [12], and in Leavenworth and Sammet [20].

Most of the above work can be viewed as problem solving where the problem is to produce a program from some formal description of its performance. Thus these systems must create the algorithms corresponding to the programs needed, whereas, as explained in the introduction, we are more concerned with the problem of algorithm communication. For us the algorithm is considered to be known in the form of examples and our system infers programs from such a specification.

The use of examples for grammatical inference was studied in Gold [13] and Feldman [9]. Here sample strings of a language are
presented and one attempts to derive a grammar for the language. A survey of grammatical inference can be found in Biermann and Feldman [6]. The use of examples was further developed by Biermann [3] in considering the inference of Turing machines from sample computations. The examples were in the form of traces of the steps performed by the Turing machine during the computation. The extension of this approach to program inference was carried out in the program synthesis project described in Biermann, Baum, and Petry [4] and in Biermann and Krishnaswamy [5]. Other studies using example computations have been made by Barzdin [2] and Raulefs [22].

Biermann and Krishnaswamy [5] describe a computer graphics display oriented program synthesizer allowing the user to execute example computations with a light pen. This system, which they call an autoprogrammer, requires that the human user specify in full detail every instruction in an example calculation in order for it to construct a program. The current research also is aimed at developing such a system but with the intent of requiring much less detail and precision from the user. Whereas Biermann and Krishnaswamy require that the user tell exactly what instruction must be executed at each step in the calculation, here we ask only that the user give the result of each executed instruction and our system will then produce the correct instruction internally. Thus a typical step on the Biermann-Krishnaswamy system might be "add A(I) to B and put the result, 3, into SUM." On our system, the same step would be expressed simply as "the result, 3, appears in SUM." Despite such a large reduction in
the amount of input information, it turns out that correct program synthesis is still possible in a short time for moderately complex programs.

In this dissertation, we will study the "pure" problem of program inference from memory traces as described in the introduction. However, we suspect that a properly designed autoprogramming system would give the user the freedom to input instructions precisely as in the Biermann-Krishnaswamy style or in terms of their results only as we have done or in other ways. Our research indicates that a correct program could still be synthesized even with these varying types of input description.
CHAPTER 3. COMPUTATIONAL ENVIRONMENT

The idea of the environment of a computation as discussed in the introduction will be formalized in this chapter. The computation is done in the given setting of memory cells, basic operations, and tests. Before the calculation begins, the memory cells are structured (single variables, arrays, etc.) and named. A computation is then performed using basic operations and tests in the environment. A particular usage of a basic operation will then be called an instruction. For example,

"Add X to Y and put the results in Z"
"Add X to Y and put the result in Y"
"Multiply X by X and put the result in Z"

etc.,

are some of the instructions which could arise during a computation in an environment allowing basic operations such as add and multiply with memory cells named X, Y, Z. In a similar manner the specific usages of basic tests are called conditions.

When it is said that a person performs a computation in the given environment, the assumption is that person has an algorithm in mind he wishes to communicate. He may not have the algorithm in an explicit form (e.g., a flowchart), but by implicitly knowing "what he wants to do" he can carry out the computation on the specific data in the environment. His implicit knowledge of the algorithm will allow him to
make the local decisions needed to perform the calculation at each step when confronted by the state of the memory contents at that time.

In order to describe this process of computation and to examine the representation of it in various traces, we will use a specific model of an algorithm and calculation. However, as we said above, it is not necessary for the user to have such an explicit representation in mind.

**Description of a Computational Environment**

**Def.** A computational environment \( E \) is a triple

\[
E = (M, F, R)
\]

\( M \) is the finite set of memory cells

\[
\{m_1, m_2, \ldots, m_n\}
\]

The contents or value of a cell \( m_i \) is an element of some set of entities, such as the set of integers or the set of strings over some alphabet. We will focus here on the integers, \( N \). Cell \( m_{n+1} \) is a specially distinguished cell usually referred to as the test cell. Its contents can only be 0 or 1. Let \( M' \) denote \( M \cup \{m_{n+1}\} \).

\( F \) is the finite set of basic operations or functions:

\[
\{f_1, f_2, f_3, \ldots\}
\]

For any \( f_i \in F \), if \( f_i \) has \( k \) arguments,

\[
f_i : N \times N \times N \times \ldots \times N \rightarrow N
\]

Let us represent any \( f \in F \) evaluated over the contents of \( k \) specific cells in \( M \) as \( f(m_{i_1}, m_{i_2}, \ldots, m_{i_k}) \). Then its result is \( n \):
In the further discussions, we will focus on binary and unary functions. For example, \( F \) could be a set of arithmetic operations \{+, -\}
or \{+, -, *, /\}

\( R \) is a finite set of basic tests or relations:
\{\( r_1, r_2, r_3, \ldots \)\}

The specific evaluation of a test in \( E \) is denoted
\( r_i(m_{i1}, \ldots m_{ik}) \).

The result of a test will be either 1 or 0, representing true or false. Some of the tests which might be used could be a set of binary relations
\{\( >, =, < \)\}
or unary relations
\{\( +, 0, - \)\}

The functions or relations used in \( E \) could be changed from one computation to another. For a particular algorithm to be communicated, it might be desirable to add an operation to the basic set, such as an exchange operation for sorts. In the particular experiments performed we have used mainly the set of basic arithmetic operations.

Given the set of functions and relations, a computation starts with the structuring and naming of the memory cells. With each cell \( m_i \) will be associated one or more names \( a_{i1}, a_{i2}, \ldots, \). More than one name is possible for cells which are involved in some data structure.
If a cell is the second element of an array A and there is an index variable I which, at the moment, has the value 2, then the cell could have names A(2) or A(I). For the structures, usually arrays, not all cells will be allowed to act as indices. The indices or pointers which will be used to form the name of the structure must be specified ahead of time. Let us illustrate this from the example of finding the average sum of the odd numbers in a list whose data structures are shown in Figure 1.1.

<table>
<thead>
<tr>
<th>Memory Cell $m_1$</th>
<th>Associated Name(s) $a_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1$</td>
<td>$a_{11} = A(I); a_{12} = A(I)$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$a_{21} = A(2); a_{22} = A(I)$</td>
</tr>
<tr>
<td>$m_3$</td>
<td>$a_{31} = A(3); a_{32} = A(I)$</td>
</tr>
<tr>
<td>$m_4$</td>
<td>$a_4 = I$</td>
</tr>
<tr>
<td>$m_5$</td>
<td>$a_5 = SUM$</td>
</tr>
<tr>
<td>$m_6$</td>
<td>$a_6 = N$</td>
</tr>
</tbody>
</table>

So I was the only declared pointer for the structure, A. Each cell of this structure has, possibly, two names. The valid use of $A(I)$ for the name of a memory cell in the array will depend on the contents of memory cell named $I$. If the contents of $m_4$ is two, then $A(I)$ is a valid name for $m_2$.

The names of the cells will be used basically outside the computational environment, E. As instructions or conditions of an algorithm are used in E, they will be understood to have been properly translated. The distinction between the cells and their names will usually be clear from context.
In doing an example computation, a user must execute instructions and note conditions in the environment \( E \). So now we will define exactly what we mean by instructions and conditions.

**Def.** For \( p \)-ary functions, an instruction, \( I \), is a \( p+2 \) tuple

\[
(m_i, f_j, m_{k1}, \ldots, m_{kp}), \quad f_j \in F
\]

\[
m_i, m_{k1}, \ldots \in M.
\]

and will be usually written

\[
m_i \leftarrow f_j(m_{k1}, \ldots, m_{kp})
\]

The instructions permitted then are all assignment type instructions. The interpretation is that \( f_j \) evaluated over \( p \) specific cells produces a result \( n \) which is then placed into a specific cell \( m_i \), called the result cell for the instruction. The right hand side of the assignment will be referred to as the result function of the instruction.

**Def.** For \( p \)-ary relations, a condition, \( C \), is a \( p+2 \) tuple

\[
(m_{h+1}, r_j, m_{k1}, \ldots, m_{kp}), \quad r_j \in R
\]

\[
m_{h+1}, m_{k1}, \ldots \in M'.
\]

and will be written

\[
m_{h+1} \leftarrow r_j(m_{k1}, \ldots, m_{kp})
\]

with a similar interpretation as above for instructions.

**Model of a Computation in an Environment**

Now we will use a particular flowchart model to have a representation of an algorithm in order to discuss the process of a computation. A computation will be considered to be a particular execution in \( E \) of
an algorithm so represented. This description of flowcharts and their execution follows common notation as in Bruno and Steiglitz [7].

**Def.** A flowchart is a finite, connected directed graph with vertices of the form:

1. One start vertex

2. One stop vertex

3. Instruction vertices where exactly one instruction in $E$ is associated with an instruction vertex

4. Test vertices where exactly one test is associated with each vertex of this type and 2 edges leave it, one labeled 0 and the other 1 to represent the result of the test.
The basic unit of the flowchart will be considered to be the instruction vertex - condition vertex pair. Thus if a flowchart of an algorithm has consecutive instruction or condition vertices, we can consider these to be formally separated by no-op instructions or null tests. This is illustrated in Figures 3.1 and 3.2.

The flowchart form can also be represented as a set of transitions which will be more convenient later. Each transition will be a triple, instruction - condition - instruction, with the condition denoted \( \neg C \) or \( C \) depending on whether the second instruction of triple lies on the edge leaving \( C \) labeled 0 or 1. For example in Figure 3.2, we have corresponding to the first 2 units of the transformed flowchart, the following 2 sets of transitions:

\[
\begin{align*}
(1I_1, \neg C_1, I_2), & \quad (1I_1, C_1, 1I_2) \quad \text{Unit 1} \\
(1I_2, \neg C_2, I_3), & \quad (1I_2, C_2, 2I_1) \quad \text{Unit 2}
\end{align*}
\]

Note that where there are multiple occurrences of an instruction, \( i_j \), in the flowchart these will be denoted in a transition as

\[1I_j, 2I_j, 3I_j \ldots\]

Next the execution of the algorithm represented by a flowchart will be discussed.

**Def.** The state, \( S_t \), of the memory is the contents of the memory cells at a particular time, \( t \). We will write

\[ [m_i]_t \quad i = 1, \ldots, h+1 \]

as the state of an individual cell \( i \), i.e., its contents at time \( t \). \( S'_t \) will denote the state of \( M' = M \cup \{m_{h+1}\} \).
Figure 3.1 A Portion of a Program P

Figure 3.2 P Represented as a Composition of Instruction Vertex - Condition Vertex Pairs.
A computation will now be described as the execution of instruction-condition pairs along with the appropriate changes to the state $S$. If in a block the instruction $I$ is no-op, then no change occurs in $S$. If the condition is a null test no change occurs in the test cell $m_{h+1}$ and both edges leaving the test vertex connect to the same daughter instruction vertex.

So now we will show the specification of an execution of an algorithm, i.e., a computation in $E$.

Let the initial state for the computation be $S_0$.

**Step 1:** Enter the start vertex. By convention we always consider the first instruction as a do-nothing start instruction. Perform test $C$, (if it isn't a null test) and exit the test vertex on the appropriate edge as indicated by the test result.

**Step $j$:** Enter the instruction vertex with state $S_{j-1}$. Execute the instruction $I$ found here, where $I$ is of the form

$$m_i = f(m_k, m_l)$$

for some $f \in F$ and some $m_i, m_k, m_l \in M$.

Then

$$[m_p]_t^j = [m_p]_{t=j-1} \quad \forall \quad p \neq i$$

$$[m_i]_j = f([m_k]_{j-1}, [m_l]_{j-1})$$

After $I$'s execution in $E$ the memory state becomes $S_j$. If $f$ is a no-op or happens to produce a result the same as $[m_i]_{j-1}$, then

$$S_j = S_{j-1}.$$
Next enter the test vertex and evaluate the condition $C$ found there, with the memory state $S_j$ as changed by $I$.

$$C: \quad m_{h+1} = r(m_i, m_k) \quad \text{for some } r \in R \text{ and some } m_i, m_k \in M.$$ and so

$$[m_{h+1}]_{t=j} = r([m_i]_{t=j}, [m_k]_{t=j}).$$

If $r$ is a null test

$$[m_{h+1}]_{j} = [m_{h+1}]_{j-1}$$

Based on $[m_{h+1}]_{j}$, the appropriate edge is chosen to the next block. Thus after passing through the given block on step $j$, the memory state is $S_j$ and we are prepared to perform step $j+1$.

**Traces of Computations**

Next the trace of a computation will be defined. A trace will consist of a sequence of entries corresponding to the steps of the computation. Let $A$ be a computation of $p$ steps in an environment $E$ with an initial memory state $S_0$.

**Def.** An IC trace, $T_{IC}$, of $A$ is a sequence of length $p$

$$(T_{11}, T_{12}); (T_{21}, T_{22}); \ldots; (T_{p1}, T_{p2})$$

The entries in the sequence are 2-tuples. $T_{j1}$ is the instruction, $I$, which was executed at the $j$-th step of $A$. $T_{j2}$ is either $C$ or $C$ (where $C$ is the condition at step $j$) depending on whether in step $j$, after the test is done, the contents of $m_{h+1}$, $[m_{h+1}]_{j}$, is 0 or 1.
An example IC trace entry from the calculation shown in the introduction would be \((I + I + 1, \text{Note: } I > N)\).

Since we are interested in the problem of inference from weaker information, traces will be defined in which first instructions, and then conditions and then both will be left out of entries of the trace. Rather than seeing the specific instruction or condition executed, its result will be given, i.e., a memory snapshot of the appropriate cells.

**Def.** An IC' trace, \(T_{IC'}\), is a sequence
\[
(T_{11}, T_{12}); \ldots (T_{p1}, T_{p2})
\]
where \(T_{j1}\) is the instruction, \(I\), of the \(j\)th step but
\[
T_{j2} = [m_{h+1}]_j
\]

Corresponding to the IC trace entry illustrated above is the IC' tuple \((I + I + 1, 1)\). The second element of the tuple is a 1 because the condition, \(I > N\), was true thus loading a 1 into the test cell \(m_{h+1}\).

**Def.** An I'C trace, \(T_{I'C}\), is a sequence
\[
(T_{11}, T_{12}, T_{13}); \ldots (T_{n1}, T_{n2}, T_{n3})
\]
with
\[
T_{j1} = m_{i,j} \quad \text{and} \quad T_{j2} = [m_{i,j}]_j
\]
where \(m_{i,j}\) is the result cell of the instruction executed at step \(j\).

It is necessary to specify the pair (result cell, result) in place of the instruction since in contrast to conditions any cell in \(M\) could have been affected. \(T_{j3}\) is \(C\) or \(^C\) as in \(T_{j2}\) of an IC trace.
Here the example entry becomes for an I'C trace, 
$m_4$, 4, NOTE: I > 3), since the instruction executed, $I + I + 1$, 
produced a result of 4 which was then put into the cell $m_4$ (named I).

**Def.** An I'C' trace, $T_{I'C'}$, is a sequence

$$(T_{11}, T_{12}, T_{12}); \ldots (T_{n1}, T_{n2}, T_{n3})$$

where

$$T_{j1} = m_i^j,$$
$$T_{j2} = [m_{ij}]_j$$

as above in an I'C trace and

$$T_{j3} = [m_{i+1}]_j.$$

These four traces may be displayed as a lattice

\[ \begin{array}{c}
I'C' \\
\text{IC} \\
I'C \\
\text{IC'}
\end{array} \]

It is clear that the traces lower in the lattice are always weaker.

In general where only a memory change is given many instructions or 
conditions could have produced the result, and so some information has 
been lost. The amount of computation needed then to provide a correct 
synthesis will often be greater.

Now let us consider how this model of computation, i.e., execu-
tion of an algorithm in the given flowchart form, fits with the
intuitive idea of a computation description as in the introduction. The representation by an I'C trace allows the freedom to just give results as in the example of the sum calculation. However the trace would require, in the summing of the array A in this example, the user to say the end of the array hadn't been reached at each step. That is, I is an index for A and so the trace entry, \( I > N \), where \( C_1 \) is I > N, must be noted all the way down the list A.

This is tedious and unnecessary here. Let the condition be noted, \( C_1 \), only once, when the end of the list is reached. When a program is produced, the blank entry will be merged with \( C_1 \) and since it represents the other branch of the test can be labeled as \( I > N \). Thus in future examples we will specify conditions in a trace only for one branch of a test. Further discussion of the handling of conditionals is found in Biermann and Krishnaswamy [5].

