INSPECTION METHODS IN PROGRAMMING

by

CHARLES RICH

B.A.Sc., University of Toronto
(1973)

S.M., Massachusetts Institute of Technology
(1975)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1980

Signature of Author_____
Department of Electrical Engineering and Computer Science
May 16, 1980

Certified by_____
Gerald Jay Sussman
Thesis Supervisor

Accepted by_____
Chairman, Departmental Graduate Committee

INSPECTION METHODS IN PROGRAMMING

by

Charles Rich

Abstract

This thesis is motivated from two directions. As research in artificial intelligence, it is a step towards a model of programming as a kind of problem solving. In particular, this thesis focusses on the aspect of programming which involves the routine application of previous experience with similar programs. I call this programming *by inspection.*

There are also practical motivations. The inadequacy of current programming technology is universally recognized. Part of the solution to this problem will be to increase the level of automation in programming. I believe (and have argued elsewhere) that the next major step in the evolution of more automated programming will be so called *programmer's apprentice* systems. These are interactive systems which provide a mixture of automated program analysis, synthesis and verification.

This thesis concentrates on the knowledge based components of a programmer's apprentice. The most important such component is a *taxonomy* of commonly used algorithms and data structures. To the extent that a programmer is able to construct and manipulate programs in terms of the forms in such a taxonomy, he may relieve himself of many details, and generally raise the conceptual level of his interaction with the system, as compared with present day programming environments. Also, since it is practical to expend a great deal of effort pre-analyzing the entries in a library, the difficulty of verifying the correctness of programs constructed this way is correspondingly reduced.

A major contribution of this thesis is a new formalism, called the *plan calculus*, for representing classes of similar computations. This formalism makes it is possible to capture many important generalizations and take multiple points of view. The use of the plan calculus is demonstrated by the construction a library of some common techniques for manipulating symbolic data, and its application to analysis, synthesis and verification of programs by inspection.

In this thesis, programming is viewed as a kind of engineering activity. This research takes place in the context of a larger study, involving a number of other researchers, of the principles underlying engineering problem solving in general. Inspection methods for analysis and synthesis are a prominent part of expertise in many other engineering disciplines, such as electrical and mechanical engineering. The notion of program understanding developed in this thesis is also motivated by similar notions of understanding for other types of engineered devices.

Thesis Supervisor: Gerald J. Sussman
Title: Associate Professor of Electrical Engineering

To My Father.

Acknowledgements

I would like to thank my thesis supervisor, Gerry Sussman, and readers, Hal Abelson, Carl Hewitt and Mike Hammer.

I would also like to thank all my friends and colleagues at the AI Lab who read parts of this document and talked to me about it. I would like to list their names, but I don't have time right now.

Thank you, Candy, for everything.

Table of Contents

1. Introduction 7

1.1 Inspection Methods 8
1.2 Multiple Points of View 11
1.3 The Plan Calculus 13
1.4 Guide to the Reader 22
1.5 Relation to Other Work 23

PART I - OVERVIEW

2. Programmer's Apprentice Scenario 30

3. Overview of Plan Library 42

3.1 Introduction 42
3.2 Functions 47
3.3 Sets 51
3.4 Directed Graphs 55
3.5 Recursive Plans 60

4. The Plan Calculus 67

4.1 Introduction 67
4.2 Plans 68
4.3 Surface Plans 78
4.4 Overlays 80

PART II - IN DEPTH SCENARIOS

5. Analysis by Inspection 89

5.1 Why Analysis? 90
5.2 Overview 91
5.3 Surface Plans 93
5.4 Loop Analysis 98
5.5 Bottom-up Recognition 112
5.6 Top-down Recognition 116

6. Synthesis by Inspection 121

6.1 Introduction 121
6.2 Data Structure Design 123
6.3 Procedure Synthesis 128

7. Verification by Inspection 156

PART III - TECHNICAL DETAILS

8. Logical Foundations 162

8.1 Introduction 162
8.2 Mutable Objects and Side Effects 163
8.3 Multiple Points of View 168
8.4 Data Plans 172
8.5 Data Overlays 176
8.6 Computations 177
8.7 Temporal Plans 183
8.8 Temporal Overlays 192
8.9 Specialization and Extension 195

9. Loops and Temporal Abstraction 198

9.1 Introduction 198
9.2 Loops 199
9.3 Temporal Abstraction 216
9.4 Recursive Structures 255

10. Plans Involving Side Effects 259

Appendix. Plan Library Reference Manual 263

Bibliography 294

Index 298

CHAPTER ONE

INTRODUCTION

This thesis is motivated from two directions. As research in artificial intelligence, it is a step towards a model of programming as a kind of problem solving. In particular, this thesis focusses on the aspect of programming which involves the routine application of previous experience with similar programs. I call this programming *by inspection.*

There are also practical motivations. The inadequacy of current programming technology is universally recognized. Part of the solution to this problem will be to increase the level of automation in programming. I believe (and have argued elsewhere) that the next major step in the evolution of more automated programming will be so called *programmer's apprentice* systems. These are interactive systems which provide a mixture of automated program analysis, synthesis and verification.

This thesis concentrates on the knowledge based components of a programmer's apprentice. The most important such component is a *taxonomy* of commonly used algorithms and data structures. To the extent that a programmer is able to construct and manipulate programs in terms of the forms in this taxonomy, he may relieve himself of many details, and generally raise the conceptual level of his interaction with the system, as compared with present day programming environments. Also, since it is practical to expend a great deal of effort pre-analyzing the entries in a library, the difficulty of verifying the correctness of programs constructed this way is correspondingly reduced.

A major contribution of this thesis is a new formalism, called the *plan calculus*, for representing classes of similar computations. This formalism makes it is possible to capture many important generalizations and take multiple points of view. The use of the plan calculus is demonstrated by the construction of a library of some common techniques for manipulating symbolic data, and its application to analysis, synthesis and verification of programs by inspection.

In this thesis, programming is viewed as a kind of engineering activity.[1] This research takes place in the context of a larger study, involving a number of other researchers, of the principles underlying engineering problem solving in general. Inspection methods for analysis and synthesis are a prominent part of expertise in many other engineering disciplines, such as electrical and

---

1. Rather than, say, as a branch of mathematics. This bias will not be defended here, but is discussed in another paper by the author.

mechanical engineering. The notion of program understanding developed in this thesis is also motivated by similar notions of understanding for other types of engineered devices.

1.1 Inspection Methods

What are inspection methods and where do they come from? Furthermore, why do inspection methods continue to be important in many domains, engineering and otherwise, where powerful general methods have been discovered to solve a wide range of problems?

Inspection methods are a distillation of the collective experience of solving many problems in a particular domain. The essence of this experience is a library of common problem *forms*. The first step of any inspection method is to *recognize* a familiar form embedded in a given problem. Associated with each such form is either an explicit solution or some schema from which a solution can easily be computed.[1]

For example, analysis of the termination conditions of a program is often done by inspection. If you recognize a loop that counts up to a fixed number greater than the initial number, then you know from experience that it terminates. Similary, for synthesis by inspection, experienced programmers typically have a repertoire of standard operations on sets which they know how to implement for variety of set representations. Once a programmer recognizes that a problem calls for one of these operations, he can implement it immediately. Verification by inspection is similar. Most of the difficult verification steps (typically the inductive arguments) are embedded in pre-proven theorems, which are associated with forms in the library. All that remains is to combine these theorems appropriately as lemmas in the proof of the particular program.

Clearly a key issue in the use of inspection methods is how the common forms and solutions in a particular domain are represented. In the domain of programming, I call these forms *plans*. Plans are a powerful new formalism in which generalizations of both data and control structure in programs can be expressed. The plan calculus will be discussed further in an upcoming section.

---

1. In sufficiently complex situations, debugging is also an unavoidable part of the use of inspection methods. The role of debugging in problem solving has been investigated by Sussman; it is not part of the focus of this thesis.

An Engineering Vocabulary

Another important feature of inspection methods is that the common forms acquire names which become part of the standard working vocabulary of experts in the field. These names for intermediate level constructs supplement the primitive vocabulary of the domain. For example, the primitive vocabulary of currents, voltages and resistances is formally adequate for specifying a wide range of electrical circuits. However, experienced engineers use a much richer vocabulary, including such concepts as series and parallel configuration, voltage divider, cascode connection, and so on. Similarly, an experienced programmer knows much more than the meaning of primitive programming constructs, such as tests, iterations, arrays, assignments, and so on. An experienced programmer is also familiar with many other, more abstract, standard plans, such as lists, hash tables, search loops, and splicing.

A shared intermediate level vocabulary is very important for communication between experts. In many fields this vocabulary has been codified and is taught as part of the standard education of novices. This implies that facility with the appropriate intermediate vocabulary is an essential component of any intelligent interactive system which is going to help experts in any field. Chapter Two demonstrates this point for a programmer's apprentice system in particular.

General Methods

We come now to the question of why inspection methods persist in the presence of more powerful general methods for the same problems. General methods, by their very nature, operate at the most primitive level of vocabulary of the domain. This causes two serious problems: the methods are inefficient; and the results are difficult for users to interpret. Because of these difficulties, experts tend to bypass general methods whenever they can recognize a familiar special case which can be solved by inspection. In fact, this behavior is usually taken as one of the distinguishing characteristics of being an expert.

For example, a very powerful general method for symbolic integration has recently been discovered by Risch. However, it is usually used only as a last resort, even by automated systems like Macsyma, because inspecting an integral for one of the many well-known forms is comparatively inexpensive, and if one is recognized, the answer can be computed much more quickly than by the Risch algorithm. Similarly, general circuit analysis techniques involving node and cut sets and the inversion of matrices are seldom employed by experts because they are so laborious in comparison to decomposing a circuit into familiar patterns.

General methods have recently been developed in the area of programming also. For example, a general method for program verification decomposes the problem into two steps. The first step is the generation of verification conditions, in which specifications of the desired behavior of

the program are combined with the axioms for each language primitive in the program, yielding a single formula to be proved valid. This formula is then passed to a general purpose theorem prover. . Unfortunately, if the program is incorrect, which is the most common case, the manner in which the proof of the verification conditions fails provides little guidance to the user about how to correct the original program. Verification by inspection, while it is not as powerful, does not suffer from this problem of incomprehensibility. Errors are detected by inspection either by recognizing a known pattern whose pre-proven properties contradict the desired specifications, or by recognizing a suspiciously close match to a known pattern. In either case, the nature of the discrepancy can be communicated to the user in terms of familiar engineering vocabulary.

The analysis of programs with side effects is another area in which general methods have failed to supplant inspection. Some work has been done on representing and reasoning about side effects in programs [56], but the general methods developed thus far are clumsy and computationally expensive. Furthermore, there is reason to believe that there are fundamental limitations to the effectiveness of general methods in this area. Programs with an unconstrained use of side effects (such as RPLACA and RPLACD) are extremely difficult to understand, even for the most expert human programmers. This has led some to advocate the extreme position of banning side effects entirely in new languages and systems. However, there are also good arguments that side effects are crucial for the modularity and efficiency of certain programs [58]. The resolution of this apparent conflict is the observation that side effects are typically used only in very stylized forms, such as to splice nodes in and out of a linked list, to update a global data base, and so on. By constructing a library of these standard plans and their properties, analysis of side effects by inspection can suffice for most practical purposes.

Education

The importance of inspection methods is reflected in the way we educate novices in a field. We first develop their intuitions by acquainting them with the standard forms. Only much later, if ever, do we teach general methods. For example, in the field of medical diagnosis (where there are as yet no reliable general methods) new doctors are trained primarily through a set of paradigm cases. Similarly, electrical engineering students are first taught how to predict the behavior of certain standard circuits (e.g. oscillators), and how to implement certain common signal processing functions (e.g. filters), before they are taught general tools for analyzing and synthesizing circuits. In programming also, we begin with the craft lore of standard algorithms and data structures before introducing any general program analysis, synthesis or verification methods, such as those of Floyd [20], Hoare [28] and others.

1.2 Multiple Points of View

The power of inspection methods rests primarily on the ability to recognize familiar forms. There are many different ways in which the recognition of familiar forms can be obscured. For example, in medical diagnosis, the presence of a second independent disease may cause certain key symptoms to be changed or absent. In electrical engineering, a standard circuit may not appear to be familiar because some components are in parallel rather than in series, or vice versa. Similar difficulties also arise in programs. For example, the placement of exit tests other than at the top or bottom of a loop can obscure recognition of standard loop plans.

Various techniques have been developed in different fields to overcome such complications. These techniques are generally called equivalences, transformations, or models. All of these can be be thought of as ways of providing the user with different *points of view* on a problem. Sometimes a different point of view is necessary in order to use inspection methods at all. Sometimes several different points of view each contribute some part of the solution. For example, in the analysis and synthesis of electrical circuits, equivalence theorems (such as Thevenin-Norton) are a basic tool for rearranging the topology of circuits to match standard forms. Electrical engineers also use views in which certain features of the problem are ignored -- the so-called AC (sinusoidal steady state) and DC (direct current) models are an example. In one model certain components become open circuits, while in the other they become shorts. Since the circuit in each model is simpler than in the full circuit view, the user is more likely to be able to use inspection. (It is also an important feature of these particular two views that results derived in them can be simply combined to give a complete analysis of the circuit.)

Multiple points of view are also important in understanding programs. Program transformations can be used to move the position of exit tests in loops,[1] and thereby increase the power of inspection methods which recognize loop plans. In the area of data structures, it is often necessary to view a single structure from the two different points of view, each of which captures a different generalization. For example, in this thesis a Lisp list can be viewed both as a recursive structure (the tail of a list is a list) and as a labelled directed graph (where the nodes are Lisp cells connected by the CDR relation and labelled by the CAR relation). The first view is appropriate for understanding CONS and CDR as push and pop operations. The second view brings to bear a programmer's experience with standard graph manipulations in order to understand RPLACD as the operation of splicing out a node. A single Lisp list may be used in both these ways in a single program.

---

1. These are the so-called loop "winding" transformations.

Another example of point of view in programming is what I call the "steady state" model of loops (and in general, recursions). In this view, exit tests are ignored in order to recognize the basic iteration and recursion plans, such as counting, summing, CAR-CDR recursion, etc. This view is similar to the AC model in electronic circuits, in that it can be simply combined with other views to construct a complete description. For example, the counting part of a loop can be abstracted as generating an infinite sequence of numbers, which is truncated by the exit test.

The plan calculus developed in this thesis includes a mechanism, called *overlays*, for representing these and other points of view. Overlays will be described further in the next section.

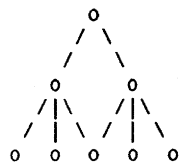
Overlapping Implementations

A general category of recognition difficulty arises in many engineering domains when the implementations of two distinct abstract functions overlap. This means that a single component at the implementation level plays a role in two distinct forms. For example, a screw in a mechanical device may fasten two plates together and also provide a fulcrum about which to pivot a lever. In a radio-frequency amplifier, an inductor may be both part of a resonant circuit in the AC model and also part of the bias network of a transistor in the DC model. Thus "bumming" is not just a feature of arcane programming -- it is an essential part of good engineering.

For example, consider the following program which, viewed abstractly, performs two functions: it finds the maximum element of a (non-empty) list, L; and it finds the minimum element.

```
(PROG (MAX MIN L)
   (SETQ L ...)
   (SETQ MAX (CAR L))
   (SETQ MIN (CAR L))
   (MAPC '(LAMBDA (N)
            (COND ((> N MAX)(SETQ MAX N)))
            (COND ((< N MIN)(SETQ MIN N))))
         (CDR L))
   ...MAX...
   ...MIN...)
```

The analysis of this program according to the plans in this thesis has the structure shown in the diagram below.



At the top level the program is logically decomposed into two functions, one which finds the maximum and one which finds the minimum. The standard loop plans for implementing these functions each have three essential components:

(i) an initialization,
(ii) a standard list iteration (MAPC), and
(iii) a comparison on each step between the current list element and the current maximum or minimum.

In the program above, the standard list iteration component is shared between the implementation of finding the maximum and finding the minimum. Thus at the third level in the diagram above there are five rather than six components -- one of them doing double duty. This type of analysis is a violation of strict hierarchical refinement, which is currently the dominant paradigm for structuring programs [11]. I have in general found, however, that it is not possible to maintain the strictly hierarchical view and at the same time analyze programs in a way which makes their similarities explicit.

In this thesis, implementation relationships are treated as points of view. This approach has the advantage of allowing the efficiency represented by the program above (as compared to a strictly hierarchical implementation with two separate loops), while still capturing the similarities between this program and programs which calculate only the maximum or only the minimum.

1.3 The Plan Calculus

The purpose of this section is to introduce the plan calculus and point out some of its important features. This formalism is an outgrowth of earlier work by the author in collaboration with Shrobe [47] and Waters. A more detailed description of the plan calculus is the topic of Chapter Four. The important features discussed in this section are as follows.

- *Wide Spectrum Specification*
- *Control and Data Abstraction*
- *Mutable Objects*
- *Programming Language Independence*
- *Multiple Points of View*
- *Additivity*
- *Verifiability*
- *Dependencies*

The plan calculus is made up of two major components: plans and overlays. Basically, a plan is the specification of a computation. Overlays represent the relationship between two different points of view on a computation, each of which is specified by a plan.

Programming is viewed in this thesis as a process involving the construction and manipulation of specifications at various levels of abstraction. In this view, there is no fundamental distinction between specifications and programs. A program (e.g. in Lisp) is simply a specification which is detailed enough to be carried out by some particular interpreter. This approach is related to the current trend in computer science towards *wide spectrum* languages. Like wide spectrum languages, the advantage of this approach is that various parts of a program design can be refined to different degrees without intervening shifts of representation.

Plans

Computations, in the view of this thesis, are made up of three types of primitives: operations, tests, and data objects. There are three corresponding types of primitive specifications in the plan calculus: *input-output* specifications, *test* specifications and *object type* specifications. Operations are specified by input-output specifications (preconditions and postconditions). Tests are specified by whether they succeed or fail when a given relation holds between the inputs. The primitive object types used in this thesis are numbers, sets and functions.

Hierarchy is represented by composite plans. Each composite plan specifies a set of local names for its parts (called *role* names) and a set of constraints which must hold between them. There are two kinds of composite plans, according to the types of the parts.

Data plans specify data structures whose parts are primitive data objects or other data structures. Data plans thus embody a kind of data abstraction. For example, List is a data plan with two roles named Head and Tail. The Head of a list may be an object of any type, but the Tail is constrained to be either an instance of List or the distinguished object, Nil ("the empty list"). Data plans are also used to represent common implementation forms. For example, a data plan called Segment is shown in Fig. 1-1. Data objects are indicated in plan diagrams by ovals. This plan has three roles named

Base (a sequence),
Upper (a natural number), and
Lower (a natural number),

and the following constraints:

(i) The Upper number is less than or equal to the length of the Base sequence.
(ii) The Lower number is less than or equal to the length of the Base sequence.
(iii) The Lower number is less than or equal to the Upper number.

This data plan (and special cases of it) is commonly used to implement other data abstractions, such as lists and queues.

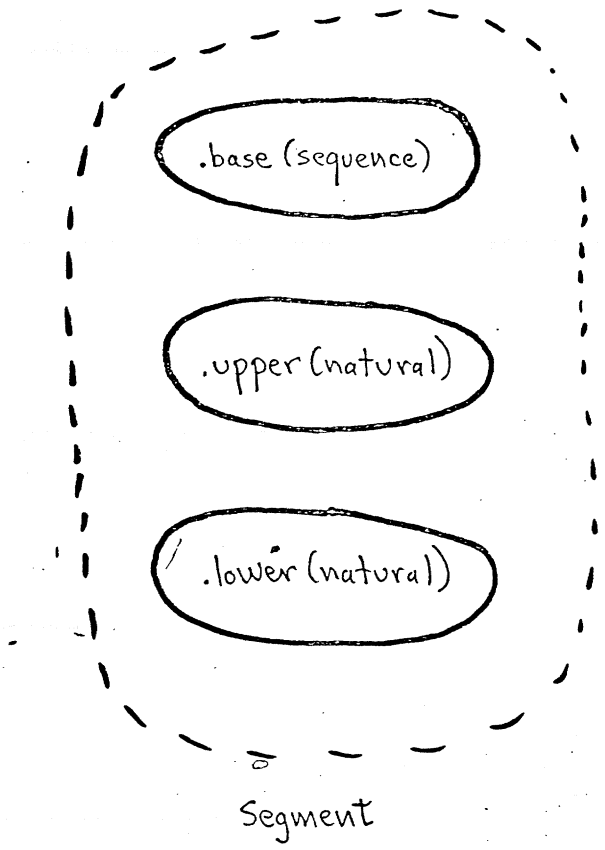


Figure 1-1. A Data Plan.

Primitive data objects and data structures are *mutable*. For primitive data objects, this means that the behavior of the object can change while its identity remains the same. For example, we can specify a set addition operation in which the identical set is both the input and output. For data structures with parts, such as instances of the Segment plan, mutability means that one or more of the parts may be replaced while the identity of the data structure remains the same. For example, a common operation on Segment data structures is to increment the Upper index. The semantics of mutability are part of the logical foundations of the plan calculus, which are discussed later in this section.

Temporal plans specify computations whose parts are operations, tests, data structures or other composite computations. In addition to various logical constraints between roles, such as "less than or equal", temporal plans also include *data flow* and *control flow* constraints. An example of the temporal plan for computing absolute value is shown in Fig. 1-2. Operations and tests are indicated in plan diagrams by rectangular boxes. The bottom half of test boxes are divided into cases labelled "F" for failure and "S" for succeed. This plan has three roles named

> If (a test for less than zero),
> Then (a negation operation), and
> End (a join).[1]

Data flow constraints (solid arrows in the figure) specify correspondences between the outputs and inputs of operations and tests. Control flow constraints (hatched arrows) specify which parts of a computation are reached depending on which tests succeed or fail. Temporal plans thus embody a kind of control abstraction.

