INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Understanding programs as devices

Allemang, Dean Thomas, Ph.D.
The Ohio State University, 1990

Copyright ©1990 by Allemang, Dean Thomas. All rights reserved.
UNDERSTANDING PROGRAMS AS DEVICES

DISSERTATION

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

By

Dean T. Allemang

The Ohio State University

1990

Dissertation Committee
B. Chandrasekaran
William Ogden
John Josephson
W. Lewis Johnson

Adviser’s Approval

Adviser

Computer and Information Sciences
For Barbara, who taught me the correct use of the word 'cogno-babble'
ACKNOWLEDGEMENTS

I wish to thank B. Chandrasekaran for providing the framework and environment that made this work possible. I wish to thank the members of my reading committee, William Ogden, John Josephson and especially W. Lewis Johnson for insightful comments that greatly helped me to understand the nature of this work. For their work on the Functional Representation, which preceded this, I wish to thank my fellow graduate students, Ashok Goel, Anne Keuneke, Bill Punch and Jon Sticklen. I would also like to thank Liz Fannin, Dave Herman, Arun Welch and Mike Tanner for making the LAIR an enjoyable place to work. I would like to thank Mike Weintraub and Richard Fox for actually taking interest in and reading this document.

This work was supported in part by DARPA/AFOSR contract #AF F 49620-89-C-0110, the Air Force Office of Scientific Research grants 87-0090 and 89-0250, and a Presidential Fellowship from The Ohio State University.
VITA

April 30, 1960 .................. Born—Columbus, Ohio

1982 ......................... B.S. Mathematics and Computer Science
                           The Ohio State University
                           Columbus, Ohio

1978-1982 ..................... Systems Programmer (part time)
                           AccuRay Corporation
                           Columbus, Ohio

1982-1984 ..................... Marshall Scholar
                           University of Cambridge,
                           Cambridge, England

1984 ......................... M. Sc. Pure Mathematics
                           University of Cambridge
                           Cambridge, England

1984-1987 ..................... NSF Graduate Fellow,
                            tenure taken at
                            Computer Science Department
                            The Ohio State University
                            Columbus, Ohio
1985 ......................... M.S. Computer Science
The Ohio State University
Columbus, Ohio

1989 ......................... OSU Presidential Fellow
The Ohio State University
Columbus, Ohio

1987-1988, 1990 ............ Graduate Research Assistant
Laboratory for Artificial Intelligence Research
The Ohio State University
Columbus, Ohio
PUBLICATIONS


B. Chandrasekaran, Ashok Goel and Dean Allemang, “Connectionism and Information Processing Abstractions: The Message Still Counts More Than the Medium”. Ai Magazine, AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE, 1988. The following two papers are shorter versions of this one.


Fields of Study

Major field: Computer and Information Sciences

Specializations:

- Artificial Intelligence: Prof. B. Chandrasekaran
- Theory of Computer Science: Prof. T. Long
- Cognitive Science: Prof. T. Milburn
Table of Contents

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Necessary Aspects of a Representation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 The Functional Representation Language</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Demonstration of the Functional Representation</td>
<td>11</td>
</tr>
<tr>
<td>1.4 The Contribution of Dudu</td>
<td>12</td>
</tr>
<tr>
<td>1.5 The Scope of Dudu</td>
<td>14</td>
</tr>
<tr>
<td>1.6 The Rest of this Work</td>
<td>15</td>
</tr>
<tr>
<td>II</td>
<td>16</td>
</tr>
<tr>
<td>An Example</td>
<td></td>
</tr>
<tr>
<td>2.1 Functional Representation</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Using Correctness Proofs</td>
<td>27</td>
</tr>
<tr>
<td>2.3 Modifying Proofs</td>
<td>32</td>
</tr>
<tr>
<td>2.4 Representing a Proof According to Intentions</td>
<td>40</td>
</tr>
<tr>
<td>2.5 Consistency of a Functional Representation</td>
<td>50</td>
</tr>
<tr>
<td>2.5.1 Consistency and Different Description Languages</td>
<td>54</td>
</tr>
<tr>
<td>2.6 Ramifications of Consistency</td>
<td>58</td>
</tr>
<tr>
<td>III</td>
<td>64</td>
</tr>
<tr>
<td>Generic Representations</td>
<td></td>
</tr>
<tr>
<td>3.1 Representing Generic Devices</td>
<td>66</td>
</tr>
<tr>
<td>3.2 Generic Loops</td>
<td>70</td>
</tr>
<tr>
<td>3.3 Generic Data Structures</td>
<td>75</td>
</tr>
<tr>
<td>3.4 Other Generic Devices</td>
<td>77</td>
</tr>
<tr>
<td>3.5 Example of use of Generic Objects: the shift loop</td>
<td>79</td>
</tr>
</tbody>
</table>

viii
### 3.6 Why Generic Objects? ............................................................. 84
- 3.6.1 Proof Strategies ......................................................... 85
- 3.6.2 Scope of Constructed Proofs ................................ 87
- 3.6.3 Providing Provisos ................................................... 88
- 3.6.4 Forming Functional Representations ...................... 88

### IV Debugging Using Device Understanding: Dudu 91
- 4.1 Matching a Functional Representation to Program Code .......... 94
- 4.2 The Role of FR in Program Debugging .............................. 98
- 4.3 The Role of Proof in Program Debugging ........................ 100
  - 4.3.1 Constructing a Proof of Correctness from the Functional Representation ............................................... 101
  - 4.3.2 Constructing an Approximate Proof from FR ................. 105
- 4.4 Dudu as a Proof System .......................................................... 111
- 4.5 A Sample Session with Dudu ................................................. 114
  - 4.5.1 A correct and expected program ............................. 115
  - 4.5.2 An incorrect program ............................................. 119
  - 4.5.3 An unexpected, but correct program ......................... 122

### V Other Work in Program Verification 128
- 5.1 Intention Based Debugging .............................................. 129
  - 5.1.1 Similarities Between Plan and Functional Representations .......................................................... 130
  - 5.1.2 Differences Between Plan and Functional Representations .................................................. 132
- 5.2 Verification Based Debugging ............................................ 136
  - 5.2.1 Reference Functions in Talus and Functional Representations .................................................. 141
- 5.3 Classical Program Verification ........................................ 143
  - 5.3.1 Teleology Apparent in Programs ............................. 145
- 5.4 Other Work in Program Understanding ............................... 146

### VI Virtues of a Functional Representation 152
- 6.1 Comparison of Representation Styles ............................ 152
  - 6.1.1 Device Representations ........................................... 153
  - 6.1.2 Plan Representations .............................................. 153
  - 6.1.3 The Functional Representation ................................... 154
- 6.2 Functional Representation and Program Debugging .......... 154
  - 6.2.1 Functional Representation and Program Proofs ........... 154
  - 6.2.2 Functional Representation and Intentions ................. 157
VII Horizons

7.1 Adaptive Design and Programs ........................................ 159
7.2 Functional Representation and Program Comprehension ............. 165
  7.2.1 Why Plan-like Representations? .............................. 166
7.3 Functional Representation and Scientific Discovery .................. 169
7.4 Limitations of Dudu ....................................................... 170
7.5 Limitations of the Functional Representation .......................... 175

APPENDICES

A Functional Representation of the Generic Search Loop .............. 177
BIBLIOGRAPHY ........................................................................ 189
# List of Figures

1.1 A grammar for the functional representation language. .................. 8

2.1 Three possible solutions to the shift problem, including one incorrect
    solution (center). ................................................................. 17

2.2 Function *induct* of the shift loop, with English language gloss of
    states. .................................................................................. 20

2.3 Abridged version of Functional Representation of the *navigator*.
    States are still English language gloss. ................................. 21

2.4 The *cover* function of the loop. ............................................ 23

2.5 The *gather* function of the collector. ....................................... 23

2.6 Augmentation to figure 2.4 to include distinctions between *for* and
    *while* loops. ....................................................................... 24

2.7 A correct solution of the shift problem which corresponds to the
    Functional Representation described so far. ........................... 25

2.8 More programs that correspond to the functional representation. .... 26
2.9 The complete functional representation for the shift loop, with En-
glish language glosses for states. .............................. 28

2.10 The complete functional representation for the navigator and col-
lector, with English language glosses for states. .............. 29

2.11 Complete Functional Representation of the Shift Loop .......... 41

2.12 Complete Functional Representation of the navigator and collector 42

2.13 A device for negating one's complement integers. .............. 56

2.14 A device for negating one's complement integers, with representa-
tion function. ................................................................. 57

3.1 Device row is both a collector and a Record. ...................... 70

3.2 Functional Representation of the Generic Total Loop ............ 72

3.3 Functional Representation of generic navigator and collector .... 73

3.4 Functional Representation of generic array ......................... 76

3.5 Functional Representation for the generic device Record .......... 78

3.6 One function (insert) of an ordered set, as a special case of an array. 80

3.7 The shift loop, represented as a special case of the generic total loop,
array and record. ................................................................. 81

3.8 The shift loop, represented as a special case of the generic total loop,
array and record (cont.) ....................................................... 82

4.1 Two programs that compute the \( n^{th} \) Fibonacci number. ........ 95

xii
4.2 A correct and expected program ................................................................. 115
4.3 An incorrect program ................................................................................... 119
4.4 An unexpected, but correct, program ...................................................... 122
5.1 A plan from PROUST's plan database ...................................................... 131
5.2 Functional representation of collector for a loop to compute a maximum .................................................. 131
5.3 Two programs that compute the maximum of the n values in an array 133
5.4 A LISP program to perform a tree sort ..................................................... 140
5.5 Another LISP program to perform a tree sort .............................................. 142
6.1 COUNTER plan from PROUST ................................................................. 157
7.1 Apparent mismatch between Dudu's expectations and program code
(example from Lewis Johnson, personal communication) ................................. 172
7.2 Two loops optimized into one. Dudu would need a separate representation for the combined loop. (example from Rich and Wills, [25]) ................................................................. 173
A.1 A program to search a binary tree ............................................................. 179
A.2 Functional representation for the generic search loop ............................... 181
A.3 Functional Representations for generic navigator and Space ................... 182
A.4 Functional Representation of generic conditional ...................................... 184

xiii
A.5 Functional Representation of a navigator for a binary search tree . . 186
A.6 Functional representation of the conditional pursuer. . . . . . . . . 187
A.7 Functional representation of a binary search tree . . . . . . . . . . . . 188
LIST OF SYMBOLS

\( \nu \) is the function which computes the next item from a navigator.

\( \sigma \) is the first item to be visited by a navigator.

\( \tau \) is the last item to be visited by a navigator.

\( U_N \) is a set to be covered by a navigator.

\( U_N(\varepsilon), \varepsilon \in U_N \) is the set of items visited by navigator \( N \) before item \( \varepsilon \)

\( \{Q\}C\{P\} \) For predicates \( Q \) and \( P \), code fragment \( C \), if \( Q \) holds before the execution of \( C \), then \( P \) will hold after execution of \( C \).

\( P^V_e \) stands for the logical expression obtained from \( P \) by substituting the expression \( E \) for all occurrences of the variable \( V \).
Humans are able to perform many complicated cognitive tasks involving devices; they are able to find faults, if any, in a device given a description of its behavior, or predict what a device will do under particular circumstances. They can explain the workings of a device to other humans. Although it is conceivable, if necessary, to accomplish these tasks by considering all possible states of the device, one seldom does this when performing such a task. Rather, one deliberately considers an appropriate subset of the states of the device, or an abstraction that aggregates various subsets of the states, as is appropriate to the task at hand.

Similar comments are true of humans who understand computer programs. In this work, I determine the requirements for a representation of program understanding to support reasoning of this nature. This dissertation describes extensions to the Functional Representation language (first described by Sembugamoorthy and Chandrasekaran in [27]), which satisfies many of these requirements, to include connecting the representation of program understanding not only to program code, but also to axiomatic proofs of program correctness.

This approach contributes to machine understanding of programs by resolving the tension between plan-based and theorem-based approaches to automatic de-
bugging. A strategy reminiscent of plan-matching approaches is used to reduce the part of a proof that must be constructed by an automatic theorem prover. This approach also contributes to machine understanding of devices by connecting the Functional Representation to a domain which has a formal definition for its semantics.

The properties of the representation are demonstrated with a program that runs in Interlisp-D on a Xerox 1108 that uses a functional representation of a computer program to verify program code, and, if the code is incorrect, to give an understandable report of the error. Dudu is able to recognize the correctness of many programs that do not correspond to the representation in an obvious way by completing a correctness proof of the program, with guidance from the functional representation.
CHAPTER I

Introduction

Humans are able to perform many complicated cognitive tasks involving devices; they are able to find faults, if any, in a device given a description of its behavior, or predict what a device will do under particular circumstances. They can explain the workings of a device to other humans. Although it is conceivable, if necessary, to accomplish these tasks by considering all possible states of the device, one seldom does this when performing such a task. Rather, one deliberately considers an appropriate subset of the states of the device, or an abstraction that aggregates various subsets of the states, as is appropriate to the task at hand.

Such behavior is characteristic of humans who truly understand computer programs; they can generate (by ‘hand simulation’) the states of the program from any input, but eschew doing so if it can possibly be avoided. Nevertheless, they can debug, redesign or explain the program. In addition, the use of formalisms such as axiomatic semantics of programs (see Pagan, [21]) makes it possible to construct proofs of correctness of computer programs. All these things are done,
for programs as for other devices, without appeal to complete simulation of the
action of the program. In this work I will examine some aspects of a representation
of a program that support reasoning of this nature. In the conclusion, I will use
this to clarify what it means to 'understand' a device.

1.1 Necessary Aspects of a Representation

In order to perform effectively any of the tasks mentioned above, it is necessary
to avoid dealing with too many low-level details. A mechanic cannot troubleshoot
an engine by thinking of every detail of the electrical, fuel mixture, cooling and
lubrication systems. Instead, faults are isolated in a particular subsystem before
details are considered. Similarly, a detailed explanation of the workings of a device
is incomprehensible unless a high level overview has first been given. The high level
explanation differs from the low level explanation in more ways than just the degree
of detail; although the detailed description of the workings of the electrical system
of an automobile engine is given in terms of voltages and currents, at a higher
level of description, it is only necessary to know that the electrical system supplies
a spark for each combustion stroke. That is, the higher level descriptions
incorporate the purposes of the various detailed descriptions. A similar situation
arises in computer programs; for example, it is not necessary to know the details
of the implementation of a data structure in order to understand how it supports
a particular algorithm. In this work I will examine a representation scheme that
explicitly incorporates mechanisms to meet these goals, that is, to represent both
detailed and abstract descriptions of a device, in which the abstract explanations
are supplied by specifying the intended purposes of the components of the device.
I will show how applying this representation to the domain of computer programs
both clarifies issues about device understanding, and solves problems of automatic
debugging.

First, I will introduce the Functional Representation that I will use for this
work, and describe how it satisfies these requirements.

1.2 The Functional Representation Language

I will use the Functional Representation (FR) language introduced by Sembug-
amoorthy and Chandrasekaran in [27] to describe program, as outlined above.
Originally, this language was intended for dealing with physical devices, such as
the buzzer used as the principal example in [27]. The guiding intuitions behind
the FR language are as follows:

- Devices have functions. This is taken as the defining characteristic of a
device; a device is something which has a known function.

- A device is made up of other devices, which are called components. The
representation of the main device achieves its functionality by employing the
services of its components. The main device achieve its effect by coordinating
these subfunctions in a specified way.
• A device may have several functions.

• How a component achieves its function(s) is irrelevant to understanding its role in another device; a device may be replaced by for another that provides the same functionality, but in a different way. This allows descriptions of the enclosing device to be given in a completely different language than those of the component.

• Functional components might not always correspond to apparent physical components.

These intuitions can be demonstrated by use of the example of a doorbell buzzer ([27]). The functionality of the buzzer is that when a button is pressed, a sound is made. This is the function of the buzzer in the above sense. The buzzer is made up of a battery, a switch, a magnetic coil and a clapper; these are its components. The components are, in turn, other devices, and have functions of their own. The clapper has two functions, that is, to make a sound when it closes, and to allow current to flow when it is closed. In order to understand the overall buzzer, it is not necessary to know, for instance, whether the clapper is made of a conducting metal in an acoustic shape, or if it strikes a bell, or if it triggers some other mechanism that makes a sound. The buzzer can be understood at a high level just by knowing that the clapper will both produce a sound and close a circuit when in its ground state. Finally, the pieces of the clapper mechanism might not
all be located together in space.

A Functional Representation of a particular device imposes a structure on the
device in which it is described as being comprised of components, which supply
various functionalities. If the device is a designed artifact, these components and
functionalities would normally correspond to the intention of the device designer;
that is, the device can be viewed as a particular set of components because the
designer built it that way. This is called a Functional Representation because it
describes objects in terms of their intended use.

In the Functional Representation language, devices are represented by their
functions. Since the Functional Representation language is a description, not a
definition, of the device, not every possible function of a device need be included.
For example, one might describe a circuit in terms of voltage drops, and in these
terms, one could describe one function of a battery as supplying a voltage drop.
One could describe another function of the battery to hold open a door, or to
weigh down loose papers. In the context of a circuit, the former would be more
appropriate, in that it probably reflects the intentions of the circuit designer.

A Function of a device is specified using input/output pairs. The inputs and
outputs are states, specified in a state language, which is independent of the func­
tional representation.\footnote{In this work, the state language is predicate calculus, with predicates taken from the formal
semantics of an underlying programming language.} In this work, as in [27], a function will be specified as a
frame with the four slots named If, ToMake, Provided and By. These slots, and
the lexical categories acceptable as slot fillers, are shown in the grammar in figure 1.1. In planning terminology, the If and ToMake slots correspond roughly to preconditions and goals respectively. The Provided slot is optional, and indicates a state (or predicate) that must hold in order for the description of the function to be meaningful. In this work I define a much more specific use of the Provided slot.

The By slot specifies a behavior, which indicates how this particular device achieves this function. That is, the function of the device describes an input/output pair that serves as a specification of what the device is to do, while the behavior describes its implementation. A behavior is specified in the Functional Representation language by an annotated directed graph with state descriptions (statements in the state language) serving as nodes in the graph. The annotations on the links describe what is understood to be responsible for the indicated state transition. In [27], behavior graphs are restricted to have the form of a chain, a single state joined by a single link to another state, which in turn may be joined to another state by a single link, and so on. Hence, in figure 1.1 behaviors are specified as alternations of states and links, starting and ending with a state.

The types of annotations allowed on a link in a behavior graph are By, Using Function and As-Per. Figure 1.1 shows these annotations, as well as the lexical categories that must follow them.
A link annotated with **By** specifies that some detail is being suppressed in the current graph. The behavior that follows specifies the details of the transition.

A link annotated with **Using Function** specifies that the transition between states is accomplished by some other function of some component. This is how the functional representation describes a device accomplishing a function by calling upon a function of a component.

A link annotated with **AsPer** refers to some knowledge beyond the functional representation; in [27], this is intended to be something like a qualitative or quantitative physics model; some description of what is going on in the device that does not refer to the intended functions of the device at all. In this work, the annotations **AsPer** and **By** are not used; only the annotation **Using Function**.

Functions are specified separately from behaviors in order to allow functions to be indexed separately from behaviors; that is, if two devices provide the same function, any differences in their behaviors is not relevant to the role they play in a larger device.

In this work, I extend the basic representation in [27] to include more complicated behavior graphs. I now also allow a behavior graph to be a simple cycle (a chain in which the last and first states are the same, e.g., (S1 L1 S2 L2 S3 L3 S1)). I also allow more than one link between two states. This indicates that the state transition is effected by all the functions referenced on the links, but in no particular order. There is no meaningful state of the system in which some of the
(device definition) ::= Device ident{(function definition)}*
(function definition) ::= Function ident
   If (state)
   ToMake (state)
   [Provided (state)]
   By behavior

(link) ::= By behavior
   | Using Function function of device
   | As Per knowledge in the context of (any)

Figure 1.1: A grammar for the functional representation language. (state) is left undefined by [27]. ident is any identifier (a name). behavior, function and device are identifiers that are specified in a (behavior definition), (function definitions) or a (device definition), respectively. A (behavior definition) is usually shown as a graph, with the states as nodes, and the links bearing the specified annotations. Sembugamoorthy and Chandrasekaran only use very simple graphs (with no branches and no cycles).
functions have been completed but others have not. Notice that this differs from the usual interpretation of state/transition graphs.

The seminal work by Sembugamoorthy and Chandrasekaran used the Functional Representation to describe mechanical devices. The state language used by Sembugamoorthy and Chandrasekaran was not formally defined, and was mostly evocative English language state descriptions. In this work, by applying the functional representation principles to a domain for which there is a more formal treatment, I have given a formal definition of a state language. This definition makes it possible to link the representation to the proof of correctness of the program, and thus defines a solution to the problem of consistency of the Functional Representation. This in turn has provided a more formal distinction between the If and Provided slots defined by Sembugamoorthy and Chandrasekaran.

Sembugamoorthy and Chandrasekaran present the problem of the 'consistency of the representation' [27]. The problem is that the functional representation language allows us to describe a device, but does not place any constraints upon the description we use the language to present. For instance, one could represent a system in which water is made to flow by closing an electric switch, and Bernoulli's equation is used to support the statement that the temperature of the water rises as a result. The consistency problem is to make sure that the components and functions referenced in a functional description in fact support the state changes they are claimed to support. Sembugamoorthy and Chandrasekaran correctly com-
ment that this cannot be done without a specification of the state language. Later in this work, I will give a criterion for consistency of a functional representation of a program, using a state language based on program semantics.

Sembugamoorthy and Chandrasekaran demonstrate some of the capabilities of the Functional Representation by presenting an algorithm to automatically generate a troubleshooting system for a device given a functional representation of it. The advantages of this approach were that it showed how the Functional Representation could be used to focus attention on parts of the device by examining whether they satisfied their intended goals. Many of the problems with this approach stemmed from its lack of a systematic language for representing states. For example, the constructed system would ask the end user if various states were obtained during the operation of the device; it had no way to tell whether such a state was observable, or if it was not, how it might relate to other, observable, states. In this work, I define the state descriptions as predicates in the formal semantics of the programming language. This facilitates reasoning about states that cannot be easily determined while a program is running, and linking them together in the form of a correctness proof of the program.

The states, or predicates, are intermediate formulas from an axiomatic proof of correctness of the program. But a proof is more than just a sequence of formulas; the remaining structure of the proof is specified by a proof template attached by the builder of the functional representation to the various parts of the functional
representation. These can often be specified generically; so, for example, any loop would have a proof template that required the construction of a loop invariant, which is proven for a base case and an inductive case. The intermediate formulas in these two cases (as well as the invariant itself) would be specified by states in the behavior description of a particular loop. In the next section, I describe a program whose job it is to construct such a proof from the various proof templates and the details of implementation of a given program.

1.3 Demonstration of the Functional Representation

To demonstrate how the functional representation can be used, I provide a method called Dudu (Dudu stands for ‘Debugging Using Device Understanding’, and has been implemented in Interlisp-D on a Xerox 1108) that automatically debugs programs. Starting with a functional representation of a program as described above, Dudu takes a program written in a Pascal-like pseudo code, and determines whether it can be described as represented. The proof templates associated with each part of the functional representation are combined, along with specific from the sample program, to construct a proposed proof of correctness of the program. This proposed proof is modified as needed to construct a proof for the input program; this modification is focused by use of the hierarchical structure of the functional representation. Not only does this provide a proof of correctness for any program that has the same intention structure as the prototype, but also allows
Dudu to pinpoint places where the input program deviates from the prototype (exactly those places where the proof modification fails). In such cases, Dudu can attempt to complete the proof (thus recognizing a novel solution), or failing that, generate an error message specific to the faulty part of the program.

1.4 The Contribution of Dudu

Dudu is not the first attempt that has been made to solve the debugging task. The most notable work in this area is that of Johnson (in a system called PROUST [10]) and Murray (in a system called Talus [20]).

The approaches taken by Johnson and Murray differ in many ways. Johnson's approach involves representing programming plans and goals, and analyzes a student's program by synthesizing a composite plan that would result in the sample code. Differences between the student's code and any known plans provide indices into a catalogue of well-known common errors. This approach is called 'intention based diagnosis' because it works by trying to anticipate the intentions of the programmer. Murray provides a set of reference functions which solves the specified task, and then uses a theorem prover in the Boyer-Moore logic to attempt to prove the equivalence of the reference function to the student's function.

Murray argues convincingly that PROUST could benefit from the ability to reason about program semantics. Since Talus does reason with program semantics, it is able to avoid some problems that PROUST encounters. Furthermore, Talus is
able to solve some problems that seem to require the interpretation of intentions. However, Talus relies on the fact that the reference function is written in such a way that its intentions are reflected by its structure, and these intentions match those of the student.

One contribution of Dudu is to combine these approaches. Dudu uses a representation that shows how knowledge about the behavior of a device arises from knowledge about its components, and how they work. At high level, these components and their functions correspond to plans in PROUST. At the lowest level, these reduce to lines of code, for which it is understood that they work because of some statement in the formal semantics of the programming language. This allows Dudu to pursue plan-like intention based reasoning, yet still consult a proof of correctness of the program.

This approach combines the advantages of the approaches taken by PROUST and Talus:

- The intentional structure of the program is explicitly represented, so, like PROUST, Dudu can consider alternative intention structures for a student's program.

- Dudu can consult the program semantics to avoid proliferation of semantically identical plans.

Furthermore, Dudu combines these approaches to gain more advantages:
• The intentional structure focuses Dudu's debugging on a single part of the Functional Representation; since this is linked to a proof of correctness, this focuses the theorem prover on only the appropriate part of the proof. That is, Dudu does not need to construct a proof of an entire program in order to debug (or verify) it.

• The Functional Representation handles proofs in fragments, and has slots (the Provided slot) that indicate when proof fragments are appropriate. Hence constructing a proof (and finding bugs) involves proving simpler conjectures than the correctness of code fragments.

Finally, Dudu is not restricted to representing programming plans, but also represents data as devices, and uses this to help organize provisos.

1.5 The Scope of Dudu

This work focuses on aspects of a representation for devices that supports cognitive tasks. It is not intended to make contributions to matching such representations (as does PROUST, [10]), or cataloging them (as does TAD [7]), or combining them (as does the Recognizer, [25]). Instead, it focuses on defining the relation such representations have with other descriptions of devices, in particular, with their observed structure and a formal justification of their behavior. In the case of computer programs, this is the relationship between the program code and the formal semantics defining its interpretation. Hence the contribution of this work lies
more in its contribution to device understanding in general than to understanding of computer programs in particular.

1.6 The Rest of this Work

The remainder of this volume is dedicated to describing in detail the use of the functional representation to represent programs. Chapter II works out a detailed example of a program fragment taken from an insertion sort. First, the example is given as in [27], that is, without any specific state language. This example, with states given in English, is intended to familiarize the reader with the use of the functional representation, particularly for programs. Then the same example is repeated, with a specific state language based on the programming language semantics for the target language. In chapter III the functional representation is separated from the example, so as to show how functional representations can be built for other tasks. In chapter IV, the implementation and design of Dudu is described, and examples of its execution. In chapter V, I compare in detail the approach in Dudu to the approaches used by Johnson [10] and Murray [20]. In chapter VI I discuss why the functional representation was appropriate for this task. Finally, in chapter VII, I discuss how the functional representation relates to human understanding of a program.
CHAPTER II

An Example

In this chapter I present an example of how the functional representation relates a program to a proof of correctness. I will start by showing three programs that are variants of a single method. Then I will present a functional representation of that method. Finally, I will show how to relate this functional representation to a correctness proof for the simplest of the three programs. In chapter IV, I will show how to construct a proof of correctness for all three sample programs.