Figure 3.3 shows an environment and algorithm for computing integer square roots. It is based on an iterative Newton's method for the integer root of \( N \) using

\[
X_{k+1} = \frac{1}{2} \cdot (X_k + N/X_k)
\]

where \( X_k, X_{k+1} \) are the \( k^{th} \) and \( k + 1^{th} \) approximation for \( \sqrt{N} \). The algorithm illustrated in the flowchart of the Figure 3.3 assumes that the memory cells of the environment have been correctly initialized. Figure 3.4 gives the four different traces as defined above for a calculation of \( \sqrt{49} \). The computation starts with an initial guess for \( X_0 \) of 2. So the initial state, \( S_0 \), for the computation is

(2, 2, 0, 49, 1)

\( m_1 \ m_2 \ m_3 \ m_4 \ m_5 \)
In these traces the tuples containing "start" and "stop" appear explicitly. Since at the start and stop of the computation there is no change in the memory contents, then for the I'C and I'C' traces the entry $T_{j2}$ in the start and stop tuples is left blank. In future examples the start and stop will not be shown explicitly in a trace.
I. Environment \( E \):

\[ M = \{m_1, m_2, m_3, m_4\}; \quad R = \{<, =, >\} \]

\[ F = \{\text{move, add, subtract, multiply, integer(truncating) divide}\} \]

- \( m_1 \) = Two (this cell contains a constant, 2)
- \( m_2 \) = \( X_k \)
- \( m_3 \) = Temp
- \( m_4 \) = \( N \)

II. Program or flowchart of the algorithm:

Figure 3.3 Environment and Flowchart for an Integer Square Root Calculation.
<table>
<thead>
<tr>
<th>IC Trace</th>
<th>IC' Trace</th>
<th>I'C Trace</th>
<th>I'C' Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (T_{j1}, T_{j2}) )</td>
<td>( (T_{j1}, T_{j2}) )</td>
<td>( (T_{j1}, T_{j2}, T_{j3}) )</td>
<td>( (T_{j1}, T_{j2}, T_{j3}) )</td>
</tr>
<tr>
<td>(Start, -)</td>
<td>(Start, 1)</td>
<td>(Start, , -)</td>
<td>(Start, , 1)</td>
</tr>
<tr>
<td>(I_1, -)</td>
<td>(I_1, 1)</td>
<td>(m_3, 24, -)</td>
<td>(m_3, 24, 1)</td>
</tr>
<tr>
<td>(I_2, -)</td>
<td>(I_2, 1)</td>
<td>(m_3, 26, -)</td>
<td>(m_3, 26, 1)</td>
</tr>
<tr>
<td>(I_3, -)</td>
<td>(I_3, 0)</td>
<td>(m_3, 13, -)</td>
<td>(m_3, 13, 0)</td>
</tr>
<tr>
<td>(I_4, -)</td>
<td>(I_4, 0)</td>
<td>(m_2, 13, -)</td>
<td>(m_2, 13, 0)</td>
</tr>
<tr>
<td>(I_1, -)</td>
<td>(I_1, 0)</td>
<td>(m_3, 3, -)</td>
<td>(m_3, 3, 0)</td>
</tr>
<tr>
<td>(I_2, -)</td>
<td>(I_2, 0)</td>
<td>(m_3, 16, -)</td>
<td>(m_3, 16, 0)</td>
</tr>
<tr>
<td>(I_3, -)</td>
<td>(I_3, 0)</td>
<td>(m_3, 8, -)</td>
<td>(m_3, 8, 0)</td>
</tr>
<tr>
<td>(I_4, -)</td>
<td>(I_4, 0)</td>
<td>(m_2, 8, -)</td>
<td>(m_2, 8, 0)</td>
</tr>
<tr>
<td>(I_1, -)</td>
<td>(I_1, 0)</td>
<td>(m_3, 6, -)</td>
<td>(m_3, 6, 0)</td>
</tr>
<tr>
<td>(I_2, -)</td>
<td>(I_2, 0)</td>
<td>(m_3, 14, -)</td>
<td>(m_3, 14, 0)</td>
</tr>
<tr>
<td>(I_3, -)</td>
<td>(I_3, 0)</td>
<td>(m_3, 7, -)</td>
<td>(m_3, 7, 0)</td>
</tr>
<tr>
<td>(I_4, -)</td>
<td>(I_4, 0)</td>
<td>(m_2, 7, -)</td>
<td>(m_2, 7, 0)</td>
</tr>
<tr>
<td>(I_1, -)</td>
<td>(I_1, 0)</td>
<td>(m_3, 7, -)</td>
<td>(m_3, 7, 0)</td>
</tr>
<tr>
<td>(I_2, -)</td>
<td>(I_2, 0)</td>
<td>(m_3, 14, -)</td>
<td>(m_3, 14, 0)</td>
</tr>
<tr>
<td>(I_3, C_1)</td>
<td>(I_3, 1)</td>
<td>(m_3, 7, C_1)</td>
<td>(m_3, 7, 1)</td>
</tr>
<tr>
<td>(Stop, -)</td>
<td>(Stop, 1)</td>
<td>(Stop, , -)</td>
<td>(Stop, , 1)</td>
</tr>
</tbody>
</table>

Figure 3.4 Traces of the Computation of \( \sqrt{49} \)
Using the Algorithm and Environment of Figure 3.3.

\(^1\) If the entry is "-" this means no condition was specified in the step.
CHAPTER 4. DECOMPUTATION

Introduction

After a trace or several traces of a computation have been obtained, the program inference process is ready to begin. However, in the case of traces weaker than IC traces, only memory changes rather than the cause of the changes, specific instructions and conditions, may be seen in the trace. Thus it is necessary to take a memory change, a computed result, and, as in the introduction, decompute it to produce the possible causes of the observed change. So let us give the following as a definition of decomputation:

Def. Given a computation environment E, with a memory state S, the process of producing all possible causes for an observed change in the contents of some memory cell m is called decomputation. The set of instructions or conditions produced by a decomputation will be called a decomputation set, or D-set.

Assume that at the jth step of a computation in E, we observed that the cell m1's contents are changed to n. That is, we may have an IC trace and

\[ T_{j1} = m_1 \]
\[ T_{j2} = n \]
The first step in a decomputation of this is to find all functions
\[ f([m_k]_{j-1}, [m_k]_{j-1}) \quad f \in F; \quad m_k, m'_k \in M \]
such that their value is \( n \) when \( S_{j-1} \) is the current memory state.

Then for each possible function, all possible instructions are

generated by considering the names of the cells involved.

As an example of the process, let us assume we have an \( I'C \) trace

of a computation in which, say,

\[ T_{71} = m_2 \quad : \text{cell affected} \]
\[ T_{72} = [m_2]_7 = 5 \quad : \text{change in cell 2} \]

Now if

\[ [m_3]_6 = 10 \]
\[ [m_5]_6 = 2 \]

then

\[ m_3/m_5 \]

is a possible result function which could have produced the change in

\( m_2 \) at this step. Let cell 5 be a cell in an array and cell 1 be the

cell corresponding to an index declared for this array. If the names

of the cells are

\[ \begin{align*}
  m_1 & \quad - \quad J \\
  m_2 & \quad - \quad X \\
  m_3 & \quad - \quad \text{TMP} \\
  m_5 & \quad - \quad A(1) \text{ or } A(J)
\end{align*} \]

and if

\[ [m_1]_6 = 1 \]

the possible instructions formed from the result function are
For the decomputation of each entry of a trace, it is necessary to know the state of memory prior to the change shown in the entry. This can be achieved simply by requiring that the initial memory state $S_0$ be given with the trace. Then the memory state can be updated by the changes in the trace at each entry yielding the needed memory states.

Comparison of Decomputation of Traces

Four types of traces, IC, IC', I'C, I'C' were described in Chapter 3, but it was decided to concentrate on I'C traces in this study. Synthesis from IC traces has been investigated elsewhere [4]. IC' traces appeared to be less interesting than I'C traces since in most computations many more instructions are executed than tests performed. Comparatively, IC' traces seem to be less amenable to decomputation. In general, the number of possible causes of the logical result, true or false, is much greater than the number of causes of an arithmetic result like 5. For example, consider 5 cells with contents

\[
\begin{array}{ccccc}
  m_1 & m_2 & m_3 & m_4 & m_5 \\
  1 & 2 & 3 & 4 & 5
\end{array}
\]

There are 7 basic arithmetic causes of 5,
\[
\begin{align*}
m_1 + m_4, & \quad m_4 + m_1 \\
m_5 / m_1, & \quad m_5 * m_1, \quad m_1 * m_5, \\
m_2 + m_3, & \quad m_3 + m_2
\end{align*}
\]
and 20 causes of "TRUE" with the relations >, <:

\[
\begin{align*}
&\quad m_1 < m_2 \\
&\quad m_1 < m_3 \\
&\quad \vdots \\
&\quad m_1 < m_5 \\
&\quad m_2 > m_1 \\
&\quad m_2 < m_3 \\
&\quad \vdots \\
&\quad \text{etc.}
\end{align*}
\]

Considering distinct causes we see there are only 4 arithmetic causes because of commutativity. Likewise if we do not consider \( m_i < m_j \) distinct from \( m_j > m_i \), there are only \( 4 + 3 + 2 + 1 = 10 \) conditions.

In general, if there are \( n \) cells and none of their values are the same, there are

\[
\sum_{i=1}^{n-1} \frac{n(n-1)}{2}
\]

distinct causes of a given logical result with a relation like >.

I'C' traces share these disadvantages with IC' traces and for these reasons we will be concerned with only I'C traces.

**Decomputation Algorithm**

The implementation of decomputation which was produced deals with the decomputation of I'C traces. A similar approach could be used in principle for IC' or I'C' traces to construct conditions from memory snapshots if that would be desired in the future.

Decomputation must produce for each trace entry \( j \), where \( T_{j2} = n \), the set of all result functions which evaluate to \( n \). Rather than create these individually for each trace entry, a table is maintained
which contains all possible result functions and their values for the memory state $S_{j-1}$. The process carried out for each trace entry is twofold:

1. If for the $j^{th}$ entry of the trace, $T_{j2} = n$, then find all result functions in the table associated to this value $n$. This will yield the decomputation set $D_j$ for the $j^{th}$ trace entry.

2. In order to do step 1 again for the $(j+1)^{th}$ entry, it will be necessary to update the table to reflect the new memory state $S_j$. The cell $m_{ij}$ in $T_{j1}$ has possibly had its contents changed and so all result functions using it as an argument must have the values associated to them updated corresponding to the change.

The structure of this basic table, called HASH, and another table called UPDATE, are determined by these two basic operations (1) and (2) described above.

**HASH** is an $M \times N$ array containing pairs of the form

$$(\text{result function, } n)$$

where $n$ is the result produced when the associated result function is evaluated for the current memory state. A given pair is located in a particular row of this table based on the value $n$. Specifically it is in the $(n \mod M)^{th}$ row of the table. For example if we have the pair $(A(I) + B, 60)$ and the table has 50 rows, the entry is in the 60 MOD 50 or 10th row of the table.

Thus the formation of the decomputation set $D_j$, where $T_{j2} = n$, simply consists of finding all result functions which are associated
with pairs having a value entry n in the (n MOD M)th row.

After the D-set is obtained, it is necessary to update this table because of the change in the cell m_{ij}. The HASH table entries involving m_{ij} are spread through the entire table. The list, UPDATE, has the values of all possible result functions for this current memory state. The list is so organized that direct calculations can be made of indices of UPDATE for the values of all functions involving m_{ij}. These values found are then used as pointers into HASH and the updating done.

The organization of UPDATE is straightforward, based on an orderly enumeration of possible result functions. For example let

\[ F = \{f_1 = +, f_2 = \ast\} \]

\[ M = \{m_1, m_2, m_3\} \]

The ordering below is by \(1^{\text{st}}\) cell, \(2^{\text{nd}}\) cell, then function index.

\[
\begin{array}{ccc}
1 & 1 & k \\
1 & 1 & 1 & m_1 + m_1 \\
1 & 1 & 2 & m_1 \ast m_1 \\
1 & 2 & 1 & m_1 + m_2 \\
1 & 2 & 2 & m_1 \ast m_2 \\
1 & 3 & 1 & m_1 + m_3 \\
1 & 3 & 2 & m_1 \ast m_3 \\
2 & 1 & 1 & m_2 + m_1 \\
\vdots \\
\end{array}
\]

Based on the number of cells and number of instructions, the position of all result functions involving, say \(m_3\), can be calculated. This entire updating procedure is summarized in the flowchart of Figure 4.1.
For given $S_0$ calculate values of all possible result functions in $E$ and construct the HASH table and UPDATE list.

Read next tuple of $T_i^*$

$T_{j1} = m_{ij}$, $T_{j2} = n$

Use $n$ as pointer into HASH and form the set of causes of $n, D_j$.

Find entries in UPDATE corresponding to $m_{ij}$ (those are old-values),

For each such entry in UPDATE replace its old-value by its new-value

Use old-values and new-values to enter HASH and make updates.

Figure 4.1 Flowchart of Decomputation Process of an I'C Trace.
Decomputation of the I'C trace resulting from the calculation of $\sqrt{49}$ shown in Figure 3.4 will be used to illustrate the entire process in an actual example. Figure 4.2 shows part of the trace of the computation, its environment, and the D-sets that resulted from the decomputation for each entry.

The next figure, 4.3, illustrates the organization of UPDATE. It shows the contents of UPDATE, at the 11th step, in the far left hand column. The other columns indicate the result functions, whose values for $S_{11}$ are in UPDATE, and the ordering of the list is illustrated by the indices $i$, $j$, $k$.

The three figures, 4.4, 4.5, and 4.6 are snapshots of some entries in UPDATE and HASH illustrating the updating done at step 12 of the decomputation where $m_{ij} = m_2$ and $n = 7$. Figure 4.4 shows two rows of HASH before the changes. In Figure 4.5 the entries in UPDATE corresponding to result functions involving $m_2$ have been changed from those of Figure 4.3. Finally in 4.6 an entry in row 4 of HASH is seen to have been eliminated because of the change in $m_2$ and new entries appear in row 5.
<table>
<thead>
<tr>
<th>State</th>
<th>Memory State</th>
<th>Memory Change</th>
<th>D-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>([m_1]_0 \ldots [m_4]_0)</td>
<td>((m_{ij}, n))</td>
<td>(D_j)</td>
</tr>
<tr>
<td>1</td>
<td>(S_1)</td>
<td>((m_3, 24))</td>
<td>({m_4/m_1, m_4/m_2})</td>
</tr>
<tr>
<td>2</td>
<td>(S_2)</td>
<td>((m_3, 26))</td>
<td>({m_1+m_3, m_2+m_3})</td>
</tr>
<tr>
<td>3</td>
<td>(S_3)</td>
<td>((m_3, 13))</td>
<td>({m_3/m_1, m_3/m_2})</td>
</tr>
<tr>
<td>4</td>
<td>(S_4)</td>
<td>((m_2, 13))</td>
<td>({\text{move } (m_3)})</td>
</tr>
<tr>
<td>5</td>
<td>(S_5)</td>
<td>((m_3, 3))</td>
<td>({m_4/m_3, m_4/m_2})</td>
</tr>
<tr>
<td>6</td>
<td>(S_6)</td>
<td>((m_3, 16))</td>
<td>({m_2+m_3, m_4/m_3})</td>
</tr>
<tr>
<td>7</td>
<td>(S_7)</td>
<td>((m_3, 8))</td>
<td>({m_3/m_1})</td>
</tr>
<tr>
<td>8</td>
<td>(S_8)</td>
<td>((m_2, 8))</td>
<td>({\text{move } (m_3)})</td>
</tr>
<tr>
<td>9</td>
<td>(S_9)</td>
<td>((m_3, 6))</td>
<td>({m_3-m_1, m_4/m_3, m_2-m_1, m_4/m_2})</td>
</tr>
<tr>
<td>10</td>
<td>(S_{10})</td>
<td>((m_3, 14))</td>
<td>({m_2+m_3})</td>
</tr>
<tr>
<td>11</td>
<td>(S_{11})</td>
<td>((m_3, 7))</td>
<td>({m_3/m_1})</td>
</tr>
<tr>
<td>12</td>
<td>(S_{12})</td>
<td>((m_2, 7))</td>
<td>({\text{move } (m_3), m_4/m_3})</td>
</tr>
<tr>
<td>13</td>
<td>(S_{13})</td>
<td>((m_3, 7))</td>
<td>({\text{move } (m_3), m_4/m_3, \text{move } (m_2), m_4/m_2})</td>
</tr>
<tr>
<td>14</td>
<td>(S_{14})</td>
<td>((m_3, 14))</td>
<td>({m_1<em>m_3, m_3+m_3, m_1</em>m_2, m_2+m_2, m_2+m_3})</td>
</tr>
<tr>
<td>15</td>
<td>(S_{15})</td>
<td>((m_3, 7))</td>
<td>({\text{move } (m_2), m_4/m_2, m_3/m_1, m_3-m_2})</td>
</tr>
</tbody>
</table>

Figure 4.2 D-Sets Resulting from Decomputation of the I'C Trace of the Computation of \(\sqrt{49}\) in Figure 3.4.
<table>
<thead>
<tr>
<th>Contents of UPDATE</th>
<th>Result functions whose values are the corresponding UPDATE entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result function</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>$f_k(m_1, m_j)$</td>
</tr>
<tr>
<td>4</td>
<td>$m_1^1 m_1$</td>
</tr>
<tr>
<td>4</td>
<td>$m_1^2 m_1$</td>
</tr>
<tr>
<td>2</td>
<td>move($m_2^1$)</td>
</tr>
<tr>
<td>10</td>
<td>$m_1^1 m_2$</td>
</tr>
<tr>
<td>-6</td>
<td>$m_1^1 m_2$</td>
</tr>
<tr>
<td>16</td>
<td>$m_1^1 m_2$</td>
</tr>
<tr>
<td>0</td>
<td>$m_1^1 / m_2$</td>
</tr>
<tr>
<td>9</td>
<td>$m_1^1 m_3$</td>
</tr>
<tr>
<td>-5</td>
<td>$m_1^1 m_3$</td>
</tr>
<tr>
<td>14</td>
<td>$m_1^1 m_3$</td>
</tr>
<tr>
<td>0</td>
<td>$m_1^1 / m_3$</td>
</tr>
<tr>
<td>51</td>
<td>$m_1^1 m_4$</td>
</tr>
<tr>
<td>-47</td>
<td>$m_1^1 m_4$</td>
</tr>
<tr>
<td>98</td>
<td>$m_1^1 m_4$</td>
</tr>
<tr>
<td>0</td>
<td>$m_1^1 / m_4$</td>
</tr>
<tr>
<td>6</td>
<td>$m_2^1 m_1$</td>
</tr>
<tr>
<td>4</td>
<td>$m_2^1 m_1$</td>
</tr>
<tr>
<td>16</td>
<td>$m_2^1 m_2$</td>
</tr>
<tr>
<td>64</td>
<td>$m_2^1 m_2$</td>
</tr>
<tr>
<td>8</td>
<td>move($m_2^2$)</td>
</tr>
<tr>
<td>15</td>
<td>$m_2^1 m_3$</td>
</tr>
<tr>
<td>49</td>
<td>move($m_4^2$)</td>
</tr>
</tbody>
</table>

Figure 4.3.—Snapshot of UPDATE at Step 11 of I'C Trace
HASH value entries
4 \((m_1 + m_1, 4); (m_1 \cdot m_1, 4); (m_2 / m_1, 4);\)
5 \((m_1 - m_3, -5); (m_3 - m_1, 5);\)

Figure 4.4 HASH Table Entries

UPDATE value corresponding result functions

\[
\begin{align*}
9 & \quad m_1 + m_2 \\
-5 & \quad m_1 - m_2 \\
14 & \quad m_1 \cdot m_2 \\
0 & \quad m_1 / m_2 \\
& \quad \vdots \\
5 & \quad m_2 - m_1 \\
3 & \quad m_2 / m_1 \\
& \quad \vdots
\end{align*}
\]

Figure 4.5 UPDATE Entry Changes

HASH value entries
4 \((m_1 + m_1, 4); (m_1 \cdot m_1, 4); (m_2 / m_1, XXX)^1;\)
5 \((m_1 - m_3, -5); (m_3 - m_1, 5); (m_1 - m_2, -5);\)
\((m_2 - m_1, 5);\)

Figure 4.6 HASH Entry Changes

---

^1 XXX in an entry indicates the entry is invalidated because of the change in the memory contents.
Decomputation and Cell Names

We have simplified for clarity the description of the entries into the HASH table. In fact the HASH table has in it the result functions formed from the cell names and not the simpler forms involving only the cell numbers. For example, if the names of $m_2$ are $A(l)$ and $A(I)$ and the name of $m_3$ is $X$, then $m_2 - m_3$ could appear in HASH as $A(l) - X$ or $A(I) - X$. Thus when the entry corresponding to $m_2 - m_3$ is used from UPDATE to enter HASH, $m_2 - m_3$ is translated into the named result functions depending on the current memory state. So an actual D-set, $D_j$, combined with the name of the result cell at the $j$th step, $a_j$, can give rise to all the possible instructions that might be used in constructing an actual program.