The plan calculus is to a large degree programming language independent (for a wide class of conventional sequential programming languages). This makes it possible to build a program understanding system which is concerned with the syntactic details of different languages only at its most superficial interface. In order to translate back and forth between a given programming language and the plan calculus, the primitives of the programming language are divided into two categories:

(i) The primitive *actions* and *tests* of the language, such as `CAR`, `CDR`, `CONS`, `NULL` and `EQ` in Lisp, are represented as input-output specifications and test specifications.

---

1. A join is a virtual entity which is needed in order to specify what the output is in each case of a conditional. Joins will be defined in Chapter Four.

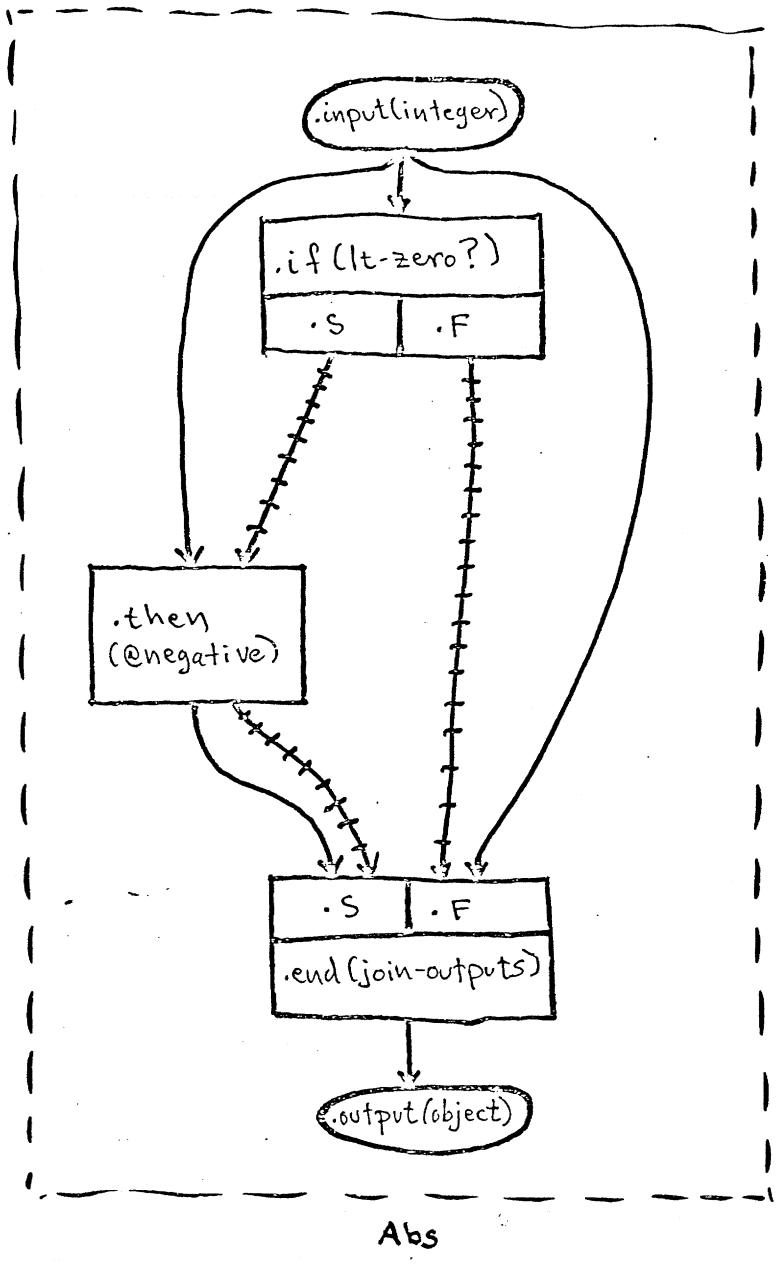


Figure 1-2. A Temporal Plan.

(ii) The primitive *connectives*, such as PROG, COND, SETQ, GO and RETURN in Lisp, are represented as patterns of control and data flow constraints between operations and tests.

The translation from standard program code to an equivalent plan representation has been implemented for reasonable subsets of Lisp [47], Fortran [60] and Cobol. The translation from suitably restricted plans to Lisp code has also been implemented.

Overlays

Overlays are the mechanism in the plan calculus for representing points of view in the programming domain. An overlay is formally a triple made up of two plans and a set of *correspondences* between roles of the two plans. Each plan represents a point of view; the correspondences express the relationship between the points of view. The development of overlays for program plans was motivated by Sussman's "slices", which he uses to represent equivalences in electronic circuit analysis and synthesis.

In addition to standard plans, there also standard overlays. For example, consider the following recursive Lisp program which copies a list.

```
(DEFINE COPYLIST
   (LAMBDA (L)
      (COND ((NULL L) NIL)
            (T (CONS (CAR L)(COPYLIST (CDR L)))))))
```

This program is an example of a singly recursive program in which there is computation "on the way up", i.e. in which the recursive invocation is not the last step in the program. Many standard recursive computations, such as list accumulation by CONSing, can be performed either "on the way down" or "on the way up." For example, the following tail recursive program, which reverses a Lisp list, performs list accumulation on the way down.

```
(DEFINE REVERSE
   (LAMBDA (L)
      (REVERSE1 L NIL))))

(DEFINE REVERSE1
   (LAMBDA (L M)
      (COND ((NULL L) M)
            (T (REVERSE1 (CDR L)(CONS (CAR L) M))))))
```

Recognition of the standard Lisp list accumulation plan in these two programs is facilitated by an overlay which expresses how, in general, to view accumulation on the way up as accumulation the way down with an intervening order reversal. This overlay is shown in Fig. 1-3. Without going

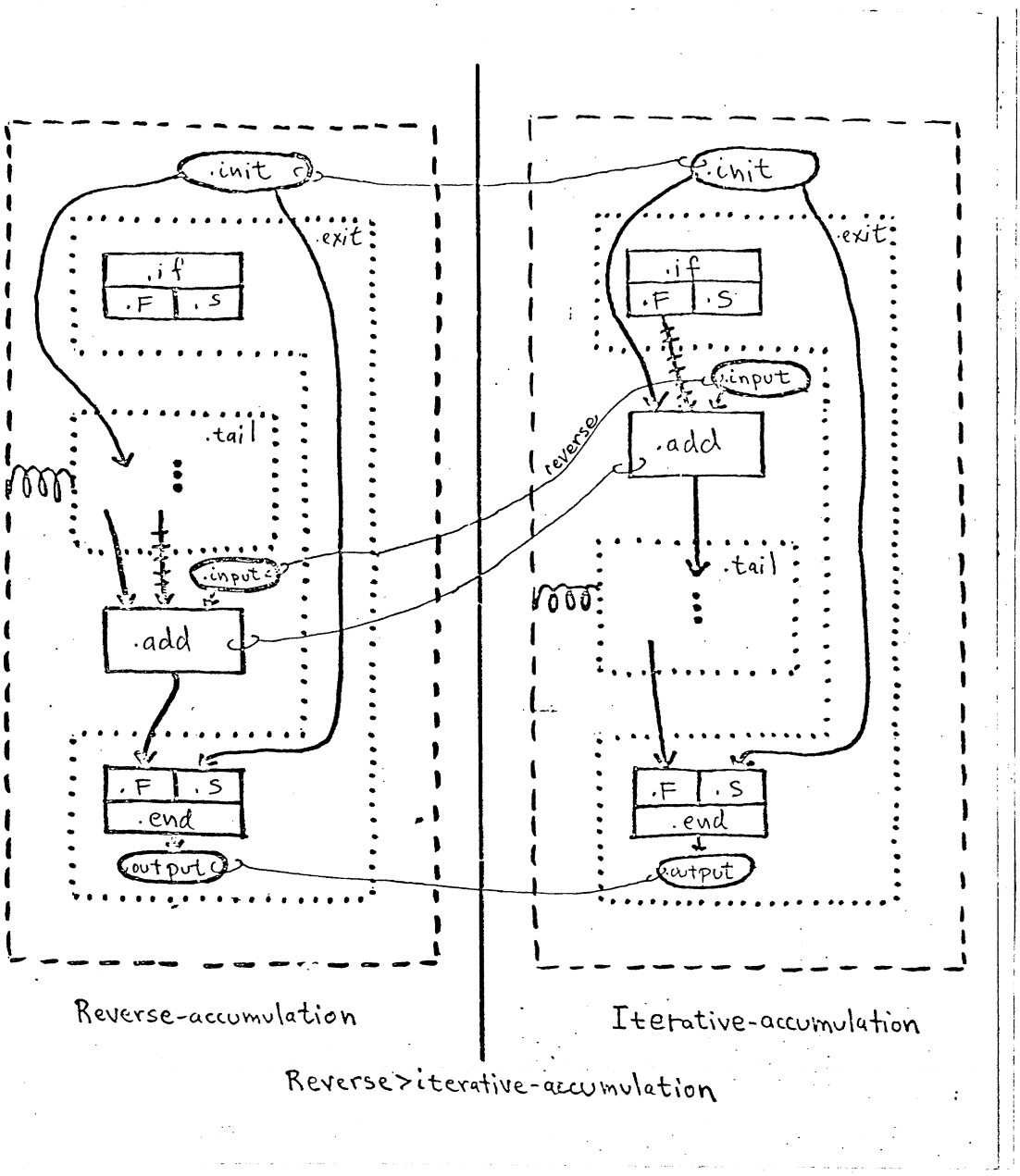


Figure 1-3. An Overlay.

into details,[1] consider that the plan on the left represents accumulation on the way up; the plan on the right represents accumulation on the way down. The four hooked lines between the two plans specify correspondences between the two points of view. Unlabelled correspondences (three out of the four in Fig. 1-3) are equalities. Thus the initialization of the accumulation (the Init role) is the same in both views. So are the input-output specifications of the accumulation operations (the Add role), and the final output. The most important correspondence, however, is the one labelled "reverse" in the figure. This is the correspondence which specifies that the order in which the elements of list L are accumulated in the COPYLIST program is the reverse of the order in which they are generated by the CAR-CDR part of that program.[2]

Notice that overlays are *symmetric*.[3] This means that either side can be used as a "pattern" (plans can be naturally thought of as patterns), which makes it possible to use the same overlays in both analysis and synthesis. The fact that correspondences are equalities (or equalities between the values of functions) means that information can propagate between points of view in both directions. For example, analysis by inspection of COPYLIST proceeds by first recognizing the standard list accumulation by CONSing plan in the point of view represented by the right hand side of the overlay in Fig. 1-3. The known properties of this plan include the fact that the final output is a list whose elements are the successive inputs to the accumulation operations, in reverse order. Propagating this information back to the original view through the correspondences and performing the algebraic simplification,

$$\text{reverse}(\text{reverse}(l)) = l,$$

leads directly to the result that the elements of the output of COPYLIST are the same as the elements of the input list, in the same order.

As mentioned earlier, implementation is also represented using overlays. One side of the overlay is a plan representing the abstract behavior, e.g. pushing an element onto the front of a queue. The other side of the overlay is an implementation plan, e.g. storing the element in an array and adding one to an index pointer. The correspondences in such an overlay propagate information between the abstract and concrete descriptions. Such overlays are used in both analysis and synthesis by inspection. In bottom-up analysis, we try to recognize known implementation plans; once an implementation plan has been recognized, we overlay it (pun intended) with the corresponding

---

1. It is not necessary to understand this diagram in detail at this point. Overlays in general will be explained more fully in Chapter Four, and this overlay in particular is discussed at length in Chapter Nine. For now, it is adequate just to get the idea that there are plans on both sides and correspondences between them.
2. The Lisp interpreter's stack is being used to effect the reversal.
3. This is not strictly true, but only for a reason which is beyond the level of detail of this introduction.

abstract plan, and continue to work upwards. Conversely, in synthesis by inspection we try to recognize abstract plans for which there are known implementation overlays.

Logical Foundations

The remaining features of plans and overlays, namely additivity, verifiability and dependencies, are all related to the logical foundations of the plan calculus. Formally, a plan is a set of axioms in a first order logic. (The details of the axiomatization are given in Chapter Eight.) Although, practically speaking, plans are not manipulated as first order axioms, the logical foundations provide a semantics and a set of proof rules against which the practical manipulations can be validated.

Placing plans in the paradigm of logic has several advantages. For example, *additivity* is a direct consequence of an axiomatic formalization. Additivity means that the result of combining two plans is always to narrow down the set of specified computations. This is a desirable property not shared by other formalisms, such as program schemas (other formalisms are reviewed at the end of this chapter). Additivity also meshes well with the principle of least commitment (which in this context means that implementation plans should have the minimum number of constraints necessary to support the implemented abstract behavior.)

The logical foundations of the plan calculus are also involved in inspection methods for program verification. Verification by inspection is based on recognizing plans and applying already verified overlays. Automating the verification of overlays is not part of this thesis research. However, the logical foundations developed here do establish what needs to be proven to verify an overlay. For example, the verification of an implementation overlay entails proving that the constraints of the abstract plan are derivable from the constraints of the implementation plan, with the correspondences also taken as premises.

In addition to simply recording that an overlay has been verified, it is also useful to keep a record of which constraints of the implementation plan were used in the proof of which constraints of the abstract plan. This information can be extracted as a by-product of the proof process [56]. Such links are in general called *dependencies*. Dependencies, as part of the plan calculus, are a network of links between specifications which trace the logical derivation of one from the other. Dependencies thus capture a dimension of logical structure which is different from the hierarchical decomposition expressed by the roles of a plan.

Dependencies make it possible for a programmer's apprentice system to explain how a program works and reason about the potential effects of a modification. For example, if you want to delete a constraint from an implementation plan, the dependences tell you exactly which constraints of the corresponding abstract plan could become invalid. Similarly, if you change the abstract

specifications of an already verified overlay, the dependencies indicate which parts of the verification need to be redone and which parts can be carried over without any extra work. The use of dependencies in reasoning about programs, especially in program evolution and modification, has been the focus of related work by Shrobe [56].

1.4 Guide to the Reader

The remaining chapters of this thesis can be broken up into three parts. The first part, consisting of Chapters Two, Three and Four, gives an overview of the main components of the thesis. Chapter Two is a scenario which illustrates the use of inspection methods in understanding an example program which implements a simple symbol table with hashing. Chapter Three outlines the scope of the current plan library. Chapter Four introduces the form of diagrams which will be used in the rest of the thesis to define plans and overlays.

Chapters Five, Six and Seven form a second part, which backs up the overview with more details. Each of these chapters is an in-depth scenario of the use of inspection methods in program analysis, synthesis or verification. The same example program as was first introduced in Chapter Two is also used in each of these chapters. The style of presentation in these chapters is to introduce and explain new plans as they are needed in the example. Also, for ease of referring to previously defined plans, an index is provided at the back. If there are two page numbers listed for each item, the first is the page on which the plan or overlay diagram appears; the second is the appendix entry for that item.

The final part of the thesis, Chapters Eight, Nine, Ten and the appendix, is the most detailed and technical. Chapter Eight lays out the logical foundations of plans and overlays. Chapter Nine gives the detailed formalization of loop plans and temporal abstraction, a way of viewing loops in which composition is simpler. Chapter Ten deals with plans involving side effects. These topics are treated in a more general way earlier in the thesis. The appendix is a reference manual for the plan library which gives the detailed specifications for each plan and overlay used in the thesis.

1.5 Relation to Other Work

It is useful to distinguish three areas of concern of this thesis. In this section I outline some connections and comparisons with other work in these areas. The three areas are:

- *Taxonomy* - Standard programming forms and the relationships between them.
- *Formalism* - For representing programming knowledge.
- *Applications* - Analysis, synthesis, and verification of programs.

More generally, at the end of this section, I discuss related work on aspects of programming other than the use of inspection methods, such as debugging and deductive methods.

Program Taxonomies

Many people in the computer science and software engineering community have been calling for the codification of standard program forms for a long time. Two major motivations for this are: to improve software reliability and correctness, and to improve the education of programmers. For example, Dijkstra in his influential *Notes on Structured Programming* [11] called for the codification of standard program forms with associated theorems about their correctness, as follows.[1]

> "$d := D$;
> **while non** prop(d) **do** $d := f(d)$" (6)
>
> When a programmer considers a construction like (6) as obviously correct, he can do so because he is familiar with the construction. I prefer to regard his behavior as an unconscious appeal to a theorem he *knows*, although perhaps he has never bothered to formulate it; and once in his life he has convinced himself of its truth, although he has probably forgotten in which way he did it and although the way was (probably) unfit for print. But we could call our assertions about program (6), say, "The Linear Search Theorem" and knowing such a name it is much easier (and more natural) to appeal to to consciously.
>
> ...it might be a useful activity to look for a body of theorems pertinent to such programs.

More recently, Floyd in his 1978 ACM Turing Award Lecture talked as follows about the importance of teaching the standard forms of programming to new programmers, as compared with

---

1. p. 10.

emphasizing the primitive programming language constructs. (Floyd calls these forms *paradigms* and is particularly interested in very general ones, such as "divide and conquer").[1]

> To the teacher of programming, even more, I say: identify the paradigms *you* use, as fully as you can, then teach them explicitly. They will serve your students when Fortran has replaced Latin and Sanskrit as the archetypal dead language.

Many people have answered these calls, using a variety of expressive tools and covering a range of programming areas. I group these efforts roughly into two categories. In the first category are those who have tried to give wide coverage of the basic forms of everyday programming, such as the standard manipulations involving of sets, directed graphs and linear data structures (lists and sequences). Most prominent in this category is the work of Knuth [32]. In three monumental volumes, Knuth uses a mixture of mathematics, example programs and expository English text to communicate his "programmer's craft" in fundamental algorithms (manipulations on linear lists and tres), seminumerical algorithms (random numbers and arithmetic), sorting and searching. There are also many one-volume text books which have a similar format, but are somewhat less comprehensive.

In the second category, I put those whose have focussed on a more particular programming domain. Not suprisingly, work in this category is also characterized by more formal representations (which will be discussed in the next section). Domains that have been studied in some depth include algorithms on sequences [43] [46], sorting [23], standard loop forms [44] [60], set implementations [51], and the implementation of associative data structures [49].

This thesis falls partly in both categories. The contents of the current plan library is mostly the result of generalizing the plans required for an in-depth understanding of a particular example program -- the implementation of a symbol table using hashing, which is introduced in the scenario in Chapter Two. This example program was chosen because it involves many different techniques which are representative of the domain of routine symbolic manipulations (sets, lists, etc.). I believe that a library which is adequate for this example is a good start towards complete coverage of the domain. The small fraction of plans in the current library which are not directly motivated by the symbol table example fall into two categories. Some of these are obviously important basic plans which don't happen to be used in the example, such as counting and accumulation loops. Other plans are included to fill gaps in the taxonomic structure of the library, such as the plan for splicing into a list (only splicing out appears in this particular symbol table). Barstow's work [3] is similar in depth and breadth.

---

1. p. 459.

Other Formalisms

In this section I review other formalisms that have been used to represent standard programming forms, and point out similarities with the plan calculus. Note, however, that no other formalism has all the features described in the preceding section.

The most obvious candidate for representing standard programming forms is partially completed program texts with constraints on the unfilled parts. These are generally called *program schemas*, and have been used by Wirth [62] to catalog programs based on recurrence relations, by Basu and Misra to represent some typical loops for which the loop invariant is already known, and by Gerhart [21] and Misra [43] to represent and prove the properties of various other common forms. Unfortunately, the syntax of conventional programming languages is not well suited for the kind of generalization needed in this endeavor. For example, the idea of a search loop, expressed informally in English, is something like the following.

> A search loop is a loop in which a given predicate (the same one each time) is applied to a succession of objects, until one satisfies the predicate; the loop is then exited, making the object which satisfied the predicate available for use outside the loop.

In Lisp, this kind of loop can be written in innumerable forms, many of which are syntactically very different, such as:

```
(PROG (X)
      ...
   LP (COND ((P X)(GO OUT)))
      ...
      (SETQ X ...)
      ...
      (GO LP)
  OUT ...X...)
```

or

```
(...(PROG (X)
          ...
       LP (COND ((P X)(RETURN X)))
          ...
          (SETQ X ...)
          ...
          (GO LP))...)
```

or

```
(DO ((X ...))
    ((P X) ...X...))  .
```

The problem here is that conventional programming languages are oriented towards specifying computations in enough detail so that a simple local interpreter can carry them out. Unfortunately a lot of this detail is often arbitrary and conceptually unimportant. Gerhart [21] combines program schemas with program transformations to help overcome this problem.

A new generation of programming languages, descended from Simula, provide better syntax for specifying computations at various levels of abstraction. For example, both CLU [33] and Alphard [55] have iteration statements that capture some of the structure of the loop plans in this thesis. Like plans, these languages also enable some of both the data and control aspects of a computation to be be unified in a single specification.

However, there are two more fundamental problems with using program schemas to represent abstract programming forms, which Simula and its descendants do not solve. First, programs are not in general additive. This means that when you combine two program schemas, the resulting schema is not guaranteed to satisfy the specifications of both of the original schemas. (This is mostly due to side effects, which are a part of all practical programming languages.) In contrast, in the plan calculus all constraints between the parts of a plan are explicitly stated, so that if the constraints of two plans do not contradict, the combination is guaranteed to satisfy both original sets of constraints.

Second, existing programming languages can only describe one point of view. I speculate that the reason for this is that a program is still basically thought of as a set of instructions to be executed. In comparison, I think of a plan as a set of blueprints in which various features of a computation are specified, each from the most appropriate point of view.