Consider the problem of moving the contents of each element of an array between indices $k$ and $n - 1$ to the next higher position, that is, $\forall i \in [k, n - 1], \ a[i + 1] = \#a[i]$ ($\#a$ denotes the original values of the array $a$. At the end of the program, $a[n]$ has the value that used to be at $a[n - 1]$).  

Three possible solutions to this are shown in figure 2.1. The first solution iterates backwards over the relevant fragment of the array, moving the element with the highest index ($n - 1$) into place first, leaving room for the next to last

---

1This problem is a subproblem of an insertion sort, when a single array is used to store the list to be sorted.
\begin{verbatim}
i := n - 1
while i ≥ k do
  a[i + 1] := a[i]
i := i - 1
end

i := k
while i ≤ n - 1 do
  a[i + 1] := a[i]
i := i + 1
end

i := k
temp := a[k]
while i ≤ n - 1 do
  save := a[i + 1]
a[i + 1] := temp
temp := save
i := i + 1
end
\end{verbatim}

Figure 2.1: Three possible solutions to the shift problem, including one incorrect solution (center).

...element, and so on. The second solution attempts to move the element with the least index (k) first, but is buggy, since this clobbers the contents of the next array location. The final example is a corrected version of the second case, in which some auxiliary variables have been introduced to take care of the value clobbering problem.

If one suppresses the details in these programs, all three of these programs are the same, in that they iterate over the set in question, and move the elements into place one by one. They differ in the details by which these are done. The first one traverses the set from highest index to lowest, while the other two go from lowest to highest. The first two simply move the appropriate element from where it started to where it belongs, while the third does a more complicated version of swapping values to get the appropriate value in the appropriate place.

Despite these differences, all three share the following features:
There is a variable responsible for navigating through the set to be processed (in all three cases the name of that variable happens to be \( i \)). Navigating through the set involves three things:

- start somewhere in the set
- move from one item to the next
- detect the end of the set

There is also some code responsible for moving a value from one position in the array to another. It is necessary that no value be clobbered by this action.

Since the correct programs can agree in overall solution strategy, but differ in the choices made for these parts, (i.e., the order in which the set is covered, how the elements are moved in the array), I will represent these parts separately, one for finding the way through the set, which I will call the *navigator*, and one to move the elements in the array, which I will call the *collector*. The job of the navigator can be broken down into three parts, which must interact with the collector in the appropriate way in order to perform the desired shift. This coordination is what is in common between the two solutions (top-to-bottom and bottom-to-top). So I will first describe the top level loop, which is common to both solutions, leaving

---

2I take the choice of the name *navigator* to be self-explanatory. The job of the *collector* is to take an item provided by the navigator, and an intermediate result, and form a new intermediate result which incorporates the new item. If the intermediate results are some sort of collection, then the *collector* has the job of adding a new item into the collection.
slots open for the roles which will be filled by the navigators and collectors, which will be customized for the two sample solutions.

2.1 Functional Representation

All representations in this work will be in the language of the Functional Representation of Sembugamoorthy and Chandrasekaran [27]. The Functional Representation does not specify the semantics for states; I will say more about the nature of states when I relate the FR description of a program to the template for correctness proofs of the program. For now, I will specify states in English.

In figure 2.2, I show the FR of the overall loop for the shift problem. This loop will have two functions specified, one of which is shown in figure 2.2. This function, which I call induct, is the overall function of the loop, hence it specifies that the loop will in fact satisfy the specification of the problem, that is, to move the items from $k$ to $n - 1$ inclusive to locations $k + 1$ to $n$ respectively. This is specified by the If and ToMake fields of the function induct. This functionality is achieved by a certain behavior, which in this case is given the name 'initialize-and-loop'. The state/transition diagram corresponding to a program object can be viewed as a flowchart of the program control which implements the function. In this case, since this is the highest level function of the loop, the flowchart is a high level flowchart. It differs from classical flowcharts by the inclusion of the states in the diagram. Hence, the state/transition diagram in figure 2.2 should be
**Function** `induct`

*If* there are values in array slots $k$ to $n - 1$

**To Make** These same elements are in slots $k + 1$ to $n$ respectively

**By initialize-and-loop**

**Behavior** `initialize-and-loop`:

---

```
T
     
Using Function `start`
of `navigator`

First zero elements have been moved

Using Function `cover`
of `shift loop`

First $n - k - 1$ elements have been moved
```

---

**Figure 2.2:** Function `induct` of the shift loop, with English language gloss of states.
Device *navigator*

**Function start**

*If True*

*ToMake* navigator initialized

*By e.g.,* $?navigator := n - 1

**Function next**

*If some element is available*

*ToMake* the next element is available

*By e.g.,* $?navigator := $?navigator - 1

**Function check**

*If some element is available*

*ToMake* element known to be appropriate

*By e.g.,* $?navigator ≥ d

Figure 2.3: Abridged version of Functional Representation of the *navigator*. States are still English language gloss.

read as a two-step method, the first of which sets up the variables for a loop, and the second runs the loop itself. This diagram defines the relationship between the initialization of the loop variables and the loop itself; the intermediate states will relate this structure to a proof of correctness for the loop.

The construct ‘Using Function’ in the FR language means that a transition will be made, not by appeal to a primitive fact about the domain, but rather by reference to some other function (either of another device, or of the same device). Both links in the behavior ‘initialize-and-loop’ are of this sort. This is what makes it a ‘high-level’ flowchart; the actions specified in it refer to functions provided elsewhere, which will eventually be specified in terms of the programming language.

The role of variable initialization is provided by the function *start* of the naviga-
tor. In figure 2.3, we show an abridged FR for the navigator. Figure 2.3 shows the three functions of a navigator which we discussed above (page 18), namely \textit{start}, \textit{next} and \textit{check}, with the states in the \texttt{If} and \texttt{ToMake} fields replaced by English language descriptions which are evocative of the relationships between them. The behaviors of the navigator are not presented as state/transition diagrams; rather, they are presented as programming language code templates, samples of what a line of code might look like which can implement these functions. The lines shown in figure 2.3 are for the navigator which traverses the set in decreasing order, from \( n - 1 \) back to \( k \). Identifiers in the figure which begin with a '?' are template variables; they stand for any identifier in the target language, but of course, all occurrences of any template identifier must be bound to the same program identifier. These sample lines of code are the lowest level flowcharts; they do not refer to any other functions, or any other flowcharts, they refer directly to the programming language constructs by which the operations are implemented. The notation 'Using Function \textit{start} of navigator' on the first link of figure 2.2 indicates that the navigator as defined here can fill the role specified by the first link in figure 2.2, in the capacity defined by its function called \textit{start}. In order to verify the consistency of using this function to fill this role it will be necessary to verify that the state which follows in figure 2.2 (in English, 'first 0 elements have been moved') in fact does follow from the function \textit{start}. This requirement will help to refine the formal description of a state.
Device *shift loop*

**Function** *cover*

If no items moved

**ToMake** \( n - k - 1 \) items moved

**By** *cycle*

**Behavior** *cycle*

- elements up to \( i - 1 \) have moved

  - Using Function *next*
    of *navigator*

  - Using Function *check*
    of *navigator*

  - elements up to \( i \) have moved

  - element \( i \) is appropriate

  - Using Function *gather*
    of *collector*

---

Figure 2.4: The *cover* function of the loop.

---

Device *collector*

**Function** *gather*

If \( i \) is appropriate, and all items before \( i \) have been moved

**ToMake** all items up to \( i \) have been moved

**By** *e.g.*, \( a[?x + 1] := a[?x] \)

---

Figure 2.5: The *gather* function of the collector.
Device shift loop

Function cover

If no items moved

ToMake $n - k - 1$ items moved

By cycle

Behavior cycle

Using Function next of navigator e.g., & by &

Using Function check of navigator e.g., while & to &

elements up to $i - 1$ have moved

elements up to $i$ have moved

Using Function gather of collector e.g., &

element $i$ is appropriate

Figure 2.6: Augmentation to figure 2.4 to include distinctions between for and while loops.

The second link in figure 2.2 refers to another function of the loop itself, namely, the function cover of the shift loop. In figure 2.4 we see the FR of this function. In its behavior, we see the roles which are filled by the other functions of the navigator, as well as the gather function of the collector, which is shown in figure 2.5.

The behavior for gather is also primitive, that is, rather than referencing another behavior, it gives a code sample for how it is to be implemented.

The behavior for cover shows the control flow relationships among the functions of the navigator and collector. These functions themselves specify how they
\[ i := n - 1 \]
\[ \text{while } i \geq k \text{ do} \]
\[ a[i + 1] := a[i] \]
\[ i := i - 1 \]
\[ \text{end} \]

Figure 2.7: A correct solution of the shift problem which corresponds to the Functional Representation described so far.

correspond to program code. But so far, we have not shown anything that corresponds to the program code that shows the syntactic relationships among the program fragments that correspond to the functions of the navigator and collector. For example, the sample code for the function `check` of the navigator shows the comparison `?index \geq k`; if the loop is implemented by a `while` construct, this comparison should immediately follow the keyword `while` and precede a `do`. That is, the code fragments `while`, `for`, `do` and `end` are associated with the intermediate level flowchart which is the behavior for `cover`. Hence, we augment the links of the FR with code samples, and show the syntactic relationship of the code samples at the level of `cover` to those at the level of the navigator. For example, at the link in `cover` which is notated `Using Function check of navigator`, we further notate it with `while &`, to indicate that whatever code the function `cover` corresponds to will follow immediately after a keyword `while`. See figure 2.6 for all the annotations for a while loop.

The abridged representation just described (figures 2.2, 2.3, 2.5 and 2.6) corresponds directly to the program in figure 2.7. Since it is an abridged version, with
many optional components omitted, and since we have not shown how to use the
notion of a proof schema to allow this to refer to other programs, so far this is a
representation of only this one program (up to variations in variable names).

It is simple enough to extend this representation so that it obviously corre­
sponds to all the programs in figures 2.1 and 2.8. This is done by adding the
appropriate code samples to the functions of the navigator and the collector, as
well as to the cover function of the shift loop. This introduces a new wrinkle into
the code fragment templates. While the initialization \(i = k\) and the check
\(i < k\) are both valid ways to implement the respective functions, it is not
the case that a correct program can have both of these. That is, the navigator
is either going up or down, and it must be presented consistently. Similarly, the
fragment 'by -1' counts as a means for decrementing a variable only if it appears
appropriately in a for loop. Hence, associated with each code sample is a set of
identifier/value pairs to coordinate among these options. For example, the samples
\(i = k\), \(i <= n - 1\) and \(i = i + 1\) for start, check and next of
the navigator respectively, are all consistent with the navigator traversing the set

\[
\begin{align*}
\text{for } i \text{ from } n - 1 \text{ to } k \text{ by } -1 \text{ do } & \quad \text{for } i \text{ from } k \text{ to } n - 1 \text{ do } \\
& a[i + 1] := a[i] \\
& a[i + 1] := a[i] \\
\end{align*}
\]
in ascending order. Hence, all of these share the identifier/value pair ‘order=asc’.

The complete representation of the shift loop, which represents all the solutions in figures 2.1 and 2.8 is shown in figure 2.9.

Unfortunately, the representation in figure 2.9 is still in trouble, since two of the programs in figures 2.1 and 2.8 are faulty. Figure 2.9 is intended to correspond to a set of programs which are understood the same way. Clearly the incorrect programs are not understood in the same way as the correct ones, so figure 2.9 corresponds to too many programs.

In order to distinguish correct programs from incorrect ones, the FR of the program must be able to refer to the semantics of the programming language. Since the structure of the loop and the role of the collector are known, as well as the structure of correctness proofs for loops, it is possible to recognize a correct collector by completing a correctness proof of the program. This method, which is the topic of the next section, is able to recognize novel collector/navigator combinations.

2.2 Using Correctness Proofs

If we have a proof of correctness for a program which uses some overall strategy, say, to move through the appropriate part of the array, moving elements up one by one,

\footnote{The problem is that when we navigate from the bottom of the array to the top, and move the array elements up one array entry, that we write to the \( i + 1 \) element of the array before we read from it, that is, we clobber the contents of the array.}
Function induct
If There are values in array slots \( k \) to \( n - 1 \)
ToMake These same elements are in slots \( k + 1 \) to \( n \) respectively
By initialize-and-loop
Behavior initialize-and-loop:

Using Function \( \text{start} \)
of navigator
e.g. for & [type=for]
& while [type=while]

First zero elements have been moved

Using Function \( \text{cover} \)
of shift loop
e.g. do & end

First \( n - k - 1 \) elements have been moved

Device shift loop
Function \( \text{cover} \)
If no items moved
ToMake \( n - k - 1 \) items moved
By cycle
Behavior cycle

Using Function \( \text{next} \)
of navigator
e.g. & [type=while]
& do [type=for]

Using Function \( \text{check} \)
of navigator
e.g. while & [type=while]
to & [type=for]

elements up to \( i - 1 \) have moved

Using Function \( \text{gather} \)
of collector
e.g. &

Figure 2.9: The complete functional representation for the shift loop, with English language glosses for states.
Devices navigator

Function start

If T

ToMake navigator initialized

By e.g., ?navigator := n - 1

?navigator from n - 1

?navigator := k

?navigator from k

Function next

If some element is available

ToMake the next element is available

By e.g., ?navigator := ?navigator − 1

?navigator := ?navigator + 1

by -1

NIL

Function check

If some element is available

ToMake element known to be appropriate

By e.g., ?navigator ≥ k

?navigator ≤ n − 1

k

n − 1

Devices collector

Function gather

If i is appropriate, and all items before i have been moved

ToMake all items up to i have been moved

By e.g., a[?x + 1] := a[?x]

Figure 2.10: The complete functional representation for the navigator and collector, with English language glosses for states.
it can be modified to form a proof of correctness of other programs using the same overall strategy (or failing to do so, declare that they are probably buggy). Above, I presented a representation for such programs which preserves the structure of the programs. I will now link that representation to a proof of correctness in such a way that it can be decided what part of the proof needs to be changed to form a proof for the new program.

First, I present a standard proof of correctness for the program in figure 2.7. This easy proof will be given in great detail, so that later it can be related to the Functional Representation of the program. This proof follows the form for inductive proofs outlined by Heisel ([16]) for proving the correctness of loops. Interestingly, Heisel uses the title 'Checklist for Understanding a Loop' for this proof form.

Let \( C \) be the code in figure 2.7, in which the identifier \( a \) is an array, and the identifiers \( k \) and \( n \) are indices into the array, with \( k < n \). Denote the values of the array \( a \) as \( \#a \).

Let \( Q \) be the condition \( \forall j, k \leq j \leq n - 1, \quad a[j] = \#a[j] \). At the start of the program, \( Q \) holds.

Let \( P \) be the condition \( \forall j, k \leq j \leq n - 1, \quad a[j + 1] = \#a[j] \).

Theorem 1. \( \{Q\}C\{P\} \) (i.e., if \( Q \) holds before the execution of the code in \( C \),

---

4There are two steps in the outline Heisel gives that are not included here, which involve proving that the loop will terminate, and dealing with the values at termination. Hence, any proofs mentioned here are proofs of partial correctness.
$P$ will hold after the execution of $C$.  

**Proof.**

After the execution of $i := n - 1$, the variable $i$ will contain the value of $n - 1$.

Form an invariant of the loop. Consider the proposition $I$:

$$
\forall j, k \leq j \leq i \quad a[j] = \#a[j]
$$

\[ \land \forall j, i + 1 \leq j \leq n - 1 \quad a[j + 1] = \#a[j] \]  

(2.1)

With $i$ bound to $n - 1$, the second conjunct is vacuous, so this reduces to $Q$. Hence, $I$ is true as the program enters the loop.

To show that $I$ is an invariant, we must show that it is true after each iteration of the loop.

After $a[i + 1] := a[i]$, since $a[i] = \#a[i]$, we have $a[i + 1] = \#a[i]$. This and $I$ imply

$$
\forall j, k \leq j \leq i + 1 \quad a[j] = \#a[j]
$$

\[ \land \forall j, i \leq j \leq n - 1 \quad a[j + 1] = \#a[j] \]  

(2.2)

After execution of $i := i - 1$, we replace all occurrences of $i - 1$ with $i$, which gives us

$$
\forall j, k \leq j \leq i + 1 \quad a[j] = \#a[j]
$$
\[\forall j, i + 1 \leq j \leq n - 1 \quad a[j + 1] = \#a[j]\]  

(2.3)

This implies, \textit{a fortiori}, \(I\).

At the loop termination, we have \(i = k - 1\); from the second conjunct of \(I\), this leaves

\[\forall j, k \leq j \leq n - 1 \quad a[j + 1] = \#a[j]\]

QED.

\section*{2.3 Modifying Proofs}

In order to be able to modify this proof to work for other programs that use a similar strategy, it is necessary to make the structure of the proof match the intended structure of the program. Then we can map it onto the functional representation of the program.

Since at the top level of description, each program in figures 2.1 and 2.8 is a single loop, the proof for each of these will follow the form for an inductive proof of a loop. Paraphrasing Heisel et. al. [16], this is

To Prove \(\{Q\}C\{P\}\),

- Propose a loop invariant \(I\), with \(\{Q\}A\{I\}\), where \(A\) is the code before the loop

- Verify that it is an invariant of the loop, that is, show that \(\{I\}B\{I\}\), where \(B\) is the body of the loop.
• Show that $I$, along with the termination condition of the loop, implies $P$.

The formation of the loop invariant itself is a bit trickier, since it involves a subtle interplay between the body of the loop, the counter of the loop, and the termination condition. If we were to change the loop invariant for this proof, say, to an invariant for a program in which the array is visited in the opposite order, we would need to swap the quantifiers around. That is, at the beginning of any execution cycle of the loop, the values of $j$ for which the initial condition $a[j] = \#a[j]$ is still true are the values greater than $i$ (the navigator variable) in the counting up case, and the values less than $i$ in the counting down case. Compare this loop invariant to equation (2.1):

$$\forall j, i \leq j \leq n - 1 \quad a[j] = \#a[j]$$

$$\land \quad \forall j, k \leq j \leq i - 1 \quad a[j + 1] = \#a[j] \quad (2.4)$$

In order to generate this invariant automatically, it is necessary to determine how the parts of the program give rise to parts of the invariant. Not surprisingly, this makes the invariant itself easier to understand.

I will do this by examining the roles that the navigator and the collector play in forming the loop invariant. First, I introduce the following notation. Let $U$ be the set of values which are to be covered by a navigator $N$. Let $\sigma \in U$ be the element at which the navigator starts (Recall that a navigator has three functions, $start$,}
next and check). Let $\nu : U \rightarrow U$ be the map which describes the next function of the navigator. $\nu$ is defined for every element of $U$ except one (the last item visited by the navigator). For that element, define $\nu$ to be $\tau$, with $\tau \notin U$. Finally, extend both the domain and co-domain of $\nu$ to include $\tau$ and define $\nu(\tau) = \text{def } \tau$. Now define $U(e)$, for $e \in U \cup \{\tau\}$ as follows:

$$U(e) = \begin{cases} \emptyset & \text{for } e = \sigma; \\ U(\nu^{-1}(e)) \cup \{\nu^{-1}(e)\} & \text{for other } e \in U; \\ U & \text{otherwise.} \end{cases} \quad (2.5)$$

That is, $U(e)$ is the set of elements traversed by $N$ before $e$ (not including $e$ itself). $U \setminus U(e)$ is the set of elements yet to be visited by the navigator (including $e$). When there may be some ambiguity about which navigator is being referred to in the definition of $U$, the navigator is specified in a subscript, e.g., $U_N \setminus U_N(e)$.

The job of the collector is to compute some function over the various sets provided by the navigator. I will denote by $\Gamma(S)$ ($S$ is a parameter of $\Gamma$; $S \in U^*$) the predicate on computation states that says the appropriate function has been computed. For example, for a collector that saves a running total in variable $\text{sum}$, $\Gamma(S) \equiv \text{sum} = \sum_{i \in S} i$. In these terms, the function of the collector is to make $\Gamma(S \cup \{e\})$ true, given $\Gamma(S)$ and $e$.

For the collector used in these examples,

$$\Gamma(S) = \forall j \in U \setminus S, \quad a[j] = \#a[j]$$

$$\land \forall j \in S, \quad a[j + 1] = \#a[j] \quad (2.6)$$
That is, for items in the set $S$, the array values are in their new positions (i.e., advanced by one), while the other items are in their original positions.

For the navigator used in the original example (figure 8, counting down the set), $\nu(i) = i - 1$, so $U = \{k..n - 1\}$, and $U(i) = \{j|i + 1 \leq j \leq n - 1\}$, so $U \setminus U(i) = \{j|k \leq j \leq i\}$. When we put these definitions into (2.6), we have (2.1), as desired.

Now we can reconstruct the proof in terms of $\nu$ and $U$.

Let $Q$ be the condition $\forall j \in U, \ a[j] = \#a[j]$. At the start of the program, $Q$ holds.

Let $P$ be the condition $\forall j \in U, \ a[j + 1] = \#a[j]$ (These conditions are the same as in the last theorem).

**Theorem 1.** \{Q\}C\{P\} (i.e., if $Q$ holds before the execution of the code in $C$, $P$ will hold after the execution of $C$. )

**Second proof (structured to match the program intentions).**

After the execution of $i := n - 1$, the variable $i$ will contain the value $n - 1$.

Form an invariant of the loop. Use the invariant $I = \Gamma(U(i))$.

or, writing out the definition of $\Gamma$,

\[
I = \forall j \in U \setminus U(i), \ a[j] = \#a[j] \\
\land \forall j \in U(i), \ a[j + 1] = \#a[j]
\]

(2.7)
With $i$ bound to $n - 1$, this reduces to $Q$, since $n - 1 = \sigma$, and by definition of the function $U$, $U(\sigma) = \emptyset$, and thus the second conjunct is vacuous.

Now we verify that $I$ is indeed an invariant of the loop.

Within the loop, $i \in U$, and by definition of $U$, $i \not\in U(i)$ always. So $i \in U \setminus U(i)$, hence $a[i] = \#a[i]$, after execution of the line $a[i + 1] := a[i]$, we have $a[i + 1] = \#a[i]$. Along with $I$, this implies

$$\forall j \in U \setminus U(i), \quad a[j] = \#a[j]$$

$$\wedge \forall j \in U(i) \cup \{i\}, \quad a[j + 1] = \#a[j] \quad (2.8)$$

But from the definition of $U(i)$ (eq. 2.5), $U(i) \cup \{i\}$ is just $U(\nu(i))$, so we have

$$\forall j \in U \setminus U(i), \quad a[j] = \#a[j]$$

$$\wedge \forall j \in U(\nu(i)), \quad a[j + 1] = \#a[j] \quad (2.9)$$

Since $U(i) \subseteq U(\nu(i))$ always, $U \setminus U(i) \supseteq U \setminus U(\nu(i))$, we have a fortiori

$$\forall j \in U \setminus U(\nu(i)), \quad a[j] = \#a[j]$$

$$\wedge \forall j \in U(\nu(i)), \quad a[j + 1] = \#a[j] \quad (2.10)$$

which is just
\[ \Gamma(U(\nu(i))) \]  

(2.11)

After execution of \( i := i - 1 \), we replace all occurrences of \( i - 1 \) with \( i \). In order to make the structure of this proof match the intention structure of the program, we notice that this line is, in fact, \( i := \nu(i) \) (this is how \( \nu \) is defined; it is the function used to compute the next value of the navigator variable), so we can replace all occurrences of \( \nu(i) \) in 2.11 with \( i \), which leaves us with \( I \) at the end of the loop, as desired.

At the loop termination, we have \( i = k - 1 = \tau \); from the second conjunct of \( I \), this leaves, by definition of \( U \),

\[ \forall j \in U, \quad a[j + 1] = \#a[j] \]  

(2.12)

QED.

To see that this proof has the appropriate structure, we will now attempt to modify it to form a proof of correctness for the while loop in figures 2.1 and 2.8. Notice that the while loop is not correct, hence the argument for its correctness cannot be a valid proof. \(^8\)

Let \( C \) be the code for the while loop in figures 2.1 and 2.8, in which the identifier \( a \) is an array, and the identifiers \( k \) and \( n \) are indices into the array, with

\(^8\)I will use the words 'argument' and 'conjecture' to refer to any line of reasoning about a program, even if it is faulty, and its result. In the case that the argument is valid, it becomes a 'proof', and the conjecture a 'theorem'.
$k < n$. Denote the initial values of the array $a$ as $\#a$.

For this while loop, the navigator starts at $k$, and finds its next value by adding one to the index. That is, $\nu(i) = i + 1$, $\sigma = k$, $\tau = n$, and thus $U = [k..n-1]$, and $U(i) = \{j|k \leq j \leq i-1\}$, so $U \setminus U(i) = \{j|i \leq j \leq n-1\}$.

Let $Q$ be the condition $\forall j \in U$, $a[j] = \#a[j]$. At the start of the program, $Q$ holds.

Let $P$ be the condition $\forall j \in U$, $a[j + 1] = \#a[j]$ (These conditions are the same as in the last theorem).

Conjecture 2 (i.e., if $Q$ holds before the execution of the code in $C$, $P$ will hold after the execution of $C$. )

Argument.

After the execution of $i := k$, the variable $i$ will contain the value of $k$.

Form the loop invariant. This is the same loop invariant as before, with the appropriate definitions of $\nu$ and $U$.

$$I = \Gamma(U(i))$$ (2.13)

$\Gamma$ is the same as before, so

$$I = \forall j \in U \setminus U(i), \quad a[j] = \#a[j]$$

$$\land \forall j \in U(i), \quad a[j + 1] = \#a[j]$$ (2.14)
Since $U(k)$ is empty, and $i = k$, $I$ is satisfied at the start of the loop by $Q$.

As in previous proofs, we try to show that $I$ is an invariant by working through the semantics of the loop line by line.

After execution of $a[i + 1] := a[i]$, we have, since $a[i] = \#a[i]$, $a[i + 1] = \#a[i]$.

So $\forall j, \ i + 2 \leq j \leq n - 1, \ a[j] = \#a[j]$, i.e., $\forall j \in U \setminus U(i + 2)$, or, in $\nu$ notation, $\forall j \in U \setminus U(\nu(\nu(i))), a[j] = \#a[j]$.

Also, $\forall j \in U(\nu(i)), \ a[j + 1] = \#a[j]$.

After $i := i + 1$, we can replace $\nu(i)$ with $i$, so we have

\[
\forall j \in U \setminus U(\nu(i)), \quad a[j] = \#a[j]
\]

\[
\land \forall j \in U(i), \quad a[j + 1] = \#a[j]
\]

which is not $I$ (there is still a $\nu$ left over in the first conjunct).

Of course, the problem occurred at the line $a[i + 1] := a[i]$. The problem arose in this proof, but not in theorem 1, because the loop invariant in theorem 1 made no commitment to the contents of $a[i + 1]$ (see eq. (2.1); the first clause commits values of $a[j]$ for $j \leq i$, while the second clause commits values for $j \geq i + 2$). In the faulty argument for conjecture 2, there is a commitment to $a[i + 1]$, that is, in the first clause, $i + 1 \in U \setminus U(i)$, since $U(i)$ is the set of values strictly smaller than $i$. That is, when the line $a[i + 1] := a[i]$ was executed, the array cell $a[i + 1]$ was not empty. This is the also the description of why the program does not work; the
third line clobbers the values of the array before they are used.

2.4 Representing a Proof According to Intentions

In order to automate this process of customizing the proof to a new program, I use the functional representation to mediate between the proof and the program. The FR plays two roles in this mediation:

- The structure of the proof is recorded in the functional representation. That is, the roles played by the navigator and collector in the entire loop are recorded in the FR. This includes a description of how the terms in the proof (such as \( \nu, \sigma \) and \( U \)) relate to parts of the functional representation (the various functions of the navigator). In turn, the FR shows how the functions of the navigator relate to fragments of program code.