Restrictions on the Environment $E$

Some restrictions must be made on the nature of the functions and relations allowed in order for decomputation to be feasible. The major point here is to allow only a finite number of possible instructions or tests in $E$ so that there can be an exhaustive testing to check if they produce the required memory change. This basic requirement for finiteness is something like a closure property of the environment $E$. Thus it was necessary to disallow the use of literals for they can admit an infinite number of explanations for a memory change. For example, for all $n$ there are an infinite number of $a$'s and $b$'s such that $n = a - b$. Also for all $n$ there are an infinite number of $a$'s such that $0 = n/a$ where we are using integer truncating division.
Number of Basic Operations Performed in the Decomputation Process

Let the basic operation be the entry into the UPDATE list and the corresponding entry into the HASH table, i.e., the complete processing of a possible result function.

I. Both tables are completely initialized by generating all possible result functions in E, i.e.,

\[ f_k(m_i, m_j) \lor m_i, m_j \in M; \lor f_k \in F \]

If there are n cells and r binary functions in F then there are \( n^2 \cdot r \) result functions. If there were p-ary functions in F, the number would be \( n^p \cdot r \). However, remember that the result functions in HASH are constructed with names of cells. So if, for a given memory state, there are \( a \) names possible in E (\( a > n \)), there are

1) \( a^2 \cdot r \)

result functions to be formed. Allowing unary and binary functions we have

2) \( a^2 \cdot r_b + a \cdot r_u \)

where \( r_b \) is the number of binary functions and \( r_u \) the number of unary functions.

Now not all of these are actually formed since there is repetition for commutative operations and also since a constant is guaranteed to result when a cell is divided by or subtracted from itself it need not be in the table. When \( T_{j2} \), the result at step \( j \), is one or zero, then the decomputation adds to the set \( D_j \) of instructions from HASH, the instructions which represent memory cells being divided by or subtracted from themselves. These instructions then are produced directly each time they are needed and so are not kept in HASH. If there
are commutative operations and 2 (subtraction and division) yielding constants then there are \( (a^2 \cdot r_c/2) + a \cdot 2 \) more result functions than needed in 2). So we arrive at the expression

\[
3) \ a^2 \cdot (r_b - r_c/2) + a(r_u - 2)
\]

For example, in the sum program of Chapter 1 there are 7 cells, and if I's contents are say 1, then the first cell of the array can be called A(1) or A(I). So there are 8 names and allowing the 4 arithmetic operations and a move operation one has

\[
8^2 \cdot (4 - 2/2) + 8(1 - 2) = 184
\]

result functions for this simple example.

II. Now in addition to this initialization work, we must analyze what is done during decompuation for each entry of a trace. For each entry it is necessary to update for all result functions involving \( m_{ij} \), the cell affected at this step. The number of such result functions can be found by calculating how many involve at least one occurrence of \( m_{ij} \). To calculate this, subtract away from the total number of possible functions, the number with no \( m_{ij} \) occurring as an argument. This is the same thing as finding the total number of result functions in an environment without that cell. Assume \( m_{ij} \) has \( b \) associated names and \( a \) is the total number of names for the current memory state. Then we get from the above discussion the result

\[
(a - b)^2 \cdot (r_b - r_c/2) + (a - b) \cdot (r_u - 2)
\]

This can be subtracted from the total number in 3) to give

\[
4) \ (a^2 - (a-b)^2) \cdot (r_b - r_c/2) + (a - (a-b)) \cdot (r_u - 2)
\]

or

\[
4') \ (2ab - b^2) \cdot (r_b - r_c/2) + b \cdot (r_u - 2)
\]
as the number of result functions to be updated at a step \( j \).

Consider again the sum calculation. In Figure 1.2 we have at the 7th entry of the memory trace, SUM being the variable changed. So in this environment for \( S_6 \), \( a = 8 \) and \( b = 1 \), and

\[
(2 \cdot 8 - 1) \cdot (4 - 1) + 1 \cdot (1 - 2) = 44
\]

In general for binary functions, the number of basic operations needed for the initialization process is approximately

\[
a^2 \cdot r
\]

where \( a \) is number of names of cells possible at initialization and \( r \) is the number of functions. Let \( \bar{a} \) be the average number of possible cell names per step during the computation. Then for each update step the number of basic operations is of the order of

\[
\bar{a} \cdot r
\]

**Automatic Indexing**

To capture the loose style of specification as illustrated in the introductory chapter we have been considering I'C traces of a computation. One can also consider allowing entire steps to be omitted from a trace. Arbitrary omission of steps is very difficult as guesses must be made as to where steps are omitted and how many steps are left out. It is possible to overcome some of these problems by considering the semantics of the environment \( E \) and supplying instructions or memory changes when they are known to be missing.

In particular we shall consider the use of the environment's semantics about array type structures to allow the omission of certain array indexing instructions. If some higher level instructions related
to arrays in the environment are available to the user then for a large number of situations explicit indexing for arrays need not be specified.

This process of allowing automatic generation of index variables for certain situations will be called **automatic indexing**. This process would be done before decompuation, taking the incomplete trace and filling in missing steps so that it can then be decomputed.

Let us first consider an example which will illustrate ideas about automatic indexing. The following is a segment of a calculation's flowchart in which every second element of an array is being squared:
If in the computation environment E we have

```
A
7
-2
3
8
6
5
```

then the sequence of memory changes corresponding to the program segment is

```
(I, 2)
(A(I), 4)
(I, 4)
(A(I), 64)
(I, 6)
(A(I), 25)
(I, 8)
```

Note: I > N

Now we will give a very important assumption or restriction on calculations carried out in an environment. This restriction is used particularly during synthesis but it also is a property which facilitates automatic indexing. What is required is that if a cell in an array structure is to be used as a result cell of a calculation there **must** be an index or pointer declared for the structure whose current contents allow it to be used in the name of the cell. For example,
if the 2nd cell of A is to be a result cell, then I must be a declared pointer and have the value 2, i.e., A(I) is allowed but not A(2) as a name for the result cell.

Thus if the second cell of A has its value changed and there is no valid index, the system can, knowing there must be one, take steps to create a valid index.

(A) If a pointer is declared but it hasn't a correct value add to the trace just before the reference to the cell of the array, an entry making the required change to the index. For example (1, 2) would be inserted as an entry in the trace just before (A(I), 4).

(B) If a pointer hasn't been declared there will be an implicit one (pair) available for use with an array. Then the entry in the trace will be made as in (A) above. If more indices than necessary are used a post-optimization can be done after the final synthesis step.

Note that we have made no assumption as to the nature of the missing instructions, e.g., I + I+1, etc. This will be accomplished as if the user had done indexing himself and we then decomputed and synthesized. It is precisely the decomputation and then synthesis which makes this possible.

Thus what the decomputation makes possible is a default: within certain restrictions references to arrays can be allowed without an explicit use of pointers. It must be noted that the indexing must nonetheless be used implicitly in an unambiguous manner. This is
necessary for the automatic indexing to be sound, i.e., the default is always what the user would have done had he been indexing explicitly.

It is doubtful if all indexing functions can become automatic. For example, in cases where there is more than one pointer, it is always ambiguous as to which should be the index.

However, it is possible to allow some additional instructions pertaining to indexing which could allow an unambiguous interpretation of a larger range of default indexing (under some restrictions). Consider the following example:

```
A
1
7
5
2
4
```

It is desired to search A for a given element, say 2. The problem here, and in many other cases, is that cells of A are being referenced but not changed, and we have no unambiguous knowledge in general of reference to a cell. There must be an equivalent to what a person would do in explaining such a process via an example. That is, a person would say things like:

"Look down the list until you find ...",

"Consider every 2\textsuperscript{nd} element ...", etc.
Most cases can be covered by the following 3 instructions:

SCAN
END SCAN
INDEX

The first instruction would be most important and most complex:

SYNTAX: SCAN, A(I_1), A(I_2), ... A(I_k), END SCAN;

The search for the element 2 in A on the previous page would then become the following sequence of references made, say with a light pen,

SCAN
1
7
5
2
Condition
END SCAN

The SCAN-END SCAN allows the system to know that an array is being referenced and thus to do automatic indexing as previously. Between pointing to elements of the array during a SCAN any number of other instructions can be performed under some necessary restrictions with respect to array references. INDEX will allow a specific reference to the implicit pointer the system has created.
CHAPTER 5. SYNTHESIS

This chapter will consider the problem of the inference or synthesis of programs from traces. The representation that will be used for these programs is the flowchart form of Chapter 3. It will be assumed for now that any specified trace is correct, i.e., the person has made no errors in the computations he has performed in E. Then the result of the synthesis process should be a program which, when run on the data of the sample computation, produces the same trace. Such a program will be said to be compatible with the trace.

**Def.** Let \( T \) be a trace which resulted from a computation with initial state \( S_0 \). Let \( P \) be a program and let \( P(S_0) \) represent the trace of the program's execution with \( S_0 \) as the initial state. Then \( P \) will be said to be **compatible** with \( T \) if

\[
T = P(S_0)
\]

In addition to compatibility with a trace, an inferred program will be required to be of minimal size by some measure. If a size bound were not imposed, then trivial programs would result. A program compatible with a trace can be produced by allowing a program instruction for each entry in the trace, i.e., effectively making the trace itself a program directly. Now we define the size measure for programs which was used in this work.
Def. The length of a program, $L(P)$, is the number of instruction vertices in the flowchart form of the program.

So a synthesis procedure will be required to yield a program of minimal length, denoted $P_{\text{MIN}}$, compatible with the given traces.

Computation traces have information not usually contained in the formal specifications used in most program synthesis work, since the sequence of entries in a trace strongly represents the control flow of the algorithm. It is possible to enumerate programs in some order and then test to see if they are compatible with the given trace(s), finally yielding a $P_{\text{MIN}}$ (See Appendix A). However, such a uniform generate and test method fails to make full use of the information in a computation trace. First an analysis of the trace can yield a lower bound, $B$, on the size of $P_{\text{MIN}}$. Thus it would be unnecessary to enumerate all programs of size less than $B$. If in fact though,

$$L(P_{\text{MIN}}) = B+k$$

it will be necessary to generate and test all programs of size $B$, $B+1$, $B+2$, ... $B+(k-1)$ and also some of size $B+k$.

Moreover, the sequence information in a trace can be used so that it isn't necessary to completely construct a program and then test it for compatibility. Instead minimal partial programs are constructed to be compatible with an increasing number of the trace entries, until a complete compatible program is created. Thus the generation and testing of a program are effectively combined.

Let us now start the description of a synthesis algorithm by stating what the result of a synthesis will be with respect to an input trace. We begin with IC traces. Synthesis will produce the simplest
possible program, \( P_{MIN} \), compatible with the trace. Since there can be multiple occurrences of the same instructions in \( P_{MIN} \), we will need the following definitions.

**Def.** A given instruction \( I_j \) can appear more than once in a program. Each such appearance of \( I_j \) will be called a **program occurrence** of \( I_j \) and the \( k^{th} \) program occurrence of \( I_j \) will be denoted \( kI_j \).

For example in the program of Figure 5.1, the first block of the flowchart contains \( I_1 \) and so is written, \( I_1 I_1 \). The second occurrence of \( I_1 \) after the test \( C_1 \) in the flowchart is then written as \( 2I_1 \).

**Def.** Each appearance of \( I_j \) as the instruction in an entry of a trace will be called a **trace occurrence** of \( I_j \).

Consider in Figure 5.2 a trace of the program of Figure 5.1. In it we see two trace occurrences of \( I_2 \) and \( I_3 \), each corresponding to one program occurrence of \( I_2 \) and \( I_3 \). There are also two trace occurrences of \( I_1 \) which correspond to different program occurrences of \( I_1 \).

Thus the final result of synthesis will be the assignment of the correct program occurrences of \( I_j \) in \( P_{MIN} \) for every trace occurrence of \( I_j \). In the trace of Figure 5.2 then it is necessary to assign all the trace occurrences of \( I_2 \) and \( I_3 \) to the one program occurrence of each. It is inferred that the individual trace occurrences represent one program occurrence, in a sense generalizing these trace occurrences.
Figure 5.1 Example Program

\[ (T_{j1}, T_{j2}) \]

1  \hspace{2cm} (I_1, -)
2  \hspace{2cm} (I_2, -)
3  \hspace{2cm} (I_3, -)
4  \hspace{2cm} (I_2, -)
5  \hspace{2cm} (I_3, C_1)
6  \hspace{2cm} (I_1, -)
7  \hspace{2cm} (I_4, -)

Figure 5.2 IC Trace of Program of Figure 5.1
Next consider the problems of synthesis from decomputed I'C traces. Here a single specific instruction is not given for each trace entry, but a set of possible instructions. Hence it is necessary to first decide on the trace occurrence for the entry and then the program occurrence. The process for decomputed I'C traces can thus be summarized as, for every trace entry, \( j = 1, \ldots, m \)

a) choose an \( I_k \in D_j \) and then

b) choose a program occurrence for \( I_k \)

so that the resulting program is minimal. There seems to be basically two ways for making these necessary choices. One way is to first select instructions for every trace entry, effectively creating an IC trace, and then to proceed with the second choice for each entry as with an IC trace. So it would then be necessary to create all possible IC traces from the decomputed I'C trace, and synthesize a program from every one until it is known that a minimal program has been produced. Further discussion of this method is found in Appendix B.

The other method is to make both of these choices each time for every entry, thus effectively creating the program as the choices are made. This represents an extension to decomputed I'C traces of the synthesis methods developed in [3] for Turing machine traces and [4] for IC traces.

---

1 Note \( D_j \) really contains result functions, not instructions. However, an instruction can be formed from a result function and a result cell name. Since the decomputed I'C trace has these names specified for each trace entry, selection of a result function from \( D_j \) is tantamount to selection of the instruction, and will be so considered in the future.
This method for the synthesis of programs for decomputed I'C traces will be described now in some detail. The process starts with some lower bound $B$ on the program size which cannot be exceeded unless all programs of smaller size have been eliminated. Then the 2 choices are made at each entry so that the resulting (partial) program is (1) compatible with the trace, (2) no larger than $B$ at this stage unless all smaller possible programs have been eliminated. When all such possible combinations of choices have been made for each entry, and it hasn't been possible to construct a compatible program of size $B$, the size bound is increased

$$B + B + 1$$

and the process started again. What this means is that we have effectively tried all programs of size $B$ and since none have been compatible, we know the correct $P_{\text{MIN}}$ must be larger. So we are indeed enumerating programs, none are skipped and we are sure we will find the minimal program.

The actual basic procedure will now be outlined and illustrated; after this we will give the techniques which make the procedure computationally feasible.

**Definitions for the Synthesis from I'C Traces**

The process of choosing an instruction $l_k$ and assigning it to some program occurrence, say $2l_k$, for any entry of a decomputed I'C trace, $T$, will be called a step of the synthesis. Some of these choices will be arbitrary, called arbitrary steps, and some constrained by previous choices, called forced steps.
As the synthesis proceeds making these choices, a transition
matrix representation of the partial program up to the current entry
is created. Now some definitions are made and illustrated to des­
cribe the synthesis.

Def. A transition, \((mI_1, C_j, nI_k)\) is applicable to the \(\ell\)th trace
entry of an I'C trace if

1) \(I_1\) has been chosen from \(D_\ell\) and assigned to program
   occurrence \(mI_1\), and

2) \(T_{j\ell} = C_j\)

Def. A forced step at entry \(j\) results in the choice of a particular
instruction, \(I_k\), from \(D_j\), and choice of a particular program occur­
rence of \(I_k\), say \(m\), thus assigning \(mI_k\) at entry \(j\). These forced
choices are due to an applicable transition at entry \(j-1\) with \(mI_k\) as
the 3\(^{rd}\) entry in the transition tuple. For example, assume \(I_7 \in D_j\)
and that \((2I_1, C_2, 2I_7)\) is applicable at entry \(j-1\). Then the choices
for entry \(j\) are forced by the already existing transition to be such
that \(2I_7\) is the assignment made at \(j\). Forced steps in the trace are
just the result of loops in the compatible program. Any step which
is not forced will be called arbitrary.

Def. A contradiction is said to occur when there is a transition,
\((mI_1, C_j, nI_k)\) applicable to the \(\ell\)th entry but \(I_k \notin D_{\ell+1}\).

A contradiction occurs because of an incorrect previous decision about
one of the two choices at the last arbitrary (non-forced) entry. In
order for the synthesis to proceed it will be necessary to backtrack
to this entry and try to make changes to the choices made there. The possible choices are (1) assignment of the given I to a new program occurrence or (2) choose a new I from the D-set. Either of these may not be possible and when all choices at a given entry are not possible, then there is a failure at the entry. Then it will be necessary to backtrack further. If all choices at all entries are unsuccessful then all possible programs of size B have failed. So it is necessary to start trying on larger possible programs and the size bound is increased and the process started again. The steps of this process will now be given, outlining the entire procedure. Figure 5.3 is a flowchart of the steps of the process explained below.