Another commonly used formalism for representing abstract programming forms is *flowchart schemas*. Originally developed by Ianov in 1960 [30], flowchart schemas are a network-like connection of test and operation boxes. This formalism has the features of being programming language independent and having logical foundations. (Manna gives an excellent tutorial on the formalization and use of flowchart schemas in his book on the mathematical theory of computation [37].) Flowchart schemas capture control flow abstraction in a very natural and intuitive way. However, the only method they provide for expressing the flow of data between operations is variable assignment. Unfortunately, the use of variables in this way destroys additivity the same as for programming languages.

This problem with flowchart schemas can be fixed by combining flowchart schemas with another network-like formalism, the *data flow schemas* of Dennis [12]. In data flow schemas, operations have local port names and data flow is represented by port-to-port connections. The synthesis of these two types of schemas is essentially the temporal plan formalism used in this thesis.

Temporal plans, however, have the additional feature that mutable objects are representable, which is not the case in data flow schemas.

A currently popular formalism for specifying data abstractions is the *algebraic axiom* approach [25] [34] [22]. Though data plans are formally equivalent to abstract data types, in practice the approach of this thesis is somewhat different (mostly due to concern with mutable objects). In the algebraic axiom framework, there are no mutable objects or side effects. For example, in the standard algebraic axiomatization of stacks one defines the following three primitive functions on stacks[1]

push: stack × object → stack
pop: stack → stack
top: stack → object

and the following set of algebraic equations.

$$\text{top}(\text{push}(x,y)) = y$$
$$\text{pop}(\text{push}(x,y)) = x$$

In this thesis, however, similar behavior is formalized differently. The only primitive functions on a data structure are its roles, which are thought of as access functions. For example, the fundamental singly recursive data structure in this thesis is called List. The two primitive access functions on lists are[2]

head: list → object
tail: list → list

In this framework, operations such as Push, Pop, and Top, are non-primitive concepts which are specified by input-output specifications roughly as follows.

(i) A **Push** operation take as input a list and an object; its output is a list whose head is the input object and whose tail is the input list.
(ii) A **Pop** operation takes as input a list; its output is the tail of the input list.
(iii) A **Top** operation takes as input a list; its output is the head of the input list.

Side effects are specified in this framework by specifying an operation in which the same object is both input and output, but in which parts of that object (i.e the values of primitive access functions) are different before and after. Recently, Guttag and Horning [26] have taken a similar

---

1. We do not worry about the empty stack in this example.
2. Again we do not worry about the empty case, since it is not relevant to the comparison being made in this section. The formalization of data plans is presented more completely in Chapter Eight.

approach. They call the part of their system in which side effects are specified "routines" and use the predicate transformer notation instead of preconditions and postconditions.

Other work on representing mutable data objects and side effects includes Early [17], Burstall [5] and Yonezawa. Of these, the V-graphs of Early are the most similar to data plans. Early also takes access paths as the only primitive functions, and specifies side effect operations as transformations on the part structure of data objects.

To my knowledge, the largest existing machine-usable codification of knowledge about non-numerical programming has been compiled by Green and Barstow [23]. They use a semantic network representation for abstract programming forms, which does achieve programming language independence (Barstow's system [3] synthesizes programs in two different languages). Implementation relationships are represented in their system as *rules* in which the left hand side is a semantic network with pattern variables, and in which the right hand side is a semantic network into which the values of those variables are substituted. As compared to overlays, which are symmetric between use in analysis or synthesis, these pattern-substitution rules are biased towards use only in program synthesis. Multiple points of view are also missing from this representation.

The most serious problem with Green and Barstow's representation, however, is that it has no formal semantics. This means that the implementation rules are not verifiable. From the point of view of this thesis, this is a crucial flaw, since one of the main practical attractions of the inspection method approach is the potential for compiling a library of already verified forms, which one can combine to construct more reliable programs.

Another formalism some have found attractive for codifying programming knowledge is formal grammars. For example, Ruth constructed a grammar (with global switches to control conditional expansions) which represented the class of programs expected to be handed as exercises in an introductory PL/1 programming class. This grammar was used in a combination of top-down, bottom-up and heuristic parsing techniques in order to recognize correct and near-correct programs. Miller and Goldstein also used a grammar formalism (implemented as an augmented transition network) to represent classes of programs in a domain of graphical programming with stick figures. The major shortcoming of these grammars is the same as with Green and Barstow's representation, namely the lack of verifiability.

The three projects discussed above have taken a "practical" approach to the representation problem. They have concentrated on getting their systems to do something useful and significant with their representations, and they have succeed in those terms. In this thesis, however, I have developed a formalism which can be practically used and also has logical foundations, with the additional benefits outlined earlier.

Computer Aided Program Development Systems

The application area to which this thesis is aimed can be generally described as computer aided environments for program development. In particular, this thesis is part of a project aimed at developing what we call a *programmer's apprentice* system. What distinguishes a programmer's apprentice from existing systems is the level of program understanding shared between the user and the system.

Existing program development systems provide various types of services at different levels of understanding. The lowest level of understanding is when the system manipulates everything as text strings. At this level, various kinds of useful bookkeeping can be provided, such as keeping track of versions of source code, test data and documentation [1] [16].

The next level of understanding is when the system is able to parse the syntax of the user's programming language. At this level it is possible to provide many more useful services, such as structure editors [14] and cross-referencing [59]. If in addition the system can interpret the semantics of the programming language, then further analysis and verification assistance is possible, such as symbolic interpreters [7] [2] and verification condition generators [45]. A slight step above the programming language understanding level are systems which support the syntax of a more abstract design formalism [61].

I believe that current systems are quickly approaching fundamental limitations to the services they can provide due to fact that they understand programs only at the level of the programming language. I believe the next major step, represented by the programmer's apprentice, is to program understanding based on a library of standard programming forms. This will make it possible for the system to apply inspection methods to the analysis, synthesis and verification of programs. The scenario in the next chapter elaborates what a programmer's apprentice could do.

Other Aspects Of Programming

Inspection methods are certainly not the whole story in programming. Programmers are not always faced with totally familiar problems. Miller has studied and catalogued some very general problem decomposition methods which programmers can apply when faced with unfamiliar problems. Sussman has explored the role of debugging when plans are "almost right". Finally, Manna and Waldinger [40] have explored the applicability of deductive methods to programming.

CHAPTER TWO

PROGRAMMER'S APPRENTICE SCENARIO

2.1 Introduction

A library of plans opens up many new possibilities for what a computer aided program development system can do to help a programmer. This chapter illustrates some of these new possibilities, without going into too much detail. Chapters Five, Six and Seven go into more depth on how the behavior illustrated here can be implemented using the results of this thesis.

Many different activities are interwoven in the programming process. These activities can be roughly dividing into three major areas: analysis, synthesis and verification. Analysis activities in general involve determining properties of a plan which are not explicit in its definition (usually by decomposing it into parts). Synthesis in general involves refining an abstract plan into one which is more detailed in the appropriate sense for some target machine. Verification in general has to do detecting errors and constructing arguments as to why a given plan works.

A program development system can aid a programmer in all three of these areas. For a programmer's apprentice system, in particular, this means the same library of plans is used for analysis, synthesis and verification by inspection. For example, suppose there is a plan which captures the idea of iteration with a "trailing" value, as illustrated by the following code.

```
(PROG (CURRENT PREVIOUS)
      ...
   LP (SETQ CURRENT (... PREVIOUS))
      ...
      (SETQ PREVIOUS CURRENT)
      (GO LP))
```

If this plan is in the library, the system should be able to recognize its use in programs it hasn't seen before; it should be able to synthesize programs using this plan; and it should be able to detect errors in the use of this plan, such as incorrect initialization. This factorization of knowledge is an important feature of the design of programmer's apprentice.

The scenario in this chapter portrays a system in which inspection methods for program analysis, synthesis and verification are fully integrated. At the time of this writing, an integrated system with these capabilities has not yet been implemented. However, several of the major functions portrayed in the scenario have been implemented separately in experimental form. Waters has implemented a system which translates Lisp code to the plan calculus and performs some further

analysis on the resulting plans. Shrobe has implemented a system which verifies plans by symbolic evaluation. Although a complete synthesis system has yet been implemented, Waters has implemented the bottom-end module for this which translates suitably detailed plans to Lisp code. Finally, an experimental program for automatically drawing plan diagrams from the system's internal representation has also been implemented.

What we will see in this scenario is basically the top-down, stepwise refinement style of program development. This should not be taken as an endorsement of this methodology to the exclusion of all others. I believe that the appropriate style of development depends greatly on the particular programming task. A strongly top-down development, such as in this scenario, is appropriate when the programming task is very familiar. On other occasions, a bottom-up or middle-out methodology may be more appropriate. The research in this thesis is aimed at developing the representations and basic capabilities which are needed to support program development in any order.

2.2 The Scenario

In this scenario, we see an expert user interacting with a programmer's apprentice system to develop the implementation of a symbol table using hashing. Lines typed by the user are in lower case; lines typed by the system are in upper case. The user communicates with the system in a simple English-like command language. The important part of this language is not its syntax, but the shared vocabulary of programming terms.

```
> let a "symbol table" be a finite set.
OK
```

The system initially doesn't know anything about symbol tables *per se*. The user thus begins by introducing "symbol table" as a new term (indicated by quotation marks). The term finite set is known to the system.

```
> implement a symbol table.

 APPLICABLE IMPLEMENTATIONS FOR: SYMBOL-TABLE (FINITE-SET)
       1. LIST
       2. SEQUENCE
       3. LABELLED-THREAD
       4. DISCRIMINATION
 ? )
```

Here the system presents a *menu* of ways its knows for implementing a finite set. A completely menu-driven scenario is shown in Chapter Six. Such menus are computed dynamically by searching the current library for overlays between finite sets and other data structures. At present there are only four choices; more implementations will be added to the library in the future. The user declines to choose an option, asking for some documentation first.

```
> define discrimination function.

A DISCRIMINATION FUNCTION IS A FUNCTION SUCH THAT
IF "B" BELONGS TO THE RANGE OF THE FUNCTION, THEN B IS A SET;
AND IF "X" BELONGS TO B, THEN THE FUNCTION APPLIED TO X EQUALS B.

> implement a symbol table as a discrimination function.
OK
```

A discrimination function is a function which partitions its domain into a disjoint set of sets (called buckets). Discrimination functions are part of the system's library of standard forms. The formal definition of discrimination function presented above is not a "canned" text string, but is constructed from the logical formula which defines discrimination function in the system's library.[1] The user's first refinement step is then to implement the symbol table as a discrimination function.

The library also contains plans and overlays involving discrimination functions. The overlay invoked by the user above says that a discrimination function can be viewed as implementing a set which is the union of its buckets.

```
> show code for addition to a symbol table by side effect.

        (DEFINE SYMBOL-TABLE-ADD
           (LAMBDA (OLD INPUT)          ;MODIFIES OLD.
              (PROG (BUCKET)
                    (SETQ BUCKET (APPLY OLD INPUT))
                    (UPDATE-FUNCTION OLD
                                     BUCKET
                                     (BUCKET-ADD BUCKET INPUT)))))

        (DEFINE BUCKET-ADD
           (LAMBDA (OLD INPUT)
              ...))

        (DEFINE UPDATE-FUNCTION
           (LAMBDA (OLD VALUE INPUT)    ;MODIFIES OLD.
              ...))
```

The system knows how to implement addition to a set implemented as a discrimination function. One way of displaying this knowledge is for the system to generate code from its current plan representation of the design thus far. As can be seen above, this code has gaps in it (indicated by three dots). These gaps correspond to the fact that it doesn't yet know how the discrimination function or the buckets are implemented.

The arguments to the procedure SYMBOL-TABLE-ADD are the symbol table to be modified (OLD) and the entry to be added INPUT. At this point in the design, all the system knows is that the old symbol table is a function, but not how that function is implemented. The discrimination function may later in the design turn out to be implemented by a Lisp procedure, or by a data structure, or (in

---

1. See the appendix for the definition in logical form. The technology for making this kind of a straightforward translation exists.

fact) a combination of the two. The code (APPLY OLD INPUT) in the first line of SYMBOL-TABLE-ADD is therefore used by the system to express the general idea of applying the discrimination function to get the appropriate bucket in which to add the given entry. Similarly, the procedure UPDATE-FUNCTION is a placeholder for the operation of modifying an OLD function such that all domain elements which used to map to a given range VALUE (in this case the bucket before addition), will afterwards map to the INPUT (in this case the bucket after addition).

Despite the arguments made earlier as why code is not an appropriate representation for the library of standard forms, code can still be a convenient language for communicating between the system and the user in certain circumstances. Also code has the short term advantage of being more familiar to the user, as compared to using the plan calculus explicitly.

Notice also that the system has annotated the code it has written with comments. We will see later in the scenario that the system also reads the comments on any code that the user types in.

```
> describe symbol-table-add.

SET ADDITION TO A DISCRIMINATION FUNCTION IS IMPLEMENTED BY THREE STEPS,
        DISCRIMINATE: APPLY THE DISCRIMINATION FUNCTION
        ACTION: SET ADDITION
        UPDATE: MODIFY THE DISCRIMINATION FUNCTION WITH A NEW VALUE
SUCH THAT THE OUTPUT OF THE DISCRIMINATE STEP IS THE OLD SET OF THE ACTION STEP;
THE INPUT OBJECT OF THE ACTION IS THE INPUT OBJECT OF THE DISCRIMINATE STEP;
THE OUTPUT OF THE DISCRIMINATE STEP IS THE VALUE INPUT TO THE UPDATE STEP;
THE NEW SET OF THE ACTION STEP IS THE INPUT OBJECT TO THE UPDATE STEP;
AND THE DISCRIMINATION FUNCTION IS THE OLD FUNCTION INPUT TO THE UPDATE STEP.
```

In response to a request from the user, the system here gives a toplevel logical decomposition of the code it has written. Again, this description is not "canned" text, but can be generated from the plan representation as needed. For this sort of information, displaying the plan diagram shown in Fig. 2-1 is probably superior.

```
> implement the discrimination function as a keyed discrimination.
OK

> define keyed discrimination.

A KEYED DISCRIMINATION IS A COMPOSITION OF TWO FUNCTIONS,
        ONE: A FUNCTION ("THE KEY FUNCTION")
        TWO: A FUNCTION ("THE BUCKET FUNCTION")
SUCH THAT THE COMPOSITION, VIEWED AS A FUNCTION, IS A DISCRIMINATION
FUNCTION.
```

The design of the symbol table continues in small steps. The next step here is to decompose the discrimination function into two functions: one which maps from objects to keys; and one that maps from keys to buckets. The strings in quotation marks above are "canned" text which is attached to roles of the plan to give better words than "the one function" and "the two function", which would be generated automatically.

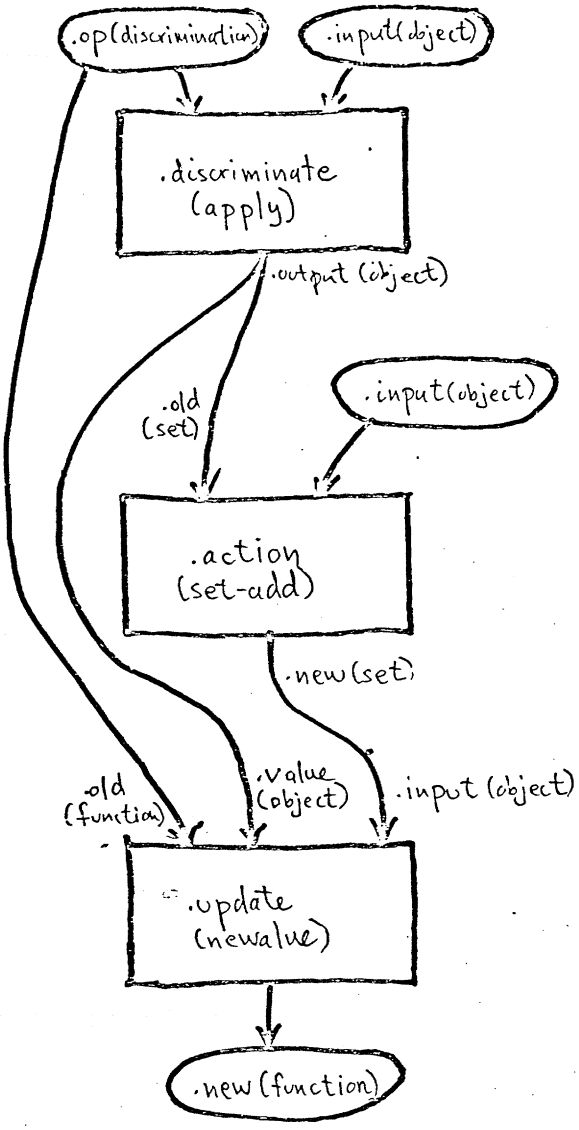


Figure 2-1. Discriminate, Action and Update Plan for Addition to Symbol Table.

The system knows quite a bit about functional compositions. For example, it knows that the range of the first function must be a subset of the domain of the second function. It also knows that to update a function implemented as the composition of two functions, it suffices to update the second function. Both of these pieces of information will be used later in the scenario.

```
> the key function of the keyed discrimination is car.
OK

> implement the bucket function of the keyed discrimination as a hashing.
OK

> define hashing.

A HASHING IS A COMPOSITION OF TWO FUNCTIONS,
        ONE: A FUNCTION ("THE HASH FUNCTION")
        TWO: AN IRREDUNDANT SEQUENCE ("THE TABLE") .
```

The final step in the refinement of the symbol table data structure is to introduce hashing. The basic idea of hashing is to decompose a function (in this case the function from keys to buckets) into two functions: a many-to-one function which maps from the original domain to an initial interval of the natural numbers; and one-to-one function which maps from the natural numbers to the original range. Sequences are formalized in this system as functions on initial intervals of the natural numbers. An irredundant sequence is one in which no two terms are equal.

```
> show symbol-table-add.

        (DEFINE SYMBOL-TABLE-ADD
           (LAMBDA (TABLE INPUT)                ;MODIFIES TABLE.
              (PROG (INDEX)
                    (SETQ INDEX (HASH (CAR INPUT)))
                    (ARRAYSTORE TABLE INDEX
                                (BUCKET-ADD (ARRAYFETCH TABLE INDEX)
                                            INPUT)))))

        (DEFINE HASH
           (LAMBDA (INPUT)
              ...))

        (DEFINE BUCKET-ADD
           (LAMBDA (OLD INPUT)
              ...))
```

The user has asked the system to regenerate code for SYMBOL-TABLE-ADD based on the additional specifications provided thus far. Notice that the system still doesn't know how the buckets are implemented, so BUCKET-ADD is still just a placeholder. However, the system does now have enough information to make some other implementation decisions. For example, since the hashing function does not need to be modified, it has been implemented as a Lisp procedure.[1] The only function that needs to be modified now is the table (a sequence). The system has also taken the

---

1. There is nothing in the current plan library of the lore concerning what makes for good hashing functions. This is really in the domain of numerical computation and theory, which is outside of the focus of this thesis.

initiative of implementing this sequence as a Lisp vector. Modifying a Lisp vector is achieved by ARRAYSTORE.

If desired, the user could have been queried about each of these decisions. However, it is probably preferable to have the system take the initiative at this low level and make it possible for the user to retract decisions later.[1]

```
> show code for associative retrieval from a symbol table.

        (DEFINE SYMBOL-TABLE-RETRIEVE
           (LAMBDA (TABLE INPUT)
              (BUCKET-RETRIEVE (ARRAYFETCH TABLE (HASH INPUT))
                               INPUT)))

        (DEFINE BUCKET-RETRIEVE
           (LAMBDA (BUCKET INPUT)
              (PROG (OUTPUT)
                    ...
                    (COND (... (RETURN NIL)))
                    ...
                    (COND ((EQ (CAR OUTPUT) INPUT)
                           (RETURN OUTPUT)))
                    ...))))
```

Associative retrieval is a standard specification known to the system. Conceptually, it has three inputs: a set, a key function, and a search key. It also two cases: if there is a member of the set such that the key function applied to it equals the search key, then the retrieval succeeds and its output is such a member; otherwise, it fails.[2]

As can be seen from the code above, the system also knows the standard plan for implementing associative retrieval from a set implemented as a keyed discrimination, namely: apply the bucket function to the search key to obtain a bucket; and then perform associative retrieval from the bucket using the same key function and search key. Notice that the respective sets (either the whole table or the bucket) and the search key (INPUT) are the formal parameters of SYMBOL-TABLE-RETRIEVE and BUCKET-RETRIEVE in the code above, while the key function, CAR, is coded in line. This coding does not cause any loss of modularity, since the purpose of that particular use of CAR is preserved in the plan representation.

The gaps in BUCKET-RETRIEVE are due to the fact that the user has not yet specified how the buckets are implemented. However, the systems does know that this procedure conceptually has two cases. Procedures in Lisp can have only one return point. The system has thus decided to use the standard technique of a *flag* to encode two cases -- a return value of NIL signals the failure case.

---

1. A truth maintenance system can facilitate this kind of retraction. The issues involved in incremental modification are an important part of the programmer's apprentice project in general, but are outside of the focus of this thesis.
2. This way of specifying associative retrieval generalizes well to multiple key retrieval.

Minor programming techniques, such as the use of flags, can also be captured in the plan representation.

The following frame illustrates another kind of interaction between the user and the system. In addition to being able to read each other's code, the user and system can also *edit* each other's code. This frame also shows how the plan library can provide the user with a sort of "macro expansion" facility for writing code more easily and correctly. For example, the user below invokes the standard plan for generating successive elements of a Lisp list by CAR and CDR, truncated by NULL. The code that is generated for this plan in this context is underlined below.

```
> insert code for truncated list generation of bucket in bucket-retrieve.

        (DEFINE BUCKET-RETRIEVE
           (LAMBDA (BUCKET INPUT)
              (PROG (OUTPUT)
                 LP (COND ((NULL BUCKET)(RETURN NIL)))
                    (SETQ OUTPUT (CAR BUCKET))
                    (COND ((EQ (CAR OUTPUT) INPUT)
                           (RETURN OUTPUT)))
                    (SETQ BUCKET (CDR BUCKET))
                    (GO LP)))))
```

The important difference between what takes place here and a conventional macro facility is that here the insertion of new material takes place in the plan calculus, where the benefits of additivity hold. Only afterwards is code regenerated from the augmented plan.