- The functional representation at any level corresponds to the part of the proof which does not change when different choices are made at a lower level. For example, the inductive forms of the proofs above are the same, since they are both loops. The programs differ only in choice of navigator, so the proofs are the same at the level of the loop. The functional representation of the loop is common for both programs.

In order to achieve the first item, I have to show how parts of the proof correspond to the various constructs in the FR. I will show how this works by using
Device shift loop

Function induct
If \( \forall j \in U, a[j] = \#a[j] \)
ToMake \( \forall j \in U, a[j + 1] = \#a[j] \)
By initialize-and-loop
Behavior initialize-and-loop:

\[
\begin{align*}
T & \quad \text{Using Function start of navigator} \\
\Gamma(U(?navigator)) & \quad \text{Using Function cover of shift loop} \\
\Gamma(U(?navigator)) ? ?navigator = r & \\
\text{Function cover} \quad \text{If } \Gamma(U(?navigator)) ? ?navigator = \#nav \in U \\
\text{ToMake } \Gamma(U(?navigator)) ? ?navigator = \#nav & \\
\text{By cycle} & \\
\text{Behavior cycle} & \\
\end{align*}
\]

Figure 2.11: Complete Functional Representation of the Shift Loop
Devices \textit{navigator}

**State Definitions**

\[ \sigma = n - 1, \tau = k - 1, \]
\[ U(i) = \{ j \mid i + 1 \leq j \leq n - 1 \}, U = \{ j \mid k \leq j \leq n - 1 \} \] [order=dec]
\[ \sigma = k, \tau = n, \]
\[ U(i) = \{ j \mid k \leq j \leq n - 1 \}, U = \{ j \mid k \leq j \leq n - 1 \} \] [order=asc]

**Function \textit{start}**

\textbf{If} \( T \)

\textbf{ToMake} \( ?n a v i g a t o r = \sigma \)

\textbf{By e.g.,} \( ?n a v i g a t o r := n - 1 \) \[\text{order=dec, type=while}\]
\( ?n a v i g a t o r \) from \( n - 1 \) \[\text{order=dec, type=for}\]
\( ?n a v i g a t o r := k \) \[\text{order=asc, type=while}\]
\( ?n a v i g a t o r \) from \( k \) \[\text{order=asc, type=for}\]

**Function \textit{next}**

\textbf{If} \( ?n a v i g a t o r = ?x \)

\textbf{ToMake} \( ?n a v i g a t o r = \nu(?x) \)

\textbf{By e.g.,} \( ?n a v i g a t o r := ?n a v i g a t o r - 1 \) \[\text{order=dec, type=while}\]
\( ?n a v i g a t o r := ?n a v i g a t o r + 1 \) \[\text{order=asc, type=while}\]
by \(-1\) \[\text{order=dec, type=for}\]
\( \text{NIL} \) \[\text{order=asc, type=for}\]

**Function \textit{check}**

\textbf{If} \( ?n a v i g a t o r \in U \)

\textbf{ToMake} \( \langle ?n a v i g a t o r \in U \rangle \)

\textbf{By e.g.,} \( ?n a v i g a t o r \geq k \) \[\text{order=dec, type=while}\]
\( ?n a v i g a t o r \leq n - 1 \) \[\text{order=asc, type=while}\]
\( k \) \[\text{order=dec, type=for}\]
\( n - 1 \) \[\text{order=asc, type=for}\]

Devices \textit{collector}

**State Definitions:**

\[ \Gamma(S) = \forall j \in U \setminus S, \quad a[j] = \#a[j] \]
\[ \land \forall j \in S, \quad a[j + 1] = \#a[j] \]

**Function \textit{gather}** \textbf{Depends On} \(?x\)

\textbf{If} \( \Gamma(U(?x)) \land (?x \in U) \)

\textbf{ToMake} \( \Gamma(U(\nu(?x))) \)

\textbf{By e.g.,} \( a[?x + 1] := a[?x] \)

\textbf{Provided} \( \text{(empty } a[?x]\text{)} \)

Figure 2.12: Complete Functional Representation of the navigator and collector
the same example of the shift loop. The complete functional representation of the shift loop appears in figures 2.11 and 2.12. In the following, I will show excerpts from the proof of theorem 1 on page (page 36) and excerpts of the functional representation in figures 2.11 and 2.12.

From the state definitions in figure 2.12 we have

\[ \Gamma(S) = \forall j \in U \setminus S, \quad a[j] = \#a[j] \]
\[ \land \forall j \in S, \quad a[j + 1] = \#a[j] \] \hspace{1cm} (2.16)

\[ U = \{ j | k \leq j \leq n - 1 \} \]
\[ \sigma = n - 1, \quad \tau = k - 1 \quad \nu(i) = i - 1 \]
\[ U(i) = \{ j | i + 1 \leq j \leq n - 1 \} \] \hspace{1cm} (2.17)

The \textit{induct} function of the shift loop

\textbf{Function \textit{induct}}

\textbf{If} \ \forall j \in U, \quad a[j] = \#a[j] \\
\textbf{ToMake} \ \forall j \in U, \quad a[j + 1] = \#a[j]

---

\(^6\)All the notation in these figures has been described, except for the angle brackets (\(\langle\rangle\)) near the bottom of figure 2.11. These are defined in more detail in Appendix I. For the purposes of this discussion, they can be ignored.
corresponds to the statement of the proof of theorem 1:

Let \( Q \) be the condition \( \forall j \in U, \ a[j] = \#a[j] \). At the start of the program, \( Q \) holds.

Let \( P \) be the condition \( \forall j \in U, \ a[j + 1] = \#a[j] \)

**Theorem 1.** \( \{Q\}C\{P\} \) (i.e., if \( Q \) holds before the execution of the code in \( C \), \( P \) will hold after the execution of \( C \).)

Notice that \( P \) and \( Q \) are just the **ToMake** and **If** fields of the **induct** function, respectively.

The behavior **initialize-and-loop** corresponds to the rest of the proof. The first link,

\[
\begin{array}{c}
T \\
\text{Using Function } \text{start} \\
\text{of } \text{navigator} \\
\Gamma(U(?navigator))
\end{array}
\]

refers to the function **start** of the navigator.

**Function start**

**If** \( T \)

**ToMake** ?navigator = \( \sigma \)

**By** e.g., ?navigator := \( n - 1 \) \[order=dec,type=while\]
and corresponds to the first line of the proof,

' ... After the execution of \( i := n - 1 \), the variable \( i \) will contain the value \( n - 1 \).

... '

The state which follows this link,

\[ \Gamma(U(?navigator)) \]

corresponds to the setting and verification of the loop invariant,

' ... Form an invariant of the loop. Use the invariant

\[ I = \Gamma(U(i)) \] (2.19)

or, writing out the definition of \( \Gamma \),

\[ I = \forall j \in U \setminus U(i), \ a[j] = \#a[j] \] (2.20)

\[ \land \ \forall j \in U(i), \ a[j + 1] = \#a[j] \] (2.21)

With \( i \) bound to \( n - 1 \), this reduces to \( Q \), since \( n - 1 = \sigma \), and by definition of \( U \), \( U(\sigma) = \emptyset \), and thus the second conjunct is vacuous. ...

The next link in initialize-and-loop

\[ \Gamma(U(?navigator)) \]

Using Function cover of shift loop

\[ \Gamma(U(?navigator)) \land ?navigator = \tau \]
refers to the *cover* function of the shift loop:

**Function cover**

\[
\text{If } \Gamma(U(?\text{navigator})) \land ?\text{navigator} = \#\text{nav} \in U \\
\text{ToMake } \Gamma(U(?\text{navigator})) \land ?\text{navigator} = \#\text{nav}
\]

which corresponds to the part of the proof which verifies that \( I \) is indeed an invariant of the loop, that is, if \( I \) is true before some iteration of the loop, \( I \) is also true after that iteration.

In order to determine which of the states in the behavior *cycle* is first, we simply determine which state we already know to be true. In this case, since we have just verified \( I \) for the current value of the navigator, it is the state at the top of the diagram for *cycle* in figure 2.11 (which is just \( I \), from eq. 2.20).

The first link in *cycle*,

\[
\Gamma(U(?\text{navigator}))
\]

refers to the function *check* of the navigator.
Function check

If \(?navigator \in U\)

ToMake \(\{?navigator \in U\}\)

By e.g., \(?navigator \geq k\) \([\text{order=dec, type=while}]\)

The angle-brackets around the state in the ToMake field of the function check indicates that the code corresponding to this function does not make the enclosed predicate true, but it verifies that it is true for future use in the proof. Hence, this corresponds in the proof to

'... Since \(i \in U\)...'

The next link is

\[
\Gamma(U(\nu(?navigator))) \leftarrow \langle ?navigator \in U \rangle
\]

Using Function gather of collector

which refers to the gather function of the collector, with \(?x\) bound to \(?index\)

Function gather

If \(\Gamma(U(?index)) \land (?index \in U)\)

ToMake \(\Gamma(U(\nu(?index)))\)

By e.g., \(a[?index + 1] := a[?index]\)
which, in turn, corresponds to the fragment of the proof which deals with the line \( a[i + 1] := a[i] \), and its conclusion, viz.:

"... after execution of the line \( a[i + 1] := a[i] \), we have \( a[i + 1] = \#a[i] \). Along with \( I \), this implies ..."

\[
\forall j \in U \setminus U(i), \quad a[j] = \#a[j]
\]

\[\wedge \forall j \in U(\nu(i)), \quad a[j + 1] = \#a[j] \quad (2.22)\]

"..."

Finally, the link

\[
\Gamma(U(?\text{navigator}))
\]

Using Function \textit{next} of \textit{navigator}

\[
\Gamma(U(\nu(?\text{navigator})))
\]

refers to the function \textit{next} of the navigator, i.e.,

\textbf{Function \textit{next}}

\textbf{If } ?\text{navigator} = ?x

\textbf{ToMake } ?\text{navigator} = \nu(?x)

\textbf{By e.g., } ?\text{navigator} := ?\text{navigator} - 1 \ [\text{order=dec, type=while}]
an corresponds to the part of the proof dealing with the semantics of the line
\( i := i - 1 \), or

' ... After execution of \( i := i - 1 \), we replace all occurrences of \( i - 1 \) with \( i \). ...'

\[ \forall j \in U \setminus U(\nu^{-1}(i)), \quad a[j] = \#a[j] \quad (2.23) \]

\[ \land \forall j \in U(i), \quad a[j+1] = \#a[j] \]

'...

Now we come full circle in the behavior cycle, by verifying the state we began
with, which is exactly \( I \).

Finally, returning to the last state of the diagram for initialize-and-loop, we
have \( I \) and that \( i = \tau \), which is stated in the proof, and finally, returning to the
original function induct of the loop, we have

\[ \forall j \in U, \quad a[j+1] = \#a[j] \quad (2.24) \]

in the ToMake slot, which is also the last line of the proof.

The functional representation does not replace the proof; there are some arg-
ments in the proof which do not appear in the functional representation. For
example, when we verify that the loop invariant holds at the start of the loop, the
functional representation tells us what the invariant should look like (i.e., \( \Gamma(U(i)) \)),
and when we will be in the position to prove it (after applying the semantics for
the line $i := n - 1$). It does not tell us how to prove that it holds (which is done by observing that $U(i) = \emptyset$ for $i = \sigma$, hence $I \equiv Q$, which has already been verified).

### 2.5 Consistency of a Functional Representation

In [27], Sembugamoorthy and Chandrasekaran raise the question of how to determine that a given functional representation is consistent. By this they are referring, for example, to the fact that the states of detailed behaviors may not correspond identically to the states of the enclosing functions. They point out in the concluding remarks that in order to determine consistency it is necessary to have ‘domain knowledge for interpreting state descriptions ...’ [27]. For example, from figure 2.11 we have the function `start` of the navigator, which is used by the function `induct` of the shift loop to reach the state described by $\Gamma(U(\text{?navigator}))$, even though the `ToMake` field of `start` states that it can be used to achieve a state described by `?navigator = \sigma`. In order to ensure the consistency of the representation, it is necessary to guarantee that the state $\Gamma(U(\text{?navigator}))$ will hold, given that `?navigator = \sigma`. In the case of the devices treated by Sembugamoorthy, since the states were interpreted only by the (human) user of the final system, there was no way to automatically verify the correctness or consistency of a representation, and without some semantics of the state descriptions, no systematic way to perform the verification by hand.

In figure 2.11, there is a semantic interpretation of the states, namely that the
'states' in the figures correspond to the intermediate formulas in a proof of correctness of a program, and hence have an interpretation in the formal semantics of the programming language. Since the functional representation of the program corresponds to a correctness proof, in order for it to be consistent, the proof must be valid. That is, we can specify the following consistency condition on a functional representation (in the following, as elsewhere, the word 'state' refers to the predicate that labels the state):

**Condition a.** A functional representation of a program is consistent exactly if

i. For every function, given the If clause and the context in which the function is used, it is possible to prove the initial state of each behavior that implements the function. Similarly, given the final state of each behavior, it is possible to prove the ToMake clause of the function.

ii. For every link of type 'UsingFunction', given the state which precedes the link and the context of the enclosing behavior, it is possible to prove the If clause of the function referred to by the link. Similarly, given the ToMake clause of that function, it is possible to prove the state which follows the link.

The consistency of the representation in figure 2.11 can be shown by examining all the appropriate links, functions and behaviors. For the function *induct* of the shift loop, it is easy to prove the first state of the behavior *initialize-and-loop,*
since it is T (which is already true, and hence needs no proof). The ToMake field, $\forall j \in U, \quad a[j] = a[j] + 1$ can be proven from $\Gamma(U(\text{navigator}))$ and the definitions of $U$ and $\Gamma$. The consistency of the first link follows since the state before the link (T) implies the If clause of the function start of the navigator (which is also T).

The ToMake clause of the function start of the navigator, $\text{navigator} = \sigma$ implies the next state of initialize-and-loop since $U(\sigma) = \emptyset$ (by definition of $U$), and the fact that $\forall j \in U, \quad a[i] = a[j]$, from the If clause of the function induct.

For the second link in the behavior initialize-and-loop, (Using Function cover of shift-loop), since it is possible for the function to be achieved even if it is not possible to derive the If clause of cover from the preceding states (e.g., in the case that the termination condition of a loop is already satisfied, so that it is unnecessary to examine the inside of the loop), I use an expanded version of criterion ii:

Let $S_1$ and $S_2$ refer to the states which precede and follow a link, respectively. Also let $P$ and $Q$ refer to the If and ToMake clauses of the function mentioned by that link, respectively.

iiia. For every link which refers to a behavior with a loop in it, $S_1 \rightarrow P \lor S_2$, and $Q \rightarrow P \lor S_2$.

Intuitively, the state which precedes the link ($S_1$) must either satisfy the If condition ($P$) to enable the function specified by the link, or, failing that, be able to guarantee the following state ($S_2$). Similarly, the ToMake condition ($Q$) must
either satisfy the following state \(S2\), or, failing that, enable the function to run again by satisfying the If condition \(P\).

So, in order to check the consistency of the representation for the function \textit{cover} of the shift loop, we have the following bindings for condition (iia):

\[
S1 = \Gamma(U(?navigator))
\]

\[
S2 = \Gamma(U(?navigator)) \land ?navigator = \tau
\]

\[
P = \Gamma(U(?navigator)) \land ?navigator = \#navigator \in U
\]

\[
Q = \Gamma(U(?navigator)) \land ?navigator = \nu(\#navigator) \hspace{1cm} (2.25)
\]

\[
(2.26)
\]

Given \(S1\), i.e., \(\Gamma(U(?navigator))\), we must show either \(S2\) or \(P\). If \(P\) does not hold, it must be that \(?navigator \notin U\), since \(S1\) guarantees \(\Gamma(U(?navigator))\). But if \(?navigator \notin U\), then it must be the case that \(?navigator = \tau\), and \(S2\) holds.

Given \(Q\), that is, \(\Gamma(U(?navigator)) \land ?navigator = \nu(\#navigator)\), if \(P\) does not hold, it must be that \(?navigator \notin U\), so \(navigator = \tau\), and \(S2\) holds.

Now we check the consistency of the link inside the \textit{cycle} behavior as before. The only problem is to determine which states will play the role of 'initial' and 'final' states in condition (i), since the behavior is a loop, and has no fixed beginning or end. For behaviors which are loops, we define the initial state and the final state to be identical, and they must be some state which can be proven given the If clause
of the function. Given this definition, the first part of condition (i) is satisfied, once an initial state is identified. In this case, the initial (and final) state is at the top of the figure, and is labeled ‘Γ(U(?navigator))’.

From the If clause of the function cover, we have that ?navigator ∈ U, so the If clause of the function check is satisfied. The Make clause of check is identical to the following state.

The If clause of the function cover is identical to the If clause of the gather function of the collector, and the Make clause of gather is identical to the next state. Since ?navigator = $\$navigator$, this implies Γ(U(ν($\$navigator$))).

The If clause of the function next is a pattern; since ?navigator does have a value, it is true with the binding of ?x to $\$navigator$. Since ?navigator = ν($\$navigator$), and from the last state we have Γ(U(ν($\$navigator$))), we have Γ(U(?navigator)), which is the final state of the behavior.

From the final state, and the assertion (from the last paragraph) that ?navigator = ν($\$navigator$), we have exactly the Make clause of the function cover.

Since all links and functions have been shown to be consistent, the entire functional representation for the shift loop is consistent.

2.5.1 Consistency and Different Description Languages

One of the motivations for the Functional Representation language was that devices and their components could potentially be described using very different languages.
This is even more important in programming than in ordinary device design, in that usual methodologies of program design suggest that the specification of a module be described independently of its implementation. One weakness that appeared in the original formulation of the Functional Representation ([27]) was that the connection between a state at a higher level of abstraction and a state at a lower level was left implicit in the Functional Representation diagrams. In [1], Allemang and Keuneke point out that phenomena that are represented by processes at one level of abstraction are treated as states at another. In [13], Keuneke proposes extensions to the Functional Representation language that allow for specifying the relationship between these states.

In the case of programs, the relation of abstract state to concrete state raises additional problems. Consider the example of an integer that is implemented by bit strings, using one’s complement notation. Figure 2.13 shows the functional representation for a function to negate such an integer.

The rules given in this section for consistency of a Functional Representation require that $x \in \mathcal{Z} \Rightarrow x = b_0 b_1 \ldots b_n$, and that $x' = \overline{b_0 b_1 \ldots b_n} \Rightarrow x' = -x$. This illustrates two problems with this notation. First, it overloads the use of the symbols $x$ and $x'$; that is, in the Function definition, they stand for integers, while in the behavior diagram, they stand for bit strings. More importantly, since nowhere is there given a description of how the bit strings correspond to integers, there is no way to prove the two conjectures above. The mechanism outlined by Keuneke in
Function *negate*

If $x \in \mathbb{Z}$

ToMake $x' = -x$

By *invert*

**Behavior* *invert***:

$$x = b_0b_1 \ldots b_n$$

Using Function *induct* of inversion loop

$$x' = \overline{b_0b_1} \ldots \overline{b_n}$$

Figure 2.13: A device for negating one's complement integers.

[13] is insufficient to solve this problem, since there is more than just a correspondence between states, but rather a schema for state correspondences that relates the integers to their bit string representations. Such a schema can be described using a 'representation function' as in Alphard [8]. Figure 2.14 shows how the representation function is attached to the functional representation. Notice that the details of the representation function has to be associated with the behavior, since different behaviors will implement different representations, but is mentioned in the function slots (If and ToMake) without reference to the particulars of any implementation. Once this is done, conditions (i) and (ii) above become identical to the proof conditions 3 and 4 given for Alphard in [8], page 24.
Function negate

If $\text{rep}(x) \in \mathbb{Z}$

To Make $\text{rep}(x') = -\text{rep}(x)$

By invert ($\text{Rep} (1 - 2b_0) \sum_{i=2}^{n} 2^i$)

Behavior invert:

\[
x = b_0 b_1 \ldots b_n
\]

Using Function induct of inversion loop

\[
x' = \bar{b}_0 \bar{b}_1 \ldots \bar{b}_n
\]

Figure 2.14: A device for negating one's complement integers, with representation function.
The representation function is not used by Dudu, since Dudu's job is to reconstruct a proof of correctness, not to build it in the first place. The representation function is used while determining the consistency of the representation, before it is used by Dudu. If there are limitations on the use of a representation function (say, the integer must be less than $2^{15}$), such limitation can be placed in the appropriate **Provided** slot. When Dudu reconstructs the proof, it can make use of such a proviso.

2.6 Ramifications of Consistency

In section 2.4 I showed how a functional representation of a program is related to a proof of correctness of the program. In the last section, I showed that the representation given in figure 2.11 for the shift loop is consistent. However, it does not follow that a proof built from figure 2.11 must be correct. This is because the functional representation does not generate the entire proof. The states in the functional representation indicate what intermediate formulas will be used in the proof, but not how to get from one to another. For example, in the proof of the shift loop, the functional representation told us that the initialization of the variable $i$ should result in being able to prove the loop invariant (2.6), but does not tell us how to prove this. The complete proof refers to the program language semantics of the lines of code which are actually used in the program, while the discussion of consistency does not take program language semantics into account.
In order to complete the proof, it is necessary to prove that the actual lines of code in fact satisfy the conditions specified by the functional representation.

Often it might be possible to prove from the program semantics of the sample lines that the appropriate functions will be achieved. For example, again in figure 2.11, the ToMake field of the function start of the navigator contains the predicate ?navigator = σ. For [order=dec], we have, from the state definitions of the navigator (also figure 2.11) that σ = n − 1. The programming language semantics of an assignment statement guarantee that after execution of ?navigator := n − 1, it will be the case that ?navigator = n − 1. Since σ = n − 1, we can prove that ?navigator = σ, as desired. A similar argument holds for the other examples of the function start.

If we can construct such a proof for every code sample in the representation, then we can say that a proof constructed from the functional representation of a program is a valid correctness proof for that program. However, for more interesting representations, this may not be the case. Consider the function gather of the collector (see figure 2.11). In order to verify this function, we need to prove that:

\[ \{ \Gamma(U(\nu(x))) \wedge \sigma \in U \} ; a[\nu(x) + 1] := a[\nu(x)] ; \{ \Gamma(U(\nu(\nu(x))) \} \]

That is, if the If clause is true before the execution of the assignment statement, then the ToMake clause will be true afterwards. For the bindings of \( \Gamma, \sigma, \tau \) and \( U \) for [order=dec], this reduces to
\[
\begin{align*}
\{ & \forall j \in [k..x] \quad a[j] = \#a[j] \\
& \wedge \forall j \in [x + 1..n - 1] \quad a[j + 1] = \#a[j] \}
\end{align*}
\]
\[a[x + 1] := a[x]; \] \hspace{1cm} (2.27)
\[
\begin{align*}
\{ & \forall j \in [k..x - 1] \quad a[j] = \#a[j] \\
& \wedge \forall j \in [x..n - 1] \quad a[j + 1] = \#a[j] \}
\end{align*}
\]

Since \(?x \in U\), we have that \(a[?x] = \#a[?x]\). Using the usual semantics of assignment, we replace all occurrences of \(a[?x]\) in the antecedent with \(a[?x + 1]\), and simplify. The only such occurrence is in the first conjunct, when \(j = ?x\). Replacing \(a[?x]\) with \(a[?x + 1]\) in the antecedent we form the proposed consequence

\[
\forall j \in [k..x - 1] \quad a[j] = \#a[j]
\]
\[
\wedge \quad a[?x + 1] = \#a[?x] \] \hspace{1cm} (2.28)
\[
\wedge \forall j \in [?x + 1..n - 1] \quad a[j + 1] = \#a[j]
\]

from which we easily derive

\[
\forall j \in [k..x - 1] \quad a[j] = \#a[j]
\]
\[
\wedge \forall j \in [?x..n - 1] \quad a[j + 1] = \#a[j] \] \hspace{1cm} (2.29)
which is the desired consequence.

For the bindings of $\Gamma, \sigma, \tau$ and $U$ for [order=asc] we have

$$
\{ \forall j \in [x..n-1] \quad a[j] = \#a[j] \\
\forall j \in [k..x-1] \quad a[j + 1] = \#a[j] \}
$$

; $a[x + 1] := a[x]$;

$$
\{ \forall j \in [k..x] \quad a[j] = \#a[j] \\
\forall j \in [x + 1..n - 1] \quad a[j + 1] = \#a[j] \}
$$

We perform the same substitution, $a[x+1]$ for $a[x]$; notice that this substitution is only valid if the antecedent is quantified over non-empty sets, i.e., $[k..x-1]$ must not be empty, i.e., $k \neq x$:

$$
\forall j \in [x + 1..n - 1] \quad a[j] = \#a[j] \\
\forall j \in [k..x - 2] \quad a[j + 1] = \#a[j]
$$

This is a contradiction, since $a[x+1]$ can have only one value. 7 Hence, we

---

7Unless, of course, $\#a[x-1] = \#a[x]$. For this to never be a contradiction, $\forall x \in U, \#a[x-1] = \#a[x]$, i.e., all the initial values in the array are the same. For this case, of course, the program is correct.
must conclude that $k = x$. In fact, if we check back to equation 2.30, we can verify that 2.30 can be true only if $k = x$.

In order to verify the consistency of the function $\textit{gather}$ of the collector, it is necessary to show, as we did above for the function $\textit{start}$ of the navigator, that all the code samples are guaranteed to satisfy the $\textit{If}$ and $\textit{ToMake}$ fields of the function. This cannot be done for the collector, because the $\textit{If}$ and $\textit{ToMake}$ fields refer to values which are defined outside of the collector, namely $\nu$ and $U$, which are defined by the navigator.

Although, in general, it is not possible to prove the consistency of a function of this sort, it is possible to specify a proviso which will give an indication of whether or not such a proof is likely to exist. For instance, in the case of the function $\textit{gather}$, the problem with the counting up case is that the value of $a[?x]$ is not committed by the description of $\Gamma$, whereas for the counting down version, it is. Hence the proviso that $a[?x]$ not be committed is an indication that a proof might be possible. Furthermore, if one were to patch a program which violates this proviso so that it satisfies it, then there is a good reason to believe that the patched program can be proven correct. That is, the proviso can supply a suggestion for how to correct a faulty program. This proviso appears in figure 2.11 as ‘$\textbf{Provided (empty } a[?x])$’.

The proviso serves as a more easily checked approximation to the condition which must be verified to guarantee a proof. A non-approximate proviso would be exactly that predicate which would guarantee that the function will be satisfied
by the code. To use such a proviso would be as much work as to use the If and 
ToMake fields, along with the programming language semantics of the program 
itself. Since a proviso is intended to be more easily checked than the consistency 
of the If and ToMake fields, there is a preference for simple provisos.

A proviso can also be used to describe why a program fails to be correct. In the 
case of the proviso for the function gather above, it is the case that if the proviso 
does not hold, then the program cannot be correct (since otherwise some value in 
the array will be lost). If a program does not satisfy this proviso, then the failure 
could be described by reporting that the proviso is not satisfied. For this reason, 
there is a preference for provisos that are easily understood by people, and which 
can be easily related to the program code.
CHAPTER III

Generic Representations

The representation presented in figure 2.11 stands for a large number of programs and proofs of correctness of those programs. If we say that a functional representation 'stands for' a program exactly if it is possible to construct (as was demonstrated in section 2.4) a correct proof of the program from the functional representation, then this representation stands for every correct program in figures 2.1 and 2.8. The representation in figure 2.11 captures the understanding that is common among all these programs. It has nothing to say about programs written for other tasks, say for computing a running total or performing an insertion sort. Nevertheless, many aspects of the shift programs are shared by these other programs. For instance, all of these involve loops, the correctness of which can be proven by an inductive argument involving a loop invariant (more specifically, that it can be formed as $\Gamma(U(n))$ for navigator $n$). Furthermore, we even know something about the structure of the loop invariant. It should be possible to represent the aspects of figure 2.11 that refer to loops in general separately from those...
aspects that are particular to the shift loop. Not only does this provide greater clarity in the representation, but it also helps describe how to construct such a representation. In the preceding chapter we have shown how to represent the shift loop as if it were the only loop ever designed; this means that we had to construct the representation from scratch. By using more generic components (such as a generic loop), we can represent programs as instances of the class of generic loops.