**Synthesis Outline**

(1) Initialize the program size bound B to any lower bound established from the trace.

(2) Set the trace index j to 1, j + 1.

(3) Choose the first instruction, I, from D_j and choose the first program occurrence of I to assign it to.

(4) Increment the trace index, j = j + 1. If we have passed the end of the trace halt; if not, go to the next step.

(5) Is there an applicable transition at entry j? If not go to step (3) to make an arbitrary choice at j, else go to the next step.

(6) Does the transition which applies here cause a contradiction? If not, this is a forced step, make the forced assignment, and proceed to the next entry, i.e., go to step (4).

If there is a contradiction, we must backup, so go to the next step, step 7.
Forced Step, Make Assignments and Go To Next Entry.

Is any transition applicable?

Is there a contradiction?

Failure at This Entry

Try to Make New Choice

Success, New Choice(s) Made. Go to Next Entry.

End of Trace

HALT

Reach Start of trace
(7) Backup to the entry corresponding to the last arbitrary (non-forced) step. If we have already reached the start of the trace, \( j = 1 \), we cannot backtrack further and must go to step 9 to increase the permitted program size. Otherwise go to the next step.

(8) Having backtracked to this entry, the last selection(s) made here must have been unsuccessful and so some choice must be changed. There are two kinds of changes possible here:

1) a change of the program occurrence for this entry

   \[ mI \to (m+1)I, \text{ or if this is not permissible,} \]

2) a change of the trace occurrence for this entry,

   i.e., if \( I \) is currently the instruction name assigned choose a new \( I' \) from the D-set.

The first change will not be made if the \((m+1)^{th}\) occurrence of \( I \) exceeds the bound \( B \) or if it doesn't represent a new program occurrence but is just a renaming of \( mI \). The second change is also not allowable if the new \( I' \) causes the bound \( B \) to be exceeded. In this case another new instruction not causing a violation of the size bound will be chosen if there are any such instructions left in \( D_j \).

If it was possible to make any kind of assignment change at this step then continue with the synthesis, i.e., go back to step 4 and the next entry. Finally if all changes are attempted and for one reason or another none are successful, then there has been a complete failure at this entry and further backup is required, i.e., go to step 7.

(9) Since this step is taken when we have backed up to the start of the trace, this means all possible programs of size \( B \) have
failed to be compatible with the trace. So we must start over allowing a larger program, i.e., increment B,
\[ B + B+1 \]
and go to step 2 beginning again at the first trace entry.

**Annotated Synthesis Example**

The synthesis process will now be illustrated by a complete sample synthesis. The synthesis input is represented in Figure 5.4 by the column of D-sets that resulted from the decomputation of an I'C trace and the conditions or \( T_j \) column. The steps of the synthesis procedure for this trace are given in Figure 5.5. Finally, the program synthesized is shown in Figure 5.6. A preliminary analysis of the information in Figure 5.4 shows that at least 3 instructions must occur in any program compatible with this data (this analysis process will be discussed later in this chapter). Thus we start synthesis with an initial bound of \( B = 3 \).

Steps 1 - 4: These are arbitrary steps in which the first instruction in each D-set is chosen. For steps 1-3 these selections also represent the first program occurrence of the instruction. At step 4 the chosen instruction, \( I_3 \), is assigned to the already existing first program occurrence of \( I_3 \), \( I_1 \), from step 2. So at this point the following transitions have been created:

1. \((I_7, -, I_3)\)
2. \((I_3, -, I_2)\)
3. \((I_2, -, I_3)\)
Figure 5.4 Decomputed Representation of an I'C Trace.

<table>
<thead>
<tr>
<th>Index j</th>
<th>$D_j$</th>
<th>$T_{j3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${I_7, I_4, I_3}$</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>${I_3, I_{11}, I_8}$</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>${I_2, I_5}$</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>${I_3, I_1}$</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>${I_1, I_9, I_5}$</td>
<td>$C_1$</td>
</tr>
</tbody>
</table>

Figure 5.5 Synthesis Procedure for Trace of Figure 5.4

<table>
<thead>
<tr>
<th>Step</th>
<th>Trace Index</th>
<th>Assignment Attempted</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$II_7$</td>
<td>arbitrary</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$II_3$</td>
<td>arbitrary</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$II_2$</td>
<td>arbitrary</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$II_3$</td>
<td>arbitrary</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$II_2$</td>
<td>contradiction</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>$2II_3$</td>
<td>Both 6 &amp; 7 fail because they exceed the initial bound $B = 3$. Since these are all possibilities at $j = 4$, step 8 backtracks to entry 3.</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>$II_1$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>$II_5$</td>
<td>Arbitrary - next possible choice after $II_2$</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>$II_3$</td>
<td>arbitrary</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>$II_5$</td>
<td>forced</td>
</tr>
</tbody>
</table>

Again here when we write $D_j$, we write it as a set of instructions, those formed from combining the result functions in the $D_j$ from de-computation and the result cell name from $T_{j1}$.
Step 5: The second transition above, \((II_3, -, II_2)\), is applicable at \(j = 4\) in the trace since \(II_3\) is assigned there currently and \(T_{j3}\) is blank. However the attempted forced assignment of \(II_2\) at \(j = 5\) is contradictory since \(I_2 \notin D_5\).

Step 6: Since there was a contradiction at the last step, backup is done to the last arbitrary (non-forced) step, and the last free choice changed. So we backup to \(j = 4\) and since the last choice was to assign \(I_3\) to the first program occurrence, this choice is changed and \(2I_3\) is assigned at entry 4 (creating a new program occurrence of \(I_3\)). Because a new program occurrence is created, we check and see this creates a program of size 4, exceeding the current bound of 3.

The choice of program occurrence cannot be changed again since we have now exhausted all the possibilities for \(I_3\) and so the prior choice in which \(I_3\) was selected must be changed.

Step 7: Here the choice changed is that of the trace occurrence for \(j = 4\). \(I_1\) is now selected and since it is the first \(I_1\) selected, it is assigned to \(II_1\). Again this then represents a new program occurrence and exceeds the current program size bound. Hence again the last choice before this must be changed. But no other choice exists for a program occurrence of \(I_1\) and so another new trace occurrence must be tried. However \(D_4\) is exhausted, so we have failed completely at level 4. A choice even further back must be changed and the last arbitrary step was at \(j = 3\).

Step 8: The last choice made at entry 3 was the assignment of \(I_2\) to the first program occurrence, but this cannot be changed to \(2I_2\) since it wouldn't really represent a new choice. The last free choice
was of \( I_2 \) for the trace occurrence at \( j = 3 \). Hence the next instruction in \( D_3 \), \( I_5 \), is chosen and since it is a new trace occurrence it creates a new program occurrence, \( II_5 \). Because only two other program occurrences are currently assigned, this new one doesn't cause the size of the partial program to exceed 3, the value of the bound \( B \).

Step 9: Returning to \( j = 4 \), the same arbitrary choice must be made, \( II_3 \). The transition matrix representing the partial program constructed so far is now

1) \((II_7, -, II_3)\)
2) \((II_3, -, II_5)\)
3) \((II_5, -, II_3)\)

Step 10: Once more as in step 5, a transition is applicable to entry 4. However this time, rather than a contradiction occurring, there is a forced choice of \( II_5 \), where \( I_5 \) is the 3rd instruction in \( D_5 \).

The end of the trace has now been reached and so a final halt transition is created

4) \((II_5, C_1, HALT)\)

These 4 transitions then represent the program shown in Figure 5.6.

![Figure 5.6 Program Produced by Synthesis](image)
Let us make 2 observations about the above synthesis. First, note that if $I_5$ had been the first instruction in $D_3$, it would have been chosen, there would have been no contradiction at step 5, and hence the backup would have been avoided. Second, even if $I_5$ were not the first instruction in $D_3$, it could be seen that a selection of $I_2$ would fail. Looking ahead in the trace to entry 5 one sees that $I_2 \notin D_5$. But this means a new program instruction would be needed to explain entry 5 implying 4 instructions in the program being constructed. So the choice of $I_2$ at entry 3 would inevitably cause the bound $B = 3$ to be exceeded. These two observations illustrate the basic ideas behind two techniques, re-ordering and look-ahead, which will be utilized to improve the synthesis process.
Trace Partitions

The concept of trace partitions will be introduced now to permit the description of the structure of a trace as it is related to the process of synthesis. First we give the definitions and some examples of the following terms: partition, refinement and decomposition. These terms will be used throughout the discussion of the synthesis process.

Def. A partition, \( P \), of a set \( S \) is a set of subsets of \( S \), \( P = \{\pi_1, \pi_2, \ldots, \pi_n\} \), such that the subsets are pairwise disjoint and their union is \( S \), i.e.,

\[
(1) \bigcup_{i=1}^{n} \pi_i = S \\
(2) \pi_i \cap \pi_j = \emptyset \quad 1 \leq i, j \leq n, i \neq j
\]

The subsets \( \pi_i, \pi_j \) in a partition are usually called blocks of the partition. A partition of a set can be thought of as having been induced by an equivalence relation over the set. Thus the blocks of the induced partition are equivalence classes of the relation, that is, the elements of a given block are in some sense equivalent.

Def. A refinement of a partition \( P \) of a set \( S \) is another partition \( P' = \{\pi'_1, \pi'_2, \ldots, \pi'_m\} \) such that for any \( \pi'_j \in P' \)

\[
\pi'_j \subset \pi_k
\]

for one and only one \( \pi_k \in P \). This means that every block of \( P' \) is contained in one and only one block of \( P \), but a block of \( P \) can contain several of the blocks of \( P' \).
Def. A decomposition, \( R \), of a set \( S \) is a set of subsets of \( S \),
\[ R = \{ \tau_1, \tau_2, \ldots, \tau_n \}. \]
The union of these subsets is still \( S \) as for a partition but these subsets need not be mutually exclusive.

For example, let \( S = \{ s_1, s_2, \ldots, s_9 \} \). Then \( P = \{ \pi_1, \pi_2, \pi_3 \} \)
is a partition of \( S \) where
\[
\begin{align*}
\pi_1 &= \{ s_1, s_2, s_3, s_4 \} \\
\pi_2 &= \{ s_5, s_6, s_7 \} \\
\pi_3 &= \{ s_8, s_9 \}
\end{align*}
\]
\( P' = \{ \pi'_1, \pi'_2, \pi'_3, \pi'_4, \pi'_5 \} \) is a refinement of \( P \)
\[
\begin{align*}
\pi'_1 &= \{ s_1, s_2 \} \subseteq \pi_1 \\
\pi'_2 &= \{ s_3, s_4 \} \subseteq \pi_1 \\
\pi'_3 &= \{ s_5, s_6 \} \subseteq \pi_2 \\
\pi'_4 &= \{ s_7 \} \subseteq \pi_2 \\
\pi'_5 &= \{ s_8, s_9 \} \subseteq \pi_3
\end{align*}
\]
\( R = \{ \tau_1, \tau_2, \tau_3, \tau_4 \} \) is a decomposition of \( S \) where
\[
\begin{align*}
\tau_1 &= \{ s_1, s_2, s_3, s_4 \} \\
\tau_2 &= \{ s_1, s_3, s_5, s_6 \} \\
\tau_3 &= \{ s_5, s_6, s_7, s_4, s_8 \} \\
\tau_4 &= \{ s_9 \}
\end{align*}
\]
These three structures are illustrated in Figures 5.7, 5.8, and 5.9.
Figure 5.7 Partition $P$

Figure 5.8 Refinement $P'$

Figure 5.9 Decomposition $R$
For any IC trace a partition of the set of trace entries is induced by the names of the instructions associated to the entries. Let a trace, $T_{IC}$, have $n$ different instruction names (trace occurrences) appearing in it. Then there are $n$ blocks of the corresponding partition, $\pi_1, \pi_2, \ldots, \pi_n$, where each block is the set of trace entries which are equivalent in that they have the same instruction name for each entry. Since every trace entry is associated with one and only one instruction, every entry is in one and only one of the $\pi_i$'s and so this clearly is a partition of the trace entries. Say the $k^{th}$ different instruction of the trace is $I_k$, then

$$\pi_k = \{ j | T_{j1} = I_k \}$$

that is, $\pi_k$ is the set of all entries which are trace occurrences of $I_k$.

Now the synthesis process can be thought of as producing a refinement of the above partition based on the multiple occurrences of instructions with the same name in the program synthesized. A block in this refinement will then be the set of all trace entries corresponding to the same program occurrence of an instruction. So if a final program has two program occurrences of $I_7$, $II_7$ and $2I_7$, then the block $\pi_k$ above would be refined or split into two blocks, the set of entries that correspond to $II_7$ and the set of entries corresponding to $2I_7$.

For example the trace of Figure 5.2 has the partition

$$\pi_1 = \{ j | T_{j1} = I_1 \} = \{1, 6\}$$
$$\pi_2 = \{ j | T_{j1} = I_2 \} = \{2, 4\}$$
$$\pi_3 = \{ j | T_{j1} = I_3 \} = \{3, 5\}$$
$$\pi_4 = \{ j | T_{j1} = I_4 \} = \{7\}$$
The first block \( \pi_1 \) must be split by the synthesis to represent
the two program occurrences of \( I_1 \) in the corresponding program of
Figure 5.1.

When a final refinement is achieved, each block of the partition
represents an instruction in the program to be constructed, a specific
program occurrence. In the final refinement of the example above,
\( \pi_1' = \{1\}, \pi_2' = \{6\} \), and since these are different program occurrences,
the trace entries 1 and 6 can be labeled \( 1_1 \) and \( 2_1 \) respectively.
After a labeling like this is done for every trace entry, the final
transition matrix representing the flow diagram of the program can be
read from the trace immediately. In summary the final refinement
yields the flowchart directly, after the conditions are properly
resolved.

In an I'C trace, since the instructions are not definitely known
for the entries, the initial partition induced by the instruction
names is not known. So synthesis for decomputed I'C traces must

(1) produce the instruction partition of \( T_{I'C} \), and

(2) then produce the refinement of it to yield a final program.

This is nothing more than the two choices we described before, choosing
a trace occurrence and then a program occurrence for each entry.

Although the instruction partition of an I'C is unknown, an
initial partition of the trace can be achieved by considering the
result cells for each entry. Since all instructions used are assign­
ment type instructions, if the result cells of two instructions are
different, so are the instructions. For example

\[ m_1 + m_2 + m_4 \quad \text{and} \quad m_3 + m_2 + m_4 \]
are different instructions. Since we have made the restriction that only one name is allowed for a result cell, no ambiguity arises due to multiple result names for an instruction. If for two trace entries, j and k, the result cell names, \( a_j^3 \) and \( a_k \), are different, then we know the instructions associated to \( j \) and \( k \) must be different.

Assume that there are \( n \) different result names appearing in the decomputed trace \( T_{1,C} \) and form the sets of entries which have the same result names, called result sets.

**Def.** A result set or, r-set, \( R_i \), for \( T_{1,C} \) is the set of indices of all trace entries having the same, \( i^{th} \), result name appearing in the trace,

\[ R_i = \{ j | a_j \text{ is the } i^{th} \text{ result name occurring in the trace} \} \]

Since every entry has a result name and an entry is only grouped with entries of the same name, the sets \( R_1, \ldots, R_n \) are a partition of the trace entries.

An instruction partition, \( \pi_1, \pi_2, \ldots, \pi_m \), must be a refinement of the partition defined by result sets. For a given \( \pi_i \), the result names associated with all entries, \( j \in \pi_i \), must be the same name, \( a_j \), since all entries in \( \pi_i \) correspond to the same instruction, \( I_k \). So all these entries must be in the same r-set, \( R_k \), where the result name, \( a_j \), of \( I_k \), is the \( k^{th} \) result name occurring in the trace. However, it is clear that different instructions, i.e., different partition blocks, \( \pi_i, \pi_j \), can have the same result name and so their corresponding entries are in the same r-set. For example, in a use

---

3 Here the term \( a_j \) refers to the name of the result cell, \( T_{j1} \), found at the \( j^{th} \) entry.
of I as the name of an index, the initialization and increment instructions

\[ I + 1 \]

\[ I + I + 1 \]

will produce in an I'C trace, entries with the same result name, I. However, these correspond to different instructions, or trace occurrences in an equivalent IC trace.

Pre-processing

In pre-processing, the r-sets defined above are examined and it is attempted to further partition them, approaching the final synthesis refinement.

Def. Two entries, i, j, in an I'C trace will be said to be distinct if it is known that the entries arose from the execution of different instructions in \( P_{\text{MIN}} \).

Entries in different r-sets are known to be distinct. The pre-processing will try to further partition each r-set by showing entries of a given r-set to be distinct. To describe this process we make the following definitions:

Def. A pair of entries i, j, (denoted \((i,j)\)) are said to be coupled if i, j are in the same r-set, \( R_k \), and \( T_{i3} = T_{j3} \).

Def. A chain of entries of length \( n + 1 \) is formed if \( n + 1 \) pairs of entries

\((i,j), (i + 1, j + 1), \ldots (i + n, j + n)\)

are each coupled entries.
If two entries \((i, j)\) are coupled and the following entries \((i + 1, j + 1)\) are distinct, then since we are considering deterministic programs, the entries \(i\) and \(j\) are distinct. Also if we have a chain 
\[(i, j), (i + 1, j + 1), \ldots (i + n, j + n)\]
d and the entries \((i + n + 1, j + n + 1)\) are distinct then by propagation back through the chain, all the pairs of entries in the chain are also distinct.

The entries \((i, j)\) are known to be distinct if

1. \(a_i \neq a_j\), i.e., \(i, j\) are not in the same \(r\)-set, or
2. \(D_i \cap D_j = \emptyset\)

The first condition is immediately obtainable from the trace. Because of the cost of performing set intersections, the second condition is checked only if the first fails to apply.