Notice that the user here has indirectly told the system that the buckets of the symbol table are implemented as lists. This gives the system enough information to complete the implementation of BUCKET-RETRIEVE by identifying the exit test of the list generation with the failure case of the associative retrieval, (RETURN NIL), and by setting OUTPUT to the successively generated list elements.

```
> show symbol-table-add.

        (DEFINE SYMBOL-TABLE-ADD
           (LAMBDA (TABLE INPUT)                ;MODIFIES TABLE.
              (PROG (INDEX)
                    (SETQ INDEX (HASH (CAR INPUT)))
                    (ARRAYSTORE TABLE INDEX
                                (CONS (ARRAYFETCH TABLE INDEX)
                                      INPUT)))))
```

Given that buckets are implemented as lists, the system assumes the most straightforward implementation for BUCKET-ADD which is simply CONS, as shown above.

```
> describe the implementation of a symbol table.

A SYMBOL TABLE IS IMPLEMENTED AS A DISCRIMINATION FUNCTION.

THE BUCKETS OF THE DISCRIMINATION FUNCTION ARE IMPLEMENTED AS LISTS.

THE DISCRIMINATION FUNCTION IS IMPLEMENTED AS A KEYED DISCRIMINATION.

THE KEY FUNCTION OF THE KEYED DISCRIMINATION IS CAR.
THE BUCKET FUNCTION OF THE KEYED DISCRIMINATION IS IMPLEMENTED AS A HASHING.

THE HASH FUNCTION OF THE HASHING IS IMPLEMENTED AS A LISP PROCEDURE.
THE TABLE OF THE HASHING IS IMPLEMENTED AS A LISP VECTOR.
```

Above we see how the programmer's apprentice can provide an interactive documentation facility. This can be particularly useful if more than one programmer is working on a program, or if a program needs to be modified a long time after it was written.

```
> show code for associative deletion from a symbol table by side effect.

        (DEFINE SYMBOL-TABLE-DELETE
           (LAMBDA (TABLE INPUT)        ;MODIFIES TABLE.
              (PROG (INDEX)
                    (SETQ INDEX (HASH INPUT))
                    (ARRAYSTORE TABLE INDEX
                                (BUCKET-DELETE (ARRAYFETCH TABLE INDEX))))))

        (DEFINE BUCKET-DELETE
           (LAMBDA (BUCKET INPUT)
              (COND ((NULL BUCKET) NIL)
                    ((EQ (CAAR BUCKET) INPUT)
                     (BUCKET-DELETE (CDR BUCKET) INPUT))
                    (T (CONS (CAR BUCKET)
                             (BUCKET-DELETE (CDR BUCKET) INPUT))))))
```

Associative deletion is also a standard specification known to the system. Like associative retrieval, it has three inputs: a set, a key function, and a key. Its output is the input set minus those members for which the key function applied to them equals the input key. The implementation of associative deletion from a set implemented as a discrimination function is a similar three step plan to the set addition plan introduced earlier, namely: apply the discrimination function to get a bucket, perform the same associative deletion on the bucket to get a new bucket, and then modify the discrimination function to incorporate the new bucket. The system has generated code for this plan as shown above.

Notice that associative deletion from the bucket list is implemented by the system in the straightforward manner which copies the list. In the next frame, we will see that the user has something more clever in mind, and therefore intervenes to provide his own "more efficient" code for deleting from the bucket by side effect.

```
> edit bucket-delete

        (define bucket-delete
           (lambda (bucket input)                ;modifies bucket.
              (prog (p q)
                    (setq p bucket)
                 lp (cond ((eq (caar p) input)
                           (rplacd q p)          ;splice out.
                           (return bucket)))
                    (setq q p)
                    (setq p (cdr p))
                    (go lp))))

WARNING! THE LOOP IN BUCKET-DELETE IS ALMOST A
TRAILING GENERATION AND SEARCH,
        CURRENT: P
        PREVIOUS: Q
        EXIT: (COND ((EQ (CAAR P) ...)))
        ACTION: (CDR P)
EXCEPT THAT THE OUTPUT OF THE ACTION IS NOT EQUAL TO THE
INPUT OF THE EXIT TEST.
```

Here we see an example of inspection methods used for verification. The user has attempted to code a generation and search loop with a trailing value and has not gotten it quite right.[1] The plan in the library for trailing generation and search has the roles for the Current value, the Previous value, the Exit test, and the generating Action on each iteration, with roughly the following constraints between them:

(i) The output of the Action is equal to the input of the Action on the next iteration.
(ii) The output of the Action is equal to the input of the Exit test.
(iii) The Current value is equal to the Input of the Exit test.
(iv) The Current value is equal to the Previous value on the next iteration.
(v) The Current value and Previous value are outputs of the loop.

In a near-miss recognition, most but not all of the constraints of a plan are satisfied. In this example, constraint (ii) is not satisfied as indicated by the system in the warning message above. The details of how this recognition takes place are explained in Chapter Seven.

Verification by inspection yields a much more meaningful diagnostic than would be given by other methods of detecting this error. For example, running the code above with certain inputs would result in the Lisp interpreter halting at the RPLACD with an error message such as the following.

```
; NIL BAD ARG - RPLACD
```

---

1. Whether or not the user should have known that this particular plan was in the library is irrelevant. The point here is to illustrate the idea of a program understanding system that "looks over the shoulder" of the user.

In general, correcting errors is more difficult than detecting them. For example, it is hard for the system to know whether a near-miss is actually an error or just a new variation on a plan it doesn't know about. The programmer's apprentice will thus in general rely on the user to correct errors. The user's response to the warning message above is shown underlined below.

```
> edit bucket-delete

        (define bucket-delete
           (lambda (bucket input)                ;modifies bucket.
              (prog (p q)
                    (setq q bucket)
                 lp (setq p (cdr q))
                    (cond ((eq (caar p) input)
                           (rplacd q p)          ;splice out.
                           (return bucket)))
                    (setq q p)
                    (go lp))))

    WARNING! THE CODE (RPLACD Q P) IS NOT RECOGNIZED AS SPLICEOUT.
    SUGGEST (RPLACD Q (CDR P)) ? yes
```

Unfortunately, there is yet another error in the user's code. The user has asserted in a comment above that the RPLACD line implements splicing an element out of the list. However, the system finds that this code does not match its library plan for splicing out. In this case, since the user has stated his intention in a comment, the system can go as far as offering a correction, which is accepted.

Notice that at different points in this scenario, Lisp lists have been viewed both recursively and as directed graphs. The first version of BUCKET-DELETE was a standard recursion on the tail of the list; while the version above is in the domain of graph manipulations involving successor nodes and modifying arcs between them.

Now that BUCKET-DELETE has been re-implemented using side effects, a more efficient implementation of SYMBOL-TABLE-DELETE is possible, in which the ARRAYSTORE is necessary only if the first element of the bucket is deleted.

```
> show symbol-table-delete.

        (DEFINE SYMBOL-TABLE-DELETE
           (LAMBDA (TABLE INPUT)          ;MODIFIES TABLE.
              (PROG (INDEX BUCKET)
                    (SETQ INDEX (HASH INPUT))
                    (SETQ BUCKET (ARRAYFETCH TABLE INDEX))
                    (COND ((EQ (CAAR BUCKET) INPUT)
                           (ARRAYSTORE TABLE INDEX (CDR BUCKET)))
                          (T (BUCKET-DELETE BUCKET))))))
```

To come to this implementation, the system has done some analysis of side effects by inspection. Specifically, there are plans and an overlay in the library which say that one way to modify a function (change the associations between domain and range elements by adding a new

range element) is to modify an old range element. Applied to this program, this overlay allows the system to view the deletion by side effect of an element from the bucket as the implementation of the modification of the discrimination function.

Analysis by inspection is also in operation here. By recognizing the user's BUCKET-DELETE code as a trailing generation and search plan, the system derives some important additional properties of this procedure. In particular, it knows that this procedure only searchs internal nodes of the bucket list, and that it only finds the first node which has the given key. With regard to the first property, there is a plan in the library which combines an internal deletion with a conditional test on the first node to achieve a complete deletion. The system has used this plan to arrive at the code above. The second property is propagated up to the specifications of SYMBOL-TABLE-DELETE, as shown below.

```
> describe preconditions of symbol-table-delete.

THERE EXISTS A UNIQUE "X" SUCH THAT X BELONGS TO THE OLD SYMBOL TABLE,
AND THE CRITERION APPLIED TO X IS TRUE.

> describe preconditions of symbol-table-insert.

THE INPUT DOES NOT BELONG TO THE OLD SYMBOL TABLE.
```

Thus analysis by inspection has revealed some important additional restrictions which the user either was not clearly aware of, or in any case, did not explicitly state. The propagation of restrictions from the specifications of BUCKET-DELETE to SYMBOL-TABLE-DELETE and SYMBOL-TABLE-ADD could be achieved by the use of general reasoning mechanisms. However, in keeping with the emphasis on inspection methods in this thesis, I argue these are familiar specializations of the most general addition and deletion specifications, and are therefore pre-compiled in the library.

CHAPTER THREE

OVERVIEW OF PLAN LIBRARY

3.1 Introduction

This chapter gives an overview of the plan library with an emphasis on taxonomy. English descriptions and examples programs are used to give a feeling for the extent and overall organization of the knowledge in the library. Formal definitions for all library entries can be found in the appendix (see index for page numbers) written in a notation which is explained in Chapter Eight. Chapters Five, Six and Seven, describe the use of the library in specific scenarios of analysis, synthesis and verification by inspection.

Let me emphasize that the taxonomy represented in the current library is only intended to be a beginning. The exact contents of the current library has been determined primarily by the requirements of giving a complete account of one significant size example program, capturing all the important generalizations. The example program that was chosen for this is the symbol table program introduced in the scenario of Chapter Two. This particular program was chosen because contains many different forms which are representative of the domain of common manipulations on symbolic data. I felt that a library which was adequate for this example would be a good start towards complete coverage of the domain. Furthermore, I felt that concentrating on understanding one example in depth would lead to better development of the relationships between many different levels of abstraction in the library.

Both of these feelings have been borne out in fact. Capturing all the important generalizations in this one program has touched upon a remarkable range of basic programming techniques. Furthermore, to give a complete account of the symbol table program has required filling the library with plans at a fairly abstract level, such as the idea of implementing a set as a discrimination function, down to the level of minor programming techniques, such as the use of flags, and at many levels in between. I found that if one looks deep enough, one can indeed "see the universe in a blade of grass".

The small fraction of plans in the current library which are not directly motivated by the symbol table example fall into two categories. Some of these are obviously important basic plans which don't happen to be used in the example, such as counting and accumulation loops. Other plans are included to fill gaps in the taxonomic structure of the library, such as the plan for splicing into a list (only splicing out appears in the symbol table program).

While I would defend the major outlines and organization of the current library, I do not expect that any reader will agree on every last detail. Many common manipulations on symbolic data are missing at present. The current library also needs be expanded in many different directions, such as to include more specialized graph algorithms, matrix manipulations, and so on. However, it will hopefully be clear after reading this chapter where many of these extensions would fit into the existing structure.

Methodology

My basic methodology in developing an taxonomy of programming forms has been to start with the technical vocabulary commonly used and understood by experienced programmers, and then to apply my own intuitions to make appropriate generalizations and distinctions. I thus take the position that if programmers have evolved a name for something, it is probably an important concept. This means, for example, that there are objects in the library which express the meaning of terms like "trailing pointer", "search loop" and "splice out".

Another method I have used to discover important programming concepts is to look for abstractions which unify the explanations of how many different programs work. For example, the concept of a directed graph makes it possible to express a number of standard algorithms independently of how the nodes and edges are represented in a particular program. This line of argument has also lead to including in the library a number of other familiar mathematical objects, such as functions, relations, sequences and sets.

The vocabulary of descriptions is not the only kind of knowledge involved in programming. A programmer also knows many ways of implementing one specification in terms of others. The idea of implementing a set as a hash table, or of removing an entry from a list by splicing it out, are examples of implementation relationships. In building up a library of programming knowledge, there is an interplay between these implementation relationships and the vocabulary of descriptions. One motivation for making a vocabulary distinction can be to separate two cases which allow different implementations. For example, finite and infinite sets are distinguished in the library because membership tests for finite sets may be implemented by a loop with two exits, which is not that case for infinite sets. (The set of natural numbers is an example of an infinite set which is part of basic programming.)

A third important kind of knowledge, which is *not* yet explicitly represented in the library, is the relative cost (according to an appropriate metric) of various computations. I have taken the approach of first studying the vocabulary of computation descriptions and implementation relationships because, in part, I believe that much of an expert programmer's knowledge about the relative cost of computations is embedded in his vocabulary. In other words, given that cost considerations are the primary motivation behind many standard programming ideas, the study of

these ideas is a logical starting place for developing an understanding of computational cost. For example, the idea of a hash table is motivated by the desire to speed up various kinds of retrieval operations. This increase in speed is due the fact that any single bucket in the table is smaller than the union of all the buckets. I hope to study the existing library further from this viewpoint in order to make this kind of knowledge more explicit.

Overall Organization

The current library contains approximately 50 input-output and test specifications, 50 data plans, and 100 temporal plans. These plans and specifications are organized in two ways: in a taxonomic hierarchy and by an interlocking network of approximately 100 overlays. There are two taxonomic relationships used in the library: specialization and extension. Furthermore, a plan may be a specialization or extension of more than one other plan, so that the taxonomic hierarchy may be tangled. An example of this appears later in this chapter.

A plan or specification is a *specialization* of another plan or specification if it has the same roles, but additional constraints. This means that the computations or data structures specified by the specialized plan are a subset of those specified by the more general plan.

A common motivation for introducing a specialization of a plan is because the properties of the specialization are exploited in some particular implementation. For example, consider the data plan, Segment, introduced in Chapter One. This data plan has three roles: a Base sequence, an Upper index, and a Lower index. One way of implementing a mutable stack is to use an instance of Segment in which only the lower index is varied -- the upper index is always equal to the length of the base sequence. I call this data plan Upper-segment; it is a specialization of Segment. Upper-segment has the same role names as Segment. Its constraints are the three constraints of Segment, i.e.

(i) The Upper number is less than or equal to the length of the Base sequence.
(ii) The Lower number is less than or equal to the length of the Base sequence.
(iii) The Lower number is less than or equal to the Upper number.

plus the following specializing constraint.

(iv) The Upper number is equal to the Length of the Base sequence.

The basic idea of *extension* is to add an additional role to a plan or sepcification. The extended plan also inherits all the constraints of the old plan.

A common kind of extension is to add an additional output to an input-output specification. For example, when Thread-find is the standard input-output specification for finding a node satisfying a given criterion in a linear directed graph (called a thread), assuming there is one. It has two input roles, named Input and Criterion, and one output role, named Output. The Output is a

node of the Input thread which satisfies the Criterion predicate. When Thread-find operations are used in conjunction with other plans, such as splicing, it is convenient to output not only the node found, but also the previous node in the thread. I call this extension Internal-thread-find. Internal-thread-find has the same input roles as Thread-find and two output roles, Output and Previous, with the additional constraint that the successor of the Previous node in the Input thread is the Output node.

Object Types

Part of the hierarchy of object types is shown in Fig. 3-1. The names in this figure are the names either of primitive object types or data plans. Similar figures later in this chapter will also include the names of input-output and test specifications, and temporal plans. Plain solid vertical lines between names in these figures denote specialization or extension relationships, with the specialized or extended plan always below. Arrows in these figures represent overlays between plans. Most overlays are many-to-one mappings from instances of one plan to another. The arrowhead for such ovelays points from the domain to the range. Overlays that are one-to-one are indicated by double-headed arrows. Dotted lines indicate "use" relations. For example, the definition of labelled directed graph data plan makes use of the definition of the directed graph (Digraph) data plan.

Referring to Fig. 3-1, note that the root node in the data object hierarchy is called **Object.** Below Object are the primitive types in the current library are **Integer, Function, Binfunction** (functions of two arguments), and **Set.** By "primitive" I mean here that systems which use the plan library are expected to have specific procedures for reasoning about these objects, and that this knowledge is not explicitly represented in the library itself.

The notion of Integer used here are a standard extension of the finite integers with a maximum element, Infinity, and a mimumum element, Minus-infinity. Integer has subsets Natural and Cardinal. Instances of **Natural** are all the integers greater than or equal to 1, and not including Infinity. Instances of **Cardinal** are all the integers greater than or equal to zero, including **Infinity.**

Subsequent main sections of this chapter give overviews of parts of the library under the other main nodes in this hierarchy. There is a section about plans involving functions, one about plans involving sets, one about directed graphs, and one about recursive structures. However, these sections will not be able to mention every last plan in the library, since that would make the figures an unreadable clutter. For example, some plans involving minor programming techniques, such as the use of flags, various ways of implementing predicate tests are described as they arise in the scenarios later in the thesis, and in the appendix.

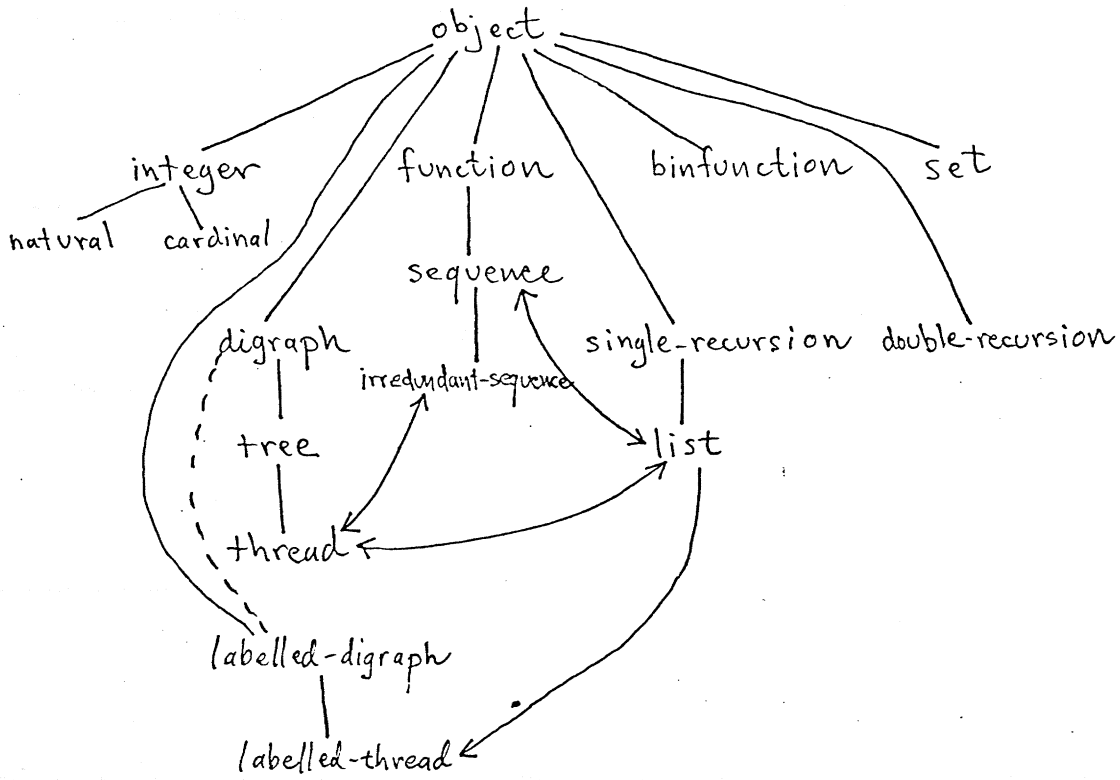


Figure 3-1. Hierarchy of Object Types.

Notice the overlays in the middle Fig. 3-1 between Sequence, List, Thread, and Labelled-thread. These overlays will be explained in more detail in subsequent sections. For now it is important just to point out this example of how multiple of points of view are catalogued in the library. Each of these data plans (Sequence is a subset of the primitive object type Function) captures an alternative point of view on what could be called *linear structures.*

3.2 Functions

Fig. 3-2 shows parts of the plan library which involve functions. At the top left are three basic input-output specifications which have functions as inputs or outputs. **@Function** is the specification for applying a function to an argument to get a value.[1]

Another common operation performed on functions is to change the value associated with a given argument. The input-output specification for this operation is called Newarg. Newarg has three inputs: the old function, the arg, and the new value. The output is a new function such that the argument maps onto the new value and the values of all other arguments remain unchanged.

A less commonly used specification is **Newvalue.** Newvalue also has three inputs: the old function, the old value, and the new value. The output is a new function such that all the arguments that used to map onto the old value now map onto the new value and the values of other arguments remain unchanged. Newvalue will be used in this thesis as part of the explanation of how hash tables work.

Notice that these specifications make no commitment as to whether the old function is copied or modified to get the new function. The copying and side effect versions will be treated as specializations. It is advantageous to work with these more abstract specifications as much as possible, since they unify the logical structure of a larger number of programs. The input-output specification, **Old+new**, of which Newarg and Newvalue are specializations, is a very general form which makes it possible to state this idea in general. These same remarks apply to all other input-output specifications in this chapter which are shown as specializations or extensions of Old+new. Plans involving side effects are discussed further in Chapter Ten.