So far we have only considered certain program fragments to be devices. But there are other sorts of components from which a program is constructed. For example, data structures do not correspond to code fragments, yet they have a distinct teleology. They have certain roles to play in larger structures or programs, and certain services they can provide. In turn, especially in a programming paradigm which supports abstract data types, data structures can call upon the services of fragments of code, that is, a data structure could have a code fragment as a component. In this sense, data structures are devices in just the same way that program fragments are.

Finally, there are concepts that play a role in program understanding and that are at a different level of abstraction than program code. An example of such a concept is a data record. Programs are often seen as shuffling around some data records (for example, in a program that sorts a list of employee names), without combining any two records, creating a record or destroying any records. In the
context of such records, assignments are interpreted with a semantics different from
the usual semantics of replacing one value with another. Rather, since records are
to be passed from place to place, the assignment is used as the primitive operation
for moving records.

In the remainder of this chapter, I will describe a generic loop, a generic data
structure, and a generic data record, and show how the shift loop can be con­
structed using these generic components.

3.1 Representing Generic Devices

The functional representation as described in [27] does not allow for the represen­
tation of generic devices and their instantiations. Thus it is necessary to extend
the functional representation language to allow for generic devices.

The approach taken here for specifying generic devices is to specify them as if
I were specifying a specific device, with variables standing in for the features of
the specific device that are not to be specified in the generic. So, for example,
the specific navigator given in figure 2.12 covered the set \([k..n - 1]\) by starting at
\(n - 1\), subtracting 1 during each iteration, until it reached \(k\). A generic navigator
will cover a set \(U\) by starting at \(\sigma\), applying some next function \(\nu\) to each item,
until it reaches \(\tau\). One problem with such an approach is that there are constraints
that hold for the particular values \([k..n - 1], n - 1, k, -1\) that are important for
the appropriateness of the navigator. One such constraint is that the set covered
by an iterator starting at \( \sigma \), moving by \( \nu \), ending at \( \tau \) in fact covers the set \( U \). In a sophisticated type definition system, like OBJ ([3]), these constraints would be specified as a *theory* that governs any particular navigator, which would make it possible to automatically determine that a particular specification satisfies the constraints supplied in the class. Dudu requires this check of consistency to be done by the designer of the functional representation. The motivation for this representation of generic devices is to allow specific devices to inherit defaults from the generic classes; thus I have used a simple default inheritance model of generic devices. The rest of this section will describe the notation I will use for specifying classes of devices, and filling in the details of instantiations of them. Figures 3.2 and 3.3 show uses of all the constructs to be described here for specifying generic devices, and figures 3.4 and 3.5 show uses of all the constructs for instantiating them.

A class of devices is specified in exactly the same way as the particular devices in the last chapter. Any of the following parts of the device representation may be left as variable, to be specified by a particular instance of the class.

- Variables to be used in state descriptions. These must be listed in the section ‘State Definitions’ at the head of the class definition. The State Definitions section may also include any constraints that must hold among these variables; e.g. the generic navigator must specify that \( \sigma, \tau, \text{ and } \nu \) bear the appropriate relationship to the set \( U \) that the navigator must cover. State
variables that may participate in constraints in higher level devices (such as the variable $U$ in these examples) are listed with the keyword \texttt{Depends On}.

- Rather than calling on functions of specific devices, the generic device may call upon functions of a device class. Any instantiation of this class must call upon the indicated function of a particular device of the named class.

- Slots of functions that are normally required may be left unfilled. The required slots (\texttt{If}, \texttt{ToMake}, \texttt{By}) must be filled in by the instance.

In order to specify a particular instance of a class, it is necessary to specify these three things, that is,

- A choice must be made for the variables mentioned in the \texttt{State Description}. This choice must be consistent with the constraints listed in the \texttt{State Definitions};

- Any device classes mentioned in the class definition must be replaced by instances of the appropriate class; and

- All details of any functions mentioned in the class definition must be specified in the instance.

An instance of a class is given a name, and is specified by the keywords \texttt{is a} (or \texttt{is an}) and \texttt{binding}. For example, \texttt{Device index is a navigator binding}
$U = [1..n]$ defines a new device, called \textit{index}, that inherits various aspects from the generic \textit{navigator} (e.g., that it has functions \textit{start}, \textit{check} and \textit{next}, and must define variables $\sigma$, $\tau$, and $\nu$), with the parameter $U$ bound to the set $[1..n]$. To bind variables for a particular function, the keyword \textbf{binding} can also appear in conjunction with the annotation \textbf{Using Function} on links in behavior diagrams.

The keyword \textbf{with} is used to specify functions of instances beyond what is inherited from the class. For example, the device class \textit{navigator} insists that all navigators have a function \textit{start}, but does not specify what the function \textit{start} must be. A particular navigator must specify this; for the navigator in figure 3.7, this is done with \textbf{Device index is a navigator with Function start} etc.

These bindings can be used together in ways analogous to the notion of 'Mixins' in object oriented programming; for instance, a \textit{Record} can be used as a \textit{collector} if the appropriate functions \textit{initialize} and \textit{gather} are specified. As a \textit{Record}, it will inherit the function \textit{move}, which can help provide the \textit{collector} functions. So, for example, we can specify that the \textit{gather} function (required for a device to be a \textit{collector}) is to be identified with the \textit{move} function (inherited from the \textit{Record}) under certain bindings, as shown in figure 3.1.

Finally, it is also possible that the role a component plays in the working of a device is that it provides a value for one of these parameters. For example, the \textit{move} function of the \textit{Record} requires two locations (to move from and to); these locations can be provided by an array. The keyword \textbf{is given by} specifies this
...@collector = Device row
    is a Record
    with gather = @move
        binding
        @loc1 = source
        @loc2 = destination
    with initialize = NIL

Figure 3.1: Device row is both a collector and a Record.

use, so we can reference function move binding @loc1 is given by location of array
binding @i = index. All of these constructs are used in the specifications of the
generic devices to follow, and the example instance of the shift loop.

3.2 Generic Loops

The shift loop described in the preceding chapter applied a process to an entire
set of elements. The process had the property that it was possible to apply it to
the entire set by acting on each item one at a time. A loop whose job it is to
compute a total of a set of numbers is also of this type, since a total for a set can
be computed by keeping a running total, adding one element at a time. Even the
top level loop of an insertion sort can be viewed in this way, since its job is to
put each item in its correct place, which it does item by item. Loops for counting
elements and finding extreme elements can be viewed the same way. I will call
loops of this kind ‘total loops’ (since they apply some function to an entire set,
coming up with a sort of total of the items).
The specification for total loops will resemble the representation given above for the shift loop, without the details which are specific to the shift itself. That is, there is a *navigator*, which supplies elements of a set $U$ to be processed one by one, and a *collector*, which applies some function $\Gamma$ to the set, by applying the function to the items one by one. In terms of the Functional Representation, the *navigator* has functions *start*, *next* and *check* which give the first element of the set $U$, find the next element of the set, and check to see whether the entire set has been delivered, respectively. The *collector* has functions *gather* and *initialize*, which specify how to compute the function $\Gamma$ of $U$, given the elements of $U$ one by one (*initialize* gives $\Gamma(\emptyset)$). Figures 3.2 and 3.3 show the complete functional representation of the generic total loop. This looks exactly like the representation in figures 2.11 and 2.12, without the code samples for the specific navigator or the specific collector.

Another difference between the representation of the generic total loop and the particular shift loop is in the representation of the behavior *initialize-and-loop*. In figure 2.11, the first link referred only to the function *start* of the navigator. In figure 3.2, there are two links between the first state ($T$) and the second state ($\Gamma(U(?navigator))$), one labeled with the function *start* of the navigator, and the other labeled with *initialize* of the collector. In general, total loops can be expected to initialize both the collector and the navigator (consider the cases of a running total, in which the sum is usually initialized to 0, and a maximum, in which the
Devices total loop Depends on $P$, $U$

Function *induct*

- If $P(\emptyset)$
- ToMake $P(U)$
- By *initialize-and-loop*

Behavior *initialize-and-loop*:

1. **Using Function** initialize of collector
2. **Using Function** start of navigator
3. $\Gamma(U(?navigator))$
4. **Using Function** cover of shift loop
5. $\Gamma(U(?navigator)) \wedge ?navigator = r$

Function cover

- If $\Gamma(U(?navigator)) \wedge ?navigator = \%nav \in U$
- ToMake $\Gamma(U(?navigator)) \wedge ?navigator = \%nav$
- By *cycle*

Behavior *cycle*

1. $\Gamma(U(?navigator))$
2. **Using Function** next of navigator
3. **Using Function** check of navigator
4. $\Gamma(U(\nu(?navigator))))$
5. $\Gamma(U(?navigator)) \wedge ?navigator \in U$
6. **Using Function** gather of collector
7. **binding** $\text{@index} = ?navigator$

Figure 3.2: Functional Representation of the Generic Total Loop
**Devices** *navigator* **Depends On** *U*

**State Definitions**

\[ \sigma, \tau, \nu, \]

\[ U(e) = \begin{cases} \emptyset & \text{for } e = \sigma; \\ U(\nu^{-1}(e)) \cup \{\nu^{-1}(e)\} & \text{for other } e \in U; \\ U & \text{otherwise}. \end{cases} \]

\[ U(\tau) = U \]

**Function** *start*

If T

ToMake ?navigator = \sigma

**Function** *next*

If ?navigator = ?x

ToMake ?navigator = \nu(?x)

**Function** *check*

If ?navigator \in U

ToMake (?navigator \in U)

**Devices** *collector**

**Depends On** *index*

**State Definitions:**

\[ \Gamma(S) \]

**Function** *gather*

If \( \Gamma(U(?index)) \land (?index \in U) \)

ToMake \( \Gamma(U(\nu(?index))) \)

**Function** *initialize*

If T

ToMake \( \Gamma(U(\sigma)) \)

Figure 3.3: Functional Representation of generic navigator and collector. The **Depends On** keyword constrains the use of the device; for example, the *navigator* will navigate the set *U*. In order to do this, any particular navigator must define \( \sigma, \tau, \) and \( \nu, \) (and hence \( U, \) which is defined in terms of these), so that \( U(\tau) = U. \) These definitions determine the detail of the functions of the *navigator* as shown.
partial maximum is initialized to some very low value). In the case of the shift loop, the collector, which was the set of values in the array, was already initialized at the start of the loop. The split (where there are two links from on state) in the diagram in figure 3.2 means that both functions start and initialize are to be called upon to achieve the following state, but the order in which they occur is not specified in this behavior. In this case, this reflects the fact that the two initializations can occur in either order.

The overall outline of the proof of correctness of a loop can be specified independently of the specifics of the navigator and collector. For example, that the loop invariant is formed by composing $\Gamma$ and $U$ as shown in both figures 3.2 and 2.11 is independent of the choice of $U$ and $\Gamma$. Even the outline of the proof, namely that one will construct a loop invariant, which is to be verified before the beginning of the loop, and shown to hold through any iteration of the loop, and hence will hold at the end of the loop when the guard condition fails, can be taken from the representation of the loop, without regard to the specific navigator and collector used for any particular loop.

It is not the case that every loop can be conveniently represented as a total loop. For instance, a loop that searches an array for a value, and makes available an index to the array cell which contains the value cannot be represented easily as a navigator and collector as outlined above. The index does not, and is not intended to, cover the entire array. There is no function which is to be applied to
the entire set of array indices, which can be computed from the indices one by one. Appendix I shows the functional representation of a generic search loop.

While it may be possible to make a representation of a loop which could be used to represent any loop at all, one of the main motivations of knowledge based systems in general, and the functional representation in particular, is to make knowledge specific to some domain available for solving otherwise intractable problems. In this case, we have seen how the loop invariant can be constructed from two relatively simple (and easy to specify) functions attached to the navigator and collector. It is not surprising that such a simple solution is not a general solution to the problem of finding appropriate loop invariants. It is possible to determine from the specification of the programming task when the collector/navigator model is appropriate (e.g., for a maximum, it is necessary to examine every item in the array, and compare each one to the current maximum; similarly for a total, every item must be summed). The functional representation of the total loop breaks the loop invariant into two more easily specified pieces, and tells how to combine them to get the working invariant.

3.3 Generic Data Structures

The description of the shift loop in figure 2.11 refers to an array of values; in fact, the shift loop does not make any sense except in the context of an array of values, since its specification is that it moves values through an array. The only reference
Devices array

Depends On I  (* this is the set of indices of the array *)

State Definitions
L a finite set of locations,
ϕ : I ↔ L

Function location
Depends On i
If i ∈ I
ToMake ϕ(i)
By e.g., ?array[i]

Function value
Depends On i
If i ∈ I
ToMake v(ϕ(i))
By e.g., ?array[i]

Figure 3.4: Functional Representation of generic array. ϕ is a bijection which connects indices to locations. The details of what it means for something to be a location are not important for any of the examples in this work. All we need to know about them is that the description of a Record (see next subsection) requires that some of its parameters be locations. v(ϕ(?i)) refers to the value stored in the location ϕ(?i)

to the array that we see in figure 2.11 is in the sample code for the collector, which is given as ?collector[?x + 1] := ?collector[?x].

While arrays can be used for many purposes, to specify the generic array it is only necessary to specify the functions which are shared by all arrays. An array indexes storage locations. The functions of the generic array (as shown in figure 3.4) are to provide, given the index, a value from a location, or to provide the location, to be assigned to another value. These function are called value and location respectively.

A particular array may have other functions than these, which call upon the
functions value and location. Later we will see how an ordered set can be built as a special case of an array, in which special functions for inserting into the ordered set call upon the generic functions of the array.

### 3.4 Other Generic Devices

The buggy programs in the last chapter (e.g., figure 2.1, center) were buggy because there was a point at which they assigned a value to a location which held another value which would be needed later. When time came to use the previous value, it had been clobbered by the assignment. This was reflected in the proofs from the last chapter by the fact that in the corresponding loop invariant, some array element was doubly committed.

For many applications, a programming metaphor of records and files is appropriate for describing the computation. Records can be pulled from the file cabinet, manipulated, and put back into the cabinet, at the same or different locations from whence they were pulled. Records can be compared, shuffled and updated. Two records cannot be kept in the same place at the same time (impenetrability). Seldom are many records combined into one, so that the old records are no longer kept. Examples of problems that match this metaphor well are payroll (updating employee records) and job scheduling (process records), since two employees are seldom combined into one, and two jobs are seldom made into one. Tasks that involve sorting also use this metaphor. Arithmetic tasks often do not match this
Devices Record

Function move Depends On locations loc1, loc2

If $P(?loc1)$
ToMake $P(?loc2) \land (empty?loc1)$
Provided (empty?loc2)
By e.g., $?loc2 := ?loc1$

Figure 3.5: Functional Representation for the generic device Record

metaphor; for a running total, the items (numbers) are combined without keeping the old numbers available.

The ordinary programming language semantics of an assignment statement is not sensitive to this distinction. If variable $x$ contains one record, it is perfectly acceptable to assign another record to it, say $x := y$. After this assignment, both $x$ and $y$ contain the record. But according to the semantics of the metaphor, the record has moved from $y$ to $x$, and is now being stored in $x$ rather than $y$. It is now acceptable to use $y$ on the left-hand side of an assignment (since no record is being stored there), but not to use $x$ in such a way (since that would store two records in the same place).

The use of such a metaphor plays a role in program understanding; a program which makes use of this metaphor is understood differently than a program which does not. For this reason, in figure 3.5 I describe the generic device Record. A Record has the function move, which enforces the property of impenetrability; records may only be moved into empty positions, and when they are so moved, the position they came from becomes empty. This constraint is enforced by the
Provided and ToMake clauses in figure 3.5.

3.5 Example of use of Generic Objects: the shift loop

Figures 2.11 and 2.12 show the functional representation of the shift loop. Figures 3.2 and 3.3 show a generic loop. Figure 3.4 shows the functional representation of an array, and figure 3.5 shows the functional representation of a Record. Figures 3.6 and 3.7 show the shift loop, specified using these generic devices.

In an insertion sort, an outer loop examines each item in the input, decides where to put it into the part of the input which has already been sorted, and puts it there. If the sorted list is represented as a linked list, this insertion involves modifying the appropriate links to include the new element. If the list is represented as an array, this insertion involves moving some of the list out of the way, so as to accommodate the new item. The shift loop discussed here accomplishes exactly this task. Since the nature of this loop (and hence the way it is understood) changes dramatically when the data structure changes, the loop is represented as a sub-device of the array, in its role as Ordered Set. That is, the shift loop is a device which provides services to the array to allow it to carry out its functions (which are provided for the sort loop). Hence figure 3.6 begins with the device called Ordered Set, which is a special case of an array. The generic array does not have a function insert, which is required of the Ordered Set, so it is specified in figure 3.6 by reference to the shift loop.
Device Ordered Set

State Definitions

\[ S \] a set of Records,
\[ \psi : [1..n] \rightarrow S \] the ordering of the set.

is an array binding \( I = [1..n] \)

Function insert

Depends On \( k \in [1..n], \) item a Record

If \( k \in [1..n], \)

ToMake \( \psi' \) and \( S' \) such that \( \psi' : [1..n + 1] \leftrightarrow S' \),

and \( \psi'(k) = \text{item}, i > j \Rightarrow \psi'^{-1}(\psi(i)) > \psi'^{-1}(\psi(j)) \)

(i.e., \( \psi' \) preserves the ordering imposed by \( \psi \))

By arraysplit

Behavior arraysplit

\[ k \in [1..n] \]

Using Function induct
of shift loop

\[ \psi \circ \varphi \] is \( \psi' \) as desired

Using Function move
of item ...

\[ S \cup \text{item} \] is \( S' \)

Function compare ...

Function least ...

Function greatest ...

Figure 3.6: An ordered set, as a special case of an array. This is the context in
which the shift loop is defined. The Ordered Set is defined in terms of the Records in
the set, while an array (3.4) is defined in terms of locations. Hence, the function \( \psi \)
in the ToMake clause of the function insert is given by composing \( \psi \) (the function that
returns a value of a location) with \( \varphi \) (which indexes array locations). Other
functions provided by an ordered set (including compare, least, and greatest) are
not shown here.
Device shift loop is a total loop binding \( U = [k...n-1] \)

Function \( \text{induct} \)

Provided (empty ?OrderedSet[n])

with \( \text{@navigator} = \text{Device index} \) binding \( \text{@U} = U \)

State Definitions
\[
\begin{align*}
\sigma &= n - 1, \tau = k - 1, \nu(i) = i - 1, \\
U(i) &= \{j|i + 1 \leq j \leq n - 1\}, \quad U = \{j|k \leq j \leq n - 1\} \quad \text{[order=dec]} \\
\sigma &= k, \tau = n, \nu(i) = i + 1, \\
U(i) &= \{j|k \leq j \leq n - 1\}, \quad U = \{j|k \leq j \leq n - 1\} \quad \text{[order=asc]} \\
\end{align*}
\]

Function \( \text{start} \)

If \( T \)

ToMake \( ?\text{navigator} = \sigma \)

By e.g., \( ?\text{navigator} := n - 1 \) \quad \text{[order=dec,type=while]}

\( ?\text{navigator} := n - 1 \) \quad \text{[order=dec,type=for]}

\( ?\text{navigator} := k \) \quad \text{[order=asc,type=while]}

\( ?\text{navigator} := k \) \quad \text{[order=asc,type=for]}

Function \( \text{next} \)

If \( ?\text{navigator} 
eq ?x \)

ToMake \( ?\text{navigator} = \nu(?x) \)

By e.g., \( ?\text{navigator} := ?\text{navigator} - 1 \) \quad \text{[order=dec,type=while]}

\( ?\text{navigator} := ?\text{navigator} + 1 \) \quad \text{[order=asc,type=while]}

by -1 \quad \text{[order=dec,type=for]}

NIL \quad \text{[order=asc,type=for]}

Function \( \text{check} \)

If \( ?\text{navigator} \in U \)

ToMake \( ?(\text{navigator} \in U) \)

By e.g., \( ?\text{navigator} \geq k \) \quad \text{[order=dec,type=while]}

\( ?\text{navigator} \leq n - 1 \) \quad \text{[order=asc,type=while]}

\( k \) \quad \text{[order=dec,type=for]}

\( n - 1 \) \quad \text{[order=asc,type=for]}

...
with \( \text{@collector} = \text{Device row} \)

StateDefinitions

\( v(i) \) is given by value of OrderedSet binding \( \text{@}i = i \)

\[
\begin{align*}
\Gamma(S) &= \forall j \in U \setminus S, \quad v[j] = \#a[j] \\
\land & \quad \forall j \in S, \quad v[j+1] = \#a[j]
\end{align*}
\]

is a Record

with \text{gather} given by \text{@move}

binding

\( \text{@loc1} \) is given by location of Ordered Set binding \( \text{@}i = \text{index} \)

\( \text{@loc2} \) is given by location of Ordered Set binding \( \text{@}i = \text{index} + 1 \)

with \text{initialize} given by NIL

Figure 3.8: The shift loop, represented as a special case of the generic total loop, array and record (cont.) As usual, \( \text{@}i \) refers to the variable \( i \) in the ordered set (that is, the array), while \text{index} is local to the collector, and is specified by behavior \text{cycle} of the generic loop, in figure 3.2, to refer to the navigator. \text{@move} refers to the function provided by \text{Record}, and \text{@loc2} and \text{@loc1} refer to its parameters.
The \textit{shift loop} is a special case of a loop. This means that the shift loop inherits all aspects of the generic total loop, i.e., the shift loop has the two function \textit{induct} and \textit{cover}, with behaviors as specified in figure 3.2. Figure 3.2 shows that a loop calls upon the services of two other devices, the \textit{navigator} and \textit{collector}, and also specifies that the \textit{navigator} must provide functions \textit{start}, \textit{next} and \textit{check} and state definitions for $\sigma$, $\tau$ and $\nu$, while the \textit{collector} must specify functions \textit{initialize} and \textit{gather}, and a state definition for $\Gamma$. The details of these functions are specific to a particular loop, so in order to specify the shift loop, it is necessary to specify what will serve as \textit{navigator} and \textit{collector}. These are notated in figure 3.7 following the keyword with.

The \textit{navigator} in figure 3.7 is specified exactly as it was in figure 2.12, that is, not as a special case of anything else.\footnote{It would be possible (and sensible) to define a generic device called \textit{linear navigator} with the single parameter an ordered set $U$, which would define \textit{start}, \textit{next}, $\nu$ etc. appropriately. This does not contribute to the discussion in this section, so I have left out the extra level of indirection.}

The \textit{collector} is specified as a special case of a \textit{Record}, which was described in figure 3.5. A \textit{Record} provides neither a function \textit{gather} nor \textit{initialize}, so it is necessary to specify these in order to use the \textit{Record} as a \textit{collector}. The \textit{initialize} function is specified as NIL, since for the shift loop in figure 2.11, no initialization was necessary. The function \textit{gather} is specified by reference to the function \textit{move} which is provided by \textit{Record}. That is, figure 3.7 states that the shift loop uses a \textit{Record} as its \textit{collector}, and that the \textit{gather} function which a collector must provide
is being provided by the function move of the Record. The function move of Record
requires two parameters, specifying where the Record will move from and to. These
are specified in figure 3.5 by loc1 and loc2, and are bound in figure 3.7 after the
keyword binding to the function location of the generic array. The function
location of array also requires a parameter i, which is bound in figure 3.8 to index
and index + 1.

3.6 Why Generic Objects?

In Chapter 2 I showed how to represent the shift loop using the functional represen-
tation, but without using any of the generic objects described in this chapter.
There are three things we gain by having generic objects available for specifications.
These are

• Proof strategies. A strategy for a proof of correctness for a program fragment
can be indexed by the type of fragment it is. For example, proofs for loops
have particular strategies.

• Providing Provisos. In the last chapter we saw how proofs of provisos could
approximate correctness proofs. Generic devices can specify generic provisos.

• Forming a Functional representation. Having generic items available simpli-

ifies the construction of a functional representation.

\[2\]The function location of the array does not have a predicate in its ToMake slot (see figure
3.4), as have all functions so far. Instead, it has an expression which is to be used (in this
case) in the predicates in the move function of the Record.
3.6.1 Proof Strategies

For any loop, the proof strategy which will be used to prove the correctness of the loop will be as described by Heisel et. al ([16]), i.e., set up a loop invariant, verify that it holds before the loop begins, and that it will hold after any iteration of the loop, given that it holds at the beginning of that iteration. This proof strategy can be related to the functional representation of the loop in figures 3.2 and 3.3 in the following way; the second state of the behavior initialize-and-loop is the loop invariant. The states in the behavior cycle correspond to the proof that the loop invariant is in fact invariant over any iteration of the loop. In chapter IV, I will show how to automatically generate a proof of correctness of a program using a template based on this proof strategy, filling in the detailed bindings of $\nu, \sigma, \tau, U$ and $\Gamma$ as provided by the specific representation and the particular program.

In addition to providing a framework for correctness proofs, the loop also provides a framework in which we can construct approximate proofs using provisos specified in the representation of the specific loop. That is, in order to guarantee a proviso in the cycle behavior, we know that we have to prove that the proviso is satisfied the first time through the loop, and each subsequent time. That is, the following method will construct a proof that guarantees that a proviso $P$ on link $L$ in cycle is satisfied:

- For each link $l$ before $L$ in cycle (moving backwards in the behavior) until
the top of the cycle is reached

- If the ToMake clause of the function on \( l \) implies \( P \),
  
  * then replace \( P \) with the If clause of \( l \);
  
  * Note this replacement.

- Replace all occurrences of \(?navigator\) with \( \sigma \) (in accordance with the function \( start \) of the navigator). Note any replacements made.

- Determine whether the If clause of the entire loop implies \( P \); If so, the notes so far (in reverse order) constitute a proof that \( P \) is satisfied the first time through the loop.

- For each link \( l \) in cycle, starting just before \( L \) and moving backwards in the behavior until the If clause of the entire loop implies \( P \)
  
  - If the ToMake clause of the function on \( l \) implies \( P \),
    
    * then replace \( P \) with the If clause of \( l \);
    
    * Note this replacement.

- If this loop terminated, the notes (in reverse order) constitute a proof that the proviso holds from one iteration of the loop until the next, and thus, that it always holds.

This method will generate a proof that the proviso holds, if such a proof exists, but the final loop will not terminate if no proof exists. There is an easy fix for
this, which is used in Dudu. A description of the implementation of this method in Dudu is given in chapter IV. The method used by Dudu can be used to generate a proof that a proviso is maintained for any loop which can be described by the navigator/collector model.