The entire trace is scanned in order to detect the conditions and chains which will allow us to say that entries of the trace are distinct (non-equivalent). A matrix, \(DV\), of such non-equivalences between entries is built as the trace is scanned. For a given entry \(j, j \in R_k\), all entries in \(R_k\) following \(j\) in the trace are checked with \(j\) for non-equivalence. So if \(\ell \in R_k\), then the entry, \(DV_{j\ell}\), of the matrix \(DV\) is

\[
DV_{j\ell} = \begin{cases} 
\text{undefined if } \ell \leq j \\
1, \text{ if } \ell \text{ is found to be distinct from } j \\
0, \text{ otherwise}
\end{cases}
\]

A given \(r\)-set, \(R_k\), can be broken into smaller sets of entries on the basis of the relation between elements in \(R_k\) as specified in \(DV\). Corresponding to each entry \(i \in R_k\) will be the set of entries not known to be distinct. This subset of \(R_k\) of entries not known to be
distinct from \(i\) will be denoted \(R_k^i\). That is

\[
R_k^i = \{ j \in R_k | DV_{ij} = 0 \text{ or } DV_{ji} = 0 \} \cup \{i\}.
\]

These sets will not generally partition \(R_k\) as a given entry can be in more than one set. If two entries \(i, j\) are distinct then \(R_k^i \neq R_k^j\) since \(i \notin R_k^j\) and \(j \notin R_k^i\). If entry \(\ell\) isn't known to be different from \(i\) or \(j\) it will be in both sets, but by the end of the synthesis \(\ell\) may be distinguished from \(i\) or \(j\), or both.

For example, if \(R_k = \{2, 3, 5, 7\}\) then a possible matrix of differences for this set might be

\[
\begin{array}{cccc}
2 & 3 & 5 & 7 \\
\hline
2 & 1 & 1 & 1 \\
3 & 0 & 0 & \ \\
5 & & & 1 \\
7 & & & \\
\end{array}
\]

These matrix entries mean that entry 2 is distinct from entries 3, 5, and 7; entry 3 is distinct from 2 but is not known to be distinct from 5, 7; etc. The resulting decomposition sets of \(R_k\) are then

\[
\begin{align*}
R_k^2 & = \{2\} \\
R_k^3 & = \{5, 7, 3\} \\
R_k^5 & = \{3, 5\} \\
R_k^7 & = \{3, 7\}
\end{align*}
\]

and can be diagrammed as
Now what does such a decomposition help us say about the partition which will be induced by the synthesized program? It will allow the calculation of a lower bound, $b^*_k$, on the number of program occurrences (blocks of the final partition) associated with entries of $R_k$. The partition that is finally produced by the synthesis must of course be compatible with the non-equivalence information in $DV$. The basic condition that must be satisfied by all possible partitions is that the trace entries known to be distinct cannot be in the same block of the final partition. Let

$$\pi^1_k, \ldots, \pi^n_k$$

be any possible partition of $R_k$ and let two entries $i, j \in R_k$ be known to be distinct. Then in order to be compatible with the non-equivalence of $i$ and $j$, if $i \in \pi^s_k$, $j$ must be in a different block $\pi^t_k$ of the partition.

For example the decomposition of $R_k = \{2, 3, 5, 7\}$ shown above is compatible with 3 possible partitions:
Partition 1: \( \pi^1_k = \{2\}; \pi^2_k = \{3, 5\}; \pi^3_k = \{7\} \)

Partition 2: \( \pi^1_k = \{2\}; \pi^2_k = \{5\}; \pi^3_k = \{3, 7\} \)

Partition 3: \( \pi^1_k = \{2\}; \pi^2_k = \{3\}; \pi^3_k = \{5\}; \pi^4_k = \{7\} \)

In general, the minimum number of blocks in any of the possible compatible partitions will represent a lower bound, \( b_k \), on the number of program occurrences which must be associated with \( R_k \). Here we see that \( b_k = 3 \).

So the final result of pre-processing is to produce the lower bound \( b_k \) for all \( r \)-sets \( R_k \). This is used during synthesis in a similar manner for each \( R_k \) to the program size bound \( B \) described at the start of this chapter.

Re-ordering

The basic synthesis search proceeds along a trace constructing the minimal partial program compatible with the trace up to a current entry. When the search reaches an entry \( j \) at which an arbitrary step is to be made, a selection must be made from \( D_j \) of an instruction to be assigned to the entry. Usually, a random choice will be incorrect, and there will be a failure or a contradiction at some future step. Eventually a choice will be made of a suitable instruction for the entry permitting the synthesis to proceed.

If the selection process is improved so that the probability of an initial choice being correct is increased, much of the search's backtracking is eliminated. This is seen in the example synthesis of the trace in Figure 5.4. As we noted there, if the order of the instructions in \( D_j \) were interchanged the synthesis would have
proceeded without backups. We will describe two techniques for this purpose: re-ordering and look-ahead.

Re-ordering is a pre-synthesis process which heuristically orders the elements of the instruction sets, the $D_i$'s. Pre-processing has established the decompositions of the $r$-sets. Consider an entry $j \in R_k$ which is in a number of the subsets of $R_k$ comprising the decomposition of $R_k$. This means $j$ could correspond to the same program occurrence as any of the entries in a set $R^j_k$ of which it is an element. The re-ordering criterion for elements of $D_j$ will then be based on the number of times it occurs in the $D$-sets of the above entries.

Def. The re-order weight of an element $e, e \in D_j$, is written $w(e,j)$, and is defined as

$$w(e,j) = \sum_{i \in Q_j} c_i(e)$$

where

$$Q_j = \{ i \mid i \text{ is a trace entry, } i \neq j, \text{ such that } i \text{ isn't known to be distinct from } j \}$$

and where

$$c_i(e) = \begin{cases} 1 & \text{if } e \in D_i \\ 0 & \text{if } e \notin D_i \end{cases}$$

The re-ordering of a set $D_j$ is done by finding the weight, $w(e,j)$, for every $e \in D_j$ and ordering the elements by decreasing weight creating the re-ordered set $0(D_j)$. That is, in the re-ordered set, $0(D_j)$,

$$w(e_1,j) \geq w(e_2,j) \geq w(e_3,j) \geq \ldots \geq w(e_n,j), \ e_1, \ldots, e_n \in D_j$$

where $e_1$ is the $i^{th}$ element of the set $0(D_j)$. 
The weights of the elements of the sets are found during the intersection of the involved sets. This will also allow an entry to be made in DV if the intersection of entries is found to be null. Let us illustrate this process using the trace of Figure 5.4. Each set will be re-ordered with respect to the rest of the sets, assuming that it isn't yet known if any of the entries are distinct. So the weights for each entry are as follows:

\[
\begin{align*}
D_1: \quad & w(I_3,1) = 2 \text{ since } I_3 \text{ is in } D_2 \text{ and } D_4 \\
& w(I_4,1) = w(I_7,1) = 0 \text{ since these are in none of the } D \text{ sets except } D_1 \\
D_2: \quad & w(I_3,2) = 2, \quad w(I_{11},2) = w(I_8,2) = 0 \\
D_3: \quad & w(I_5,3) = 1 \text{ since } I_5 \in D_5 \text{ also} \\
& w(I_2,3) = 0 \\
D_4: \quad & w(I_3,4) = 2, \quad w(I_1,4) = 1 \text{ since } I_1 \in D_3 \\
D_5: \quad & w(I_1,5) = 1, \quad w(I_5,5) = 1, \quad w(I_9,5) = 0
\end{align*}
\]

After the sets are re-ordered we have

\[
\begin{align*}
0(D_1) &= \{I_3, I_4, I_7\} \\
0(D_2) &= \{I_3, I_{11}, I_8\} \\
0(D_3) &= \{I_5, I_2\} \\
0(D_4) &= \{I_3, I_1\} \\
0(D_5) &= \{I_1, I_5, I_9\}
\end{align*}
\]

Elimination of Weight Zero Instructions

The synthesis process we have been discussing is truly enumerative in that it will not produce a program of size \( B' \) if there exists a program of size \( B < B' \) compatible with the given trace. However, it
doesn't use a formal enumeration which generates programs in the same order every synthesis. This is due to the fact that the enumeration essentially proceeds by construction of partial programs compatible with a given trace. Thus the order of the enumeration is dependent on the given traces. If more than one program of a given size is compatible with the trace, then it is possible that different traces would cause different, equivalent programs to be synthesized. In a formal enumeration the same program would have been arrived at each time.

Often it is possible to determine prior to synthesis that some instructions in the various D_j's are redundant in the sense that their omission will not prevent the synthesis of a minimal size program. Therefore such redundant elements of the D_j's should be eliminated in order to speed the search for the program. Let us make some definitions now in order to describe synthesis results and omissions of instructions and then give a theorem about the effects of such omissions.

**Def.** Programs of the same size produced from different syntheses are said to be **size equivalent** programs.

**Def.** Programs with the same graph structure but having some instruction nodes with different instruction names are said to be **structurally equivalent** programs.

**Def.** The subset of instructions in D_j which are found to be of weight zero will be called Z_j. That is, \( Z_j = \{ I | I \in D_j \text{ and } w(1,j) = 0 \} \).
Thm. Given a decomputed trace, $T'_1$, let $P_{\text{MIN}}$ be the program produced by synthesis using this trace. If for every $j$, we replace $0(D_j)$ in the above trace by

$$0(D_j)' = (0(D_j) - Z_j) \cup \{e_r\}$$

where $e_r$ is any arbitrary element of $Z_j$

then $P'$ will be structurally equivalent to $P$, where $P'$ is the program produced by the synthesis from the altered trace.

Pf. There are two possibilities:

a) no instruction in $Z_j$ can cause a successful synthesis. So if the search ever selects any instruction in $Z_j$, it tries and fails on all in $Z_j$. If $Z_j$ is eliminated then the backtrack passes the entry without the unnecessary work due to the elements of $Z_j$ and the order of enumeration is unaffected.

b) at least one instruction in $Z_j$ will allow a successful synthesis. Actually if one instruction in $Z_j$ is correct, then all are and any one can be used as the representative at the entry $j$.

Structurally, if an instruction of weight zero can be successfully assigned, this means the entry $j$ is the only trace occurrence of its corresponding program instruction. Consider all entries $i$ which could be the same as $j$, i.e., also trace occurrences. By the definition of weight zero, the instruction isn't in any of the other sets, $D_i$. So these entries cannot be assigned that instruction, and must be distinct from $j$. Since $j$ is thus isolated it never interacts, i.e., merges, causes forced moves, etc., with the rest of the trace. The particular instruction name is immaterial, any one can be used and the rest
eliminated. Thus the elimination of these instructions does allow a structurally equivalent program $P'$ to be synthesized.

It is probably true that other instructions can be eliminated on the basis of their patterns of occurrence and program size equivalence could be achieved. However, the amount of work required in such an analysis for the number being eliminated could be prohibitive. Here for the weight zero instructions, we have a side benefit from re-ordering and need do no extra work. Also, the number of weight zero instructions observed at most entries is considerable, often $1/3$ to $1/2$ of the total. Thus this particular heuristic is especially effective and cheap.

**An Example of Unsuccessful Re-ordering**

There are many ways of pruning a search and the methods used should be appropriate to the problem. In this search it was observed that without a successful re-ordering, synthesis becomes extremely difficult. Hence we will discuss problems and limitations of re-ordering and how to overcome such difficulties.

For given ordering of the set $D_j$, let $I_j^{\text{MIN}}$ be the first instruction in $D_j$ for which a $P_j^{\text{MIN}}$ can be successfully synthesized. If $I_j^{\text{MIN}}$ is the $k^{\text{th}}$ instruction in a given $D_j$, then the first $k-1$ must be tried and fail eventually. If these failures occur far enough along the trace, the required backtrack enumeration becomes huge. The purpose of re-ordering is to hopefully place $I_j^{\text{MIN}}$ as the first instruction in $D_j$, but this is not always successful.
An example of a case in which re-ordering will be unsuccessful is given in Figure 5.10. Here we show an I'C trace after decomputation, giving the result cell names, D set, and $T_{ij}$ or the condition noted for each entry.

Entry Index

<table>
<thead>
<tr>
<th>$j$</th>
<th>$a_j$</th>
<th>$D_j$</th>
<th>$T_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>${I_1, I_3}$</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A(I)</td>
<td>${I_4}$</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>${I_3}$</td>
<td>$C_1$</td>
</tr>
<tr>
<td>4</td>
<td>TEMP</td>
<td>${I_5}$</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>${I_2}$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>6</td>
<td>TEMP</td>
<td>${I_5}$</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>${I_1, I_2}$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>8</td>
<td>TEMP</td>
<td>${I_5}$</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>${I_1, I_2}$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>10</td>
<td>TEMP</td>
<td>${I_5}$</td>
<td>$C_3$</td>
</tr>
</tbody>
</table>

Figure 5.10  I'C Trace After Decomputation

Since $X$ is the first result name, in Figure 5.10 the first result set is

$$R_1 = \{1, 3, 5, 7, 9\}$$

The values in DV produced by pre-processing for the entries of $R_1$ are the following:
After the pre-processing it can be seen that the decomposition sets of $R_1$ will appear as

The smallest partition compatible with this decomposition has two blocks and so we have an initial bound $b_1 = 2$ for $r$-set $R_1$. The bound is correct in that a synthesis produces two program occurrences for the entries of $R_1$. 
However, due to the decomposition pattern, when re-ordering is performed the following happens for entries 1 and 7:

a) Since $R_1^1 = \{1, 3, 7, 9\}$ the weighting of the elements of $D_1 = \{I_1, I_3\}$ is done with respect to entries 3, 7, and 9. Since $I_1 \in D_7$ and $D_9$,

$$w(I_1, 1) = 2$$

and since $I_3 \in D_3$ only,

$$w(I_3, 1) = 1.$$ 

Therefore $O(D_1) = \{I_1, I_3\}$.

b) $R_1^7 = \{1, 5, 7, 9\}$ and so for $D_7 = \{I_1, I_2\}$

$$w(I_1, 7) = 2$$

since $I_1 \in D_1$ and $D_9$. Similarly $I_2 \in D_5$ and $D_9$ and then

$$w(I_2, 7) = 2.$$ 

So $O(D_7) = \{I_1, I_2\}$.

The case for entry 9 is the same as for 7. Hence we see that $I_1^{\text{MIN}} = I_3$ is not the first instruction in $O(D_1)$ nor is $I_7^{\text{MIN}} = I_9^{\text{MIN}} = I_2$ placed ahead of $I_1$ in $O(D_7)$ or $O(D_9)$ since it has the same exact weight as $I_1$. So re-ordering for this case is indeed unsuccessful.

**Look-Ahead**

In the above example we saw that the re-ordering wasn't able to produce a correct order in the sets $D_1$, $D_7$, and $D_9$. The look-ahead process is able, for some cases, to avoid the selection of an instruction which has been placed before the correct instruction in a
re-ordered D-set. For the arbitrary steps of the search, at which a
new instruction is about to be selected, a look-ahead in the trace
can detect information which indicates that, for the present state of
the synthesis, the selection will prove to be unsuccessful.

It would be possible to check for a variety of potential problems,
but by observation of many syntheses, an incorrect instruction most
often caused a failure of the search due to a bound $b_k$ being exceeded.
Checking for this is accomplished by tentatively assigning the
instruction being considered at entry $j$ and then re-computing the
decomposition of $R_k, j \in R_k$. So a new bound can be calculated for $R_k$
on the basis of this new decomposition. For example at the entry $j = 1$
in the example above, if $I_1$ is assigned here, this entry must be
distinct from entry 3. The resulting decomposition is illustrated
below.

[Diagram of decomposition with labeled entries and arrows indicating transitions between entries.]

$R^5_1 \rightarrow 5\bullet$

$R^1_1 \rightarrow 1\bullet$

$R^7_1, R^9_1$

$\bullet 3$

$R^3_3$
This new decomposition would imply that at least 3 program instructions are needed to represent the entries in $R_k$, exceeding the initial bound of 2. The next instruction $I_3$ in $D_1$ can be chosen instead without exceeding the bound and the synthesis then proceeds without the need for backtracking.

Examples of the effects of look-ahead and re-ordering will be presented in the chapter describing the experimental results.

We have seen above an example in which the re-ordering of the $D$-sets was not successful. Now we will explain how such a failure of re-ordering occurred and examine if it is possible to guarantee a correct re-ordering under certain conditions.

Consider a given trace or concatenation of traces $T_{I'C}$. Let $P_{MIN}$ be a possible minimal program compatible with $T_{I'C}$ and let $\pi^I$ be the instruction induced partition of $T_{I'C}$ corresponding to $P_{MIN}$. The blocks of this partition $\pi^I$ are then written as

$$\pi_1^I, \pi_2^I, \ldots, \pi_m^I.$$  

The re-ordering of instructions can be unsuccessful if the pre-processing cannot produce enough distinctions between entries belonging to different blocks of the partition $\pi^I$ of the trace. In the previous example of an unsuccessful re-ordering on page 78, the trace's partition $\pi^I$ has the following two blocks:

$$\pi_j^I = \{1, 3\} \quad \text{corresponding to } I_3$$  
$$\pi_k^I = \{5, 7, 9\} \quad \text{corresponding to } I_2$$

Pre-processing couldn't distinguish the entry 1 in the first block from the entries 7 and 9 in the second and because of this the re-ordering was unsuccessful.
Let us discuss another example to illustrate exactly why the failure of re-ordering occurs. Consider one r-set, $R_k$, having two corresponding blocks, $\pi^I_1$, $\pi^I_2$, of the instruction partition $\pi^I$ of a trace. So let

$$R_k = \{2, 3, 4, 5, 6, 7, 9\}$$

$$\pi^I_1 = \{2, 4, 5\} \quad \text{corresponding to } I_1$$

$$\pi^I_2 = \{3, 6, 7, 9\} \quad \text{corresponding to } I_2$$

$$\pi^I_1: \begin{bmatrix}
D_2 = \{\ldots, I_1, \ldots\} \\
D_4 = \{\ldots, I_1, \ldots\} \\
D_5 = \{\ldots, I_1, \ldots\}
\end{bmatrix} \quad \pi^I_2: \begin{bmatrix}
D_3 = \{\ldots, I_2, \ldots\} \\
D_6 = \{\ldots, I_2, \ldots\} \\
D_7 = \{\ldots, I_2, \ldots\} \\
D_9 = \{\ldots, I_2, \ldots\}
\end{bmatrix}$$

Let pre-processing be unable to distinguish the entries 6, 7, 9 of $\pi^I_2$ from entry 5 of $\pi^I_1$ and so $R_k^S = \{2, 4, 5, 6, 7, 9\}$. Next assume $I_2 \in D_5$ and that $I_1 \not\in D_6, D_7, \text{ or } D_9$. We can now represent the situation as

Because entry 5 has enough overlap with a block whose representative instruction, $I_2$, happens to be in the entry's D-set, re-ordering fails to place $I_1$ correctly. That is, $w(I_1, 5) = 2$, but $w(I_2, 5) = 3$ and so $I_2$ is placed before $I_1$ in $O(D_5)$. 
Now let us investigate what can be said if the pre-processing of a trace $T_{I',C}$ succeeds in creating an instruction partitioning $\pi^I$ of the trace.