At the middle left of Fig. 3-2 are some plans having to do with implementing a function as the composition of two functions, i.e. by the data plan **Composed-Functions.**

---

1. The character "@" is intended to be read as "apply".

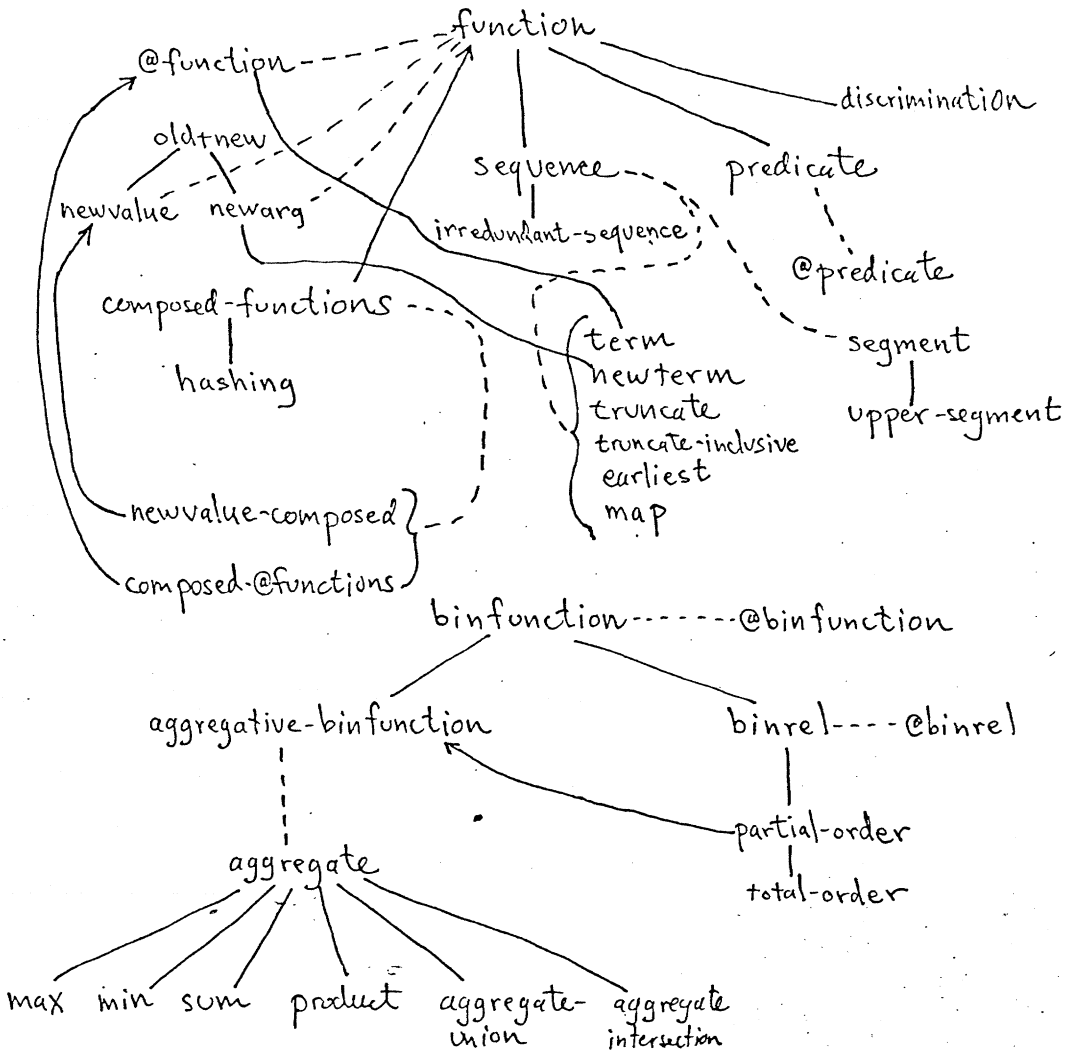


Figure 3-2. Plans Involving Functions.

Composed-@functions is the temporal plan for applying the second function of a composition to the output of applying the first function to the given argument, which is the implementation of @Function for a function implemented as a composition.

The plan **Newvalue-composed**, and the overlay betwween it and Newvalue express the fact that a Newvalue operation on a function implemented by a composition can be implemented by a Newvalue operation on the second function only. This plan arises in the symbol table example where the hash table is viewed as the composition of two functions: a numerical hash function which doesn't change, and an array that is modified to insert new entries.

Notice that the data plan **Hashing** is a specialization of Composed-Functions. As we have seen in the scenario, the first function in this case is referred to as the hash function, and the second (a sequence) is referred to as the table. A discrimination function can be implemented as a hash table, in which case the table is a sequence of sets, called the buckets. The significance of this implementation is that changes (e.g. Newvalue operations) to a discrimination decomposed this way may be achieved by changing only the table, as specified by the Newvalue-composed plan discussed above. Discrimination functions will be discussed further in the next section on sets.

Sequences

Sequences in this thesis are viewed formally as a subset of functions on the natural numbers which are defined on some initial interval (up to the **Length** of the sequence) and undefined elsewhere. A common specialization is **Irredundant-sequence**, i.e. sequences in which no two terms are equal.

A number of common operations on linear structures are most naturally specified in terms of sequences. Fig. 3-2 shows several such input-output specifications. The first two specifications, **Term** and **Newterm**, are simply specializations of @Function and Newarg to the case when the functions involved are sequences.

The next two specifications have to do with truncating sequences according to some criterion predicate. In both cases a precondition is that there exist some term of the input sequence which satisfies the criterion. The output sequence in both cases is a finite initial subsequence of the input sequence. In the case of **Truncate-inclusive**, all but the last term of the output sequence fail the criterion; the last term passes. In the case of **Truncate**, all terms of the output sequence fail the criterion and the length of the output sequence is one less than the index of the first term in the input sequence that passes the criterion.

A closely related input-output specification is **Earliest**. Again the inputs are a sequence and a criterion, and a precondition is that there exist some term of the input sequence which satisfies the

criterion. The output is the earliest term of the sequence which passes the criterion, i.e. all terms with indices lower than the index of the output fail the criterion.

The final input-output specification on sequences in Fig. 3-2 is **Map**. Here the input and output are sequences of the same length. An additional input (Op) is a function such that each term of the output is the result of applying that function to the corresponding term of the input.

Aggregations

This section introduces a bit of elementary algebra to capture the similarity in structure between programs which compute sums, products, set unions and intersections, maximums and miminums. What all of these have in common is that they involve functions of two arguments which are commutative, associative and have identity elements. I call such functions *aggregative.*[1]

The input-output specification which is the generalization of all these operations is called **Aggregate**. Aggregate takes as input a (non-empty finite) set of objects and an instance of **Aggregative-binfunction**. The output of Aggregate is the result of composing the application of the given binary function to the members of the input set. The algebraic properties of aggregative functions guarantee that the order of this composition doesn't matter.[2]

Fig. 3-2 also names six common specializations of Aggregate for particular common aggregative functions: **Sum** (for Plus), **Product** (for Times), **Aggregate-union** (for Union), **Aggregate-intersection** (for Intersection), **Max** (for Greater), and **Min** (for Lesser).

Relations

Note in Fig. 3-2 note that relations are treated as boolean valued functions. In particular, unary relations, **Predicate**, are a subset of functions of one argument, and binary relations, **Binrel** are a subset of functions of two arguments. Correspondingly, **@Predicate** is the specialization of @Function to predicates, and **@Binrel** is the specialization to binary relations.

Finally in Fig. 3-2 note the overlay between **Partial-order**, a specialization of Binrel, and Aggregative-binfunction. This overlay allows the following code

---

1. If in addition there is an inverse function, then we have the structure of an Abelian group. Of the six functions mentioned above, Plus, Times, and Union have this structure.

2. Which is why the input is a set rather than a list or sequence. Also there is some subtlety being suppresed here concerning whether the input should be a set or a multiset. In the case of union, intersection, maximum and minimum, the occurence of duplicates doesn't matter, and therefore the set abstraction is definitely appropriate. Sum and product, however, do not have this property. Nevertheless, I argue that, conceptually, the input to a summation operation is a set of objects in the sense that even though viewed as integers they may have the same behavior, they represent conceptually distinct quantities and are therefore not identical. See Chapter Eight for more on the notion of behavior versus identity.

```
(COND ((> N MAX)(SETQ MAX N)))
```

to be analyzed as an application of the Lesser function (and similarly, when the test is "<", an application of Greater), which then facilitates analyzing a loop with this code in the body as implementing a Max or Min operation.

3.3 Sets

Fig. 3-3 shows part of the plan library which involves sets. At the left of the figure we first have some common input-output and test specifications with sets. **Member?** tests whether a given object is a member of a given set. **Any** is a more complicated test: given a set and a predicate as inputs, if there exists a member of the set which satisfies the predicate, it succeeds and returns such a member as its output; otherwise it fails. **Set-find** is a related input-output specification: it has the precondition that there that there exists a member of the input set which satisfies the input predicate, and simply returns such a member as its output.

The next two input-output specifications each have a set as input and a set as output. **Each** is a specification used to abstract programs like `(MAPCAR 'SQRT L)`, where the input list, `L`, is viewed as a set and `SQRT` is a function applied to each element of the set to get an output set. **Restrict** takes as input a set and a predicate and returns the subset which satisfies the predicate. As in the case of functions, no commitment is made in these specifications as to whether the old set is copied or modified to get the new set.

Finally, **Set-add** and **Set-remove** specify addition and removal of a given input object to or from a set. The general specification Old+input+new-set, of which both Set-add and Set-remove are specializations, makes it possible to capture what the implementations of these specifications have in common for sets implemented as discrimination functions, (which will be discussed later in this section).

The implementation of sets is a very rich area of programming technique [52]. It is not the goal of this thesis to be exhaustive of all of the possibilities, but rather to show by example how to go about formalizing such implementations using the plan calculus. In addition to the standard simple implementations of sets as sequences and lists, this section presents two examples of non-trivial set implementations, which are involved in understanding the the symbol table program.

The overlay for viewing a list as the implementation of a set is recursively defined: an object is a member of the implemented iff it is the head of the list or it is a member of the set implemented by the tail of the list. The empty set is usually implemented by Nil. There are also overlays in the library for viewing Push and Pop operations as Set-add and Set-remove operations.

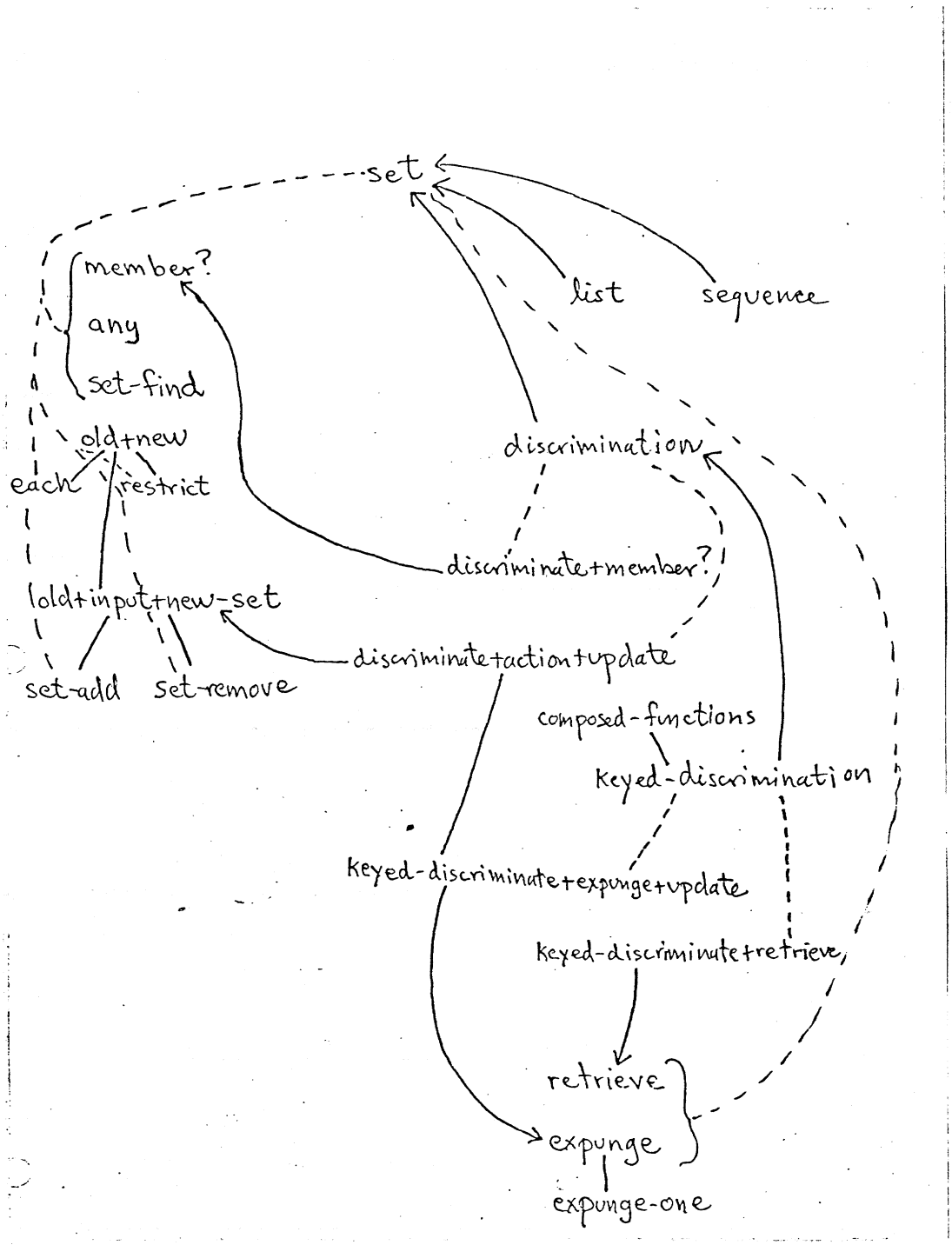


Figure 3-3. Plans Involving Sets.

The implementation of other set operations is more naturally expressed taking the point of view of the list as a directed graph, which will be discussed in the next section.

A sequence can be viewed as the implementation of the set of terms of the sequence.

Discrimination

One basic idea underlying many set implementations is to introduce a function (called a **Discrimination**), whose range is a set of sets (called buckets). Such a function can be viewed as implementing a set wherein a given object is a member iff it is a member of the bucket obtained by applying the discrimination function to that object. This is the basic "divide and conquer" strategy underlying both hash tables and discrimination nets.

Testing for membership in a set implemented as a discrimination is implemented by the two step plan Discriminate+member?, as shown in Fig. 3-3. The first step is to apply the discrimination function to the given object to determine which bucket to look in. The second step is an instance of Member?, with the set input being the bucket fetched by the first step. Since any single bucket in a discrimination is smaller than the overall implemented set, (except in the case of a degenerate discrimination function which maps all objects to a single bucket), this implementation leads to a increase in speed.

Both Set-add and Set-remove for for input and output sets implemented as discriminations, are implemented by specializations of the same three step plan: first, apply the discrimination function to the input object to obtain a bucket; second, perform the appropriate operation on that bucket to get a new bucket; and finally, update the discrimination function so that all domain objects which used to map onto the old bucket now map onto the new bucket (i.e. a Newvalue operation). These three steps are expressed by the **Discriminate+action+update plan.**

Associative Retrieval

Associative retrieval adds to basic set operations the concept of a key. The function which associates members of a set with keys is called the key function. Given a set, such as the entries in a symbol table, we are often more interested in finding a member with a given key, than just testing for membership. The most basic specification for associative retrieval is called **Retrieve** (see bottom of Fig. 3-3). Given a set, a key function and an input key, Retrieve has two cases: if there exists a member of the set with the given key, then it succeeds, and its output is such a member; otherwise it fails. The other main associative retrieval specification, **Expunge**, removes all members of an input set which have a given key. **Expunge-one** is a common specialization of Expunge which often allows a simpler implementation. Expunge-one has the additional precondition that there exists exactly one member of the input set with the given key.

Keyed Discrimination

To speed up associative retrieval for a given key function, a discrimination function can be used which is itself the composition of two functions. This is the data plan **Keyed-discrimination** (see middle of figure). The first function is called the key function. The second function, called the bucket function, maps from the set of keys to the buckets. In typical usage, the bucket function may itself be decomposed further into a Hashing (or another keyed discrimination, as will be discussed shortly).

The implementation of Retrieve from a keyed discrimination has the same two step structure as the implementation of Member? for a discrimination: first apply a function to obtain a bucket; then perform the appropriate operation on the bucket. In the case of a keyed discrimination, however, the appropriate bucket is obtained by applying the bucket function (which is the second half of the composed functions which implement the discrimination) to a given key, instead of applying the full discrimination function to an object which might be a member of the set. This plan is called **Keyed-discriminate+retrieve** and is used in the analysis of SYMBOL-TABLE-RETRIEVE.

For Set-add and Set-remove, the fact that a discrimination is further implemented as a keyed discrimination makes no difference. For example, SYMBOL-TABLE-ADD is analyzed in terms of the Discriminate+action+update plan described earlier.

Associative deletion (Expunge) from a keyed discrimination is implemented by a three step temporal plan, **Keyed-discriminate+expunge+update**, which is an extension of the Discriminate+action+update plan described earlier (see figure). Keyed-discriminate+expunge+update has the following three steps. (This is the plan used to understand SYMBOL-TABLE-DELETE.)

(i) First, the appropriate bucket is obtained by applying the bucket function of the keyed discrimination to the given key.

(ii) Then, just as in Discriminate+action+update, the action on the whole set reduces to a corresponding action on the bucket. The Action step here is an instance of Expunge.

(iii) The final Update step is similarly a Newvalue operation on the discrimination function so that all domain objects which used to map the old bucket, map to the new bucket. Furthermore, in the case of a keyed discrimination, only the bucket function needs to be updated; the key function stays unchanged.

The idea of keyed discrimination can be generalized to multiple key data bases in two ways. One approach is to have separate discrimination functions for each key function which map into a shared set of buckets. Associative retrieval on a pattern of keys is then implemented by intersecting

the appropriate buckets. Alternatively, the discrimination functions for different keys can be composed, so that each function maps to a bucket which is itself a set implemented as a discrimination on the next key. This is the basic idea underlying discrimination nets.

3.4 Directed Graphs

Directed graphs are one of the most common programming data structures. A **Digraph** in this thesis is a set of nodes and an edge relation. For example, a Lisp list may be thought of as a directed graph wherein the nodes are Lisp cells, the edge relation is Lisp-cdr, and Lisp-car is a function which attaches a label to each node. The nodes of a standard Lisp binary tree structure may also be viewed as a directed graph in which the edge relation is the union of the Lisp-car and Lisp-cdr relations between the nodes. This view is particularly appropriate for programs which splice objects in and out of lists or trees.

Barstow has recently developed a set of rules for generating many standard programming algorithms for operating on directed graphs in the general case. Some time in the future his rules should be incorporated into the present library. This section concentrates on the special case of acyclic graphs with a single root, i.e. trees, and furthermore on the linear case of trees, which are here called *threads*.

Fig. 3-4 shows some standard specializations of Digraph. **Tree** is a directed graph in which there is a root and no cycles.[1] A **Bintree** is a a tree in which each node is either a terminal or it has exactly two successors. A **Thread** is a specialization of Tree in which the successor of each node is unique. This also means that the predecessor of each node in a thread (if it exists) and the terminal node are unique.

The vocabulary of partial orders is often applied to trees and threads. For example, it is common to think of a nodes in a tree or thread being "before" other nodes. This viewpoint is formalized by an overlay from Tree to Partial-order indicated in Fig. 3-4. A tree is viewed as a partial order in which two nodes are less than or equal iff they are successor* (the transitive closure of the successor relation) in the tree or are the same node. The root of the tree in this view becomes the minimum element of the partial order. Furthermore, if the tree is a thread, then the partial order is total.

---

1. Notice that this definition of tree does not constrain a node to have a unique precedecessor, i.e. there can be sharing of substructure in the tree. In later versions of the library it will be necessary to distinguish between acyclic rooted directed graphs in which nodes do and do not have unique predecessors.

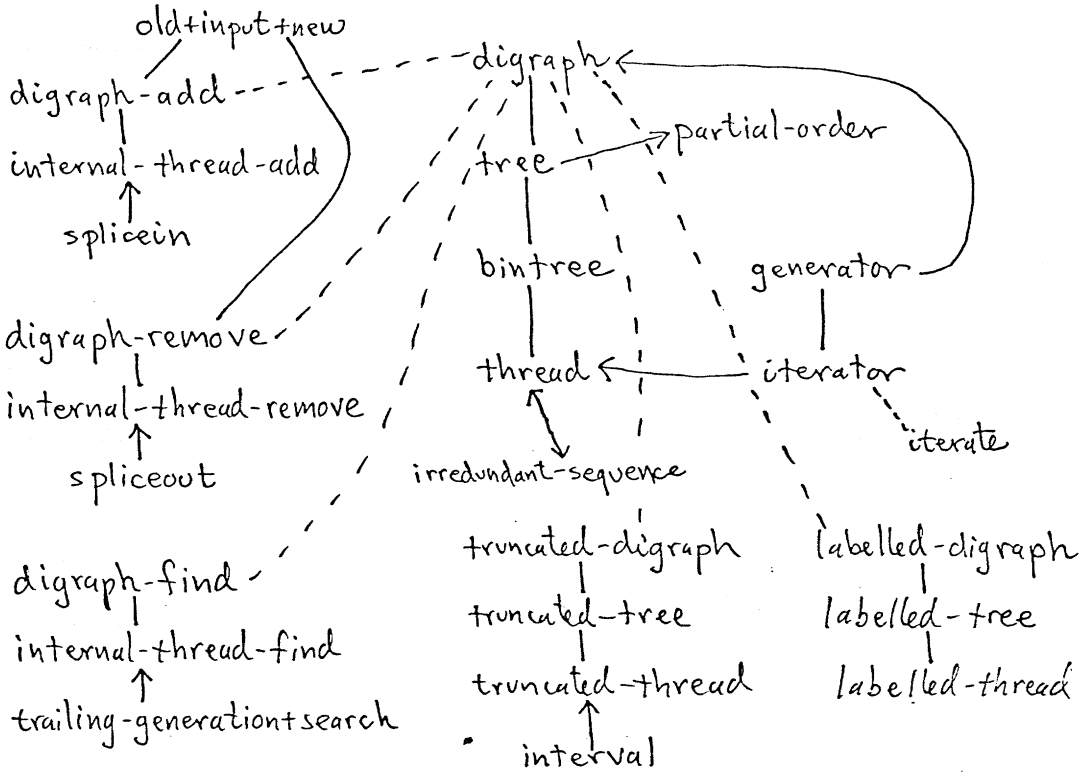


Figure 3-4. Plans Involving Directed Graphs.