3.6.2 Scope of Constructed Proofs

The states in the functional representation of a program correspond to intermediate formulas in a proof of partial correctness for that program. As long as the program matches the expectations of the functional representation so that the proof is relevant, the 'frame problem' of deciding what tacit changes occur from one state to the next is taken care of by the original proof; the scope of a predicate in the functional representation is the same as the scope of the corresponding formula in the proof. When Dudu begins to modify proofs, it is no longer necessarily the case that these formulas will have the same scope; that is, in the algorithm specified in the last section, when a proof fragment is constructed to show that '... the ToMake clause of the function on l implies P', it is possible that in order to show this, it is necessary to consult any amount of the enclosing proof for the entire program. If this is the case, then Dudu will fail to complete a proof. Even in such a case, the functional representation has still focused the theorem prover on an appropriate conjecture to prove, and suggested a context (i.e., a position in the original proof) in which to prove it.
3.6.3 Providing Provisos

Not only can the methods by which provisos are proven be inherited from a generic object, but also the provisos themselves can be supplied from a generic object. For example, the proviso that the collector in the shift loop not assign two values to one location at a time is a special case of the proviso that two objects cannot be in the same place at the same time. This is true of anything which is described as a Record in the sense described in section 3.4. Thus the collector in the shift loop, which is represented as a Record, provides the gather function which is required of all collectors by appealing to the move function it inherits as a Record. The move function of Record includes the proviso that the left-hand side of the assignment be empty, or in figure 3.4, '(empty ?loc1)'.

3.6.4 Forming Functional Representations

The representation of the shift loop given in the last chapter was presented as if it was built from scratch. That is, it was necessary to write a prototype program which solved the problem, decompose it into functional pieces, work out the behaviors of those pieces, and how they relate to a proof of correctness, and what provisos could make appropriate shortcuts in the proof construction. The generic representations have simplified every stage of this process. The functional pieces of the program can be chosen from the library of known pieces. Many of these correspond to ordinary program structures (like loops and arrays), and the others
correspond to the entities used in describing the program (records). The relationships between these generic devices and the proofs are already worked out; for example, the loop invariant can be computed from the navigator an the collector, and the loop body shows how that is to be done. Finally, provisos are inherited from the programming metaphors.

One could wonder why these particular generic parts (along with the ones presented in the appendix) have been identified. That is, while it is possible to represent all of the example programs in this work (as well as the examples in [10]) using these generic parts, might it be the case that another set of generic parts would do as well, or better? Such a state of affairs is always possible when a set of primitives is proposed to describe some range of phenomena; another set of primitives might do as well. A set of primitives can be justified by observing that it can in fact cover an interesting set of phenomena, and has some desirable properties. The set of plans/goals used by PROUST in [10] can be used to reason about any example presented there, and furthermore there is an automatic procedure (PROUST itself) that can decide how a particular program is decomposed into those plans. Similarly, The primitives presented in this chapter and in the appendix can be used to represent a large number of programs (all the examples in this work, which include examples of abstracted data structures, as well as the examples in [10]). For each primitive in this description, there is a connection already worked out from it to a proof of correctness of the program, and an automatic pro-
cedure (Dudu, described in the next chapter), that can match a representation in terms of these primitives to a sample program, and reason with the correctness proof of the program, as described below. 3 Other decompositions into primitives may be possible, but in order to justify them, it would be necessary to demonstrate how they can be related to correctness proofs.

---

3A complete debugging system would incorporate both of these features, i.e., constructing a plan representation as in PROUST, and relating them to correctness proofs, as in Dudu. In chapter V, I show how PROUST's plans can be cast in terms of Dudu's generic devices, thus bringing the power of program semantics to the plan structures of PROUST.
CHAPTER IV

Debugging Using Device Understanding: Dudu

The functional representation described in the preceding chapters, including the
generic devices described in the last chapter, has been implemented on a Xerox
1108 running Interlisp and Loops \(^1\). In order to demonstrate the utility of the
Functional Representation in directing the manipulation of program code, I have
implemented a program called Dudu (Debugging Using Device Understanding)
which, given the functional representation of a program, and a fragment of code
that is purported to be an implementation of that program, does the following:

- Constructs an approximate proof of correctness of the program, if it is correct
- If the program is not correct, identifies the bug, and suggests a correction.

The proof is approximate in the sense described in chapter II, that is, Dudu
outlines the parts of the proof that are the same for all programs that are under-

\(^1\) There is an unfortunate collision of terminology here, since most of the examples in this work
have to do with loop constructs in programs, and the Lisp Object Oriented Programming
System provided by Xerox is also called Loops. Except in this paragraph, the word 'loops' will
refer to the programming construct, not the object oriented system.
stood the same way. If part of the proof depends on details of the implementation, Dudu uses a simple theorem prover that uses the traditional semantics of the program code. The theorem prover used by Dudu is not at all sophisticated, so this often results in formulating queries to the human user. Since these queries are only required when the Functional Representation no longer provides any structure to the proof, and the FR does provide the proof structure (i.e., invariants) for loops, the supplementary proofs are certain to be simple (i.e., they will not involve the determination of any loop invariants). Furthermore, when the FR specifies appropriate provisos for the program fragments, Dudu will complete the proof, using the proviso as a guide. As was discussed in chapter II, such a proviso-based proof is typically only an approximation to a proof.

If no proof can be constructed, the offending proviso indicates exactly where the problem was, and what the difficulty was. In the example from chapter II, the proviso that an array element must be empty points directly to the line of code which violates it, so that Dudu can blame that line for overloading an array location.

Dudu makes use of the functional representation in pursuit of this task in the following manner:

Given a fragment of code,

- **Match** the functional representation to the code. This involves determining which program variables are playing which roles in the functional repre-
sentation, which loops in the functional representation correspond to which looping constructs in the program, and what values are appropriate for the state definitions in the functional representation.

- If no match can be made, one of three things may be going on:
  
  - The sample program cannot be understood in the way described by the functional representation. That is, the sample solution is completely novel,
  
  - The sample program has novel aspects to it, but can still be interpreted as a program that is understood in the way specified by the functional representation. The mismatches are the result of novel ways to provide functions described in the functional representation, or
  
  - The sample program follows the outlined specified by the functional representation, but has one or more bugs.

The functional representation can discriminate the first case by allowing a heuristic for weak matches; if a program cannot be made to even weakly match the functional representation, then the first case holds. The last two can be discriminated by appeal to the proof of correctness underlying the functional representation, as made specific by the match.
• For the parts of the functional representation that matched the sample code, a proof can be generated, as was shown in chapter II. For the parts of the proof that depend on the details of the particular code sample, Dudu checks the provisos listed in the functional representation.

• The appropriate proof form provided by the functional representation (see chapter III) is consulted, and a proof that the proviso is maintained is constructed. Failure to construct this proof results in a report of a bug.

4.1 Matching a Functional Representation to Program Code

The matcher provided in Dudu is not intended to be a sophisticated matcher. More sophisticated treatments of various approaches to matching plan structures to programs have been done by Johnson with PROUST ([10]) and Wills as part of the Programmer's Apprentice ([25]). The intent of Dudu is to show that the functional representation in fact does represent the structure of a program in such a way as to guide appropriate problem solving behavior (in the case of Dudu, this is to apply program language semantics to the appropriate parts of a program to determine the presence and nature of program bugs). The matcher is willing to make a strong commitment to the conjecture that the sample program actually corresponds to the functional representation. If this is not the case, the matcher will fail to make any coherent statements about the program at all. This behavior
$eigen1 := (1 + \sqrt{5})/2$

$eigen2 := (1 - \sqrt{5})/2$

print $(eigen1 \uparrow n - eigen2 \uparrow n) * \sqrt{5}/5$

$last := 0$

$this := 1$

for $i$ from 1 to $n - 1$ do
  $next := this + last$
  $last := this$
  $this := next$
end

print $this$

Figure 4.1: Two programs that compute the $n^{th}$ Fibonacci number, which are understood in very different ways. If a grader who was expecting to see a loop solution (like the one on the right) were to be presented with the solution on the left, he would be confused. In fact, unless he has studied the appropriate theorems in linear algebra (the spectral theorem, [15]), he might decide (incorrectly) that the spectral solution is wrong. If he was familiar with the solution from linear algebra, he would quickly determine that the solution on the left should be evaluated in those terms, since it contains no loop. In any case, nothing the grader knows about loops will help him to grade the solution on the left, and similarly, no loop invariants appear in the proof of correctness of the spectral solution.

is acceptable (and even desirable) for a representation that is inspired by human understanding. Consider a human grader, whose job it is to mark as correct or incorrect a hundred proposed solutions to a problem. If he were to be presented with a solution for the wrong problem (or a solution using a profoundly different method) he would certainly be confused at first, and then mark the solution as incorrect without determining what problem it is a solution for (see figure 4.1).

The matcher is presented with a functional representation of a program, and a proposed fragment of code. The top level of the functional representation is a Device with a top level function. These describe the expectations for the structure of the code. The matcher begins by examining the behavior (or behaviors)
associated with the top level function, and tries to make a rough structural match with the code as presented. This is done by examining the example keywords on the links in the behavior (see figure 2.9 for some of these example keywords. In figure 2.9, these are the program language keywords 'for' and 'while' that appear in the behavior *initialize-and-loop*), and searching for lines in the proposed code that match these forms. This rough match allows the structure of the proposed program to guide the selection of the appropriate function. So, in the case of the Fibonacci sequence calculations in figure 4.1, a functional representation might specify two possible behaviors that can realize the overall function of computing the $n$th Fibonacci number. The rough match would distinguish which one is appropriate for describing the spectral solution from the looping solution, since none of the loop keywords in figure 2.9 appear in the solution on the left of figure 4.1.

After the rough match has determined the specific behavior which corresponds to the proposed code, the rest of the matching is done strictly top-down, that is, using expectations from the functional representation to guide the matching process.

For any behavior $B$, the proposed code is scanned for lines which match each link in $B$. If a link in $B$ refers to a function for which the behaviors are specified as code template (see section 2.1), such a match is easy to do; simply find a line in the proposed code which matches the template. For links that refer in turn to other behaviors (the flowcharts in section 2.1), the match is pursued recursively,
under the constraints placed by the matching completed so far. These constraints include ordering constraints (for example, the code fragment that matches a link late in a behavior must follow a code fragment that matches a link earlier in that behavior), as well as consistency checks (for example, the keyword from can only be used in a for loop).

If the matcher were to insist that the program match the functional representation exactly, then the failure to match a link would have to result in the matcher backtracking to some earlier decision that could have been made differently so as to affect the failed match. Since Dudu can potentially be given a buggy program to match, it could be the case that some part of the functional representation will fail to match, even though it is a correct interpretation of how the program is supposed to work. For this reason, the matcher will not backtrack so easily. If most of a behavior matches, this is a good indication that the behavior in fact does represent what the program is supposed to be doing. In this case, Dudu is justified in expecting the rest of the behavior to match. Therefore, unless a significant portion of the behavior fails to match, Dudu will proceed with the matching process (as opposed to backtracking), keeping a note of the problematic mismatch.

At the end of the match, each link in the functional representation will have matched to some fragment of the proposed code. Links that refer to functions which have code templates as behaviors will match to single lines of code, while others will match to larger fragments. The meta-variables in the functional representation
(written in figures 2.11 and 2.12 as names beginning with a ‘?’) will be bound to program variables. All state variables mentioned in the devices used by the proposed program will be bound.

4.2 The Role of FR in Program Debugging

Certain sorts of debugging can be done simply from the functional representation and matcher as described so far. If a significant part of a behavior matches the proposed code, this provides strong justification to expect the rest of the behavior to match. If this fails to happen, Dudu could, at this point, simply correct the program by suggesting that the offending lines be changed to what the behavior expected to see. Since the match keeps track of which line matched which link in the functional representation, it is possible to determine exactly which lines need to be replaced. This can be done in the context of the program as written, that is, using the variable names and choices of loop constructs that were chosen by the programmer.

Such behavior would seem to be a bit imperialistic; tantamount to saying “I don’t understand what you did here, so do it my way.” For this reason, Dudu works a bit harder before suggesting such a correction. There are three possible situations in this case:

1. Dudu’s expectations are entirely wrong; the code did not match Dudu’s expectations, and was not intended to do so.
2. The offending code was not intended to be the code that Dudu expected, but was intended for the same purpose. It is, in fact, correct, but not in the way that Dudu naïvely expected.

3. The offending code in fact was intended to be the code that Dudu expected to see. Examples of this are mistyped variable names. In such a case, code substitution amounts to a sort of plan-driven spelling checker.

Dudu is able to detect the first case, and to perform appropriately in the last two cases. In the first case, the appropriate behavior would be to find another functional representation to use. The current implementation of Dudu does not do this (see chapter VII). The appropriate behavior in the second case is to discover how the code manages to do what Dudu expected. Dudu uses the program semantics to make this determination.

If the code nearly matches Dudu's expectation, then Dudu assumes that case (3) holds, and suggests the change. A fragment of code is said to 'nearly match' if it can be made to match by using some simple transformations (such as replacing one variable with another). Thus, \( i := 1 \) would nearly match \( j := 1 \), but \( i := 0 \) would not.

If the code does not nearly match Dudu's expectations, then perhaps (2) is the case. In order to determine this, it is necessary to prove that the actual code satisfies the purpose that Dudu expects of it. Since the functional representation
represents what the expected line was intended for as well as what it was intended to be, it is easy to construct a conjecture which, if proven, would demonstrate the correctness of the proposed code. If the proof fails, then either the code is both novel and wrong, or Dudu’s expectation was inappropriate. In the former case, Dudu can still imperialistically suggest a modification.

If neither case (3) nor case (2) holds, then this functional representation is inappropriate for this program. Since Dudu is intended to demonstrate how to apply the functional representation in its appropriate context, Dudu, like the grader who does not know the spectral theorem, will not be able to make any useful statement about the proposed program code.

4.3 The Role of Proof in Program Debugging

Although the functional representation can construct a conjecture, which, if proven, would demonstrate the correctness of a fragment of the proposed code, it is not always the case that the conjecture can be proven outside of the context of the rest of the proof of correctness. For example, it might be the case that the assignment $a := b$ assign the value $x$ to $a$; this is true so long as $b$ contains $x$. If this assignment appears within the body of a loop, the proof that $b$ always contains $x$ would involve an induction on the loop. Fortunately, the functional representation indexes into a complete correctness proof for the program, so that the context in which it is necessary to prove the conjecture is available.
First, I will show how the functional representation indexes into a proof of correctness of the program. Then I will show how this index helps Dudu to construct a proof for a conjecture which was constructed to account for mismatches between the functional representation and the proposed code. The same method is used to complete the proof when parts of it cannot be specified by the functional representation, even if it were to match completely, because they rely on the specific semantics of the actual program. Finally, I will show how to construct approximate proofs when an actual proof is too expensive.

4.3.1 Constructing a Proof of Correctness from the Functional Representation

Once the matching is complete, a proof of correctness can be generated from the functional representation. If part of the functional representation failed to match the sample program, Dudu generates provisos that guarantee that the unexpected code actually fulfills the expected functionality. So, for example, if the sample code has an assignment $a[i] := \text{save}$ where Dudu expected $a[i] := \text{temp}$, the proviso that $\text{temp} = \text{save}$ is generated to support the conjecture that the sample line actually fills the expected role.

The proof is generated starting at the top of the functional representation. The top level device is processed first. Devices may inherit proof forms from the generic devices of which they are particulars; for example, the total loop form given in chapter III may specify a form for the proof to take. I will describe
the default proof form for devices in general, and then describe the proof form for the total loop (which is similar in many ways to the default). The default form is a recursive descent through the links of the behaviors of the appropriate functions of the device. The states are propositions in the axiomatic semantics of the programming language. They appear in the proof as the intermediate formulas in the proof. The links connect one state to another, and they are translated in the proof as the justifications that lead from one formula to another.

For a default object, the top level function of the top level device is processed first. For function $F$

Function $F$

If $Q$

ToMake $P$

By $B$

...  

the proof is constructed by forming the statement in formal semantics

$\{Q\}C\{P\}$

where $C$ is the code that corresponds to behavior $B$. Any state variables bound by the matcher are substituted into the state descriptions $P$ and $Q$. $C$ is available from the match, since the lines of code are notated with the links in the functional representation.

The remainder of the proof is constructed from the behavior $B$. That $Q \vdash s$ for
s the first state in the behavior $B$ follows from the consistency of the functional representation (see page 52). This is part of the proof, so Dudu states it. This is also the case for $t \vdash P$, for $t$ the final state of the behavior.

A behavior $B$ is either a code template or a state/transition diagram. For $B$ a diagram, $B$ is either cyclic or linear. The default proof form for a device in general does not take care of the case of a cyclic behavior (but the proof form for a loop does). For a linear diagram, the proof is constructed one link at a time. For a link of the form

\[
(P) \xrightarrow{\text{Using Function } F} \xrightarrow{\text{of } D \text{ binding}} \ldots \xrightarrow{(Q)}
\]

the proof consists of three things:

- $P \vdash I$ where $I$ is the state specified by the If clause of $F$,

- the proof generated by function $F$, and

- $T \vdash Q$, where $T$ is the state specified by the ToMake clause of $F$.

If the consistency of the representation has been verified, then the parts of the proof for $P \vdash I$ and $T \vdash Q$ are already done. The proof for function $F$ is generated as outlined above.
For behavior $B$ a code template, the proof process becomes more problematic. This is because, depending on the detailed semantics of the actual code that appears in the proposed code fragment, the proof might not exist. That is, for a function $F$

**Function $F$**

If $Q$

ToMake $P$

By e.g., *code templates*

... for which the *code template* matches a particular *line*, it is necessary to prove $\{Q\}line\{P\}$. Since this depends on *line*, the functional representation cannot tell whether or not this is true. At this point, Dudu appeals first to an automatic theorem prover, and then, if that fails, to the human reader of the proof to complete the proof. In either case, the problem yet to be solved has been greatly reduced.

For a device that is an instance of the generic type *total loop*, a slightly different proof form is used. The *induct* function is treated exactly like a function of a generic device, that is, its behavior is traced link by link, referring to the behavior of the functions on the links as appropriate.

In figures 3.2 and 3.3 I show the representation of the generic total loop. In the behavior *initialize-and-loop*, the middle state is $\Gamma(U(?navigator))$. This formula is proven by appeal to the links and states above it in the behavior diagram, as
is usual for any behavior. For the function cycle, which follows $\Gamma(U(?navigator))$, the default proof form cannot be used, since cycle has a cyclic behavior. For a total loop, $\Gamma(U(?navigator))$ serves as the loop invariant to prove the correctness of the loop.

The cycle behavior is processed as if it were a linear behavior starting and ending at the state labeled $\Gamma(U(?navigator))$ (the loop invariant). Because of the consistency of the functional representation of the total loop, if this proof is successful, then the loop invariant is maintained by the actual loop.

The final state of initialize-and-loop states that the loop invariant holds when the loop has reached its terminal condition ($?navigator = \tau$). The default proof generation strategy continues from here to state that this implies the ToMake field of the total loop.

4.3.2 Constructing an Approximate Proof from FR

Before describing how Dudu completes the proofs described in section 4.3.1, I will digress briefly to discuss two different sorts of semantics used by Dudu in constructing proofs, and ultimately, in reporting bugs.

Functional Semantics versus Program Language Semantics

In traditional axiomatic semantics, each statement in some programming language is associated with an axiom schema, which allows us to construct proofs about the
formal properties of the program. Other axiom schemata deal with the various ways the programming language has for composing statements into programs (say, as blocks, loops or procedures). These axiom schemata tell how to interpret particular statements regardless of the intentions of the programmer. A typical schema for the assignment statement $V := E$ might be $\{P_E^V\} V := E \{P\}$, where $P_E^V$ stands for the logical expression obtained from $P$ by substituting the expression $E$ for all occurrences of the variable $V$ [21].

When a program has been matched to a functional representation, the statements (and other entities, see chapter III) have their semantics specified by the roles they play in the program. These semantics are given by the If and ToMake fields of the functions. To distinguish this sort of semantics from the traditional semantics, I will call this the functional semantics of the program. The consistency check of the functional representation verifies that the statements made about the functional semantics is consistent with similar statements that could be made from traditional semantics. The assignment $?navigator := n - 1$ in its context in the function start of the navigator (page 45) has the usual semantics of

$$\{P_{n-1}^?navigator\} ?navigator := n - 1 \{P\}$$

but in the larger context of the induct function (also page 45), has the more specific semantics of

$$\{\Gamma(U(n - 1))\} ?navigator := n - 1 \{\Gamma(U(?navigator))\}$$
Note that $\Gamma(U(n - 1))$ is just $T$ in this context, since the bindings of $\Gamma$ and $U$, along with the If field of the function induct guarantee $\Gamma(U(n - 1))$.

In another context (that is, in another device), an assignment statement could have entirely different semantics. An assignment statement that is being used as the function move of a Record (figure 3.5) has the semantics that a location has been emptied and another one filled.

These two different systems of semantics have profound impact on the generation of a proof. A proof can be generated from functional semantics by simply tracing through the functional representation. Since the meaning of each statement is given by its role in the functional representation, there are very few choices to be made about what intermediate formulas need to be proven.

The problem with using functional semantics is that unexpected lines of code do not have any functional semantics defined, since they do not match any part of the functional representation. This is a virtue of traditional semantics; since the semantics are defined independent of their context, the semantics of a line is always defined.

Proof Completion

There are two reasons why Dudu might not have been able to build a complete proof using the method defined in the last section. It could be the case that the details of a line of code encountered in the proposed code matches one of the
templates specified in some behavior, but it is not clear that this particular instantiation of a function is consistent with the choices made for the other functions. For example, the proposed code may make use of a completely expected navigator, and an expected collector, but the combination of navigator and collector could be invalid (this happens in the shift loop if the programmer chooses to have the navigator move up the array, while the collector simply moves an item from one array cell to the next; each of these choices is valid, but the combination does not work). In such cases, it is possible to reason with the functional semantics of the program.

Dudu might fail to complete a proof because the proposed code contains one or more lines of code which were entirely unexpected. Since there is no part of the functional representation that corresponds to the novel code, the functional semantics of that code is undefined. In this case, the proof must rely on the traditional semantics of the program.

Whether the proof relies on traditional or functional semantics, the method for generating the proof is the same. Dudu uses an extension of the method outlined on page 85.

The method on page 85 works entirely with the functional semantics of the program, that is, it only traces the conditions specified in the functional representation of the program. The algorithm used by Dudu also makes use of the traditional semantics of the program, when they are available. The current implementation of
Dudu only knows about the traditional semantics of assignments. That is, novel lines of code can be used by Dudu in a proof only if they are assignment statements.

Dudu incorporates traditional semantics into the method on page 85 by allowing for interpretation of novel lines of code. Dudu also guarantees that the final loop will terminate. The full algorithm used by Dudu to generate a proof for a total loop is as follows; in the following, $P$ refers to the proviso to be proven.

- For each link $l$ before $L$ in cycle (moving backwards in the behavior) until the top of the cycle is reached

  - If the ToMake clause of the function specified by $l$ implies $P$,
    - * then replace $P$ with the If clause of $l$;
    - * Note this replacement.

  Please note that the function specified by $l$ can be interpreted as an axiom schema, just as in traditional semantics.

  - For any novel line $c$ that appear immediately before the line matched by $l$ (that is, they appear before $l$ but after any line that matches any link before $l$) do
    - * Apply the traditional semantics of $c$ to $P$. If $c$ is the assignment $V := E$, the amounts to replacing $P$ with $P^V_E$.

- Process any novel lines occurring before the start of the loop, but after its initialization, as above
• Treat the links in initialize-and-loop as the links in cycle were treated. For the function start of the navigator, this amounts to the substitutions mentioned on page 85. Note any replacements made.

• Determine whether the If clause of the entire loop implies $P$; If so, the notes so far (in reverse order) constitute a proof that $P$ is satisfied the first time through the loop.

• For each link $l$ in cycle, starting just before $L$ and moving backwards in the behavior until the If clause of the entire loop implies $P$, or an entire cycle has been made of the loop in cycle without any line affecting $P$ for the first time,

  – If the ToMake clause of the function on $l$ implies $P$,

    * then replace $P$ with the If clause of $l$;

    * Note this replacement.

  – Treat the novel lines immediately before $l$ as before

• If $P$ is a tautology, the notes (in reverse order) constitute a proof that the proviso holds from one iteration of the loop until the next, and thus, that it always holds.

This algorithm differs from the algorithm on page 85 in two ways:
• Each time a novel line is encountered, its traditional semantics are consulted to determine the effect on the proviso $P$.

• The final loop is guaranteed to terminate. Each line gets to modify $P$ once; if a cycle goes by without any new modifications to $P$, the proof is abandoned. Hence a proof could trace through several iterations of the loop, so long as something new happens at each iteration.

4.4 Dudu as a Proof System

Since Dudu uses the Functional Representation to construct proofs of correctness of programs, it is natural to ask what formal properties Dudu has as a proof generation system. That is, is it always possible for Dudu to construct a proof of correctness for a correct program (completeness)? Is it impossible for Dudu to construct a proof of correctness for an incorrect program (soundness)?

When Dudu encounters a program with no unexpected parts, its proof generation capabilities amount to recalling a proof that was constructed when the functional representation was built. In such a case, the completeness of the proof generation capability of Dudu cannot exceed the capabilities of the proof system in which the recalled proof is written. In the work presented here, Dudu uses an axiomatic system based on the system presented by Pagan ([21]), which in turn is based on Hoare ([9]), and the representational completeness of the functional representation language. The question of the adequacy of the functional represen-
tation hinges on determining which programs can be represented.

In the examples presented so far, I have shown how to represent programs made up of assignment statements and certain kinds of loops (loops that can be broken down into a navigator and a collector), and that make use of simple variables and arrays. The discussion in appendix I extends this to include programs using basic conditionals and a second kind of loop (search loops). That is, any program made up of these components could be represented by the Functional Representation using the generic devices discussed here, and consequently, a proof can be recalled for any such program. The motivation for using the Functional Representation is that the loop schemas should match program code in a natural, understandable way. However, In order to show the logical completeness of representation, it is only necessary to show that a particular schema can be used (however unnaturally) to represent any loop. In fact, the total loop can be made to match any loop in a language in which the loop is defined by Hoare's axiom schema

$$
\{P \land B\} S \{P\} \\
\{P\} \text{ while } B \text{ do } S \text{ end } \{P \land \neg B\}
$$

(4.1)

(from [21], p. 204). The total loop described in figures 3.2 and 3.3 can be specialized in such a way that the proof that Dudu will generate from it will be an application of this axiom schema.

For code $S$, and appropriate predicates $P$ and $B$, specialize the collector (figure 3.2) by binding $\Gamma(S)$ identically to $P$ (recall that $\Gamma$ is the function that the collector
applies to the set covered by the navigator), the By slot of the function \textit{gather} to be code fragment \textit{S}. Specialize the function \textit{initialize} of the \textit{navigator} by filling its ToMake slot with \textit{B}. Then in the proof generated by Dudu, as described in chapter II, the loop invariant is exactly \textit{P}, and the termination condition is exactly $\neg\textit{B}$, and the assertions made in the proof correspond exactly to Hoare's axiom schema. Thus Dudu has the capacity to recall proofs as complex as is allowed by the semantic axioms in which the proof fragments are stated.