**Def.** For any block $\pi^I_k$ of a partition $\pi^I$ of an I'C trace, $V_k$ is the set of instructions any one of which could be the representative trace occurrence corresponding to $\pi^I_k$. That is, $V_k$ is the set of instructions which occur in every D-set of the entries of $\pi^I_k$,

$$V_k = \{ I | I \in D_j, \forall j \in \pi^I_k \}.$$

Any instruction in $V_k$ can be chosen and assigned to each entry and the resultant programs are structurally equivalent. (Note $V_k \subset D_j$ for every $j \in \pi^I_k$).

It is clear that if the pre-processing can correctly produce a partition $\pi^I$ of a $T_{I',C}$, the instructions in the set $V_k$ corresponding to any entry $j \in \pi^I_k$ will be placed ahead of any other instructions in $O(D_j)$. Since pre-processing has established the block $\pi^I_k$, we know $j$ is distinct from any entries not in $\pi^I_k$. Hence the set of entries, $Q_j$, for which the re-order weight of any $I \in D_j$ is defined is in fact $\pi^I_k - \{ j \}$. Now by the definition of $V_k$, if $I \in V_k$, $I$ is in the D-set of every entry in $\pi^I_k$. So if the number of entries in $\pi^I_k$ is $m$, then $w(I,j) = m-1$ for any $I \in V_k$. If $I' \in D_j$ but $I' \notin V_k$ then $I'$ cannot be in all the D-sets of entries in $Q_j$ and so $w(I', j) < m-1$ for any $I' \notin V_k$.

The instructions in $V_k$ are all correct representatives of the block $\pi^I_k$ of the partition established for this trace. So the first one in the $O(D_j)$ sets will be chosen and used in the program.
synthesized. Thus the re-ordering will be correct for all entries of this trace.

Consider the instructions in $V_k$ in the above discussion. They all produce at entry $j$ the memory change for the given $T_{I'c}$, but because of the data dependence of the decompuation, all of them may not be able to explain memory changes in all possible I'C traces over the domain of the computation. For example consider the following environment.

Here ONE is a constant always needed with a value of 1, but LIST(1) just happens to have a 1 as its contents. Now in a computation in which

$$AVG \leftarrow TOT$$

is performed, then

$$AVG \leftarrow TOT \times LIST(1)$$
$$AVG \leftarrow TOT/\text{LIST}(1)$$
$$AVG \leftarrow TOT \times \text{ONE}$$
$$AVG \leftarrow TOT/\text{ONE}$$
$$AVG \leftarrow TOT$$

are all equivalent if LIST(1) = 1. Then they will all have the same weight by re-ordering if pre-processing discovers $I^1$ and so any one of these could be selected for the final program synthesized for this computation. However if an execution of the same algorithm with other
data doesn't have the contents of LIST(1) equal to 1 at the step corresponding to the instruction AVG + TOT, the first and second alternate instructions above are not equivalent to the others for this case. It is possible then that an incorrect instruction such as AVG + TOT * LIST(1) could be used to synthesize a program if only one computation trace was known.

There will be a subset of instructions of $V_k$ which are equivalent explanations of the given $\pi_k^I$ over the entire domain of the computation. This set will be called the set of domain equivalent instructions of $V_k$, denoted

$$V_k^{de} \subset V_k$$

So for traces of computations in some domains the instructions in $V_k - V_k^{de}$ will not appear in the corresponding D-sets after a decompuation. Hence if these traces are used and a partition $\pi^I$ of the combined traces is achieved, then the re-ordering will push the instructions in $V_k^{de}$ forward in the $O(D_j)$'s.

This means that in a sense the re-ordering can be correct "in the limit," or after enough traces have been seen by the synthesis. Loosely, let there be an enumeration of domains for a given computation, and some orderly presentation of the various traces. Then at some point in the presentation enough traces will have seen that, if a partition $\pi^I$ is produced, the instructions in $V_k^{de}$ will be in the correct places and so chosen for any synthesis.

Now let us restate these conclusions in a summary. We will always assume that a partition $\pi^I$ of any trace has been found.

(1) If, at entry $j$, there is an instruction $I \in D_j$ such that
\[ w(I,j) > w(I',j) \] for every \( I' \in D_j \), \( I' \neq I \) then \( I \) is a correct representative instruction for entry \( j \). This would mean that the rest of the instructions in \( D_j \) could be eliminated without affecting the program synthesized.

(2) If, at entry \( j \), there are several instructions with a maximum weight, we want to examine this subset of \( D_j \), that is

\[ \{ I \mid I \in D_j \text{ and } w(I,j) \geq w(I',j) \text{ for every } I' \in D_j \} \]

If it were possible to know semantically that these are all equivalent instructions, then any one of them could be chosen as the representative instruction and the rest eliminated as in (1). It might be possible to resolve a question of such an equivalence at certain points in an interactive dialogue with the user.

(3) Further, assume in (1) or (2) that we know exactly which entries were in the same block of a partition. Then the first time any entry \( j \) in a block behaves as in (1) or can be isolated as in (2), all entries in the block can be assigned the instruction found for entry \( j \).

If case (3) would actually occur for all blocks of the partition \( \pi' \), then we would have convergence of an \( I'C \) trace to an IC trace. Then there is no need for a search for the trace occurrences of \( T_{I'C} \).

So if the assumptions for case (3) are met and convergence occurs we have resolved the problem of which examples are required in order to identify the instruction names of a program. Note that the sufficiency of examples for the program occurrences (the program structure) is not being addressed here. The comments above just reveal how to insure that the information lost by taking the weaker \( I'C \) traces rather than IC traces can be recovered.
The appropriate question now should be are there any classes of computations or properties of computations that will allow the existence and form of the partition $\pi^I$ to be known? A fairly natural class for which this is possible will be called the class of pure computations. Halstead in his work [15] on software physics, gives a classification of so-called impurities in programs. One of these impurities involved the dual usage of operands in programs, that is, the use of a variable name for several different purposes in the program.

In a similar fashion, we define a pure computation in our computational environment.

Def. A pure computation in $E$ allows each memory cell or result name, $a_i$, to be used in only one instruction occurring in the algorithm. That is, if SUM is a variable name, then only one instruction in an algorithm, such as $SUM \leftarrow SUM + A(I)$ puts a result in SUM. There may be several occurrences of the instruction named $SUM \leftarrow SUM + A(I)$ in the algorithm but there cannot be another instruction such as $SUM \leftarrow SUM/N$ affecting the variable SUM. If a computation is known to be pure, then the r-sets of its trace are known to compose a $\pi^I$ partition. This meets all the requirements of case (3) above and so it is possible for many pure computations that the I'C trace could be simplified to an IC trace.
A system for the inference of programs from I'C traces has been developed based on the techniques discussed previously. The implementation was done on a PDP-10 in Algol. The system is diagrammed in Figure 6.1.

![Diagram of Inference System](image)

**Figure 6.1 Inference System**

Experiments were performed using a number of different example calculations. Some of the example calculations were done by hand and then input as an I'C trace, others by a programmed simulation of the computation. Also utilized was the CRT display system described in Chapters 1 and 2.

Let us consider a simple calculation, finding the average of a list of integers, and examine the associated computational environment and I'C trace. If the calculation were done on the CRT display, the initial state of the environment would appear as in Figure 6.2.
Figure 6.2 Data Structures for the Calculation of the Average of a List.

The system's light pen would then be used to move numbers from the bottom of the screen (not shown) into the cells representing the results of the on-going calculation. Note that the literal "ONE" actually used in the calculation must appear in the environment. This calculation started with I and AVG initialized to zero as might be done in a procedure call. N is also a parameter of the calculation representing the size of the array in order that a condition can be noted when the end of the array is reached. This would not be necessary if automatic indexing as discussed in Chapter 4 were implemented.

The environment's description and trace input to the decomputation are shown in Figure 6.3. The environment's specification also contains the information that I is an index for the array A. The blanks in the
### Environment Specification

<table>
<thead>
<tr>
<th>Cell</th>
<th>Possible Cell Names</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n₁</td>
<td>A(1), A(I)</td>
<td>7</td>
</tr>
<tr>
<td>n₂</td>
<td>A(2), A(I)</td>
<td>13</td>
</tr>
<tr>
<td>n₃</td>
<td>A(3), A(I)</td>
<td>-4</td>
</tr>
<tr>
<td>n₄</td>
<td>A(4), A(I)</td>
<td>11</td>
</tr>
<tr>
<td>n₅</td>
<td>A(5), A(I)</td>
<td>3</td>
</tr>
<tr>
<td>n₆</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td>n₇</td>
<td>N</td>
<td>5</td>
</tr>
<tr>
<td>n₈</td>
<td>ONE</td>
<td>1</td>
</tr>
<tr>
<td>n₉</td>
<td>AVG</td>
<td>0</td>
</tr>
</tbody>
</table>

### Trace Specification

<table>
<thead>
<tr>
<th>T₁₁</th>
<th>T₁₂</th>
<th>T₁₃</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Changed</td>
<td>New Value</td>
<td>Value</td>
<td>Condition</td>
</tr>
<tr>
<td>n₁</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₄</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₁</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₄</td>
<td>20</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₁</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₄</td>
<td>16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₄</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₁</td>
<td>27</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₄</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₄</td>
<td>30</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n₁</td>
<td>6</td>
<td>NOTE: I &gt; N</td>
<td></td>
</tr>
<tr>
<td>n₄</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 6.3** Initial Environment and Trace Input
condition column of the trace specification simply mean that no 
condition was specifically noted at these steps.

The output of the synthesis and its corresponding flowchart are 
in Figures 6.4 and 6.5. The actual decomputation and synthesis times 
for this example were 7.2 seconds and 1.6 seconds, respectively.

Notice in Figures 6.4, 6.5 that the last instruction in the pro-
gram, AVG + I, is incorrect. In this particular calculation the ave-
rage of the list was 6 and when the average was to be found, the con-
tents of the variable I was 6. Thus the "incorrect" instruction does 
indeed produce the correct result for this particular computation, and 
so another example is needed. This problem can easily occur when an 
instruction is executed only once during an example. Looping of the 
program through an instruction more than once or seeing another execu-
tion of it in another example will help the system to converge to a 
correct instruction. If we add an additional trace to the one which 
was insufficient in the example above, the synthesis does indeed con-
verge to AVG + AVG/N as the last instruction of the program. The ad-
ditional trace was of a simple calculation of the average of a two ele-
ment list containing the numbers 4 and 6.
\[(\text{START}, -, I + I + 1)\]
\[(I + I + 1, -, \text{AVG} + \text{AVG} + A(I))\]
\[(\text{AVG} + \text{AVG} + A(I), -, I + I + 1)\]
\[(I + I + 1, I > N, \text{AVG} + I)\]
\[(\text{AVG} + I, -, \text{HALT})\]

Figure 6.4 Transitions Produced by Synthesis

Figure 6.5 Flowchart Corresponding to Transitions of Figure 6.4
Next let us examine several examples each of factorial and Fibonacci number calculations and the synthesis of their associated programs. The results are given in Tables 6.1 and 6.2. In these tables and in others in this chapter the column headed "number of instructions" gives the total number of instructions produced by the deconstruction of the entries of that trace. As indicated in Table 6.1, calculations of 1! and 2! were not sufficient to produce a correct program. The execution of 1! requires no loops and so as seen in Figure 6.6, none are produced. This simply affirms the fact that the trace(s) must go through all possible edges and vertices of a program at least once.

In Figure 6.7 it can be seen that the program synthesized from the trace of 2! was also unsuccessful. The correct loop structure is created but the last instructions are incorrect. Figure 6.8 shows the correct program produced from examples of computing 3! and the larger factorials.

The first two examples of Fibonacci calculations shown in Table 6.2 give rise to incorrect programs. In the Table 6.2, these are labeled \( F_3 \) and \( F_4 \) denoting the 3rd and 4th Fibonacci numbers 3, and 5 respectively. Figures 6.9 and 6.10 show the programs synthesized from the traces of the computations of \( F_3 \) and \( F_4 \). Note in these figures that the loop structures of the programs are seen to be incorrect when compared to a correct program shown in Figure 6.11. These examples simply required fewer instructions in their minimal compatible programs than the correct one in Figure 6.11. More than one trace can be used in a given synthesis by a concatenation or head-to-tail joining yielding a combined trace. Now even when the traces of the first
<table>
<thead>
<tr>
<th>Example Trace</th>
<th>Trace Length</th>
<th>Number of Instructions</th>
<th>Decomputation Time (Sec)</th>
<th>Synthesis Time (Sec)</th>
<th>Success?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1!</td>
<td>4</td>
<td>58</td>
<td>.8</td>
<td>.4</td>
<td>N</td>
</tr>
<tr>
<td>2!</td>
<td>6</td>
<td>59</td>
<td>1.2</td>
<td>.3</td>
<td>N</td>
</tr>
<tr>
<td>3!</td>
<td>8</td>
<td>55</td>
<td>1.3</td>
<td>.4</td>
<td>Y</td>
</tr>
<tr>
<td>4!</td>
<td>10</td>
<td>54</td>
<td>1.5</td>
<td>.6</td>
<td>Y</td>
</tr>
<tr>
<td>5!</td>
<td>12</td>
<td>56</td>
<td>1.8</td>
<td>.7</td>
<td>Y</td>
</tr>
<tr>
<td>6!</td>
<td>14</td>
<td>88</td>
<td>2.1</td>
<td>1.3</td>
<td>Y</td>
</tr>
<tr>
<td>7!</td>
<td>16</td>
<td>91</td>
<td>2.3</td>
<td>1.4</td>
<td>Y</td>
</tr>
</tbody>
</table>
Figure 6.6 Incorrect Factorial Program

Figure 6.7 Incorrect Factorial Program

Figure 6.8 Correct Factorial Program
<table>
<thead>
<tr>
<th>Example Trace</th>
<th>Trace Length</th>
<th>Number of Instructions</th>
<th>Decomputation Time (Sec)</th>
<th>Synthesis Time (Sec)</th>
<th>Success?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_3$</td>
<td>9</td>
<td>108</td>
<td>2.4</td>
<td>0.4</td>
<td>N</td>
</tr>
<tr>
<td>$F_4$</td>
<td>13</td>
<td>128</td>
<td>3.1</td>
<td>0.7</td>
<td>N</td>
</tr>
<tr>
<td>$F_3 \cdot F_4$</td>
<td>23</td>
<td>237</td>
<td>5.5</td>
<td>3.1</td>
<td>N</td>
</tr>
<tr>
<td>$F_4 \cdot F_3$</td>
<td>23</td>
<td>237</td>
<td>5.5</td>
<td>2.6</td>
<td>N</td>
</tr>
<tr>
<td>$F_5$</td>
<td>17</td>
<td>148</td>
<td>3.8</td>
<td>1.2</td>
<td>Y</td>
</tr>
<tr>
<td>$F_6$</td>
<td>21</td>
<td>162</td>
<td>4.4</td>
<td>1.6</td>
<td>Y</td>
</tr>
<tr>
<td>$F_7$</td>
<td>25</td>
<td>168</td>
<td>5.3</td>
<td>2.1</td>
<td>Y</td>
</tr>
<tr>
<td>$F_8$</td>
<td>29</td>
<td>197</td>
<td>5.6</td>
<td>2.8</td>
<td>Y</td>
</tr>
<tr>
<td>$F_9$</td>
<td>33</td>
<td>191</td>
<td>6.3</td>
<td>3.3</td>
<td>Y</td>
</tr>
</tbody>
</table>
Figure 6.9 Incorrect Fibonacci Program

Figure 6.10 Incorrect Fibonacci Program
Figure 6.11 Correct Fibonacci Program
examples were combined, denoted \( F_3 \times F_4 \) and \( F_4 \times F_3 \) in Table 6.2, basically the same incorrect program as in Figure 6.10 was produced.

These results are really not surprising as examples of calculations of 1! and 2! or the third and fourth Fibonacci numbers, 3 and 5, aren't what one might consider a priori to be representative examples. What is indicated is that for rather reasonable examples, there is sufficient information to produce a correct program from only one such example.

The times that appear in the tables simply indicate that longer traces take longer to decompute. Also being longer, they produce more instructions and so take longer times to synthesize. The synthesis time is in particular dependent on the number of instructions since the intersections of the D-sets required in the pre-processing and re-ordering steps is a function of the square of the number of instructions in the sets.
Table 6.3 shows the synthesis times required for the traces of the Fibonacci calculations when the inference program doesn't use reordering and look-ahead.