Fig. 3-4 also shows an overlay between Irredundant-Sequence and Sequence. An irredundant sequence can be viewed as a thread in which the first term of the sequence corresponds to the root of the thread and any two consecutively numbered terms in the sequence are successors in the thread. Notice also that this overlay is one-to-one, which means that for each instance of Thread there is a unique corresponding instance of Irredundant-Sequence, and vice versa. This gives the user complete flexibility to use both the standard vocabulary of sequences (such as length and the idea of the *n*th element) and of directed graphs (such as the idea of successors) in specifying properties and relationships between objects.

Generators

One of the most common ways of implementing directed graphs in programming is to specify a single node (called the "seed") and a binary relation such that the nodes of the desired graph are the transitive closure of the given node under the given relation. This implementation is captured by the data plan **Generator.**

Iterator is the specialization of Generator which generates threads. This constrains the binary relation of an iterator to be many-to-one and have has no cycles within the transitive closure of the seed. This data plan is involved in the analysis of the part of loops which generate, such counting or CDRing down a list. The effect of the generating part of loops is also abstracted further in terms of the input-output specification **Iterate,** which takes an iterator as input and outputs the sequence of generated nodes. Loop plans and temporal abstraction will be discussed further in the next section.

Truncated Directed Graphs

Another common way of specifying a directed graph is as part of another directed graph. This is particularly used for specifying finite parts of infinite graphs such as intervals of the natural numbers.

The most general plan for describing this technique is **Truncated-digraph.** This data plan has two roles: the Base graph and a predicate called the Criterion. The criterion must divide the nodes of the base graph into three sets: a set of boundary nodes which satisfy the criterion; interior nodes, from which boundary nodes can be reached (in a finite number of successor steps); and exterior nodes, which can be reached from boundary nodes. In the case when the base graph is a thread (**Truncated-thread**), this means more simply that some node of the thread (either the root or a finite successor of the root) satisfies the criterion. Each such criterion thus determines a finite subgraph of interior nodes, either including or not including the boundary nodes.

Examples of truncated directed graphs in Lisp programming are CDR threads truncated by NULL and CAR-CDR binary trees truncated by ATOM.

A closely related way of specifying truncated threads, is in terms of upper and lower bounds on some total order. This is called an **Interval.** Thus for example, the integers from 10 to 100 are specified as an instance of Interval in which the the total order is numerical I.e, the lower bound is 10, and the upper bound is 100.

Splicing Plans

Thinking in terms of directed graphs is particularly appropriate for understanding programs which add or remove nodes in the middle of lists or trees. This section introduces a number of plans related to adding or removing internal nodes of threads in particular. These plans are used for example in analyzing the SYMBOL-TABLE-DELETE.

At the left of Fig. 3-4 are some basic input-output specifications on directed graphs which are involved in understanding splicing plans. **Digraph-add** is the basic specification for adding a node to a directed graph. It takes an old graph and a node as inputs and gives a new graph as output. All that can be said at this level of abstraction is that the input is a node of the new graph, and that all the successor relationships in the graph not involving the added node remain unchanged. Digraph-add does not specify where in the directed graph the node is to be added. **Internal-thread-add** is a specialization of Digraph-add in which the old and new graphs are threads and the new node is added anywhere but at the root.

The basic input-output specification for removing a node from a directed graph is **Digraph-remove.** Like Digraph-add, it takes an old graph and a node as input, and returns a new graph. All the successor relationships in the directed graph not involving the removed node remain unchanged. The successors of the removed node in the old graph become the successors of the predecessor of the removed node in the new graph. **Internal-thread-remove** is the specialization of Digraph-remove in which the old and new graphs are threads and the node to be removed is not the root.

Programs which splice nodes in or out of a thread typically have two steps. The first step is to find the place in the thread where the addition or removal is to occur. The output of this step is usually a pair of successor nodes, such that either the new node is to be added between them or the second node is the one to be removed. If the thread is implemented as a iterator, the second step is then to modify the generating function so as to either splice in or splice out a node, as the case may be.

The input-output specifications of the first step, finding internal nodes, which is shared between add and remove programs, are called **Internal-thread-find.** Given a thread and a criterion, Internal-thread-find returns a node of the thread (other than the root) which satisfies the criterion, and its predecessor. The typical implementation of this specification is to use a search loop which

keeps track of both the current and the immediately preceding node. This loop pattern is captured by the recursive temporal plan Trailing-generation+search plan, which will be discussed further in the next section.

The second step of implementing removal of a node is a Newarg operation in which the association between the predecessor of the node to be removed and the node is modified to be an association between the predecessor and the successor of the node to be removed. So for example in BUCKET-DELETE the node to be removed is in P and its predecessor is in Q; the generating function is CDR. The code for splicing out in BUCKET-DELETE is as follows.[1]

```
(RPLACD Q (CDR P))
```

The plan for this form of code in general is called **Splicecout.**

The second step of implementing addition of a node requires two Newarg operations: one to make the new node point to its successor, and one to make what was the predecessor of that node point instead to the new node. For example, addition of a node to a Lisp list iterator might be coded as follows.

```
(RPLACD NEW CURRENT)
(RPLACD PREVIOUS NEW)
```

The plan for this form of code in general is called **Splicein.**

Finally in Fig. 3-4, **Labelled-digraph** is a data plan with two roles: Spine and Label. Spine is a directed graph and Label is a function on the nodes of that graph. An important specialization is **Labelled-thread,** in which the spine is further constrained to be a thread. This plan captures the idea of viewing a Lisp list as a CDR thread with objects attached at each node by CAR. As demonstrated in this section, this view is particulary natural for understanding programs which splice in and out of lists.

---

1. RPLACD is modelled as Newarg, where the first argument to RPLACD is the domain element and the second argument is the new range element.

3.5 Recursive Plans

As in many other formalisms, the plan calculus uses recursive definitions to represent unbounded structures. A recursive plan is one in which one or more roles are constrained to be instances of the plan itself. This section will discuss only the special case of singly recursive plans, since the plans and overlays for doubly and mutiply recursive structures tend to be long and more detailed than those for singly recursive structures, without introducing any fundamentally new ideas.

At the top of the hierarchy of recursive plans in Fig. 3-5 is a minimal plan, **Single-recursion**, which says nothing more than that there is a role, Tail, constrained to be either an instance of Nil or itself a Single-recursion. **Nil** is a distinguished object used to terminate singly recursive structures.

The most basic singly recursive data plan, List, will be discussed first in the following section. The most basic singly recursive temporal plans, loops, will be discussed in the section following that. Finally, *temporal abstraction* will be introduced as a point of view which links singly recursive temporal plans with singly reucrsive data plans. Chapter Nine treats loops and temporal abstraction in much more detail.

Lists

List is a recursive data plan with two roles, Head and Tail. The head may be any object, but the tail must be an instance of List or Nil. It is important not to think of this data plan too concretely. The List plan is trying to capture what all recursive views of data structures have in common. List is the point of view which is used for making (linear) inductive arguments about data structures. Thus the reader should not identify the data plan List too closely with, for example, the Lisp list. Think of the data plan List as if it were called "singly recursive data structure".

Two basic input-output specifications on lists are shown at the top left of Fig. 3-5. **Push** takes as input a list (or Nil) and an object, and returns a new list, whose tail is the input list and whose head is the input object. **Pop** takes a list and returns its head and tail as its two outputs.

A common implementation of lists is using a sequence (e.g. an array) with a index to where the current head is stored. The data plan which captures this implementation is called **Upper-segment**. This plan is a specialization of **Segment**, which has three roles: the Base, which is a sequence, and the Upper and Lower bounds, which must be valid indices for the base. Upper-segment is the specialization in which the upper bound is equal to the length of the base sequence. Push and Pop operations on this implementation are implemented by the two-step temporal plans, **Bump+update** and **Fetch+update**, respectively. The second step in each of these plans is either to add

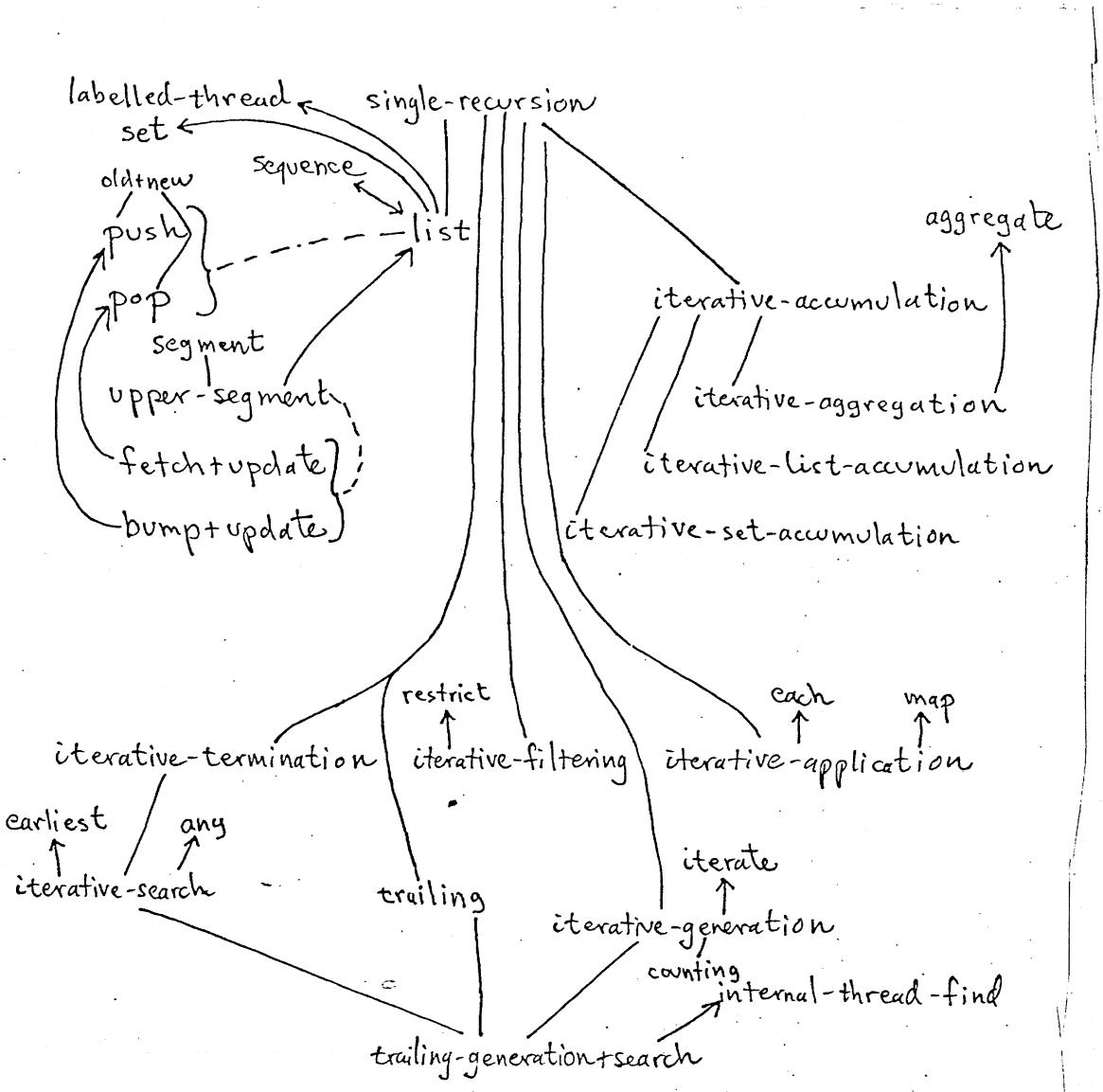


Figure 3-5. Recursive Plans.

or substract one from the old lower bound to get a new lower bound.[1] The first step in implementation of Push is a Newterm operation, which makes the given object the head of the new list. The first step in the implementation of Pop is a Term operation, which fetches the current head of the list.

Multiple Views of Linear Structures

Fig. 3-5 also indicates overlays between lists and other linear structures, such as sequences and threads. For example, whether a given data structure is viewed as a list or as a sequence depends on what we want to say about it. Certain properties are easier to specify inductively, in which case the list view is appropriate. In other cases, explicit quantification over the indices of a sequence is more conveninet. In the overlay between List and Sequence, the head of the list corresponds to the first term of the sequence, and the head of the nth tail of the list corresponds to the $N+1$th term of the sequence.

In the overlay between List and Labelled-thread, the nodes of the spine of the thread are the list and all of its tails. The edge function on the nodes of the spine is the Tail function, and the label function is Head. Thus we now have two ways of viewing Lisp cells which have Lisp cells or NIL as their CDR. We can view such a Lisp cell as implementing a list in which the CAR of the cell is its head and the CDR is its tail; or we can view the same Lisp cell as the seed for generating a CDR thread which is labelled by CAR.

Linear structures may also be viewed as (i.e. implement) sets. In particular, a list may be viewed as the set whose members are the head of the list unioned with the tail of the list viewed as a set. Nil is usually viewed as the empty set. In this view, neither the order of occurence of elements of the list nor the occurence of duplicates matters. Under this view, Push and Pop operations on a list are one implementation of Set-add and Set-remove operations on the set. Taking the view of lists as labelled threads, Splicein and Spliceout also implement Set-add and Set-remove. Both of these points of view are needed to understand how entries and added and removed from the bucket sets in the symbol table example. In SYMBOL-TABLE-ADD, entries are added to the bucket a Push operation (CONS); in BUCKET-DELETE, entries are removed by a Spliceout plan (RPLACD).

---

1. Again, at this level of abstraction no commitment is made in these plans as to whether the instance of Upper-segment is modified by side effect or copied. These are treated as specializations, just as the "pure" and "impure" versions of Push and Pop are treated as specializations.

Loops

The taxonomy of loop structures used in this thesis is based on Waters' [60] method for analyzing loop programs. Waters' method decomposes loops into fragments which correspond to "easily understood sterotyped fragments of looping behavior." The next section also introduces a point of view under which these fragments are logically composed, rather than interleaved (as they are in the raw loop), which makes their net effect easier to understand. For example, consider the following program, which sums up the non-nil elements of a list.

```
(DEFINE SIGMA
   (LAMBDA (L)
      (PROG (S N)
            (SETQ S 0)
         LP (COND ((NULL L)(RETURN S)))
            (COND ((SETQ N (CAR L))
                   (SETQ S (PLUS S N))))
            (SETQ L (CDR L))
            (GO LP))))
```

Waters distinguishes three types of fragments (he calls them *plan building methods*) in loops with one exit test. The first type he calls "basic loops". A basic loop is characterized by the fact that all of the computation in the body of the loop can potentially affect the termination of the loop. For example, the basic loop part of SIGMA is the following.

```
(LAMBDA (L)
   (PROG (...)
         ...
      LP (COND ((NULL L)...))
         ...
         (SETQ L (CDR L))
         (GO LP)))
```

In this thesis, basic loops are further decomposed into a generation part (e.g. the part involving CDR above) and a termination part (e.g. the NULL test above). The temporal plan which captures the form of the generating part of loops in general is called **Iterative-generation.** The plan which captures the form of single exit tests is called **Iterative-termination.** Both of these are are extensions of Single-recursion (see Fig. 3-5). The advantage of this further decomposition is it allows us to capture the similarity between loops which have the same generation part but different terminations. For example, the generation part of many loops is **Counting** (a specialization of of Iterative-generation in which the generating function is Oneplus) although they may have different terminations.

Waters' second category of plan building method is called "augmentations". Augmentations are characterized by the fact that they consume values produced by other parts of the loop and produce values which may be used by other augmentations. In this thesis, augmentations are further divided into *application* and *accumulation.* The distinction between these two types of augmentations rests on whether there is any "feedback", i.e. whether the augmentation consumes its

own values from previous iterations -- accumulation does, application does not. For example, the following is the application part of SIGMA.

```
(PROG (...N)
      ...
   LP ...
      ...(SETQ N (CAR L))...
      ...
      (GO LP))
```

The plan for this form of code in general is called **Iterative-application.** SIGMA also has an example of accumulation, as shown below.

```
(PROG (S...)
      (SETQ S 0)
   LP ...(RETURN S)...
      ...(SETQ S (PLUS S ...))...
      ...
      (GO LP))
```

The plan for this form of code in general is called **Iterative-accumulation.** Three common specializations of Iterative-accumulation are shown in Fig. 3-5. **Iterative-set-accumulation** is the specialization in which the accumulation operation (e.g. PLUS above) is Set-add and the initial accumulation is the empty set. **Iterative-list-accumulation** is the specializtion in which the accumulation operation is Push and the initial accumulation is Nil. **Iterative-aggregation** is the specialization in which the accumulation operation is the application of an aggregative function (as discussed earlier in the section on functions) and the initial accumulation is the identity element for that function.

Waters' final type of plan building method is called "filtering". It is the special case of an augmentation whose body is a conditional. The purpose of filtering usually is to restrict the values that will be consumed by some other augmentation. For example, in SIGMA the following is the filtering part of the loop which restricts the accumulation part to consuming only the non-nil inputs.

```
(PROG (...N)
      ...
   LP ...
      (COND (...N...))
      ...
      (GO LP))
```

The plan for this form of code in general is called **Iterative-filtering.**

Finally, the **Trailing-generation+search** plan at the bottom of Fig. 3-5 illustrates an important feature of the taxonomy in this thesis, namely that it is a *tangled* hierarchy. Trailing-generation+search combines the features of three plans. One of these plans is Iterative-generation, an example of which is the following.

```
(PROG (P ...)
      ...
   LP (SETQ P (CDR P))
      ...
      (GO LP))
```

The second plan is **Iterative-search.** Iterative-search is a specialization of Iterative-termination wherein the exit test is the application of a predicate which doesn't change as the computation proceeds, and in which the final object which satisfied the exit test is available outside the loop. This plan is suggested by the following code.

```
(PROG (P ...)
      ...
   LP ...
      (COND (...P...
             ...P...
             (RETURN ...)))
      ...
      (GO LP))
```

The final plan is **Trailing**, which captures the idea of keeping track of the immediately previouls value of some loop variable, as suggested by the following code.

```
(PROG (P Q)
      ...
   LP (SETQ P ...)
      ...
      (SETQ Q P)
      (GO LP))
```

Tailing-generation+search inherits the roles and constraints of all three of these plans. For example, the combination[1] of the three example fragments above gives the essential loop structure of BUCKET-DELETE, as shown below.

```
(PROG (P Q)
      (SETQ Q BUCKET)
   LP (SETQ P (CDR Q))
      (COND ((EQUAL (CAAR P)) INPUT)
             (RPLACD Q (CDR P))          ;SPLICE OUT.
             (RETURN BUCKET))
      (SETQ Q P)
      (GO LP))
```

As mentioned earlier, this plan implements Internal-thread-find.

---

1. The code fragments above cannot literally be combined to get the loop of BUCKET-DELETE. The appropriate domain for this combination is the plan calculus.

Temporal Abstraction

The basic idea of temporal abstraction is to view all the objects which fill a given role in a recursive temporal plan as a single data structure. In programming language terms, this often corresponds to having an explicit representation for the sequence of values taken on by a particular variable at a particular point in a loop. This idea is also present in the work of both Waters [60] and Shrobe [56]. Chapter Ten, however, explains how this point of view can be formalized as overlays for the various loop plans described in the preceding section.

Fig. 3-5 shows some of these overlays. For example, Iterative-generation can be abstracted as Iterate. The input to Iterate in this overlay is an iterator whose seed is the initial value of the relevant loop variable (e.g. P above) and whose generating function is the function applied each time around the loop (e.g. CDR above). The output of Iterate corresponds to the sequence of values taken on by the loop variable.

The relationship between the sequences of values consumed and produced in an instance of Iterative-application can be similary viewed as a Map operation. In programs where order and occurence of duplicates in the loop values doesn't matter, a further temporal abstraction can be made by viewing the values consumed and produced as sets. In this view, Iterative-application implements Each.

Similarly, Iterative-search can be viewed as implementing either Earliest or Any, depending on whether the inputs over time to the exit tests are viewed as a sequence or a set. One temporal abstraction of Iterative-filtering is as Restrict.

Thus using temporal abstraction the recursively defined plan for a loop can be viewed much more simply as a simple composition of operations on sequences or sets.

CHAPTER FOUR

THE PLAN CALCULUS

4.1 Introduction

This chapter deals with the practical side of the plan calculus. (For a more formal treatment, see Chapter Eight.) Practically speaking, the plan calculus is a network-like representation. There are many well-known ways of storing such representations in a computer to facilitate various kinds of pattern matching and associative retrieval. Several different computer implementations of the plan calculus using fully inverted assertional data bases have already been implemented and successfully used by the author [47], Shrobe [56] and Waters [60]. The details of this level of implementation are therefore not going to be disucssed in this thesis. In this document we will use a *diagram* language to represent plans and overlays. The purpose of this chapter is to introduce these diagrams and specify their intuitive meaning.