Similar remarks apply to the soundness of the proofs produced by Dudu; that is, Dudu can recall any proof that can be written in the underlying axiomatic system. However, when Dudu completes an unfinished proof, it is possible that it might construct an incorrect proof. This is because the reconstructed proof proves that some proviso mentioned in the representation is maintained, rather than that the appropriate fragment of code is correct. It would be possible for Dudu to require that all the provisos are specified conservatively, that is, each proviso simply repeats the specification of the fragment of code to which it applies. In such a case, Dudu reduces to a theorem prover over the logic that underlies its proof templates. Dudu does not do this, because Dudu makes use of heuristic provisos that are analogues of the potentially unsound proof heuristics used in transformation based automatic programming systems, like the heuristic transformation rules used by DIOGENES ([19]).
4.5 A Sample Session with Dudu

In this section, I will show Dudu's output for several examples programs, all of them relating to the shift loops described in earlier chapters. This will demonstrate how Dudu can recognize that a program matches a functional representation, thus defining the functional semantics for the program constructs. In the first example, Dudu uses the functional semantics, along with a proof template inherited from the generic total loop, to construct a proof of correctness of the program. This involves completing the proof by use of a proviso inherited from the Record metaphor, and the method for completing approximate proofs inherited from the total loop, as shown on page 85. In the second example, the approximate proof fails, so a bug report is generated from a generic bug report attached to the proviso in the Record metaphor. Finally, in the third example, novel lines appear in the sample program; Dudu constructs the proof of correctness using the same template as in the previous examples, using the functional semantics for the expected lines, and resorting to the traditional semantics for the novel lines.
4.5.1 A correct and expected program

Dudu was presented with the following program:

\[
i := n - 1 \\
\text{while } i \geq k \text{ do} \\
\quad a[i + 1] := a[i] \\
\quad i := i - 1 \\
\text{end}
\]

This is one of the examples that motivated the construction of the shift loop as presented in the last few chapters. Dudu matches this program to the functional representation, determining that the collector appears in the program as the variable \( a \), and the navigator is \( i \). \( \sigma = n - 1 \), \( \nu(j) = j - 1 \), \( \tau = k - 1 \), \( U(j) = [j+1..n-1] \) and

\[
\Gamma(S) = \forall j \in U \setminus S, \quad a[j] = \#a[j] \\
\land \forall j \in S, \quad a[j + 1] = \#a[j]
\]

Every line is expected, that is, there is a link in some behavior in the functional representation that corresponds to each line in the program. This allows Dudu to generate a proof, just as was shown on page 44. The proof built by Dudu is as follows (paragraphs beginning with "[" are comments included for this discussion; all other lines are (typeset from) Dudu's output):
While analyzing the program, Dudu asks the user '(empty a[n])?', that is, it asks whether $a[n]$ empty when the loop begins. When the user answers 'yes', Dudu presents the following proof:

To Prove: $\{\forall j \in [k..n - 1], (a[j] = \#a[j]) \land \forall j \in \emptyset, (a[j + 1] = \#a[j])\}$

\[i := n - 1\]

while $i \geq k$ do

\[a[i + 1] := a[i]\]

\[i := i - 1\]

$\{\forall j \in \emptyset, (a[j] = \#a[j]) \land \forall j \in [k..n - 1], (a[j + 1] = \#a[j])\}$

In the antecedent, after line $i := n - 1$, replace all occurrences of $n - 1$ with $i$.

This verifies the loop invariant

\[\forall j \in [k..n - 1] \setminus [i+1..n-1], \quad a[j] = \#a[j]\]

\[\land \forall j \in [i+1..n-1], \quad a[j + 1] = \#a[j],\]

at the beginning of the loop.

Since after the line while $i \geq k$ do, that $i$ in $[k..n - 1]$ is guaranteed,

after the line $a[i + 1] := a[i]$, it is necessary to show that

\[\forall j \in [k..n - 1] \setminus [i..n - 1], \quad a[j] = \#a[j]\]

\[\land \forall j \in [i..n - 1], \quad a[j + 1] = \#a[j],\]
[At this point, Dudu has found that there is a possibility that the proof will not work; this is because the choice of *navigator* (including the bindings of \(\tau, \sigma, \nu\) and \(U\)) might possibly be inconsistent with the choice of *collector* (including the binding of \(\Gamma\)). Dudu detected this possibility of such a problem in the proof because there was a proviso present in the functional representation. Namely, the *gather* function of the *collector* is a special case of the *move* function of the *Record*, which has a proviso guarding against placing two records in the same place at the same time. If no proviso is present, Dudu presumes that the functional representation is consistent; that is, the code matched to a function guarantees that the function will be achieved. Dudu continues by proving that the proviso was satisfied. This completes an approximate proof, which Dudu notes.]

I can't prove that this is so, but it is likely since the fact that \(a[i + 1]\) is empty can be shown to hold, as follows:

["\(a[i + 1]\) is empty" is the proviso inherited from *Record*; the form for the proof that it always holds is inherited from the generic total loop]

\(a[i + 1]\) is empty: Presuming that \(a[n]\) is empty, the loop initialization tells us that \(i\) is \(n - 1\). So, at the start of the loop, \(a[i + 1]\) is empty.

[The shift loop moves the items from \(k\) to \(n - 1\) up one; this works provided \(a[n]\) is empty (so that one can move \(a[n - 1]\) into it). This proviso is specified in figure 3.8. The proof Dudu gives reduces the proviso "\(a[i + 1]\) is empty" from the generic *Record* to the proviso on the loop.]
From the previous iteration, the line $a[i + 1] := a[i]$ gives us that $a[i - 1 + 1]$ is empty. The line $i := i - 1$ gives us that $a[i + 1]$ is empty.

[The line $a[i + 1] := a[i]$ is matched (eventually) to the move function of Record, so its functional semantics is given by the If and ToMake fields of move, with $?loc1$ and $?loc2$ bound to $a[i]$ and $a[i + 1]$ respectively. The ToMake field of move includes (see fig. 3.5) (empty $a[i + 1]$). Dudu uses the functional semantics whenever it can, as it does in this case.]

Thus, $a[i + 1]$ is empty for the current iteration.

[Having proven the proviso, Dudu returns to the main proof.]

After line $i := i - 1$, replace all occurrences of $i - 1$ with $i$.

[For this line, the functional and traditional semantics are identical.]

Hence,

$$\forall j \in [k..n - 1] \setminus [i + 1..n - 1], \quad a[j] = \#a[j]$$

$$\land \forall j \in [i + 1..n - 1], \quad a[j + 1] = \#a[j],$$

$$\land i = k - 1$$

[This is just the loop invariant, plus the termination condition on the loop]

That is,

$$\forall j \in \emptyset, \quad a[j] = \#a[j]$$
\( \land \ \forall j \in [k..n - 1], \quad a[j + 1] = \#a[j] \)

as desired.

4.5.2 An incorrect program

Dudu was presented with the following program:

\begin{verbatim}
 i := k
 while i \leq n - 1 do
  a[i + 1] := a[i]
  i := i + 1
end
\end{verbatim}

Figure 4.3: An incorrect program

This example looks very similar to the last example, except that the counter \( i \) moves up the set rather than down. Thus it matches the functional representation of the shift loop completely, and Dudu can use the functional semantics to attempt a proof of correctness. Since the program is not correct, this attempt will fail, and Dudu will report a bug. Dudu finds, by matching this program to the functional representation, that the \textit{collector} is again the variable \( a \), and the \textit{navigator} is again \( \nu(j) = j + 1 \), \( \sigma = k \), \( \tau = n \), \( U(j) = [k..j - 1] \) and

\[
\Gamma(S) = \forall j \in U \setminus S, \quad a[j] = \#a[j]
\]

\begin{align*}
&\land \ \forall j \in S, \quad a[j + 1] = \#a[j] \\
(\text{the definition of } \Gamma \text{ is the same as before}). \text{ The proof attempt by Dudu follows; again, paragraphs in "[]" are comments for this discussion:} \end{align*}
To Prove:

\[ \{ \forall j \in [k..n-1], (a[j] = \#a[j]) \land \forall j \in \emptyset, (a[j+1] = \#a[j]) \} \]

\[ i := k \]

while \( i \leq n - 1 \) do

\[ a[i+1] := a[i] \]

\[ i := i + 1 \]

end

\[ \{ \forall j \in \emptyset, (a[j] = \#a[j]) \land \forall j \in [k..n-1], (a[j+1] = \#a[j]) \} \]

In the antecedent, after line \( i := k \), replace all occurrences of \( k \) with \( i \). This verifies the loop invariant,

\[ \forall j \in [k..n-1] \setminus [k..i-1], \quad a[j] = \#a[j] \land \forall j \in [k..i-1], \quad a[j+1] = \#a[j], \]

at the beginning of the loop.

Since after the line while \( i \leq n - 1 \) do, that \( i \) in \([k..n-1]\) is guaranteed,

after the line \( a[i+1] := a[1] \), it is necessary to show that

\[ \forall j \in [k..n-1] \setminus [k..(i+1)-1], \quad a[j] = \#a[j] \land \forall j \in [k..(i+1)-1], \quad a[j+1] = \#a[j] \]

[So far, the proof has been just like the last one, except for the details of the]
initialization of $i$. At this point, the proviso that \(a[i + 1]\) is empty" is again generated. Dudu searches backwards through the behavior cycle shown in figure 3.2, searching for some code that will satisfy this proviso. In particular, the proviso was generated by the function \texttt{gather}; the function \texttt{check} has no effect on the proviso \(a[i + 1]\) is empty", so the search continues. The function \texttt{next} replaces the \texttt{navigator} with \(\nu(\texttt{navigator})\), that is, $i$ with $i + 1$. Now Dudu is trying to prove the proviso \(a[i + 2]\) is empty". From figures 3.8 and 3.5, we see that the function \texttt{gather} can satisfy the proviso \(a[i]\) is empty", but says nothing about \(a[i + 2]\) is empty". Dudu continues its search around the loop in cycle; when it makes an entire loop with nothing new happening (that is, no links make a contribution to the proviso except those that have contributed before), it abandons the search, and declares that the proviso cannot be proven. Associated with the proviso is a bug report for what happens when that proviso fail. Dudu reports the failure of the proof as a bug.]

This is unlikely, since on this line you are moving a value from $a[i]$ to $a[i + 1]$, but $a[i + 1]$ already has another value in it. This will clobber the value in $a[i + 1]$.

[Dudu continues the proof anyway, as if the proviso had been satisfied. Since the proviso is only an approximate proof, it might be the case that the actual proof is valid, despite the error in the approximate proof. ]

after line $i := i + 1$, replace all occurrences of $i + 1$ with $i$.

Hence,
\[ \forall j \in [k..n - 1] \setminus [k..i - 1], \quad a[j] = \#a[j] \]
\[ \land \quad \forall j \in [k..i - 1], \quad a[j + 1] = \#a[j], \]
\[ \land i = n \]

That is,

\[ \forall j \in \emptyset, \quad a[j] = \#a[j] \]
\[ \land \quad \forall j \in [k..n - 1], \quad a[j + 1] = \#a[j] \]

as desired.

### 4.5.3 An unexpected, but correct program

Dudu was presented with the following program:

\[
i := k \\
temp := a[k] \\
while \ i \leq n - 1 \ do \\
\quad save := a[i + 1] \\
\quad a[i + 1] := temp \\
\quad temp := save \\
\quad i := i + 1
\]

Figure 4.4: An unexpected, but correct, program

This example is similar to the last one, in that the counter \( i \) moves up the set. However, the rest of the program is not as expected, that is, there is no line
\textcolor{black}{a}[i + 1] := a[i] to move the element to the desired location. Nevertheless, Dudu is able to construct a proof of this program.}

Dudu begins, as before, by matching this program to the functional representation, binding the \textit{navigator} to \( i \), \( \nu(j) = j + 1 \), \( \sigma = k \), \( \tau = n \), \( U(j) = [k..j - 1] \) and

\[\Gamma(S) = \forall j \in U \setminus S, \quad a[j] = \#a[j] \]
\[\land \quad \forall j \in S, \quad a[j + 1] = \#a[j] \]

as before. In order to bind the collector, it is necessary to match the inside of the loop to the expected line \( a[i + 1] := a[i] \). The line \( a[i + 1] := temp \) is a near match, provided that \textit{temp} contains the value that was expected in \( a[i] \). Dudu will generate this as a proviso for the line \( a[i + 1] := temp \), and will complete the proof as above. The output Dudu produces for this program is as follows:

\textbf{To Prove:}

\[\{\forall j \in [k..n - 1], (a[j] = \#a[j]) \land \forall j \in \emptyset, (a[j + 1] = \#a[j])\}\]

\[i := k\]
\[temp := a[k]\]

while \( i \leq n - 1 \) do

\[\text{save} := a[i + 1]\]
\[a[i + 1] := temp\]
\[ \text{temp} := \text{save} \]
\[ i := i + 1 \]

end

\{ \forall j \in \emptyset, (a[j] = \#a[j]) \land \forall j \in [k..n - 1], (a[j + 1] = \#a[j]) \}

In the antecedent, after line \( i := k \), replace all occurrences of \( k \) with \( i \). This verifies the loop invariant,

\[ \forall j \in [k..n - 1] \setminus [k..i - 1], \quad a[j] = \#a[j] \]
\[ \land \forall j \in [k..i - 1], \quad a[j + 1] = \#a[j] \]

[The functional representation expected to find a line that said \( a[i + 1] := a[i] \); instead, it found three lines. Dudu tries to show that these three lines are actually doing the work that the one line was expected to do. ]

at the beginning of the loop.

Since after the line while \( i \leq n - 1 \) do, that \( i \) in \([k..n - 1]\) is guaranteed, after the lines

\[ \text{save} := a[i + 1] \]
\[ a[i + 1] := \text{temp} \]
\[ \text{temp} := \text{save} \]

it is necessary to show that
\begin{align*}
\forall j \in [k..n - 1 \setminus [k..(i + 1) - 1], \quad a[j] = \#a[j] \\
\land \forall j \in [k..(i + 1) - 1], \quad a[j + 1] = \#a[j] \quad (4.2)
\end{align*}

[First, Dudu notices that the second of the three unexpected lines might be able to provide the function expected by the missing line, provided that the variable \(\text{temp}\) contains the value that was expected on the right hand side of the assignment, that is, \(\#a[i]\). Dudu traces back through the loop, using the functional semantics of the expected lines (the loop check, and the loop increment) and the traditional semantics of the unexpected lines (at present, Dudu only knows the traditional semantics of assignment statements).]

I can't prove this is so, but it is likely since the facts that \(a[i + 1]\) is empty, and \(\text{temp}\) contains \(\#a[i]\) can be shown to hold, as follows:

\(a[i + 1]\) is empty: The line \(\text{save} := a[i + 1]\) gives us that \(a[i + 1]\) is empty.

[Since Dudu has hypothesized that the line \(a[i + 1] := \text{temp}\) is playing the role of the missing assignment \(a[i + 1] := a[i]\), it is necessary to prove any proviso provided by the functional description of that line. Hence, Dudu proves that \(a[i + 1]\) is empty. In this case, that is easy, since the assignment \(\text{save} := a[i + 1]\) guarantees that.]

\(\text{temp}\) contains \(\#a[i]\):

The line \(\text{temp} := a[k]\) gives us that \(\text{temp}\) contains \(\text{sharpa}[k]\).
The loop initialization tells us that $i$ is $k$, so at the start of the loop, $temp$ contains $a[i]$.

From the previous iteration,

the line $save := a[i + 1]$ gives us that $save$ contains $a[i + 1]$

the line $temp := save$ gives us that $temp$ contains $a[i + 1]$

the line $i := i + 1$ gives us that $temp$ contains $a[i]$.

Thus, $temp$ contains $a[i]$ for the current iteration.

[The proof that the generated proviso holds requires an inductive proof. The first part of the proof simply traces back from the line that generated the proviso ($a[i + 1] := temp$) to the start of the loop. The second part of the proof traces back through the loop, trying to prove the proviso "$temp$ contains $a[i]$". Moving backward through the loop from $a[i + 1] := temp$, the line $save := a[i + 1]$ has no impact on the proviso; the next previous line is $i := i + 1$; this does have an impact on the proviso, so now Dudu tries to prove that $temp$ contains $a[i + 1]$; the line $temp := save$ has an impact on this proviso, so now Dudu tries to prove that $save$ contains $a[i + 1]$. So far, Dudu has examined every line in the loop once, and still the proviso has not been proven. Dudu continues through the loop, and finds that the line $save := a[i + 1]$, which was irrelevant before, is now sufficient to prove the proviso. Unless Dudu makes an entire pass through the loop with no line making its first contribution to the proof, it will continue to traces through the loop. Having proven these provisos, Dudu returns to the main proof. The next]
line refers to a replacement to be made in formula 4.2.

After line $i := i + 1$, replace all occurrences of $i + 1$ with $i$.

Hence,

$$
\forall j \in [k..n - 1] \setminus [k..i - 1], \quad a[j] = \#a[j] \\
\wedge \forall j \in [k..i - 1], \quad a[j + 1] = \#a[j]
$$

$\wedge i = n$

That is,

$$
\forall j \in \emptyset, \quad a[j] = \#a[j] \\
\wedge \forall j \in [k..n - 1], \quad a[j + 1] = \#a[j]
$$

as desired.
CHAPTER V

Other Work in Program Verification

When a human understands a program, there are several things he can do easily with the program. One of these is to verify its correctness, or, if it is not correct, locate and correct bugs. Dudu has demonstrated that the functional representation of the program supports this task; that is, when Dudu "understands" a program (when Dudu has a functional representation of the program), Dudu is able to verify the correctness of the program, or, if the program is not correct, Dudu can locate and correct bugs. Other tasks that human understanders are able to perform include modifying a program to accommodate a change in its specifications, and justifying a program to another human. In this chapter I will compare the use of the functional representation used by Dudu to the program/plan representations used in other systems that perform one of these tasks. The most interesting systems, for comparison to the Functional Representation as used in Dudu are those that make a statement about the way a programmer understands a program. Probably the most influential work of this sort is the work by Johnson and Soloway ([10],

128
In [10], Lewis Johnson introduces the notion of intention-based diagnosis. Intention-based diagnosis uses the intended structure and function of a device to help to localize faults in the manifest structure of the device. Johnson has implemented a computer program called PROUST, which performs intention-based diagnosis on computer programs.

PROUST does not begin with a representation of the understanding of the program, as Dudu does. Rather, it begins with a specification of the problem the program is required to solve. The specification is represented as a set of goals to be achieved. PROUST is also presented with a program which purports to implement the specified problem.

PROUST has a library of programming plans, which are indexed by the goals they can achieve. A goal is chosen from the specification, and a plan that can realize that goal is selected. The plan contains code templates, as well as references to other goals. If the plan matches the input, PROUST puts the goals generated by the matched plan onto the goal agenda. Otherwise, PROUST must account for the mismatch, or conclude that the plan does not match the program.

Differences between plans and encountered code are resolved by using plan-difference rules. Plan-difference rules look at the plan that did not match and
the goal it was trying to achieve (as well as several other things) to determine
whether the difference can be accounted for. If the plan difference amounts to
some expected sort of bug (or correctness-preserving transformation), then the
appropriate plan-difference rule fires. The plan difference rules can report bugs,
determine misconceptions responsible for the bugs, or they may modify the goal
agenda.

5.1.1 Similarities Between Plan and Functional Representations

A plan representation in PROUST consists of plan templates and goals. Figure
5.1 shows an example of a PROUST plan template. The template is matched to
the student’s code, and plans are selected to realize the goals that were spawned
by the match. These plans in turn have code templates, and other goals.

For comparison, figure 5.2 shows the functional representation for a loop that
computes a maximum. Only the collector is shown, since the collector is the
part of the loop that corresponds to the MAXIMUM plan in figure 5.1. Both
representations describe the programmer’s intentions for the maximum, that is,
that a running maximum be initialized, and somewhere inside of a loop, it is
updated according to the outcome of a check that the current item is larger than
the running maximum. Both leave the details of the loop to some higher level

1The plans used by PROUST also make use of many other annotations for specifying subgoals,
which help it to determine certain error conditions, such as computing the average of no
elements.
MAXIMUM PLAN

Variables:
?New, ?Max

Template:
Init:  
?Max := 0  
(in Process: component of goal Read & Process)

Guard:  
if ?New > ?Max then

Update:  
subgoal Supercede Value(?Max, ?New)

Figure 5.1: A plan from PROUST's plan database, from [10]. Lines of Pascal code are interspersed with subgoals. This plan specifies how to compute a maximum; the variable ?Max holds the running maximum. It is initialized to a small value (here, 0). The stipulation "in Process: component of goal Read & Process" specifies that the code that follows appears in the Process: component of whatever plan realizes the Read & Process goal.

Device max loop is a Loop

... with @collector = Device max

StateDefinitions
\[ \Gamma(S) \equiv \ ?max = \text{MAX}(S) \]

Function gather
If \( \Gamma(U(?\text{index})) \land (?\text{index} \in U) \)
ToMake \( \Gamma(U(\nu(?\text{index}))) \)
By e.g., if ?\text{index} > ?max then ?max := ?\text{new}

Function init
If T
ToMake \( \Gamma(U(\sigma)) \)
By e.g., ?max = 0

Figure 5.2: Functional representation of collector for a loop to compute a maximum. This is mostly reprinted from the generic collector, with \( \Gamma \) bound as indicated. Most any navigator would go well with this, including the navigator specified for the shift loop shown earlier, or a navigator that uses read(?navigator) for its next function.
construct; the PROUST plan refers to the *Read & Process* goal, which will be realized by a loop, while the functional representation specifies that the update is part of the *gather* function of a *collector*. The Process: reference in the PROUST representation refers to the part of the loop that processes the input, i.e., exactly the part of the loop revered to by the *gather* function of the *collector*. Both representations acknowledge that although the initialization and the update of the running maximum appear in different positions in the program code, they must be dealt with as different parts of a single entity (in PROUST, they are components of the same plan, in the functional representation, they are functions of the same device). These similarities make it possible to combine the use of the plan difference rules developed for PROUST with the functional representation.

5.1.2 Differences Between Plan and Functional Representations

The difference between the functional representation and the plan representation used by PROUST is the presence of *states* in the former. This is important, because it allows Dudu to make use of the distinction between an implementation of a function (*how* it is achieved) and its role in a larger component (*why* it is present). It is the ability to make this distinction that allows Dudu to consult a proof of correctness for a program when plan template matching fails.

As an example, consider the two programs shown in figure 5.3. Both compute the maximum of the *n* items in the array *a*. They work in basically the same
max := 0
item := 1
while item ≤ n do
  if a(item) > max
    then max := a(item)
  item := item + 1
end

max := a(1)
item := 2
while item ≤ n do
  if a(item) > max
    then max := a(item)
  item := item + 1
end

Figure 5.3: Two programs that compute the maximum of the $n$ values in an array, provided $n$ is at least 1 and 2 respectively.

way, except that the second one begins one step later in the processing, and takes advantage of the fact that $\text{MAX} \{i\} = i$. Proust can recognize the program on the left easily as a straightforward example of the MAXIMUM plan shown in figure 5.1, in combination with some plan to implement the loop. Similarly, Dudu can recognize this as a simple example of a total loop, with its collector bound as shown in figure 5.2.

In order to recognize the second program, Proust would need to have a plan-difference rule, perhaps called ‘Front-Trailing-Iteration’, that checks to see whether the variables were initialized in such a way that one cycle through the loop is completed before the loop begins. It is hard to imagine what such a plan-difference rule would look like, since there is no representation of what it means for one cycle of the loop to be completed.

Dudu does have a representation of what it means for one cycle of the loop to be completed, or for that matter, for any number of cycles of the loop, since Dudu...
has a representation of the loop invariant. Furthermore, Dudu knows where in the
program the loop invariant must hold, namely, after the initializations shown in
the diagram for *initialize-and-loop* in figure 3.2.

To demonstrate how Dudu makes use of this representation, Dudu was pre­
sented with the programs in figure 5.3. The output from Dudu follows (annotations
are flagged with brackets ([]), as before):

[First, Dudu inquires:]

Is it the case that \(a[1] = \text{MAX}(a[1..2 - 1])\)?

[I answer “yes”; then Dudu proceeds:]

To Prove: \(\{\text{max} = \text{MAX}(a[0])\}\)

\[
\text{max} := a(1) \\
\text{item} := 2 \\
\text{while } \text{item} \leq n \text{ do} \\
\hspace{1cm} \text{if } a(\text{item}) > \text{max} \text{ then } \text{max} := a(\text{item}) \\
\hspace{1cm} \text{item} := \text{item} + 1 \\
\text{end}
\]

\(\{\text{max} = \text{MAX}(a[1..n])\}\)

In the antecedent, given that \(a[1] = \text{MAX}(a[1..2 - 1])\), after the line \(\text{max} := a[1]\),
replace all occurrences of \(a[1]\) with \(\text{max}\). After the line \(\text{item} := 2\), replace all occur­
rences of \(2\) with \(\text{item}\). This verifies the loop invariant, \(\text{max} := \text{MAX}(a[1..\text{item} - 1])\),
at the beginning of the loop.
[Dudu recognized the loop, since it is exactly as expected. This allowed Dudu to construct the loop invariant. The initializations were not quite as expected; they initialized the right variables, but the values were not anything like what was expected. Dudu was able to resort to the traditional semantics of the assignments, replacing variables with their values in the loop invariant. This resulted in the inquiry $a[1] = \text{MAX}(a[1..2 - 1])$; the reported proof is this derivation shown in reverse.]

[The rest of the proof is done just as was shown already for the shift example.] Since after the line while item $\leq n$ do, that item $\in [1..n]$ is guaranteed, after the line if $a[item] > max$ then max $:= a[item]$, max $= \text{MAX}(a[1..(item+1)-1])$.

After line item $:= item + 1$, replace all occurrences of item + 1 with item.

Hence, max $= \text{MAX}(a[1..item - 1])$ and item $= n + 1$,

that is, max $= \text{MAX}(a[1..n])$, as desired.

Dudu was able to complete analysis of this program by resorting to the traditional semantics of the initializations. This was made possible by the fact that the functional representation indicated not only that the variables must be initialized, but also why those initializations were being done (in particular, to guarantee the particular loop invariant). Dudu does not need to form the complete proof using the traditional semantics of the program, it only needs to complete the proof for the parts that were unexpected.

In summary, Dudu's devices correspond to plans in PROUST. In addition to
indexing from one plan to others (as goals do in PROUST), Dudu also specifies
the state of the computation that is intended to hold after a plan is completed. 
This allows Dudu to make use of traditional programming language semantics to
detect correct program transformations. In PROUST, this behavior would re­
quire a family of plan-difference rules, indexed on the particular plans. Because
PROUST plans are so similar to Dudu’s Devices, Dudu’s approach can be in­
tegrated smoothly with PROUST’s plan-difference rules, allowing them to take
advantage of knowledge of programming language semantics.

5.2 Verification Based Debugging

In [20], William Murray presents a program called ‘Talus’ that demonstrates how
to take advantage of a Boyer-Moore theorem prover to perform program debugging
tasks. Talus compares a student function to a reference function that performs the
required task. Talus does not have to worry about a separate language for repre­
senting task specifications and implementations, since the reference function, which
is written in LISP, provides the specification. The Boyer-Moore logic provides the
formal semantics of the pure LISP functions that are used both as reference func­
tions and as proposed solutions by students. Murray demonstrates that automatic
debuggers suffer if they cannot reason about program semantics, and uses Talus
to show how the Boyer-Moore logic can be used to automate such reasoning.

The strategy used by Talus is to compare the student function definition to a
reference function definition by constructing an induction proof that they compute the same function. This is done with a standard induction form in the Boyer-Moore logic. Conditionals are broken down into steps in the inductive form (the conditional branches without any recursive calls correspond to the base step in an inductive proof, while those with recursive calls correspond to the induction step(s)). The steps in the proof are of the form 'if (some conditions) then some fragment of the student function is equivalent to a fragment of the reference function'. When such a step fails, Talus is in the position to isolate the bug to the fragment of the student function mentioned in the step; the proof strategy used by Talus guarantees that any such fragment will contain no conditionals or recursive calls, so it is a fairly low-level fragment of the student's code. Talus applies a 'least-change' heuristic to suggest a modification to the student function.