**TABLE 6.3**

**EFFECTS OF REORDERING AND LOOK-AHEAD ON THE SYNTHESIS TIMES FOR FIBONACCI PROGRAMS**

<table>
<thead>
<tr>
<th>Example Traces</th>
<th>Trace Reordered</th>
<th>Trace Not Reordered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Look-Ahead</td>
<td>No Look-Ahead</td>
</tr>
<tr>
<td>$F_3$</td>
<td>.4</td>
<td>.6</td>
</tr>
<tr>
<td>$F_4$</td>
<td>.7</td>
<td>6.4</td>
</tr>
<tr>
<td>$F_3*F_4$</td>
<td>3.1/2313.5</td>
<td>3.4</td>
</tr>
<tr>
<td>$F_4*F_3$</td>
<td>2.6/2.5</td>
<td>3.0</td>
</tr>
<tr>
<td>$F_5$</td>
<td>1.2</td>
<td>81.2</td>
</tr>
<tr>
<td>$F_6$</td>
<td>1.6</td>
<td>87.2</td>
</tr>
<tr>
<td>$F_7$</td>
<td>2.1</td>
<td>76.1</td>
</tr>
<tr>
<td>$F_8$</td>
<td>2.8</td>
<td>92.4</td>
</tr>
<tr>
<td>$F_9$</td>
<td>3.3</td>
<td>97.2</td>
</tr>
</tbody>
</table>

With the trace reordered, the look-ahead synthesis times are given in the 2nd column. Except for the case of the joined traces $F_3*F_4$, $F_4*F_3$, times with and without look-ahead are approximately the same and so only one is given. When reordering is done, it is usually effective enough that look-ahead is not needed to improve the search. For simple
examples then, it can be the case that the time for the look-ahead algorithm is slightly greater since it is doing unneeded look-ahead. However, if as happened in $F_3 \ast F_4$, reordering is not completely successful, especially for the first few trace entries, the look-ahead process becomes critical. The reordered trace without look-ahead takes 15 minutes less than the non-reordered trace without look-ahead for $F_3 \ast F_4$. The main reason for this is that when reordering is done, all but one of the weight zero instructions are eliminated at each entry, considerably lessening the number of possible instructions in each entry which must be considered during synthesis.

For the $F_3 \ast F_4$ trace in the very first entry an incorrect instruction was placed first in $D_1$ by reordering and a very large number of other possibilities are tried enumeratively before the second, correct, instruction in $D_1$ is finally chosen. However look-ahead can in this case see that the first instruction in $D_1$ will cause an initial bound to be exceeded and so immediately selects the second, correct, instruction. This allows the program to be produced in 1/8000 of the time required without a look-ahead.

For $F_4 \ast F_3$, the problem occurs halfway down (at the start of $F_3$) the trace, is immediately seen as exceeding the initial bounds and so the same difficulty isn't encountered.

Other than these exceptional cases, for $F_5$ to $F_9$, where a correct program is produced, reordering allows synthesis in $\approx 1/40$ of the time otherwise required.

Thus it is clear that effort spent reordering is well worth the cost. Look-ahead doesn't seem to be needed uniformly for all examples, but must be included in general since it cannot be predicted when the reordering will fail.
A number of other programs of various types and complexities were selected to illustrate synthesis from memory snapshots. The summary of these experiments is given in Table 6.4. The synthesized programs and descriptions of each individual experiment are given in appendix C.

In most of these cases the synthesis could be done with a simple example, often the first one chosen. Where more than one trace was required, this was usually necessary because a number of different cases were needed to represent all possibilities. For example, in a search it is necessary to have an example in which the element searched for is not found and an example where it is found.

More difficulty was encountered with the algorithm FIND which required a bit of analysis to select sufficient traces. The problem with FIND was not in finding the examples needed to converge on correct instructions, but in examples required to produce the correct graph structure. The problem is due to the two index increment instructions,

\[ I + I + 1 \]
\[ J + J - 1 \]

each of which have multiple program occurrences. It is necessary that examples are chosen which allow these to be distinguished. This problem is discussed further in the appendix entry for FIND.

In general no attempt was made to insure that the examples for any algorithms represented the best behavior of the system, but where several cases were done the best was chosen for the table.

The decompuation times shown clearly reflect the greater complexity involved in a larger number of cells and the use of arrays in the environment.
The synthesis times shown generally increase with the total number of instructions in all D-sets in the decomputed trace. A basic cause of the increase is the dependence of the necessary D-set intersections on the number of instructions in the set.
TABLE 6.4
SUMMARY OF PROGRAMS SYNTHESIZED

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Traces</th>
<th>Total Trace Length</th>
<th>Total Number of Instructions</th>
<th>Decomputation Time (Sec)</th>
<th>Synthesis Time (Sec)</th>
<th>Program Length</th>
<th>Number of Cells in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>1</td>
<td>8</td>
<td>55</td>
<td>1.3</td>
<td>.4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Fibonacci Numbers</td>
<td>1</td>
<td>17</td>
<td>148</td>
<td>3.8</td>
<td>1.2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Sequence Generation</td>
<td>1</td>
<td>23</td>
<td>177</td>
<td>3.8</td>
<td>2.7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Variance Calculation</td>
<td>1</td>
<td>23</td>
<td>199</td>
<td>13.5</td>
<td>1.9</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>Variance Calculation</td>
<td>2</td>
<td>33</td>
<td>504</td>
<td>27.3</td>
<td>5.0</td>
<td>10</td>
<td>8-9</td>
</tr>
<tr>
<td>Binary Search</td>
<td>4</td>
<td>42</td>
<td>623</td>
<td>27.4</td>
<td>14.7</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Fibonacci Search</td>
<td>6</td>
<td>34</td>
<td>311</td>
<td>13.9</td>
<td>4.7</td>
<td>7</td>
<td>7-12</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>1</td>
<td>29</td>
<td>600</td>
<td>18.4</td>
<td>3.7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>Sort</td>
<td>1</td>
<td>25</td>
<td>438</td>
<td>13.1</td>
<td>4.0</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Newton’s Method Nth Root</td>
<td>1</td>
<td>35</td>
<td>262</td>
<td>7.6</td>
<td>6.1</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>1</td>
<td>49</td>
<td>2217</td>
<td>36.7</td>
<td>118.5</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Find</td>
<td>6</td>
<td>121</td>
<td>2300</td>
<td>84.1</td>
<td>90.6</td>
<td>13</td>
<td>11-15</td>
</tr>
</tbody>
</table>
CHAPTER 7. CONCLUDING REMARKS

The results discussed in the previous chapter seem to indicate that the synthesis of programs from I'C traces is feasible for the moderate size problems attempted. There is no apparent reason to expect that these results would not extend to all examples of similar complexity. In more complex problems, however, it becomes tedious to carry out the computation and difficult to perform the synthesis. The synthesis of large programs can be dealt with by decomposing the computation into parts. Thus a set of subroutines can be synthesized individually and then assembled to build a larger program.

Consider now that it may not be necessary to work with a purely I'C trace. Review the verbal description of a computation as given on page 2 in the introduction. In this example, especially at its beginning, specific instructions are sometimes used rather than just asserting the value changes of the variables. This appears to be a fairly natural way of expressing this example. It seems that when some particular instances are given initially, it is then more natural later in a computation to show just the memory changes since a specific instance has been given.

Now when working in the computational environment at the display, the first few steps of a calculation can be somewhat tentative and explicit instructions might be used for the user’s own reassurance.
Later with the particular instructions more firmly in mind, it could be easier to use just memory changes where desired.

As the calculation proceeds, an individual might want the system to function as a calculator for him. Let's say the next step of a calculation requires a division of 117 by 3. He might not want to do this operation in his head as done for the previous memory changes and so the explicit instruction is entered with the light pen. Then the result, 39, is produced for him.

The idea to be emphasized is that the user of the system should, as far as possible, be allowed to do what he wants during a computation. He should be able to carry it out in the manner most comfortable for him, utilizing specific instructions when he desires, or giving the other steps as direct memory changes if that is easier.

The above discussion implies that a mixed IC-I'C trace would be seen more often in the system. Such a trace is, of course, easier to deal with than a strictly I'C trace. No decompuation needs to be done on steps where a particular instruction is used. For the synthesis process this means that the D-set of that entry contains only one instruction. This helps by reducing the intersection costs and reducing the entire space it might be necessary to search. With just one instruction in a D-set for a given entry, any entries distinct from it are guaranteed to be distinguished when they are intersected. This could not be deduced if more than one instruction existed in the D-set. So this guarantee yields a better decomposition of the corresponding r-set. Then an improved initial bound could possibly be found during pre-processing and also a better re-ordering might result.
Further Work

A mixed trace was not actually used in any of the experiments performed since at present the mode of operation of the display system admits only strictly IC or I'C traces. The implementation of the capacity for mixed traces would allow a study to be made of the naturalness of specifying example computations in this way and provide valid mixed traces for deconstruction and synthesis studies. An additional feature for the user's ease would be the implementation of some of the automatic indexing techniques discussed in Chapter 4.

Extending deconstruction to higher level instructions, e.g.,

\[ A + X \times (Y + Z) \]

would, if feasible, allow even greater user flexibility. Also if deconstruction were possible in domains other than the integers, then other types of computations such as string manipulations would be possible.

With capabilities such as these it would then be very desirable to perform competitive testing in several user classes with other means of programming. Also of interest is the ease of teaching beginning programmers to specify algorithms in the domain of examples. There should be a number of novel techniques to be developed in such an approach.

Another area which deserves further investigation is the question of the sufficient example computations in each case for the inference of a correct program. Certainly it is first necessary to answer the question of how to specify the needed example traces if we already know the form of the actual program. Let us consider this initially for IC traces.
If one views the flowcharts we have used as Moore machines then the results about distinguishing experiments (traces) such as found in Moore [21] are applicable here. An immediate result is that it is necessary but not sufficient in general to require that traces cover every edge of the graph of the program which is to be synthesized. If there were no multiple occurrences of the same instruction in the program, such a trace would be sufficient. However it is precisely the need to differentiate among such multiple occurrences which requires a distinguishing sequence of entries in the trace for each occurrence. In Moore's work general bounds on the length of such sequences are given, but better results should be obtainable for machines representing various interesting classes of programs.

Since specific instructions do not appear in an I'C trace an example computation which was sufficient when represented by the stronger IC trace may not be sufficient when represented by one of the weaker kinds of traces, in particular an I'C trace. This is illustrated in the synthesized program shown in Figure 6.5. The program has the correct flow-of-control structure but since it was produced from an I'C trace, the name of the last instruction in the program happens to be incorrect. The problem is that an instruction which is an explanation of a memory change only by chance cannot be detected unless the real reason for the memory change occurs again in the example. If it does not then another example would be required simply to provide a case in which the instruction fails to yield the correct memory change.

As discussed in Chapter 5, pure calculations would help to avoid such problems and it is likely that similar techniques for the
specification of example computations could be developed to overcome these sorts of ambiguities. In an interactive system a possible way of resolving at least some of these ambiguities would be by developing an effective methodology of directly querying the user.

So, in conclusion, this work has established something like a point of reference for the difficulty of inferring programs from traces. The traces used here were strictly I'C traces, and as indicated above this was probably a weaker specification than would be normally given by a user. Hence it should be computationally feasible to synthesize programs whose complexity is even greater than those appearing in Chapter 6. From the user point of view, with the improvements suggested in the computational environment, the performance of the necessary computations should be practicable and natural.

While this approach might not represent the entire answer to automating programmer functions, it is one facet which should be integrated into any larger system in order to allow specification by example calculations where it is desirable and natural.
APPENDIX A. ENUMERATION OF PROGRAMS

For a given computation a trace generally represents much more information than a specification such as an input-output pair for the computation. Since there is a definite bound imposed by the length of the trace and the primitive operations of E in this work are sufficiently simple that their execution is not an issue, the compatibility of a program P with a trace T is decidable.

Thm. For any program P in E with low-level primitives and any trace T with an initial state $S_0$ it is decidable if

$$T = P(S_0)$$

where $P(S_0)$ indicates the trace of the program P when executed starting with the initial memory state $S_0$.

Pf. Let the length of the sequence of tuples, T, be n. Start P on $S_0$. Then

a) $P(S_0)$ is of length $\leq n$. It can then be compared to T to see if it is the same trace.

b) $P(S_0)$ is of length $> n$, or doesn't halt. As soon as the number of tuples in the trace is seen to exceed n the answer can be given that the trace isn't compatible with the trace T. Q.E.D.

We are primarily concerned with the production or inference of programs from traces. So we will be considering the class of programs, C, possible in an environment E to which the theorem above applies.
Such a class of programs can be effectively enumerated as discussed in Feldman and Shields [11].

Let us say that we have a set of traces for some program P in the class C. What is desired is an algorithm A which produces the smallest program for the traces seen so far. This algorithm should produce the best program for all traces so far (compatible with them), and still have good behavior in the limit. To discuss a smallest program, we need to have a measure of program size for a program P, denoted \( L(P) \). The size measure will be required to admit an effective enumeration of programs \( P_1, P_2, \ldots \) such that:

1. the number of programs of a given size \( L \) is finite and,
2. it is always possible to decide if all programs of a given size have been seen.

In the class C we are concerned with, some possible length measures are the number of basic blocks, number of instruction vertices, total number of vertices, etc. To satisfy the two conditions we have set down, it is necessary to exclude the use of literals in instructions formed in \( E \). Here if literals were allowed then instructions of the form

\[
X \leftarrow X \text{ op} \ \text{Literal}
\]

would admit an infinite number of instructions and then an infinite number of programs of a given size violating the conditions set for the size measure.

With these conditions it is possible to order programs in increasing size.
Def. An Occam's enumeration of the class $C$ is an effective enumeration of $C$ such that for any two programs $P_i, P_j$ of the enumeration

$$L(P_i) \preceq L(P_j), \quad i < j$$

Thm. Given the class $C$ of programs over $E$ and an Occam's enumeration of them relative to $L$, then for any set of traces $\Omega$ of some program $P'$, there is an algorithm $A$ for which

$$L(A(\Omega)) \preceq L(P)$$

for all $P$ in $C$ compatible with $\Omega$, where $A(\Omega)$ represents the program generated by $A$ for the set of traces, $\Omega$.

Pf. The algorithm $A$ generates the programs by the given Occam's enumeration,

$$P_1, P_2, P_3, \ldots$$

As a program $P_i$ is generated, it is tested,

$$T^j = ? \quad T(P_i(S^j_o)) \quad \forall T^j \in \Omega$$

where $S^j_o$ gives the initial memory contents corresponding to the trace $T^j$.

By the previous theorem the above process of testing is decidable, and therefore the first program $P_k$ compatible with $\Omega$ will be the program guessed by $A$, i.e.,

$$P_k = A(\Omega)$$

Clearly, since this is the first program compatible with $\Omega$ in the Occam's enumeration

$$L(P_k) = L(A(\Omega)) \preceq L(P) \quad \text{for all } P \in C \text{ compatible with } \Omega$$

Q.E.D.
APPENDIX B. AN ALTERNATE APPROACH TO SYNTHESIS FROM I'C TRACES

Rather than combining the necessary synthesis choices of 1) a trace occurrence of an instruction and 2) a program occurrence of the selected instruction for each entry of an I'C trace as described in Chapter 5, here we will consider making these choices separately. That is, criteria are set up for the choice of instructions from the D_j's at each entry j. When a choice has then been made for every entry of the trace, an IC trace has effectively been created and the basic synthesis algorithm for IC traces can be used to produce a program. The process of creating an IC trace and then synthesizing a program can be iterated until one finds that a P_MIN has been produced.

To select instructions for trace entries, we will make some new definitions. Let T_I'C be the I'C trace given and consider the r-sets for this trace. Let R_k be any r-set for T_I'C and let

\[ D_{i_1}, D_{i_2}, \ldots D_{i_j}, \ldots \quad \forall j \in R_k \]

be the D-sets of instructions associated with the trace entries in R_k. We will want to assign representative instructions from each of these D-sets to represent the trace occurrence for the entries in R_k.

Def. Let \(|R_k| = n_k\). An R-assignment, P_k, for R_k is a list of n_k instructions, with an instruction being chosen from each D_j, \(\forall j \in R_k\).

Def. The instruction size of an R-assignment is the number of different instructions in the list, denoted \(||P_k||\). Thus we can give
an enumeration of the R-assignments for a given $R_k$ in order of size

$$p^1_k, p^2_k, \ldots$$

where $$||p^i_k|| \leq ||p^j_k|| \quad i = 1, \ldots, j \in R_k \quad |D_j|$$

As an example, consider the r-set and associated D-sets for the second result name, $m_2$, found in the trace of Figure 4.2:

$$R_2 = \{4, 8, 12\}$$

$$D_4 = \{\text{move} \ (m_4)\}$$

$$D_8 = \{\text{move} \ (m_8)\}$$

$$D_{12} = \{\text{move} \ (m_4), m_4/m_3\}$$

Then let

$$p^1_k = \{\text{move} \ (m_3), \text{move} (m_4), \text{move} (m_3)\}$$

$$p^2_k = \{\text{move} \ (m_3), \text{move} (m_3), m_4/m_3\}$$

with their sizes being

$$1 = ||p^1_k|| < ||p^2_k|| = 2$$

Def. A trace assignment $T^j_{IC}$ of a memory trace, $T_{IC}$, is produced by selecting an R-assignment, $P^j_k$, for every r-set, $R_k$, associated with the trace.

A trace assignment then results in an instruction being assigned to every entry of the trace yielding an IC trace, $T_{IC}$.

Def. The instruction size of an IC trace, $||T_{IC}||$, is the number of different instructions in the trace. So the trace assignments can be enumerated in order of size

$$T^1_{IC}, T^2_{IC}, \ldots$$
where $|T_{IC}^i| \leq |T_{IC}^i| \bigwedge_{k \in R_k} |D_j| \bigwedge_{i = 1, \ldots, \infty} |D_j|$

**Thm.** Given an I'C trace $T_{I'C}$, there is an algorithm which synthesizes a program, $P_{MIN}$, of smallest total size compatible with trace $T_{I'C}$.

**Pf:** Let:

- $s_P$ = total size (total number of instructions) in program $P$,
- $\alpha_P$ = the instruction size (number of different instructions) of $P$,
- $\beta_P$ = the number of multiple occurrences of instructions in $P$, i.e., $\beta = s - \alpha$.