The plan calculus is made up of two major components: plans and overlays. The first major section of this chapter discusses plan diagrams. At the end of this section, there is also some discussion of the relationship between plans and Lisp code. The second major section of this chapter discusses overlay diagrams. At the end of this section, there are some general observations on the use of overlays as a preview of the next three chapters.

Note that the issue of side effects and mutable objects will only be mentioned in passing in this chapter, since to give a more in depth treatment requires the logical foundations developed in Chapter Eight. Chapter Ten is devoted entirely to the topic of plans involving side effects.

4.2 Plans

The basic idea of a plan, as used in this thesis comes from an analogy between programming and other engineering activities. In most other engineering disciplines plans are very prominent. For example, electrical engineering has circuit diagrams and block diagrams at various levels of abstraction; structural engineering uses large-scale and detailed blue prints of the architectural framework of a building and also of various subsystems such as heating, wiring and plumbing; and in mechanical engineering there are overlapping hierarchical descriptions of the interconnections between mechanical parts and assemblies.

A fundamental characteristic shared by all these types of engineering plans is that at each level there is a set of *parts* with *constraints* between them. Sometimes these parts correspond to discrete physical components, such as transistors in a circuit diagram, but more often the decomposition is in terms of function. For example, a simple amplifier in an electrical block diagram has the functional description $V_2 = kV_1$, where V_1 and V_2 are the output and input signals of other blocks to which it is connected. As far as this level of plan is concerned the amplification may be realized in any number of ways. A primitive component may be used or another plan may be provided which decomposes the amplifier further.

By analogy, plans in programming specify the parts of a computation and constraints between them. The parts of a plan are called *roles*. It is natural to think of the roles of a plan as selector functions. For example, consider the Segment plan discussed in Chapter One, which has three roles named Base, Upper and Lower. We will write Segment.Base to refer to Base sequence, Segment.Upper to refer to the Upper index, and so on. The character, point ("."), in this notation is intended to have an intuitive meaning similar to the way it is used for record structures in programming languages such as PL/1. If a role is filled by an instance of another plan, the point notation can be used several times, as in Iterate.Input.Seed, meaning the Seed of the Input (which is an instance of Iterator, which has a role named Seed) of the Iterate operation. A nested expression like Iterate.Input.Seed is called a *path* name.

All plans are built up, using roles and constraints, out of three primitives: *input-output* specifications, *test* specifications and primitive *objects* (integers, sets and functions). Plans built up exclusively out of objects are called *data plans*. Plans involving objects, test and input-output specifications are called *temporal plans*.

Input-Output Specifications

An example of an input-output specification is shown at top of Fig. 4-1. An input-output specification is drawn as a solid rectangular box with solid arrows entering at the top and leaving the bottom. Each arrow entering at the top represents an input; each arrow leaving the bottom represents an output. Each input and output has name, which is used the same as a role name.[1]

For example, the input-output specification depicted in Fig. 4-1 is Newterm. It has three inputs, named Old, Arg and Input; and one output, named New. Note that when the input or output of an input-output specification is not connected to any other the inputs or outputs, e.g. when an input-output specification is drawn in isolation, the arrows are terminated in solid ovals. When this is the case, the input or output role name is written inside the oval rather than beside the arrow.

Input-output specification also have *preconditions* and *postconditions*. The preconditions involve only the inputs; the postconditions involve both the inputs and the outputs. The simplest kind of such conditions are restrictions on the type of each role individually. These are usually written directly on the plan diagrams, in parentheses after the role name. For example, in Fig. 4-1 we see that the Newterm.Old is expected to be a sequence; and that Newterm.Arg is expected to be a natural number. "Object" as a type restriction, as for Newterm.Input, means that there is no individual restriction on the given role.

Constraints between roles are written in a standard logical language, the details of which are explained in Chapter Eight. In this chapter and the following three, the relationships between the inputs and outputs of an input-output specification will be described informally in English, as they are relevant to the current discussion. The interested reader may also refer to the appendix for the formal preconditions and postconditions of any particular input-output specification (use index to find page number).

To reduce the clutter in more complicated plan diagrams later in this document, some of the information described above will omitted when it can easily be inferred by the reader. For example, type restrictions (especially "object") will often be omitted for input-output specifications which should be familiar by that point in the discussion. Role names will also sometimes be omitted, in which case the left-to-right order used when the specification was first defined (and which is listed in the appendix) is to be understood.

---

1. In this chapter input-output specifications are primitive. In the formal semantics, however, input-output specification specifications are in fact treated as composite plans whose parts are objects and situations. This is why the inputs and outputs are treated here as roles.

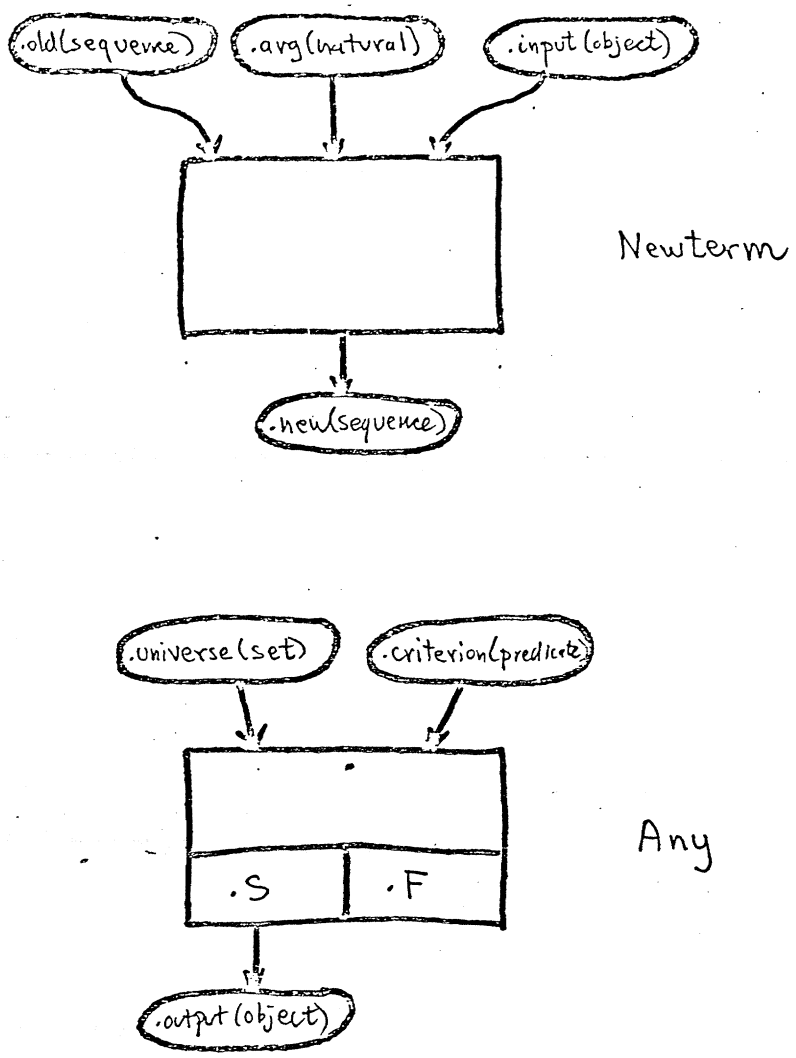


Figure 4-1. An Input-Output and a Test Specification.

Test Specifications

A test specification is drawn as a solid rectangular box with a divided bottom part, as shown in the lower part of Fig. 4-1. The inputs and outputs of a test specification are notated in the same way as the inputs and outputs of an input-output specification. For example, the test shown in Fig. 4-1, Any, has two inputs named Universe (a set) and Criterion (a predicate); and one output named Output (an object). A test also has preconditions and postconditions, just like an input-output specification.

A test specification differs from an input-output specification in that two distinct output situations are specified. Which one occurs depends on whether or not a given relation (called the *condition* of the test) holds true between the inputs. If the test condition is true, then the test is said to *succeed* and the outputs indicated on the "S" side of the box are available; otherwise the test is said to *fail*, and the outputs indicated on the "F" side of the box are available. For example, Any succeeds when there exists a member of Any.Universe which satisfies Any.Criterion, in which case Any.Output is such an object; otherwise it fails and there are no output roles.[1]

As in the case of preconditions and postconditions, test conditions are specified formally in a logical language in the appendix, but will will be described informally in English in the body of the following chapters.

More complicated tests with more than two cases can be represented by composing the binary tests described above. Alternatively, the test notation may be generalized to more than two cases.

Control Flow

Fig. 4-2 shows how control flow arcs (hatched arrows) are used to connect input-output and test specifications to specify conditional behavior. This plan, called **Cond**, is the basic "if-then-else" construct in the plan calculus. The If role is restricted to be an instance of **Test**, which is the minimal test specification, i.e. all other test specifications are extensions of it. The Then and Else roles of Cond are restricted to be instances of **In+Out**, which is the minimal input-output specification.

The End role of Cond introduces the use of a third primitive in the same category as input-output and test specification, namely *join* specifications.[2] Joins are like the mirror images of tests. A join specification is drawn as a solid rectangular box with the top part divided in "S" and "F", corresponding to the succeed and fail cases of the matching test. Unlike tests, however, joins do not

---

1. Note that at this level of abstraction, no commitment is made as to whether or not this test modifies its inputs. This property will be specified when necessary in the constraints of plans of which this test is a part.
2. Joins were first introduced into the plan calculus by Waters [60].

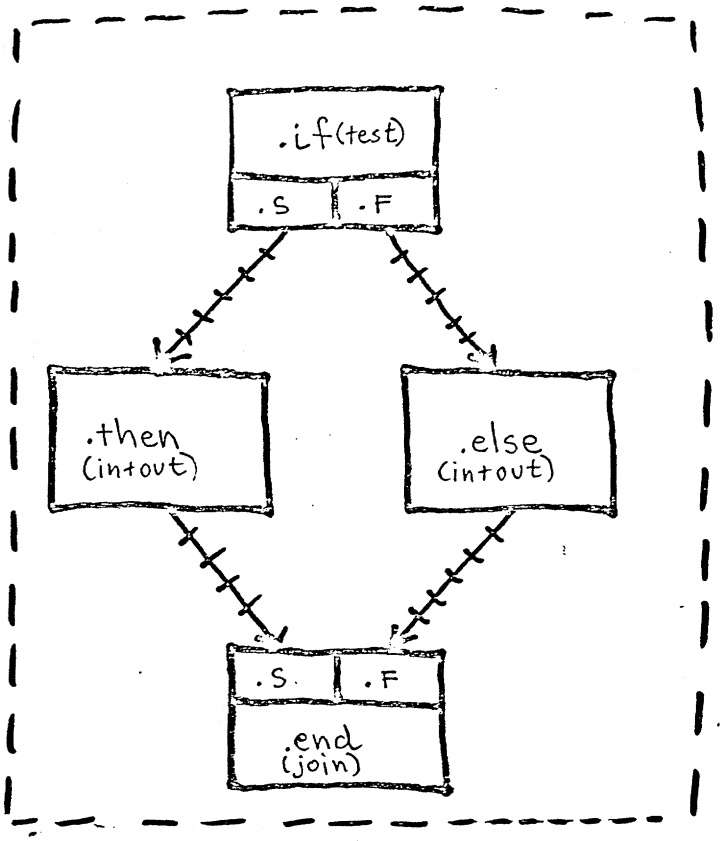


Figure 4-2. A Conditional Plan.

represent any real computation. Joins are a technical artifact used to rejoin the two "wings" of a conditional block, as in Cond. **Join** is the minimal join specification. An extension of Join, called **Join-output**, is used to specify the connection between which way a test goes and which of two possible outputs is made available for further computation, as in the following fragment of code.

```
(SETQ C (COND ((...) A)
              (T B)))
```

Data Flow

The basic idea of data flow is to specify equality between roles in a temporal plan, especially between the output of an input-output specification or test and the input of another one. Data flow is indicated in plan diagrams by solid arrows, as shown in Fig. 4-3.

Fig. 4-3 defines a plan with two roles, Discriminate and If. The Discriminate role is restricted to be an instance of @Discrimination. @Discrimination is a specialization of @Function in which the Op input is restricted to be a disrimination function, and in which the Output is therefore a set. The If role is restricted to be an instance of Member?, which tests whether the Input is a member of the Universe set. The data flow arc between Discriminate.Output and If.Universe means that the output of the Discriminate operation is the same as the Universe set of the If test.

Note however, that the data flow arc in Fig. 4-3 does *not* mean that the If test must immediately follow the Discriminate operation. An arbitrary amount of computation may occur between the end of the Discriminate operation and the beginning of the If test, as long as the set involved is the same at the time the If test begins as when the Discrminate operation ended.

Temporal Plans

Fig. 4-3 is an example of a temporal plan. Such plans in general have data flow and control flow arcs between input-output, test and join specifications, and are drawn with a dashed box enclosing the entire plan definition. This section describes a very natural way of interpreting the meaning of such diagrams in terms of the propagation of data and control tokens through an acyclic[1] directed graph according to a specified set of rules. This interpretation is essentially the one used in data flow schemas [12].

From this standpoint, control flow arcs are treated no differently than data flow arcs. When an input-output box has received tokens on all of its incoming arcs, it is "activated" and generates tokens with the appropriate properties (according to its input-output specifications) on all of its

---

1. Loops are represented as tail recursions.

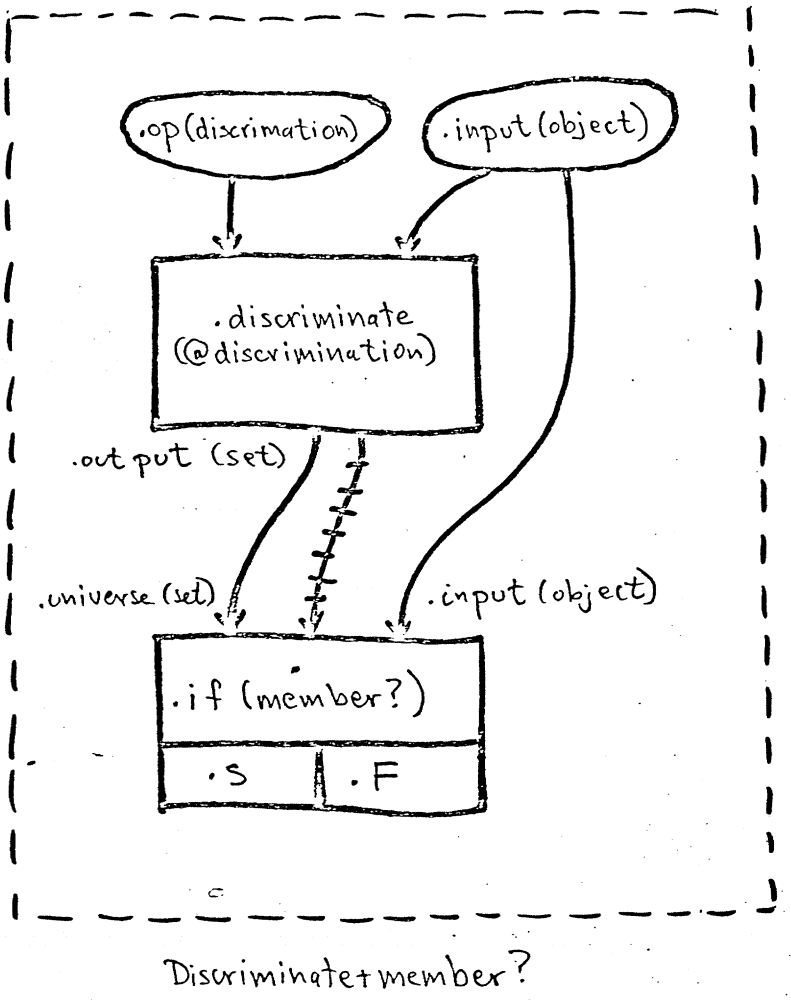


Figure 4-3. A Plan With Data Flow.

outgoing arcs.[2] If an output goes to the inputs of several other boxes (i.e. an arc splits along its way into two or more arcs), then tokens passing over that arc are duplicated the appropriate number of times so that the same object may be available at each input location. Control flow tokens have no properties; their only function is to enable activitation.

Test and join boxes also have activation rules. A test box is activated the same way as an input-output box, i.e. when it has received tokens on all of its incoming arcs. It then generates tokens either on all of the arcs leaving the success side of the box, or on all those leaving the failure side, depending on the properties of the incoming objects. A join has the complementary behavior. It is not activated until it has received all the tokens on one or the other input side. It then generates all its output tokens with properties according to its specifications (since joins involve no computation, the output tokens are always the identical to the input tokens).

Data Plans

Data plans are plans whose roles are restricted to primitive data objects or other data plans. Data plans are drawn as dashed ovals. Primitive data objects are drawn as solid ovals. For example, the data plan **Segment**, shown in has three roles named Base, Upper and Lower, restricted to be a sequence and two natural numbers, respectively. The constraints between roles specify the both the Upper and Lower numbers are less than or equal to the length of the Base sequence, and that the Lower number is less than or equal to the Upper number. (Again, these constraints are written formally in a logical language, the details of which are being suppressed until Chapter Eight.)

Recursive Plans

Recursion in plan diagrams is indicated by a spiral line as shown in Fig. 4-5. The minimal singly recursive plan is called Single-recursion. It has only one role, Tail, which is constrained to be an instance of itself. All other singly recursive plans are extensions of Single-recursion.

---

2. Thus input-output specifications require termination.

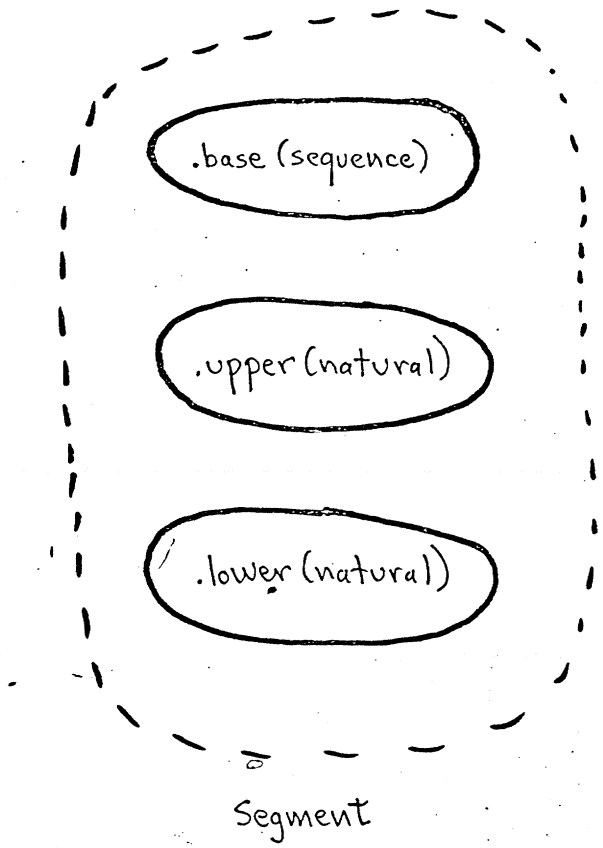


Figure 4-4. A Data Plan.

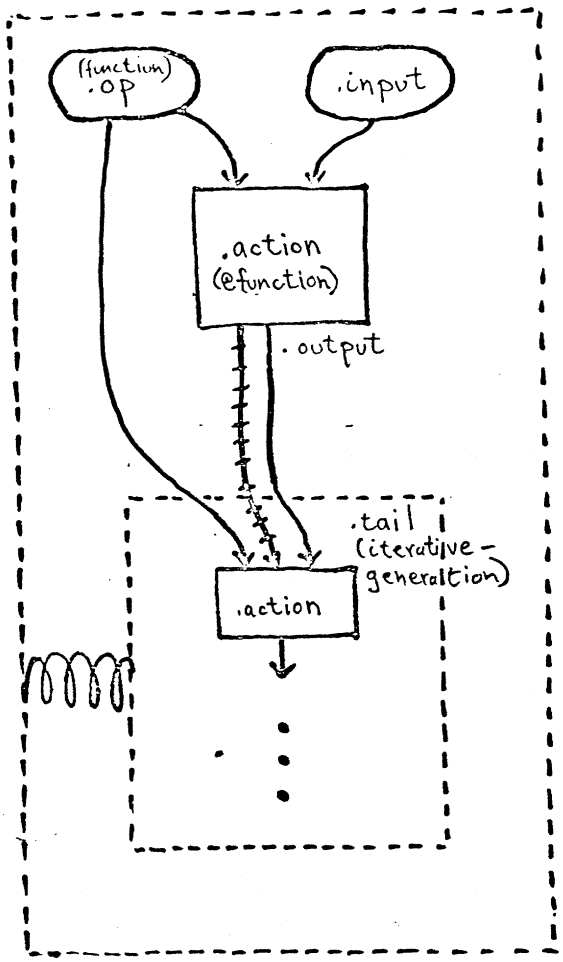


Figure 4-5. A Recursive Plan.

4.3 Surface Plans

In most common programming languages, such as Lisp, Fortran or PL/1, it is possible to construct many different programs which, from the point of view of this thesis, specify the same computations. Difference in the names of variables is the most trivial example of this kind of uninteresting variability. Most programming languages also provide many different mechanisms for achieving the flow of data from one operation to another. For example, in Lisp we could write either

```
(SETQ X (F ...))
...
(G X)
```

or

```
(G (F ...))  .
```

Similarly, the following two constructions specify essentially the same control flow.

```
(PROG (...)
      ...
   LP (COND (P (RETURN NIL)))
      ...
      (GO LP))

(PROG (...)
      ...
   LP (COND (P)
            (T ...
               (GO LP))))
```

Finally, compare the following two codings of BUCKET-RETRIEVE (the first is from the scenario), which differ in all three of the superficial ways mentioned above.

```
(DEFINE BUCKET-RETRIEVE
   (LAMBDA (BUCKET INPUT)
      (PROG (OUTPUT)
         LP (COND ((NULL BUCKET)(RETURN NIL)))
            (SETQ OUTPUT (CAR BUCKET))
            (COND ((EQUAL (CAR OUTPUT) INPUT)
                   (RETURN OUTPUT)))
            (SETQ BUCKET (CDR BUCKET))
            (GO LP))))

(DEFINE BUCKET-RETRIEVE
   (LAMBDA (B I)
      (PROG (O)
         LP (COND ((NULL B))
                  ((EQUAL (SETQ O (CAR O)) I)
                   (RETURN O))
                  (T (SETQ B (CDR B))
                     (GO LP))))))
```

The importance of surface plans is that both of these versions translate to the same surface plan (which will be shown in the next chapter). From the standpoint of analysis, a surface plan can be thought of as an abstraction of the data flow and control flow in a program, without abstracting the primitive data structures and operations. From the standpoint of synthesis, this amounts to the lowest level plan before translating the result of synthesis into code in a standard programming language.