Since it is, in general, impossible to determine whether two programs are equivalent, it would seem, on the face of it, that Talus cannot succeed in its task. But this is misleading, since the task of program debugging in general faces this problem. The performance of Talus depends on the reference function and the student function implementing the same algorithm. Toward this end, Talus begins by recognizing which of its known algorithms for a problem is being attempted by the student. This is done by extracting algorithmic features from the student solution, such as what form of recursion is used, termination criteria, and the number of recursive calls. This information is used to select which of several reference func-
tions will most likely match the student function well enough so that Talus will be able to construct a proof of equivalence.

Although it would seem that this strategy rejects the thesis that it is necessary to infer the student’s intentions in order to adequately debug his programs, this is not the case. A well-structured program in pure LISP (which is what the reference functions are) is decomposed in a way that is similar to the sort of plan decompositions used in systems that reason more explicitly about student plans, such as Dudu and PROUST. So, for instance, a single loop in a language such as Pascal appears in pure LISP as a single function; the function for combining partial results (the collector in Dudu parlance) is the body of the function, the iteration (or navigator) is specified by the recursive call; the overall loop is uniformly specified by the details of the LISP interpreter (and formalized by the principle of induction used in the Boyer-Moore logic). Since the Boyer-Moore logic deals with these things in pure LISP, it is in fact reasoning with programming plans and goals.

The fact that a well-structured LISP program can often reflect a plan/goal decomposition of an algorithm allows Talus to make sensible bug fixes by selecting fragments of the reference function to replace defective fragments of the buggy solution. Talus will fail in cases in which the reference program does not faithfully reflect the intention structure of the solution. This could happen when some detail of implementation is included in the reference function, which is incidental to the teleology of the program. In pure LISP, function definitions are not very big, so
most implementation details are put into other functions. An exception to this arises in the choice of data structures; it is possible in pure LISP to record data in list structures of arbitrary form, so long as the parts of the programs that build the structures agree with the part that references them. Any function definition must make a commitment to some list structure to be complete. An example of this is shown in figure 5.4

A tree sort sorts a list of elements by constructing a binary search tree, and then traversing the tree in infix order to output the sorted list. The binary search tree has a value stored at each node. The value stored at a node is greater than the value stored at any node in its left subtree, and less or equal to any node in its right subtree.

The tree sort shown in figure 5.4 represents a binary tree as a list of three items; the first is the value stored at the node, the second is the left subtree, the third is the right subtree. This structure is maintained by the function ‘add-tree’, and used by the function ‘report’. This implementation detail is not relevant to understanding how the sort works, but must be included in the reference function, since the reference function solves the programming task.

Another tree sort solution is given in figure 5.5. This solution is almost identical to the solution in figure 5.4, except that the representation of the tree is given differently. In figure 5.5, a node in the tree is represented as a dotted pair, its CAR is the value at the node, its CDR is another dotted pair, which is (CONS
(DEFUN tsort (l)
  (report (make-tree l NIL)))

(DEFUN make-tree (l tree)
  (IF (NULL l) tree
      (make-tree (CDR l) (add-tree (CAR l) tree))))

(DEFUN add-tree (item tree)
  (IF tree
      (IF (LESSP item (CAR tree))
          (LIST (CAR tree) (add-tree item (CADR tree)) (CADDR tree))
          (LIST (CAR tree) (CADR tree) (add-tree item (CADDR tree))))
      (LIST item NIL NIL) ))

(DEFUN report (tree)
  (IF tree
      (APPEND (report (CADR tree)
                      (CONS (CAR tree) (report (CADDR tree)))))
      NIL)

Figure 5.4: A LISP program to perform a tree sort. Items are inserted one by one into a binary sort tree; a node in the tree is represented as a list of three elements. The first is the value in the node itself, the second is the tree of items less than this node, the third is the tree of items greater than this node. The function ‘make-tree’ builds such a tree from a list of items, while the function ‘report’ performs an infix traversal of the tree.
If Talus were to use the definition in 5.4 as a reference function, and encountered the definition of 'record' given in figure 5.5, it would have to conclude that the appearance of CDDR in the latter was a bug, and would correct it to CADDR. Similar changes would be made in 'add-tree'. This happens because Talus has no explicit representation for the intentions of the parts of the program, that is, Talus does not know what the data structure of the tree is for. This intention is not reflected in the structure of the LISP code. In Dudu or PROUST, the two solutions could be represented as implementational variations on the entity that is intended to be used as a tree.

5.2.1 Reference Functions in Talus and Functional Representations

In Talus, the reference function plays the role of a task specification. The job of Talus is to show that the sample program computes the same function as the reference function. Since the reference program is itself a working program, it must make implementation decisions, such as which termination test to use (e.g., LISTP versus ATOM). Because of the flexibility of the Boyer-Moore logic and theorem prover, these commitments are usually not important, and Talus can recognize programs that make very different decisions. Dudu uses a much simpler processor, that is, Dudu simply performs replacements as specified by the semantics of assignment. Hence Dudu uses a much more flexible representation of the program.
(DEFUN tsort (1)
  (report (make-tree 1 NIL)))

(DEFUN make-tree (1 tree)
  (IF (NULL 1) tree
       (make-tree (CDR 1) (add-tree (CAR 1) tree))))

(DEFUN add-tree (item tree)
  (IF tree
      (IF (LESSP item (CAR tree))
          (CONS (CAR tree)
                (CONS (add-tree item (CADR tree)) (CDDR tree)))
       (CONS (CAR tree)
                (CONS (CADR tree) (add-tree item (CDDR tree)))))
       (LIST item NIL NIL)))

(DEFUN report (tree)
  (IF tree
      (APPEND (report (CADR tree)
                      (CONS (CAR tree) (report (CDDR tree))))))
       NIL))

Figure 5.5: Another LISP program to perform a tree sort. Items are inserted one by one into a binary sort tree; a node in the tree is represented as a dotted pair; the CAR is the value in the node itself, the CDR is a dotted pair made up of the tree of items less than this node and the tree of items greater than this node. The function 'make-tree' builds such a tree from a list of items, while the function 'report' performs an infix traversal of the tree. This program is identical to the one in figure 5.4 except it uses CDDR (in place of CADDR) for the right-hand branch of the tree.
The functional representation used by Dudu does not make any commitment to a particular implementation.

Since the Dudu processor does not have the power of the Boyer-Moore theorem prover, and in particular it does not have the principle of induction provided by the Boyer-Moore logic, the principle of induction must be dealt with elsewhere. Dudu does this with the proof schema that is provided with the generic type total loop. Since the total loop is not intended to represent all loops, the inductive schema associated with it need not be fully general (whereas the principle of induction in the Boyer-Moore logic is fully general, and must decide what form the induction proof must take).

Finally, since the functional representation does not specify the implementation in needless detail, it is not necessary to prove that the details of the student program match the details of the reference program. Instead, Dudu is able to provide an appropriate proviso that indicates what needs to be proven to complete the proof. Dudu does not construct a complete proof whenever it is presented with a program, only the parts that need to be verified.

5.3 Classical Program Verification

Many semantics-based approaches to program debugging are not targeted particularly to debugging student programs, but rather to assisting a trained, professional programmer to write reusable, verifiable code. An example of such a system is the
REFORMS system developed by Joan Krone [14].

REFORMS stands for Reusable, Efficient, Formally specified and verified, Modular Software system [Krone, 1988]. As the acronym implies, the motivation for this work is to provide a software system which will allow a programmer to write programs with the desirable properties of reusability, efficiency, formal verifiability and modularity. In return for these services (for example, the formal verifier), certain disciplines are enforced on the programmer. This is not unlike the approach in structured programming languages, where certain modularity and understandability of the code is guaranteed, provided the programmer follows the disciplines of structured programming. In the case of REFORMS, this discipline is enforced, in part, by the automatic program verifier.

These motivations help to determine what is represented in the REFORMS language. For example, Krone takes very seriously the notion of reusability, in the sense that she wants to allow a programmer to specify a program which can satisfy its specifications in a variety of implementational contexts. For instance, the operations for the abstract data type stack (e.g., push, pop, etc.) should be usable for stacks of integers, strings, or more complicated structures, even including other stacks. Furthermore, an application calling for a stack should not be concerned with the details of how the stack is, in fact, implemented. These concerns have motivated Krone to make a separation between the conceptual module and realization module, i.e., between the specification of the stack at an abstract level and
the details of its implementation at a more concrete level.

This treatment is not reserved for operations on abstract data types; all procedures are described with a conceptual module as well as a realization module. In this way, Krone allows the programmer to represent his program in terms of data types and program plans. She allows for a representation of teleology, separate from implementation. She allows for generic types of data objects.

5.3.1 Teleology Apparent in Programs

REFORMS takes advantage of the fact that in a well-structured program written by a disciplined, professional programmer, the statements, modules and data structures will reflect the intended teleology of the program. Every module in REFORMS (whether it is a data type or a simple subroutine) is specified along with its conceptual model, so she avoids the problem in Talus of having the teleology of data structures obscured by their implementation details.

Since teleology in REFORMS is specified in terms of the program itself, the only entities for which REFORMS can specify teleology are entities in the program itself. This includes data structures and control structures (and abstractions of these). This does not include entities in programming metaphors, such as the Record described above. Since Krone is interested not in tracking down bugs and providing tutorial diagnostics, but rather with providing provably correct code for a professional setting, REFORMS can rely on the programmer to enforce whatever
metaphors he finds useful.

Finally, REFORMS is not intended to infer program teleology, only to make use of it in providing reusable, verifiable code. Thus REFORMS can require that the programmer possess sufficient mathematical sophistication to formulate loop invariants and other specification of code teleology to allow REFORMS to automatically guarantee the correctness of the program.

### 5.4 Other Work in Program Understanding

A good deal of work in program understanding has been done in the context of 'automatic programming', in which the goal is to automatically design algorithms or automatically complete implementations of algorithms. Such approaches are typically more interested in understanding program specifications than in understanding programs. I present a few such approaches here to examine what issues in program understanding they need to deal with.

One class of automatic design system is the transformational approach. An example of such a system is the DIOGENES system of Mostow [19]. In DIOGENES, algorithms for heuristic search are specified in terms of constraints which must be met by a solution. An inefficient 'algorithm' (the word 'algorithm' is used in scare quotes because the initial structure might specify such non-algorithmic operations as 'searching exhaustively an infinite set') is generated to search for a solution satisfying the constraints. Then, transformations are applied to the 'algorithm'
to optimize (and to make it actually computable). The transformations used by DIOGENES include rules to replace a mention of a set with an algorithm for generating it in some order, eliminating redundant steps, and moving tests to more efficient parts of the code. These rules are all heuristics for optimizing algorithm fragments. Transformational approaches take care of all the structuring of the program internally, consulting the user on specification questions only, so there is no notion of representing understanding or intentions.

Another notable transformation based system is KIDS, developed by Doug Smith at the Kestrel Institute ([29], [28]). Unlike DIOGENES, the transformations used by KIDS are all correctness preserving. The transformations are based on finite difference methods and partial evaluation, which together can produce highly optimized code. As in other transformational systems, intentions are not explicitly represented. KIDS is used by offering some high-level transformation choices to the programmer (e.g., select algorithm type, divide-and-conquer, generate-and-test etc.), which result in a high-level specification that is transformed into an efficient program. That is, KIDS (and transformational systems in general) suggests a new style of programming, in which the program is understood in terms of the user-supplied transformations.

The Programmer's Apprentice [24] project is based upon inspection methods, that is, a program is synthesized or analyzed based upon the appearance of certain well-known programming clichés. These are called inspection methods because they
depend upon inspecting the code in search of instances of these clichés. Since the
Programmer's Apprentice is directed toward assisting program synthesis, the use
of these clichés can be seen as a specification of the programmer's intentions (this
is similar to the situation in REFORMS).

Clichés are represented in a formalism called the *plan calculus*. The purpose of
the plan calculus is to abstract away from the details of algorithms and data struc­
tures that rely only upon the particulars of some programming language. That is,
the same serious consideration of reusability motivates both REFORMS and the
Programmer's Apprentice. Just as in REFORMS, the Programmer's Apprentice
makes a clear distinction between the specification of a cliché and its implementa­
tion. Unlike REFORMS, the Programmer's Apprentice does not offer provability
of the completed programs, hence it does not explicitly enforce any methodology
on the programmer.

The Programmer's Apprentice, unlike REFORMS, has spent a good deal of en­
ergy on acquiring a library of useful clichés. Hence, the Programmer's Apprentice
pays more attention to the relationship between clichés. Clichés can be specializa­
tions of other clichés, with the specialization inheriting the capabilities of the more
general cliché. This is done by bundling together several clichés related to some
kind of object into an abstract data type. These can be related in a hierarchy of
specialization. If a programmer specifies some data type, and subsequently calls for
a cliché which is only supported by some specialization of that type, the apprentice
automatically replaces the more general type with the appropriate specialization (informing the programmer of this change). Since the representation in the form of the Plan Calculus is flexible with respect to reusability, such a change is easy to make.

The plan calculus of the programmer's apprentice shares many features in common with the functional representation of Dudu. It supports a hierarchy of detail, that is, it is possible to express an algorithm in great detail, or in high level terms suppressing detail. Unlike the representation in Dudu, the plan calculus does not provide a capability to connect the program representation to the program semantics.

Finally, Designer/SOAR ([30]) is similar to transformational approaches in that it designs algorithms, rather than programs, and hence is not concerned with details of data structures or variable types. It also is based on transforming knowledge of the algorithm to be solved from some domain specification to an algorithmic specification. The task of Designer/SOAR can be seen as that of learning how to accomplish some well-understood task given some particular computational primitives.

Designer/SOAR begins with a problem space (the application domain) which represents the problem domain, including operators for working within that domain. These operators include an operator for performing the actual operation for which an algorithm is to be designed, e.g., sorting. The task of Designer/SOAR
is to express sorting in the *computational model*, which is another problem space, whose operators include algorithmic primitives and high-level strategic operators, such as Divide-and-Conquer. That Designer/SOAR begins with knowledge of how to sort is not really begging the question, since “no one designs a sorting algorithm who does not know how to sort.” [30]

Designer/SOAR proceeds by trying to achieve the sort goal in the algorithmic space. Since no sort operator is defined, Designer/SOAR decides to try the Divide-and-Conquer strategy to accomplish a sort (this is chosen for its abstract similarity to sorting). This decision is learned by SOAR’s chunking mechanism as a solution to sorting problems.

The steps of Divide-and-Conquer are approached in the same way. High level operators are chosen either from their abstract similarity to the desired effect, or they are chosen at random from several applicable operators (different choices at these points result in different sort algorithms). For elementary operations, test cases are generated and operators are applied in the application domain to determine what computational operator is appropriate.

Each decision point is an *impasse* in the computational model, so when SOAR chooses an operator to solve some problem, that operator is learned by the chunking mechanism for that impasse. Hence, the final representation of the algorithm is chunked operators in the computational model.

While it does not seem to be the intent of the authors of Designer/SOAR to
make statements about representation of program understanding, it is interesting
to see what the Designer/SOAR experiment has to say about this. We can view the
application domain as a representation of the programmer's formal understanding
of the domain for which he is programming, including an understanding of how
he plans to solve the problem. The designed program is just a reformulation of
this procedural understanding in the terms of the algorithmic constraints, as repre­
sented in the computational model. If we view Designer/SOAR's representation as
a theory of program understanding, this states that a programmer's understanding
of his program is just his understanding of the domain, appropriately translated.
Since SOAR's theory of learning is based entirely on chunking, this leads to the
result that algorithms are merely appropriate chunks of domain knowledge. Hence
Designer/SOAR has little to say about program understanding, but rather tells
how understanding of an application domain can lead to an algorithm.

Since Designer/Soar designs algorithms rather than programs, it makes no
sense for it to be able to reason about semantics of a programming language.
Designer/Soar makes no attempt to analyze programs or algorithms, only to design
them, hence it does not model the intentions of the original algorithm designer.
This chapter summarizes the contribution Dudu makes to the topic of automatic debugging. First, I will examine the various types of representation styles that could be used for dealing with programs, and describe how the Functional Representation combines important features of several representation styles. Then I show how this representation allows Dudu to retain the power of intention based systems like PROUST, while being able to reason with program semantics, as does Talus. Not surprisingly, the intentional reasoning allows a more focussed approach to correcting proofs, which allows Dudu to get by with a much weaker theorem prover than was used by Talus.

6.1 Comparison of Representation Styles

In the previous chapter I mentioned several representations of programs. These have been used to support programming systems, algorithm design systems, and debugging systems. Nevertheless, in this work, I have chosen to use the Functional
Representation as a representation for program understanding. In this section I will outline the features of the various representations, and why the Functional Representation combines the features appropriate for program understanding.

6.1.1 Device Representations

Device representations, such as the one described by Rieger and Grinberg ([26]) represent device behavior by first representing device states and transitions between them. Device representations may also allow for detail suppression in state/transition diagrams (e.g., ABEL ([22]). Such representations are usually used to support simulation reasoning, that is, to simulate the behavior of the device under certain conditions, or for diagnostic tasks. States in such representations are associated with observable or reportable aspects of a device. Links are usually interpreted causally, that is, a link indicates that one state causes another. Device representations are not usually used for planning tasks (an exception to this is [2]).

6.1.2 Plan Representations

Representations for planning domains (like that used by Sacerdoti in NOAH [4] or by Johnson in PROUST [10]) typically view plans as intentional objects, that is, parts of plans are present for a purpose. The purpose of a plan is represented as a goal, and the body of the plan consists of further goals, or subgoals, which must be achieved to complete the plan. Planning consists of selecting plans to achieve goals, and selecting further plans to achieve the subgoals, until all goals are met.
PROUST uses such a representation to perform debugging using a method called 'analysis by synthesis'.

Goals in plan representations are often formulated in terms of predicates, e.g., a goal might be 'to achieve (AND (ON a b) (ON b c))'. Thus a goal plays the double role of indexing into plans (since plans are chosen by the goals they may address), and specifying the state of the system after the goal is achieved.

### 6.1.3 The Functional Representation

The Functional Representation is a device representation based on the teleology of devices. As such, it combines the intentional structure of plan representations with the behavior descriptions of device representations. Behaviors in the Functional Representation are described with state/transition graphs, as are behaviors in device representations, but are indexed by the intended function of the component, as in plan representations. The transitions in the state/transition graphs index other functions, and thus play the indexing role of goals in plan representations.

### 6.2 Functional Representation and Program Debugging

#### 6.2.1 Functional Representation and Program Proofs

In section 5.1, I showed how the functional representation, when applied to programming as in Dudu, can duplicate the plan structures used in PROUST. This is done by identifying behaviors in the functional representation with PROUST plans, functions in the FR with the indexes for PROUST plans that link plans
to the goals they address, and the links in the FR behaviors to the subgoals in PROUST.

In [20], Murray notes that PROUST could benefit from being able to reason about program semantics. The solution given by Murray is to replace the explicit plan structures used by PROUST with a reference function, and to make use of the Boyer-Moore logic and theorem prover to prove the equivalence of the reference function to the student's function. He maintains many of the advantages of a plan-based system by virtue of the fact that the well-structured reference functions tend to reflect the intentions of the programmer, and the Boyer-Moore theorem prover follows this structure in constructing its proofs. In section 5.2, I showed how for even a well-structured LISP program, its intentional structure might not be apparent, and thus it can lead Talus to imperialistically modify a correct student program. That is, Talus could benefit from being able to reason explicitly about the intentional structure of programs.

Dudu demonstrates another way to bring knowledge of programming semantics to an intentional structure; Dudu matches the states in its device representation with formulas in the proof of correctness of the program. This allows Dudu to match program just like PROUST does, but also to examine exactly the part of the proof that needs to be repaired to account for unexpected variation in the examined code. Two examples of this performance have been given. In the example of the shift loop (see figure 4.4), the initialization and looping code was matched to stored
templates; the body of the loop was novel. Dudu was able to recognize the expected code; but without having an explicit plan for the particular variation in figure 4.4, it was able to consult the programming language semantics of the novel code, and patch them into the correctness proof of the expected program. Dudu was able to do this even though the novel code was embedded in a loop, and hence the changes to the proof involved integrating the novel semantics into an induction proof. That is, Dudu was able to act like PROUST as long as the program conformed to its expectations, and was then able to work with the program semantics of the novel code to complete the verification.

Dudu merges the two approaches in an even more profound way. Since Dudu is intention based, before Dudu attempts to complete a correctness proof, it has a good idea what the proof will look like (if the program is completely mundane, Dudu already knows exactly what the proof is like). This allows Dudu not only to restrict the proof completion to the novel parts of the proof, but also to anticipate what problems the reconstructed proof might encounter. That is, Dudu encodes in provisos the expected problems the proof might encounter. For example, in figures 4.2 and 4.3, Dudu generates the proviso that the left-hand side of the assignment statement must be empty; this allows Dudu to construct an approximate proof of correctness for the program. Although the approximate proof does not prove the correctness of the program, it does provide convincing evidence. Furthermore, if Dudu fails to construct a proof for the proviso, then Dudu is in a position to report
COUNTER

Variables: ?Count

Templates:
Init:  
  ?Count := 0
  (in Process: component of goal Read & Process)
Update:  
  ?Count := ?Count + 1

Figure 6.1: COUNTER plan from PROUST [10]. The variable ?Count appears in various places in the program, but has a clear intention, that is, to count the input. The various parts of the plan do not have to refer to contiguous code.

6.2.2 Functional Representation and Intentions

PROUST represents the intentions of the programmer in terms of program plans. These are eventually mapped onto fragments of program code. Certain programming entities (variables, for example), have a clear intentional interpretation (e.g., this variable counts the valid items encountered so far), but do not appear in contiguous fragments of code. PROUST handles this by allowing plans to specify non-contiguous fragments of code; for example, the COUNTER plan shown in figure 6.1 specifies that the counter must be initialized to 0, and that its increment must appear in the 'Process:' component of whatever plan realizes the Read & Process goal.

Dudu would represent this counter as a special case of a collector, and the various components of the plan would appear as different functions of the device.
counter. Other plans, such as the running total, maximum and running average would be represented by Dudu in the same way. Dudu joins all such devices together in the class of devices called collector, in which the common features of these plans, such as the appropriate provisos, are stored.

Furthermore, since Dudu represents a program as a device rather than a plan, Dudu can represent programming metaphors, which never appear as lines of code at all. For example, Dudu can represent that a program is using the metaphor of a record, which can neither be created nor destroyed. The records themselves do not appear in the program code, but the fact that they are neither created nor destroyed can provide provisos that assist in constructing approximate proofs of correctness (see section 3.6.3).
CHAPTER VII

Horizons

In this work, I have described how the Functional Representation can be used to support the task of program debugging. But debugging is not the only task that experienced programmers can do with their programming expertise. In this chapter I will outline how the approach of a Functional Representation can be used for other tasks concerning programs, such as program design and maintenance, and explaining a program to a human reader, and how this relates to a description of understanding. Finally, I will suggest how the description of understanding provided by a functional representation can help to apply this work to other domains that involve understanding, in particular, for theories of scientific discovery.

7.1 Adaptive Design and Programs

In [7], Goel has shown how to use the functional representation to perform a task he calls 'adaptive design'. He outlines his method with the following pedagogical example of adaptive design; a circuit has been built to shine red light when a button
is pressed. A new specification is given, in which it is desired that a blue light will shine when a button is pressed. First, Goel determines what needs to be changed in the overall function of the original circuit. The original circuit is represented with a functional representation, based on the representation in [27]. Then this representation is searched to locate the appropriate constituent responsible for the change in function. A modification to the device structure is proposed; in this case, the proposed modification is to replace the red bulb with a blue one. Finally, it is necessary to verify that the modified structure in fact delivers the appropriate functionality. In this case, this amounts to verifying that the circuit actually does produce a blue light when the button is pressed. Because of the structure of the functional representation, it is not necessary that Goel’s system verify the entire system (e.g., that current is delivered to the bulb when the button is pressed), only that the bulb will behave as desired when the button is pressed.

The adaptive design task is important in the software engineering life cycle, in that it represents program maintenance activities. It is often necessary to modify a program to satisfy new specifications. Also, it is often effective programming practice to be able to use program code from one task again in another. In this section I will examine how well the adaptive design strategies used by Goel are suited to similar tasks in the programming domain.

Goel keeps functional representation fragments in a ‘case base’ of designs; that is, he maintains a library of functional representation fragments. A great deal
of the problem of adaptive design lies in the retrieval of the appropriate design fragments, both of the original design to be modified (in the example above, the design of the original circuit), and the component(s) to be replaced (in the above, this is the light bulb). In the case of program maintenance, one can presume that the functional representation of the original program is known and the library will be used only to find the replacement components; in the case of software reuse, the library will have to be used, as in Goel's system, both to find an original design and a modification to it.

Goel's case base is indexed by the functions of the devices within. These are specified, as are the functions in this work, by the states that activate them and that they obtain. Thus, it is necessary that Goel defines a state language in order to be able to search for these indices. Goel uses a graph formalism to represent states, whereas in this work I use a logical formalism. Nevertheless, I will show by an example that Goel's method can be applied to the functional representations presented here.

Consider the functional representation of the shift loop as described in figures 2.11 and 2.12. The programs described by this functional representation satisfy the specification that they move the elements from \( k \) to \( n - 1 \) in an array 'up one place', that is, to locations \( k + 1 \) to \( n \) respectively. Suppose instead that we want to build a program instead of moving the items up one, moves them down one.

The original program had the functionality that if \( \forall j \in [k..n - 1], a[j] = \%a[j], \)
then after the program completes, \( \forall j \in [k..n - 1], a[j + 1] = \#a[j] \). The new program has the similar functionality that if \( \forall j \in [k..n - 1], a[j] = \#a[j] \), then after the program completes, \( \forall j \in [k..n - 1], a[j - 1] = \#a[j] \). From figure 3.2, we find that the original program can be represented as a total loop, with \( U \) bound to \([k..n - 1]\). The new functionality uses the same set \( U \), but needs a different binding for \( \Gamma \). In the original case, \( \Gamma_1(S) = \forall j \in U \setminus S, a[j] = \#a[j] \land \forall j \in S, a[j + 1] = \#a[j] \).

All that is known for the new case is the value of \( \Gamma(S) \) for \( S = U \) and \( S = \emptyset \). \( \Gamma_1 \) does fine for the former, but for the latter, \( \Gamma_1 \) needs to be changed to \( \Gamma_2(S) = \forall j \in U \setminus S, a[j] = \#a[j] \land \forall j \in S, a[j - 1] = \#a[j] \). This change must be reflected in the program by a change in the collector, rather than in the navigator. Since for both definitions of \( \Gamma \), \( \Gamma(U(\sigma)) \) is the same, the function initialize of the collector need not be changed. Finally, since the function gather of the collector is not the same for the two specifications, the functional representation can localize the change to the line of code that implements that function, i.e. the line \( a[?x + 1] := a[?x] \). So far this outline follows the general algorithm of redesign presented by Goel.

In order to decide how to change the line of code, Goel searches a plan library to find another plan that can be substituted for the current line to achieve the desired functionality. In this case, it is necessary to find a line that will guarantee \( \Gamma_2(U(\nu(?x))) \) given \( \Gamma_2(U(?x)) \) (from the definition of the function gather of the collector in figure 2.12), that is, given \( \forall j \in [k..?x], a[j] = \#a[j] \land \forall j \in [?x + 1..n - 1], a[j - 1] = \#a[j], \forall j \in [k..?x - 1], a[j] = \#a[j] \land \forall j \in [?x..n - 1], a[j - 1] = \#a[j] \)
(substituting in the values of $U$ and $\nu$ from the navigator). It is easy to show (that is, it could be shown automatically) that in order to do this, it is necessary to change the value of $a[?x - 1]$ to $a[?x]$. In the plan library, single lines can be indexed by their axiom schema in the programming language semantics, and an assignment of $a[?x - 1] := a[?x]$ could be chosen. However, such a choice (based on the traditional semantics of the line of code) would make it difficult to prove the new program correct. Instead, the plan library should be indexed according to the *functional* semantics of the code (see page 106 for a description of functional semantics), that is, the plan to be selected should be a *move* function of a *record*, rather than a simple assignment. It remains a question of further research to determine whether the memory organization outlined in Goel is sophisticated enough to support indices based on functional semantics.