Let $A$ be a synthesis algorithm for IC traces. Since an IC trace is just a special case of an I'C trace in which each D-set has only one instruction, a simple variant of the I'C algorithm can be used for synthesis. The program $A$ synthesizes from an IC trace, $T_{IC}$, will be denoted as

$$P = A(T_{IC})$$

where $P$ is a minimal program compatible with $T_{IC}$. Since $A$ must produce the minimal program compatible with $T_{IC}$, it is true that

$$\alpha_P = |T_{IC}|$$

and in general that

$$s_P > |T_{IC}|.$$  

From here on we will use $\alpha$ to represent the instruction size of a trace $T_{IC}$ or its minimal compatible program.

The synthesis algorithm for an I'C trace $T_{I'C}$:

1. Decompute $T_{I'C}$
2. As described above, start to generate the possible IC traces for $T_{I'C}$ in order of instruction size,

$$T_{IC}^1, T_{IC}^2, T_{IC}^3, \ldots$$
and their corresponding instructions sizes are

\[ a_1 \leq a_2 \leq a_3 \ldots \]

where \( a_1 \) is the instruction size of \( T^1_{IC} \).

3. When each \( T^j_{IC} \) is generated, use A to produce its corresponding program written as

\[ P_j = A(T^j_{IC}) \]

Steps 1 and 3 are algorithms and 2 is certainly finite. Bounds can be found for step 2 as discussed next. This discussion is also illustrated in Figure B.1.

Apply A to \( T^1_{IC} \) yielding

\[ P_1 = A(T^1_{IC}) \]

Then the size of \( P_1 \) is

\[ s_1 = a_1 + \beta_1 \]

This is an upper bound on the size of \( P^\text{MIN} \) and so also the number of traces as no trace of instruction size \( a_j \geq s_1 \) need be considered since for any \( T^j_{IC} \) with \( a_j \geq s_1 \), the size of \( P_j = A(T^j_{IC}) \) is

\[ s_j = (a_j + \beta_j) \geq a_j \geq s_1. \]

Now each time another program is synthesized the bound can be refined if needed. At some point \( P^\text{MIN} \) of size \( s^\text{MIN} \) is produced. This \( s^\text{MIN} \) is the lowest bound and so only the following \( k \) traces will ever have to be considered (See Figure B.1).

traces: \( T^1_{IC}, T^2_{IC}, \ldots, T^{k-1}_{IC}, T^k_{IC}, T^{k+1}_{IC} \)

sizes: \( a_1 \leq a_2 \leq \ldots \leq a_{k-1} \leq a_k \leq a_{k+1} = s^\text{MIN} \)

Q.E.D.

The algorithm above can be improved by analyzing the trace's structure before synthesis. For example, it would not be necessary to actually synthesize a program for each trace \( T^j_{IC} \) generated, as a
Figure B.1 Program Enumeration
lower bound $\bar{\beta}_1$ on $\beta_1$ can be calculated. This could sometimes result in a lower bound on $\bar{\beta}_1 \cdot \alpha_1 + \beta_1$ ≥ current upper bound on $P_{MIN}$ making it unnecessary to actually use A for this case.

Such considerations of the trace's structure before the R-assignments or after them is more like the approach used in handling the I'C traces directly. The generation of IC traces in this manner from an I'C trace seems to resemble closely an assignment or set covering kind of problem, and such computational complexity led to the decision to choose the method of direct synthesis from I'C traces.
APPENDIX C. DESCRIPTIONS OF EXPERIMENTS AND RESULTS

Introduction

This appendix gives more detail about the experiments resulting in the synthesized programs listed in Table 6.4. Each experiment is listed by the name of the program synthesized from the I'C trace(s). For each one the algorithm may be briefly described and then the variables and data structures in the computational environment are named and described. The initial memory state(s) which were used to yield the trace(s) are then given. Some further comments may then be made on the overall experiment and the program which was successfully synthesized is given in a corresponding figure.
Factorial

Data and programs produced are given in the chapter on results, Chapter 6.

4 variables: ONE, FACT, CNT, N;

[ONE] = 1; N is the number for which N! is to be produced and this result is placed in FACT.

Fibonacci

Data and programs produced are in Chapter 6.

6 variables: F_{I-2}, F_{I-1}, F_I, I, N, ONE;

[ONE] = 1; N - It is N^{th} Fibonacci number that is found by the algorithm.

Sequence Generation

This is just a curious little algorithm which does different steps depending on whether a number is even or odd, but which always seems to halt by converging to one.

5 variables: NUM, TMP, THREE, TWO, ONE;

[THREE] = 3; [TWO] = 2; [ONE] = 1; TMP is used to test if NUM is even or odd.

Experiment: Data is in the summary table 6.4. The initial value of NUM was 3.

Binary Search

6 variables: ONE, TWO, U, I, L, K;

[ONE] = 1; [TWO] = 2; I is the declared index for the array A, K contains the element being searched for in array A, U and L represent the upper and lower bounds of the search (as it progresses). Initially U contains the size of the array A.
1 array:  A: 1 column, [U] rows;

Experiment: The program produced resulted from a combination of the traces of 4 examples with the following initial data:

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>K</th>
<th>A</th>
<th>K</th>
<th>A</th>
<th>K</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>-23</td>
<td>17</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>17</td>
<td>-23</td>
</tr>
<tr>
<td>-5</td>
<td>5</td>
<td>-5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>17</td>
<td>16</td>
<td>17</td>
<td>53</td>
</tr>
</tbody>
</table>

Fibonacci Search

The particular algorithm used here will work only if the array being searched is of a proper size, N. The size of the array, N, must be such that the number, N+1, is a Fibonacci number. For example, if the array is of length 7, then 7+1 = 8, the fifth Fibonacci number, F_5. Let such a Fibonacci number be F_k. Then when the search procedure is called, it is assumed that the three variables I, P, Q have been initialized with Fibonacci numbers as follows

\[
\begin{align*}
I & = F_{k-1} \\
P & = F_{k-2} \\
Q & = F_{k-3}
\end{align*}
\]

5 variables:  I, P, Q, K, TMP;

I is the index pointer declared for array A.

K is the number being searched for in A.

1 array A: 1 column, number of rows as discussed.

Experiments: The program produced resulted from a combination of the traces of 6 examples with the following initial data.
A) Arrays of length 2: \( I + 2, P + 1, Q + 1 \)

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
K & A & K & A & K & A & K & A \\
57 & 57 & 51 & -173 & 51 & -173 & 51 & -173 \\
251 & 251 & 51 & -51 & & & & \\
\end{array}
\]

B) Arrays of length 7: \( I + 5, P + 3, Q + 2 \)

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
K & A & K & A & K & A \\
-731 & & & & 53 & -731 & & \\
-653 & & & & 23 & -653 & & \\
-117 & & & & 103 & -217 & & \\
23 & & & & 135 & -203 & & \\
103 & & & & 273 & -103 & & \\
135 & & & & 273 & 35 & & \\
273 & & & & & & & \\
\end{array}
\]

In both of these searches several examples were required. It is necessary, at least, to see all paths of an algorithm, and in a search this isn't possible with one example. The reason is, of course, that one must fail in several ways to find the number being searched for.

The form of the search algorithms used here were taken from Knuth [19].

**Exchange Sort**

4 variables: \( \text{ONE}, \text{TMP}, I, J; \)

\([\text{ONE}] = 1; I, J \) are index variables declared for array \( A; \) the initial contents of \( I \) is 0.

1 array: \( A: \) 1 column, \( N \) rows;

Example: \( \text{array: } (5, 7, 2, 3) \)
### Bubble Sort

8 variables: ONE, I, J, BND, T, TMP, ZERO, N;

\([\text{ONE}] = 1;\) I and J are index variables declared for array A, T indicates if an interchange has been done during a scan, BND is the bound for a scan of array A

\([\text{ZERO}] = 0;\) N is number of elements in array A;

1 array A: 1 column, N rows;

Example: array \((17, 11, 5)\)

This example requires 3 interchanges. If the first two elements are interchanged giving initially \((11, 17, 5)\), only 2 interchanges are required during this sort. This example in contrast to the successful case using 3 interchanges failed to create a program from its trace after 1 hour, 4 minutes of computation time.

### Variance Calculation

\[\begin{align*} \text{This is a calculation of } & \sum_{i=1}^{N} (A(i) - \bar{x})^2 \text{ where } \bar{x} = \frac{\sum_{i=1}^{N} A(i)}{N}. \end{align*}\]

7 variables: ONE, SUM, SQ, TMP, I, N, ZERO;

\([\text{ONE}] = 1; [\text{ZERO}] = 0;\) I is the index declared for array A, N is the number of rows of A, SUM accumulates the sum of A and then represents \(\bar{x}\) for the final calculation.

1 array: A has 1 column by N rows. I is only variable allowed as pointer.

**Experiments:**

<table>
<thead>
<tr>
<th>Contents of A</th>
<th>Trace Length</th>
<th>Number of Instructions</th>
<th>Decomputation Time (Sec)</th>
<th>Synthesis Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((7, 5))</td>
<td>17</td>
<td>293</td>
<td>9.3</td>
<td>1.4</td>
</tr>
<tr>
<td>((7, 5, 15))</td>
<td>23</td>
<td>199</td>
<td>13.5</td>
<td>1.9</td>
</tr>
<tr>
<td>((7, 5, 15, 9))</td>
<td>29</td>
<td>365</td>
<td>20.1</td>
<td>2.8</td>
</tr>
<tr>
<td>((7, 5, 15, 9, 4))</td>
<td>35</td>
<td>435</td>
<td>28.4</td>
<td>4.5</td>
</tr>
</tbody>
</table>
The first experiment with 2 elements in A was unsuccessful, 3 instructions were incorrect in the synthesized program. With 3 or more elements, the program was correct.

**Variance Calculation II**

This is a calculation of a variance using the alternative expression of variance

\[
\text{variance} = \frac{\sum (A(I)^2)}{N} - \left( \frac{\sum A(I)}{N} \right)^2
\]

6 variables: ONE, I, J, SUM, SQ, N;

[ONE] = 1; I is a row index, J a column index for the array A. N is the number of rows in A. SQ is used to represent the first part of the expression for the variance, SUM, the second. They are both assumed to be initialized to zero at the entry to the procedure.

1 array: A: 2 columns, N rows.

**Experiments:**

<table>
<thead>
<tr>
<th>Elements of A</th>
<th>Trace Length</th>
<th>Number of Instructions</th>
<th>Decomputation Time (Sec)</th>
<th>Synthesis Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 8)</td>
<td>14</td>
<td>194</td>
<td>9.7</td>
<td>1.0</td>
</tr>
<tr>
<td>(9, -3, 3)</td>
<td>18</td>
<td>308</td>
<td>17.6</td>
<td>1.8</td>
</tr>
<tr>
<td>Combination of two above traces</td>
<td>33</td>
<td>504</td>
<td>27.3</td>
<td>5.0</td>
</tr>
</tbody>
</table>

The first experiment resulted in 4 incorrect instructions in the program of which the second experiment corrected all but one. The combination of these traces was successful.

**Newton's Method for Calculation of the p\textsuperscript{th} Integer Root, p \geq 2**

This algorithm is based on iterative use of the formula for \( p \sqrt[2]{N} \)

\[
X_{k+1} = \frac{1}{p} (px_k - x_k + N/x_k^{p-1})
\]
7 variables: ONE, PWR, TMP1, TMP2, XK, N, CNT.

[ONE] = 1; PWR corresponds to p, the root desired.
N is the number of which the \( p^{\text{th}} \) root is taken.
CNT is needed for keeping track of raising XK to the p-1 power.

Experiments:

<table>
<thead>
<tr>
<th>PWR</th>
<th>N</th>
<th>Initial Guess</th>
<th>Trace Length</th>
<th>Number of Instructions</th>
<th>Decomputation Time</th>
<th>Synthesis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>64</td>
<td>3</td>
<td>19</td>
<td>131</td>
<td>4.3</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>81</td>
<td>4</td>
<td>23</td>
<td>149</td>
<td>4.9</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>81</td>
<td>2</td>
<td>35</td>
<td>262</td>
<td>7.6</td>
<td>6.1</td>
</tr>
<tr>
<td>4</td>
<td>81</td>
<td>2</td>
<td>35</td>
<td>262</td>
<td>7.7</td>
<td>21.6</td>
</tr>
</tbody>
</table>

The first two experiments produced incorrect programs, having 3 and 1 wrong instructions respectively. The last two examples are the same, both giving a correct program, the second one having greater synthesis time because of no look-ahead.

Matrix Multiplication

This is intended to be a procedure which will take square matrices and produce the product matrix one element at a time for return to a calling routine or for output.

8 variables: ONE, PROD, IA, JB, JA, SUM, IB, ZERO;

[ONE] = 1; [ZERO] = 0; SUM is a variable in which elements of the product matrix appear. IA, IB are row indices for arrays A,B; JA, JB are column indices for arrays A, B.

2 arrays: A, B - both \( N \times N \) arrays.
Experiments:

<table>
<thead>
<tr>
<th></th>
<th>Trace Length</th>
<th>Number of Instructions</th>
<th>Decomputation Time</th>
<th>Synthesis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A:</td>
<td>[3 8]</td>
<td>49</td>
<td>2217</td>
<td>118.5</td>
</tr>
<tr>
<td></td>
<td>[4 10]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:</td>
<td>[12 9]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[5 6]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A:</td>
<td>[3 8]</td>
<td>49</td>
<td>3306</td>
<td>151.3</td>
</tr>
<tr>
<td></td>
<td>[4 7]</td>
<td></td>
<td></td>
<td>66.1</td>
</tr>
<tr>
<td>B:</td>
<td>[6 4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[5 6]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIND**

The algorithm FIND is one which has been commonly used as an example for proofs of correctness of programs [18,26]. It orders the given array so that

\[
\]

where F is given at the start of the procedure.

8 variables: ONE, I, J, M, N, TMP, R, F;

[ONE] = 1; I, J are declared index pointer to the array A. M, N are bounds on I, J for each pass through A. F is the entry in the array A around which the array is ordered. R is the current value in A[F] on the basis of which comparisons are done.

1 array: 1 column, [N]₀ rows;

Experiments: The initial values of M, N on entry to the procedure FIND are 1 and the size of the array A. A number of
different experiments were attempted before it was possible to correctly synthesize FIND. There are several different conditional tests which are used in different circumstances. Thus several examples are required just to represent all the cases with which the algorithm is designed to cope. The main difficulty lies in the multiple program occurrences of the index incrementing instructions,

\[ I \leftarrow I + \text{ONE} \]
\[ J \leftarrow J - \text{ONE} \]

Each of these has several possible paths leading from it in the program, and the problem arises in associating the paths with the correct program occurrence. This was achieved by selecting examples in which it was known, by prior information in the trace or previous traces, which program occurrence corresponded with a trace occurrence. This required a careful study of the program's structure and then choosing and executing examples which properly characterize this structure.

Examples used:

(1) \( F = 5 \) \( A: (12, 51, 23, 37, 79) \)
(2) \( F = 4 \) \( A: (37, 11, 7, 13, 54, 67, 10) \)
(3) \( F = 1 \) \( A: (2, 23, 11, 17, 37) \)
(4) \( F = 3 \) \( A: (7, 11, 3, 2, 19) \)
(5) \( F = 4 \quad A : (57, 23, 37, 13) \)

(6) \( F = 2 \quad A : (41, 59, 23) \)

It must be emphasized that the other examples were mainly chosen on the basis of being representative calculations and not selected by an analysis such as required here. This program would then seem to represent an approximate level of complexity beyond which arbitrary examples are not sufficient for synthesis with this current system.
Sequence Generation
Binary Search

Success

HALT

YES

K = A(I)

NO

U = I - ONE

L = I + ONE

L < U

YES

HALT

Failure
Fibonacci Search
Exchange Sort

Flowchart:

1. Initialization:
   - I ← I + ONE
   - J ← I + ONE
   - HALT ← NO

2. Comparison:
   - A(I) < A(J) → YES
     - TMP ← A(I)
     - A(I) ← A(J)
     - A(J) ← TMP

3. Increment:
   - J ← J + ONE

4. Loop Condition:
   - I = N → YES
     - HALT ← YES
   - J ≤ N → YES

5. Termination:
   - HALT ← YES
Bubble Sort
Variance I

1. \( I \leftarrow \text{ONE} \)
2. \( \text{SUM} \leftarrow \text{ZERO} \)
3. \( \text{SUM} \leftarrow \text{SUM} + \text{SUM} + A(I) \)
4. \( I \leftarrow I + \text{ONE} \)
5. \( \text{SUM} \leftarrow \text{SUM} + \text{SUM}/N \)
6. \( I \leftarrow \text{ONE} \)
7. \( \text{TMP} \leftarrow \text{ZERO} \)
8. \( I \leftarrow I + \text{ONE} \)
9. \( \text{SQ} \leftarrow \text{A(I)} \cdot \text{SUM} \)
10. \( \text{SQ} \leftarrow \text{SQ} \cdot \text{SQ} \)
11. \( \text{TMP} \leftarrow \text{TMP} + \text{SQ} \)
12. \( I \leftarrow I + \text{ONE} \)
13. \( I > N \)
   - NO: \( \text{I > N} \)
   - YES: \( \text{HALT} \)
Variance II

1. **J + ONE + ONE**
2. **I + ONE**
3. **SUM + SUM + A(I, 1)**
4. **A(I, J) + A(1, 1) \* A(I, 1)**
5. **SQ + SQ + A(I, J)**
6. **I + I + ONE**
7. **NO**
8. **I > N**
9. **YES**
10. **SUM + SUM / N**
11. **SUM + SUM \* SUM**
12. **SQ + SQ / N**
13. **SQ + SQ - SUM**
14. **HALT**
Newton's Method for pth Root of N
Matrix Multiplication

IA + ONE

JB + ONE

SUM + ZERO

JA + ONE

LB + JA

PROD(A(IA, JA) * B(JB, JB))

SUM + SUM + PROD

JA + JA + ONE

...YES

JA > N

NO

IB < ONE

SUM + ZERO

JA + ONE

LB + JA

PROD(A(IA, JA) * B(JB, JB))

SUM + SUM + PROD

JA + JA + ONE

...YES

JA > N

NO

IA > N

YES

HALT

JB > N

NO

JB + JB + ONE

...YES

JB > N

NO

IA > N

YES

HALT
BIBLIOGRAPHY