Now it is necessary to demonstrate that the corrected device in fact does work as desired. For Goel, this involves using a simulation model of the designed device to verify that the appropriate behavior is obtained. In this case, since a functional semantics is defined for the replacement fragment of code, this involves verifying any proviso that comes with the new code fragment. In this case, since the new assignment, $a[?x - 1] := a[?x]$ is represented as a *record move*, the appropriate proviso (see figure 3.5) is that $(\text{empty} a[?x - 1])$. This can be verified in exactly the same manner as the proviso $(\text{empty} a[?x + 1])$ was verified in chapter II, that is, by tracing back through the functional description of the program as dictated
by the inductive form of the proof provided by the generic description of the total loop. In this case, as in the second example in chapter IV, the proof of the proviso fails (since a navigator that moves down the set does not empty the destination for the move).

Since the modification has failed to produce a correct program, any of many steps might be taken. At this point, Goel abandons this particular modification, and seeks another way to correct the program. One such plan would be to perform a protected move, that is, to have a plan that introduces extra variables to guarantee that the destination is empty. This would result in a program similar to the program shown in figure 4.4. Another possible continuation would be to further patch the new program; since the proof fragment that demonstrated the fault involves the functional semantics both of the collector and navigator, either of them might be replaced in order to correct the program. A topic for future research is to determine how to determine when to abandon such a line of corrections. In this case, the cleverest solution is to replace the navigator with one that traverses the set in the opposite order, that is, to pursue this sort of patchwork reasoning for one more iteration.
7.2 Functional Representation and Program Comprehension

A task that is sometimes required of human programmers is to examine a program, and come to some 'understanding' of how it works. Usually the programmer has some idea what the program is expected to do, but very little idea how it goes about doing it. Often the programmer is guided by comments left by the original program designer. Usually a programmer is asked to do this in order to support the performance of other software maintenance tasks, such as debugging the program, proving it correct, modifying it, or explaining its abilities and limitations. That is, this task can be seen as building a functional representation of the program.

In order to perform this task, it will be necessary to be able to recognize parts of the program as matching some functional representation fragments in a library of such fragments. Johnson treats a version of this problem in [10] with the matcher PROUST uses to recognize plans in student programs, along with the heuristics for deciding which match is best. However, in order to build a useful functional representation of the program, it is necessary that the various library units define a functional semantics that is different from the traditional semantics, that is, the levels in the functional representation genuinely mark a difference in descriptive level of abstraction. Selecting between different abstractions that match the observed code equally well is a difficult problem; this is where humans resort to hints from program comments (if any such exist), or simply fail to complete the
task.

7.2.1 Why Plan-like Representations?

One could question whether it is necessary at all to construct a plan-like representation of a program in order to comprehend it. In [23], Pennington presents evidence that human expert programmers do not represent programs primarily as plan structures, but rather as syntactic chunks whose organization matches that of the program text. Nevertheless, my own preliminary examination of programming instructors shows that a mastery of various plan-like concepts is necessary for the communication of programming expertise. The examination procedure consisted of selecting, at random, three students from a beginning programming course, and assigning them the following problem (the course instructor estimated that this problem would be a challenge for these students):

Write a program to sort a list of \( n \) integers into increasing order. For example, for \( n = 5 \), and the input array containing the values 12 7 3 5 7, your program should finish with the array contents in the order 3 5 7 7 12. A lot of research has been done on different methods for sorting lists. You are to use a method called insertion sort. An insertion sort works on the following principles:

- a list of one element is already sorted.
• given a sorted list of some length and a single item, you can build
  a sorted list one element longer by inserting the element in the
  list at the right spot.
• you can find the right spot by making sure that every item before
  the right spot is less than the item to be inserted.

Your job is to decide how to use a single FORTRAN array (it will be
used as both input and output) to implement these things to form a
program for insertion sort. Write your program as a single procedure.
Then write a detailed description (at least a page long) of why your
program works. Tell what role each variable plays in your solution.

Two students correctly solved the problem. The incorrect solution was badly
flawed, and bore very little resemblance to the instructor's solution. In a person
to person session with the instructor and the student, the instructor began by
verifying that the student understood the first goal mentioned in the assignment,
namely, how to make use of the fact that a list of one item is always sorted, and
asking the student how that goal was achieved by his code. When it was discovered
that the student had not addressed that goal, the instructor tried to find where
the student might have addressed the other goals in the algorithm, namely how he
chose to insert an element into a sorted list. Along the way, the instructor used
metaphors for data objects to explicate the interaction of the various goals. It was
eventually determined that the student did not implement the assigned algorithm, and had a completely different goal structure. At this point, the discussion changed to an examination of the code itself.

This suggests that plan-like descriptions are preferred for communicating programming concepts, at least in the presence of uncertain code. In the cases examined by Pennington, the programs were known to be correct, but the subjects did not know what the programs were intended to do, while in the tutorial situation, the instructor knows what was intended by the programmer, but does not know that the program is correct. In such cases, the instructor makes use of knowledge of the program plans and metaphors in order to organize the possibly faulty code.

The instructor was quite strict in his use of program plans. In fact, in order to teach the algorithm to the student, he tried to convey the appropriate goal structure, as a 'better' solution to the problem. That is, a particular plan structure represents how a student programmer is being trained to think about a program. The understanding that is to be represented in this work is this sort of understanding; the appropriate goal structures that are to be taught to novice programmers.

This is the sense in which the functional representation of a program constitutes a representation of the understanding of a program. The way we attribute understanding to a human programmer who purports to understand a program is to check whether 1) the programmer can report how a set of goals are met by the program, and 2) the programmer can use this organization to perform various
tasks, like debugging, maintenance and explanation. The Functional Representation demonstrates how goals can be related to a program and its semantics, so that the program can be justified in terms of its goals, and Dudu shows how the functional representation can be brought to bear on a programming task.

7.3 Functional Representation and Scientific Discovery

Some work has been done very recently by Moberg, Darden and Josephson ([17]) on using the functional representation to represent a scientific theory, in particular, a biological theory. The parallels that work draws between devices and theories are even more poignant when compared to the device representations of programs used in this work.

- Biological systems have a functional organization; they are made of parts that play roles in achieving goals for larger systems.

- Just as different programs can have superficial differences that are not essential to the algorithm they (both) implement, so can different scientific theories have superficial differences that are not essential to what commitments they really make.

- Just as programs may be similar at a high level, but use different components to accomplish certain goals, scientific theories can agree at a high level, but hypothesize different mechanisms for various details.
In [17], scientific theory change is modeled as a process of theory diagnosis in the face of an anomaly, followed by theory redesign. It has already been shown ([5], [18]) that the Functional Representation is useful for locating faults in a theory that is presented with an anomaly. A topic for future research both in the scientific theory domain as well as on the device representation problem is to apply the methods provided in this dissertation to the theory change problem. This work provides a method for defining the meaning of states in a functional representation, a definition of the consistency of the functional representation, a method for finding bugs in a functional representation in some cases in which the states are not observable, and a proposed method for rebuilding a functional representation if it no longer satisfies its specification. These capabilities match well with the requirements mentioned in [17].

7.4 Limitations of Dudu

The point of this work is to study the representational requirements of reasoning about programs. I have shown how purely plan-based representations and purely proof-based representations have severe drawbacks. In this work I use the functional representation to gain the advantages of both of these approaches, and thus cover for the shortcomings of each. Since the focus of this work is on representation, I have not stressed some aspects of a debugging or automatic programming system that have been treated more completely elsewhere. These include issues of
matching a representation to actual source code (which is treated in some depth in PROUST ([10]) and the Programmer's Apprentice ([25])), and program presentation (which is treated in great depth in the Programmer's Apprentice ([25])).

Dudu deals directly with source code, without processing it into any intermediate form (such as data flow graphs, or plan structures). Dudu matches program code directly to the functional representation. If the program matches the anticipated plan structure reflected by the functional representation, as is the case for many of the examples in this work, this is sufficient to determine a match. For the examples in which novel code is encountered, Dudu is still able to find a match, because the plan structure of the novel programs is still similar to the plan structure reflected by the functional representation.

In cases in which the variation in the source code results in an apparent variation in the plan structure, the use of the functional representation by Dudu is too restrictive. Consider the example in figure 7.1. Suppose that the functional representation of the program has matched enough so that Dudu is expecting the first two lines in the figure, but the actual program is as shown in the bottom of the figure.

Since Dudu is demonstrating the properties of the representation, for the goals of this work it is sufficient to show that the functional representation can represent a plan of this form. This solution is not adequate for using the functional representation on problems of any size, since it presupposes functional representation
Dudu's expectations:
\[
\text{if } \text{max} > \text{item} \text{ then } \text{max} := \text{item}; \\
i := i + 1
\]

Program code:
\[
\text{if } \text{max} > \text{item} \text{ then do;} \\
\hspace{1em} \text{max} := \text{item}; \\
\hspace{1em} i := i + 1; \\
\hspace{1em} \text{end;} \\
\text{else } i := i + 1;
\]

Figure 7.1: Apparent mismatch between Dudu's expectations and program code (example from Lewis Johnson, personal communication).

One way to cope with this sort of problem was used by Johnson in PROUST ([10]). That is to have 'plan difference rules' that will generically unwind tangled plans like the ones shown in figure 7.1.

Dudu could cope with the example in figure 7.1 if it were able to realize that the if ... then ... else structure in the figure was supposed to correspond to the two lines given; this is not necessarily obvious, if other novel lines surround these code fragments, and since the matcher used by Dudu does not make any sophisticated use of program syntax. Were these problems to be addressed, Dudu would be able to focus semantic reasoning on showing the equivalence of the two code fragments in the figure (which is a fairly short proof, given the semantics of a conditional statement).

This approach is not sufficient for more complex examples, such as the one shown in figure 7.2. In this example, there are two collectors for a single navigator.
Dudu's expectation:
for i from 1 to n do;
    if max > a[i] then max := a[i];
end;
for i from 1 to n do;
    if min < a[i] then min := a[i];
end;

Actual code:
for i from 1 to n do;
    if max > a[i] then max := a[i];
    if min < a[i] then min := a[i];
end;

Figure 7.2: Two loops optimized into one. Dudu would need a separate representation for the combined loop. (example from Rich and Wills, [25])

Again, it is possible to represent such a loop using the functional representation as described in this work and used in Dudu, but for a practical system, this would require an unbounded combination of devices to be represented separately. Proving the equivalence of these two fragments from the basic axiom schemata of the programming language is now much more difficult.

The Programmer's Apprentice has a component called the Recognizer ([25]) whose job is to sort out exactly this sort of combination of plans. It works by converting the student's code into a data- and control-flow graph, which is parsed to find instances of programming clichés; even if two clichés overlap, as they do in this example, the Recognizer finds them in this graph.

The data- and control-flow graph serves two purposes in this solution; it is created using the syntax of the original program, so it takes initial program struc-
ture into account (which Dudu does not do). It also suppresses certain syntactic
details, which simplifies the representation necessary for clichés. This feature is
also a drawback, in that the suppressed syntactic detail might hold the clue to a
misconception (in the case in which the system is to be used for computer assisted
instruction), or to a more mundane error, such as a misspelled identifier. Such
mundane errors can affect the syntactic structure, which could be very misleading
to a structure based analysis like the one used by the Recognizer. For pattern-
based matchers like the ones used by Dudu and PROUST, this is not a problem,
since the code is directly matched to plans.

There is a classic tension here; direct device/plan matching systems (like Dudu
and PROUST) can easily bring high knowledge of the program goals down to the
code to find basic errors (or basic misconceptions, in PROUST's case). They pay
the price by possibly requiring a proliferation of plans to account for the diversity
of the input code. This diversity can be removed by a pre-processor like the
Recognizer, before the top level plan structure has been considered. This solves the
problem of proliferation of devices, but removes information that was potentially
useful to high-level device recognition (like the spelling of variable names).

The advantage that Dudu's approach brings to program recognition is to be
able to recognize novel variants by consulting appropriate parts of a proof of cor-
rectness. As such, it is not committed to either matching scheme (although the
implementation uses a code matching scheme). Since the Recognizer depends only
on having clichés represented as graphs, and the graphs used by the functional representation are similar to the cliché graphs used in the Programmer's Apprentice Plan Calculus, the approach used by the Recognizer should be applicable to the functional representation as well; this would allow Dudu to recognize its class of novel variants without the proliferation of devices it requires now. Such a system would still not have PROUST's ability to debug misspelled identifiers.

7.5 Limitations of the Functional Representation

All of the work on the Functional Representation, either in this dissertation or others ([27],[13]), has presumed that a device is already understood at the outset, that is, that the functional representation is already known. It is not a trivial problem to construct a functional representation of a device, even if the device is well-understood. The generic devices described in chapter III provide some assistance in this process, as does the library of functional representations presented by Goel ([7]).

If a device is not understood at the outset, as is the case in a creative design task, then in order to take advantage of the structures presented in the functional representation, it is necessary to build a functional representation of the device as design proceeds. The naïve way to construct such a representation would be to enforce the 'no-structure-in-function' ([6],[12]) principle on the items in the case base. Then design could proceed by selecting some functional representation of a
component that satisfies the design goals, instantiate it in the current context, find what design goals were spawned by this choice, and continue. A representation built in this way would lose many of the advantages that made the Functional Representation attractive in the first place. In particular, a motivation for the functional representation was that the description at a higher level of abstraction could be in a different language (say, a language of bags of records) than the description at a lower level (say, as arrays of integers). If the various components were to be described without reference to their use (as proscribed by the no-function-in-structure principle), they could not refer to the functions they are intended to support. On the other hand, representing parts with no attention to the no-function-in-structure principle would make the library of functional parts much too large to be practical, since every possible use of every device must be represented. It is not surprising that this is a difficult problem, since creative design is a respected ability in talented humans.

In this work, the problem of the meaning of the states in a functional representation was solved by relating the states to a formal object, namely a proof of correctness of a program. The structure of the proof constrained the use of these states. In domains where no such formal object underlies the justification for device representation, it remains to be seen whether there is a semantics for states that provides all the capabilities provided by the state descriptions in this work.
Appendix A

Functional Representation of the Generic Search Loop

The total loop, as described in chapter III, is not a description of loops in general. It represents loops that compute a function of an entire set of items, by computing the partial result of that function applied to the items one by one. A running total is a simple example of this sort of computation, hence the name total loop. Another common sort of loop is one in which there is no intention that every item in a set will be processed. A search is an example of such a loop; often the point of a search is to avoid processing some elements of a set. The loops represented by the description of a search loop in this appendix share the following properties:

- There is a set of items to be searched. This set is arranged in a space, which includes nodes that correspond to the items, along with some landmarks and directions that link nodes together.

- There is a key, which identifies when a cell satisfies the search. That is, the program is searching the space for the key.
• There is some way of navigating the space that exploits its landmarks to find a node. There may be several sorts of landmarks to be exploited, so a navigator may have to be rather complicated. The navigator must be able to tell when the space is exhausted, and when it has found a node it is looking for.

A linear search through an array can be cast in these terms by allowing the array to play the role of the space, and an index into the array the navigator. There is only a single sort of landmark in a simple array; any node has a next node, given by the index plus one. The navigator moves from one cell to the next, checking for a key. A sorted array has more structure; a more complicated navigator can take advantage of this. If the navigator is made up of two indices, which constitute an interval in the array, then the structure of the array provides intervals that are less than the current node and intervals that are greater than the current node. Three arrays arranged as a binary tree (an array of the values of the nodes, an array of pointers from each node to the left branch of the tree, and an array of pointers to the right branch) also constitutes a space, with the pointers serving as explicit landmarks. A navigator for such a space needs to test the key against a node, and pursue the appropriate subtree. That is, such a navigator is typically implemented with a conditional (if) statement. In the sequel I present the generic representation of a search loop, specifying the generic space and navigator. I also specify a tree as described above as a particular instance of the generic class space, and a navigator
\[ s := top \]
whiles \( s \neq 0 \land v(s) \neq key \) do
  if \( key < v(s) \) then \( s := left(s) \)
  else \( s := right(s) \)
end

Figure A.1: A program to search a binary tree.

to search it. This involves a functional specification for a conditional, which is specified in general, and then specialized for the tree navigator. A sample program that these structures describe is shown in figure A.1.

Figure A.2 shows the functional representation of the generic search loop. Just like the total loop, there are two functions of the loop; here they are called \textit{find} and \textit{cover}, corresponding to \textit{induct} and \textit{cover} of the total loop, respectively. The second state in the diagram for the behavior \textit{initialize-and-search} will be the loop invariant in the proof of correctness of the loop. The function \( \varphi \) is provided by the navigator (see figure A.3), and denotes the set of nodes that the navigator might reach from the current node. Thus, the loop invariant can be read as ‘if there is a node anywhere in the space that matches the key, then it is in the space yet to be searched by the navigator’. The final state of the \textit{initialize-and-search} loop is the same invariant, along with the termination conditions of the loop.

The diagram for the behavior \textit{cycle} shows the maintenance of the loop invariant; the two links on the right of the figure correspond to the two tests provided by the
navigator. The first checks that the space yet to be searched is not empty, while the second checks to see whether the key has been found; in either case, the loop terminates; otherwise, the navigator proceeds to the next node in the space, and the loop continues.

The *navigator* in figure A.3 is very like the one in chapter III, except for the *next* function. Since a *navigator* for a search does not have its course through the set laid out in advance (as was the case with the *navigator* for the total loop), it is necessary to make certain that the loop will terminate. This is done by requiring a well-ordered metric on the nodes in the space. The *next* function of the *navigator* must decrease this metric; since it is well-ordered (and positive), this guarantees that the loop will terminate. A particular space may have any number of functions *direct*, which provide access to the landmarks in the space. Typically each different *direct* will have a different *Provided* or *If* clause restricting its use.

Before showing a specific example of a search loop, it is necessary to define one more generic object, the conditional statement. This is shown in figure A.4. A simple conditional has four functions. The functions *truebranch* and *falsebranch* correspond to the code in the *then* and *else* clauses of the *if* statement (not necessarily respectively). For some condition $C$, they guarantee that $Q$ will hold after the execution of the conditional, given $C$ and $\neg C$ (respectively).

The function *test* determines the truth value of $C$; the functional representation cannot represent this outcome (since it changes from one execution to the next),
Devices search loops Depend on key, K, S, v : S → K

Function find
  If key
    ToMake ?node ∈ S ∧ v(?node) = key
  By initialize-and-search

Behavior initialize-and-search:

∃n ∈ Ss.t.v(n) = key ⇒ ∃n ∈ φ(?navigator)s.t.v(n) = key

Using Function start of navigator

T

Using Function search of search loop

⋯ ∧ (?nav = null ∨ v(?navigator) = key)

Function cover
  If ∃n ∈ Ss.t.v(n) = key ⇒ ∃n ∈ φ(?navigator)s.t.v(n) = key
  ToMake ⋯ ∧ (?nav = null ∨ v(?navigator) = key
  By cycle

Behavior cycle

∃n ∈ Ss.t.v(n) = key ⇒ ∃n ∈ φ(?navigator)s.t.v(n) = key

Using Function next of navigator

Using Function emptyp of navigator

Using Function foundp of navigator

?nav ⊄ null ∧ v(?nav) ⊄ key

Figure A.2: Functional representation for the generic search loop.
Devices navigator

**Depend on** $S$ a Space, $\sigma \in S$, key, $\varphi : S \rightarrow S^*$

**Function start**

If T
ToMake $?navigator = \sigma$

**Function empty**

If $\varphi(?navigator) = \emptyset$
ToMake $\langle \varphi(?navigator) = \emptyset \rangle$

**Function found**

If $v(?navigator) = \text{key}$
ToMake $\langle v(?navigator) = \text{key} \rangle$

**Function next**

If $P(?navigator) \land \mu(?navigator) = m$
ToMake $P(?navigator) \land \mu(?navigator) < m$

Device Space **Depend on**

$K$ a set of values,$N$ a set of nodes and $v : N \rightarrow K$

$\mu : N \rightarrow \mathbb{N}$(the natural numbers)

**Function yield**

If $?node \in N$
ToMake $v(?node)$

**Functions direct**

If $?node1 \in N$
ToMake $?node2 \in N$, $\mu(node2) < \mu(?node1)$

**Function empty**

If $?node \in N \land \mu(?node) = \text{MIN}_{n \in N}\mu(n)$
ToMake $\langle \mu(?node) \text{ minimal} \rangle$

---

Figure A.3: Functional Representations for generic navigator and Space. A Space is made up of nodes, each one of which is associated ($v$) with a value ($K$). The Space provides landmarks (access to these landmarks is made available by the function direct). The navigator begins at some node in the space, and travels from one node to the next, following the landmarks. Hence the details of the next function make use of the direct function of the Space.
but it can represent that the outcome is determined. This is done with the notation \( (C) \mid (\neg C) \), that is, either \( C \) is available for the proof, or \( \neg C \) is. The proof must follow both lines to be complete. The proofs corresponding to \( \text{truebranch} \) and \( \text{falsebranch} \) respectively provide these lines.

The function \( \text{guarantee} \) is the most familiar. It guarantees that after the execution of the conditional, some predicate \( Q \) will hold, given some predicate \( P \) holds before the conditional. This is done by guaranteeing, for the condition \( C \), that \( Q \) will hold after the conditional regardless of the truth value of \( C \). The behavior \( \text{bothwork} \) accomplishes this by first checking \( C \), and then using both \( \text{truebranch} \) and \( \text{falsebranch} \) to complete the guarantee. The code of the \( \text{truebranch} \) is tagged with the PASCAL keyword \texttt{then} if the \texttt{test} function tests for \( C \); if it tests for \( \neg C \) (which is just as valid a test), then the \( \text{truebranch} \) appear with the tag \texttt{else}. The reverse is true of the \( \text{falsebranch} \).

Now it is possible to specify a particular search loop by specifying its \texttt{navigator} and \texttt{Space}. This is done in figures A.5 and A.6. The function \texttt{next} is implemented using a conditional(\texttt{pursuer}), which is described in figure A.6. The function \texttt{less} of the tree is used in the case that \( ?\text{key} < ?\text{current} \), while the function \texttt{greater} is used otherwise. The function \texttt{test} of the \texttt{pursuer} might appear in the code as \( ?\text{key} < ?\text{current} \) or as \( ?\text{key} \geq ?\text{current} \); in the former case, the \( \text{truebranch} \) will be tagged with \texttt{then}, and in the latter case, with \texttt{else} (see figure A.4).
Figure A.4: Functional Representation of generic conditional. The test allows a proof to use one of $C$ or $\neg C$ (exactly one of which is true). The conditional statement must guarantee that $Q$ holds regardless of which of one is true. Thus, the conditional must be prepared to enforce $Q$ in either case. As usual, the parallel branches in the behavior diagram mean that both functions must be present.
Finally, in figure A.7 a tree is defined as a special case of a space. $\mu(n)$ is defined as the depth of the subtree starting at $n$. The tree has two sorts of landmarks (i.e., two functions of the form of the function $direct$ in figure A.3); one for the less branch of the tree, the other for the greater. The definitions of the functions $less$ and $greater$ show these implemented in the left and right branches of the tree, respectively.
Device binary-nav is a *navigator*

Depends on $\sigma = \text{top}$

with Function *next*

If $\mu(?\text{navigator}) = ?m \land \exists n \in \varphi(?\text{navigator}) s.t. v(n) = \text{key}$

ToMake $\mu(?\text{navigator}) < ?m \land \exists n \in \varphi(?\text{navigator}) s.t. v(n) = \text{key}$

By *bothnext*

Behavior *bothnext*

$\mu(?\text{navigator}) = ?m \land \exists n \in \varphi(?\text{navigator}) s.t. v(n) = \text{key}$

Using Function *guarantee* of *pursuer*

$\mu(?\text{navigator}) < ?m \land \exists n \in \varphi(?\text{navigator}) s.t. v(n) = \text{key}$

**Function start**

If $T$

ToMake $?\text{navigator} = \sigma$

By e.g., $?\text{navigator} := \sigma$

**Function empty**

If $\varphi(?\text{navigator}) = \emptyset$

ToMake $\varphi(?\text{navigator}) = \emptyset$

By e.g., $s = 0$

**Function foundp**

Depends On $?\text{val}$ is given by *yield of tree*

If $v(?\text{navigator}) = \text{key}$

ToMake $v(?\text{navigator}) = \text{key}$

By e.g., $?\text{val} = ?\text{key}$

Figure A.5: Functional Representation of a navigator for a binary search tree. The *next* function is provided by a conditional that is called *pursuer*, and is specified in the next figure.
Device pursuer is a conditional binding $C = ?key < ?current$,

$P = \exists n \in \varphi(?navigator) \land \mu(?navigator) = m,$

$Q = \exists n \in \varphi(?navigator) \land \mu(?navigator) < m,$

with Function test

If $T$

ToMake $(?key < ?current) \mid (\neg ?key < ?current)$

By e.g., $?key < ?current$ [prefix=""]

$?key \geq ?current$ [prefix="\"]

Function truebranch

If $(?key < ?current)$

ToMake $\exists n \in \varphi(?nav) \land \mu(?nav) < ?m$

By lesschase

$(?key < ?current)$

Using Function less of tree

binding @node $?navigator$

$?navigator = \hat{t}$

$\exists n \in \varphi(?navigator)s.t.v(n) = key \land \mu(?nav) < ?m$

Function falsebranch

If $(\neg ?key < ?current)$

ToMake $\exists n \in \varphi(?nav) \land \mu(?nav) < ?m$

By greatchase

$(\neg ?key < ?current)$

Using Function greater of tree

binding @node $?navigator$

$?navigator = \hat{t}$

$\exists n \in \varphi(?navigator)s.t.v(n) = key \land \mu(?nav) < ?m$

Figure A.6: Functional representation of the conditional pursuer. The pursuer tests which way to go in the tree, and makes the appropriate assignment.
Device tree is a *Space*

**Depends On** $N$ a set of nodes, $K$ a set of values  
$top \in N$, $v : N \to K$, $\prec$ a relation on $K$,

$$\mu(n) = \begin{cases} 
0 & \text{left}(n) = \text{right}(n) = 0; \\
1 + \max(\mu(left(n)), \mu(right(n))) & \text{otherwise}
\end{cases}$$

$$\varphi(n) = \begin{cases} 
\emptyset & n = 0; \\
\{n\} \cup \varphi(left(n)) \cup \varphi(right(n)) & \text{otherwise}
\end{cases}$$

**Function yield**

If $?node \in N$
ToMake $v(?node)$
Provided $?node = 0$
By e.g., $v(?node)$

**Function less**

If $?node \in N$
ToMake $?t \in N, \text{MAX}(\varphi(?t)) \prec v(?node) \land \mu(?t) < \mu(?node)$
Provided $?node = 0$
By e.g., $?t := left(?node)$

**Function greater**

If $?node \in N$
ToMake $?t \in N, \text{MIN}(\varphi(?t)) \succ v(?node) \land \mu(?t) < \mu(?node)$
Provided $?node = 0$
By e.g., $?t := right(?node)$

**Function emptyp**

If $?node = 0$
ToMake $?node = 0$
By e.g., $?node = 0$

Figure A.7: Functional representation of a binary search tree. $\varphi(n)$ is the set of nodes below node $n$. $left(n)$ are the nodes to the left in the tree; in the example lines of code, these are the nodes that are $\prec$ the top node.
BIBLIOGRAPHY


Wulf, London and Shaw