Programming Language Semantics

In order to translate from a given programming language to surface plans, the primitives of the programming language are divided into two categories: connectives, such as PROG, COND, SETQ, GO and RETURN in Lisp, which are concerned soley with implementing data and control flow; and the objects, relations, and actions of the language, such as numbers, dotted pairs, arithmetic relations, CAR, CDR and CONS. The first category of primitives is translated into the pattern of control and data flow relationships (including tests and joins) between other specifications defined in terms of the second category of primitives.

The translation of the second (non-connectives) category of primitives into the plan calculus is done in three steps, each of which involves some discretion. The first step is to identify the set of basic object types in the language. In this thesis, a basic Lisp is used in which there are four types of objects: atoms, dotted pairs, vectors, and integers.[1] A fifth type, Lisp datum, is the union of these four.

The next step is to choose an appropriate set of basic relationships between objects. These relationships are not the same as the primitive actions of the programming language, but are the vocabulary in terms of which these computations will be specified. For example, in this thesis I use two primitive functions relations on dotted pairs, Car and Cdr, with functionalities as shown below.

Cdr: dotted-pair → datum
Car: dotted-pair → datum

This approach separates the notion of Car as a relationship between two objects (existing at a given time), from a computation which has as input a dotted pair and as output the object which is in the Car relationship to it.

The final step in translating from Lisp to surface plans is to translate code primitives such as (CAR ...) into input-output specifications in terms of the primitive relations Car and Cdr, and similarly for CDR, CONS, RPLACA and RPLACD. CONS becomes a specification which takes as input two

---

1. This is the mathematical notion of an integer. The distinction between this and the fixed width computer representation of an integer in Lisp is not made here, because I have codified no knowledge having to do with this distinction.

Lisp data, and returns as output a dotted pair such that Car holds between it and the first input, and Cdr holds between it and the second input. RPLACA and RPLACD specify modification of the Car or Cdr relation to make the relation hold between a given dotted pair and object.

Two primitive relations on Lisp data are shown below.

Null: datum → boolean
Eq: datum × datum → boolean

Again, the distinction is made here between the relation and a computation which tests whether that relation holds for a given tuple of objects. Thus code such as the following constructions with COND is translated into the plan calculus as tests involving these relations respectively.

```
(COND ((NULL ...) ...))
(COND ((EQ ...) ...))
```

The final two primitives used to express the semantics of Lisp in the plan calculus concern Lisp vectors (one dimensional arrays).

Dim: vector → integer
Element: vector × integer → datum

The input-output specifications of the vector creation (ARRAY) and accessing (ARRAYFETCH and ARRAYSTORE) primitives of Lisp are then written in terms of these.

4.4 Overlays

An overlay is formally a triple made up of two plans and a set of correspondences between roles of the two plans. An overlay can also be thought of as a projection from the set of computations (or data structures) specified by one plan to the set specified by the other, especially if one of the two plans is a primitive object type.

For example, the following overlay,[1]

Composed>function: composed-functions → function

is a projection from instances of Composed-functions to instances of Function. **Composed-functions** is a data plan whose two roles, named One and Two are functions, with the constraint that the range of function One is a subset of the domain of function Two. Given an instance of Composed-functions, the definition of Composed>function (which is written out formally in the appendix)

1. The character ">" is intended to be read as "as".

specifies how to view it as a single function from the domain of function One to the range of function Two. Clearly, this overlay is a many-to-one projection. Other overlays, such as between List and Sequence, are one-to-one, which amounts to an isomorphism between the two sets of computations.

An additional important property of overlays is that each overlay and its inverse function must be *total* on the specified domain and range. For the use of overlays in analysis, it is important that, given an instance of the domain type, there exist a corresponding instance of the range type. For example, for the overlay Composed>function, if we recognize an instance of Composed-functions, it is important to know that there exists an corresponding instance of Function which it implements. Conversely, for synthesis it is important to know that for every instance of the range type of an overlay, there exists an instance of the domain type which is a valid implementation of it.

Fig. 4-6 shows the kind of diagram which is used to represent an overlay between two composite plans. An overlay diagram is divided in two halves by a line down the middle. The left side shows the plan diagram for the domain of the overlay; the right hand side shows the plan diagram for the range. Correspondences are drawn as lines with hooks on the ends which connect roles on one side with roles on the other.

The domain of the overlay in Fig. 4-6 is **Composed-@functions**, which has three roles: One and Two are instances of @Function, and Composite is an instance of Composed-functions. Data flow constraints in the Composed-@functions plan are such that the functions Composite.One and Composite.Two become the inputs One.Op and Two.Op, respectively; and One.Output becomes Two.Input. The range of the overlay is @Function. Basically, this overlay expresses how to view the composed application of two compatible functions as the application of a composed function.

Correspondences in overlay diagrams are either labelled or unlabelled. Unlabelled correspondences represent equality between the indicated roles; labelled correspondences represent equality between the value of labelled function applied to the role on the left and the role on the right. The function involved in such correspondences is very often another overlay.

For example, there are three correspondences in Fig. 4-6. The topmost correspondence says that the instance of Composed-functions on the left side which is the Composite role of Composed-@functions, viewed as a function according to the overlay Composed>function, is equal to the Op role of @Function on the right. Thus Composed>function encapsulates a chunk of implementation knowledge which then can be employed to define larger chunks. The next overlay we will discuss in this section uses Composed>function twice.

The other two correspondences in Fig. 4-6 are simple equalities. The first one means that for an instance of Composed-@functions and an instance of @Function related as Composed>@function, the object filling Composed-@functions.One.Input is equal to the object filling @Function.Input. Similary Composed-@functions.Two.Output corresponds to @Function.Output

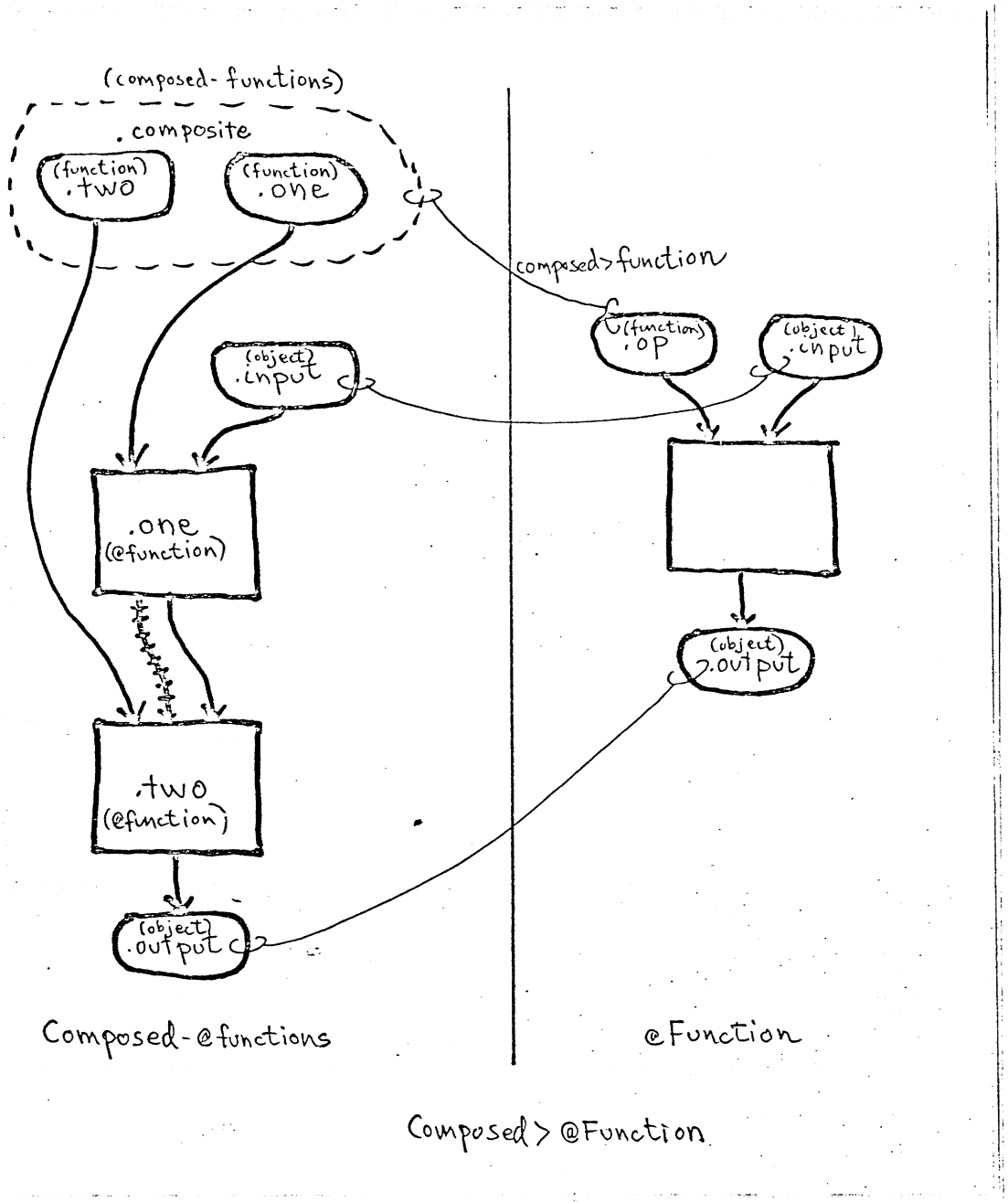


Figure 4-6. Applying a Functional Composition.

Note that in the formal definition of Composed>@Function (see appendix) there are two more correspondences. The input situation of Composed-@functions.One on the left is identified with the input situation of @Function on the right; and the output situation of Composed-@functions.Two on the left is identified with the output situation of @Function on the right. To avoid clutter, correspondences between input and output situations will usually omitted in overlay diagrams.

We can now move more quickly through the another overlay, involving composed functions. **Newvalue-composite>newvalue**, shown in Fig. 4-7, captures the idea that, given a function implemented as a composition, a Newvalue operation on the component Two of the composition can be viewed as a Newvalue operation on the implemented function. This overlay is used in the description of the symbol table program introduced in Chapter Two. The hash table in that example is viewed as a function implemented as the composition of two functions: a numerical hash which doesn't change, and an array which is viewed as a sequence that is modified to insert new entries.

Notice the overlapping of plans on the left hand side in Fig. 4-7. This style of building up larger plans by making use of instances of already defined plans and constraining certain components to correspond, allows us to be very concise. More important, we have separated what is novel about a particular plan, like Newvalue-composite, from what it has in common with other plans. Similarly for overlays, it is significant that Newvalue-composite>newvalue makes use of Composed>function rather than restating the same knowledge several times.

A Standard Implementation

As a second introductory example of overlays, I have chosen the implementation of lists using an array and an index. This particular implementation is not used in the symbol table program, but is included here because it is a standard and familiar example for many other papers on representing programming knowledge.

We begin with the idea of viewing a segment of a sequence between two bounds as a sequence.[1] This is formalized as the overlay **Segment>sequence**, which says (see appendix) that the terms of the implemented sequence correspond to the terms of the base sequence, offset by the lower bound.[2]

---

1. We are skipping the step of modelling an array as a sequence, which is part of the surface plan translation.
2. This implementation "wastes" the first and and last terms of the base sequence. It can be improved by adding Oneplus and Oneminus in various places, but this would just make the example more complicated without adding any new ideas.

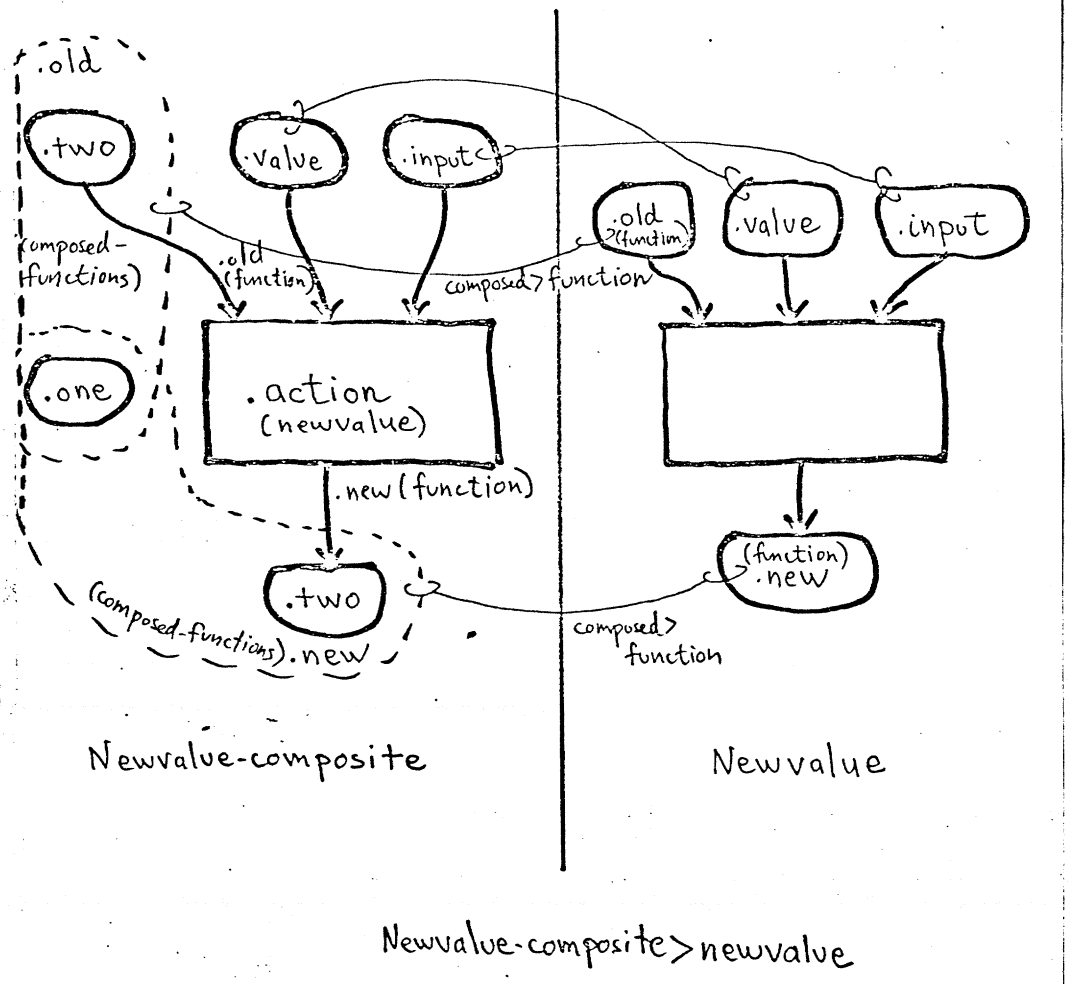


Figure 4-7. Implementing Newvalue for Composed Functions.

A specialization of Segment is Upper-segment, in which the upper bound is the length of the base sequence. This is a data plan often used to implement a list. The head of the implemented list corresponds to the term of the base sequence indexed by the lower bound, and the tail of the list is recursively defined as the list implemented by the upper segment which has the same base sequence with one plus the lower bound. Nil is implemented by a segment in which the lower bound meets the upper bound, i.e. when the lower bound is the length of the sequence. This implementation is described formally by the overlay **Upper-segment>list**, in the appendix.

Fig. 4-8 defines the overlay **Bump+update>push**, which shows how to implement a Push operation on a list implemented by Upper-segment>list. The plan on the left hand side, Bump+update, has four roles: Bump, an instance of **@Oneminus** (the specialization of @Function when the Op is Oneminus); Update, an instance of Newterm; and Old and New, instances of Upper-segment. The essence of the plan is that a new term is updated at one minus the lower bound. Bump+update>push specifies how this plan can be viewed as a Push operation if the Old input to Update together with the input to Bump are viewed as the Old input of Push (implemented by Upper-segment>list), if the Input of Update corresponds to Input of Push, and if the New output of Update together with the output of Bump are viewed as the New output of Push (also implemented by Upper-segment>list).

Similarly, Fig. 4-9 defines the overlay **Fetch+Update>pop**, which specifies how to implement a Pop operation on a list implemented by Upper-segment>list. Here we see that the base sequences of the old and new upper segments are the same. One is added to the lower bound. The Output of Fetch corresponds to the Output of Pop. The Fetch and Bump operations may occur in any order since neither uses the output of the other.

Using Overlays

We will see many more examples of overlays in this and the following chapters. In Chapters Five and Six we will also see how overlays are used in analysis and synthesis. For now I would like to make make just a few general introductory remarks on using overlays.

We have already seen that overlays are an important tool for codifying programming knowledge. An overlay can encapsulate a chunk of implementation knowledge so that it may be used many times in building up larger chunks. Such overlays express a generalization of many specific implementation strategies.

In analysis and synthesis scenarios, overlays are invoked by pattern matching against one side of the overlay and instantiating the other. For example, suppose we are in the midst of synthesizing a program and at some point we have a plan involving an instance of Push. One thing we could do is search in the plan library for an overlay which has Push on one side, for example Bump+update>push, and instantiate the other side, in this case Bump+update. The are obviously

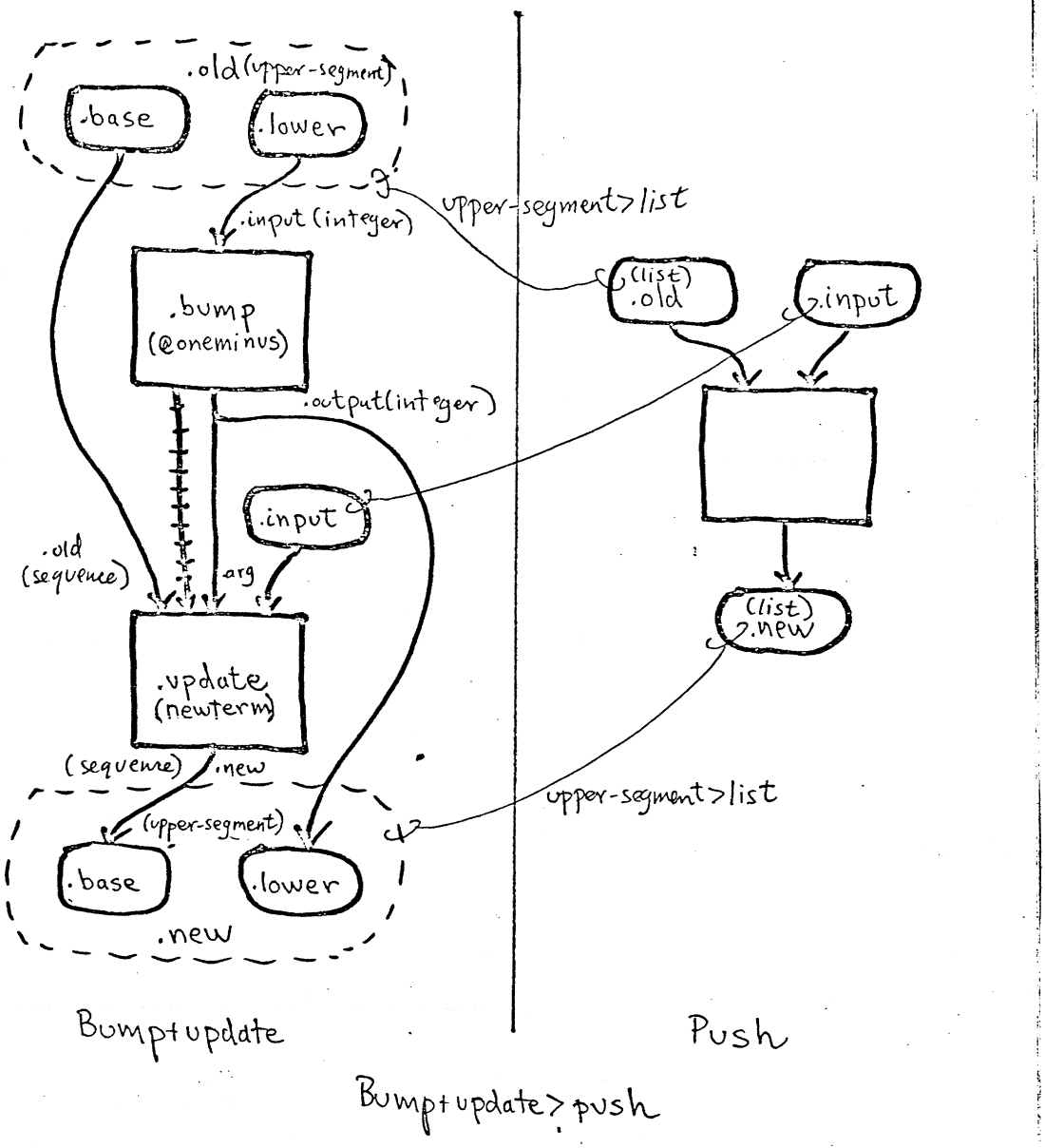


Figure 4-8. Implementation of Push.

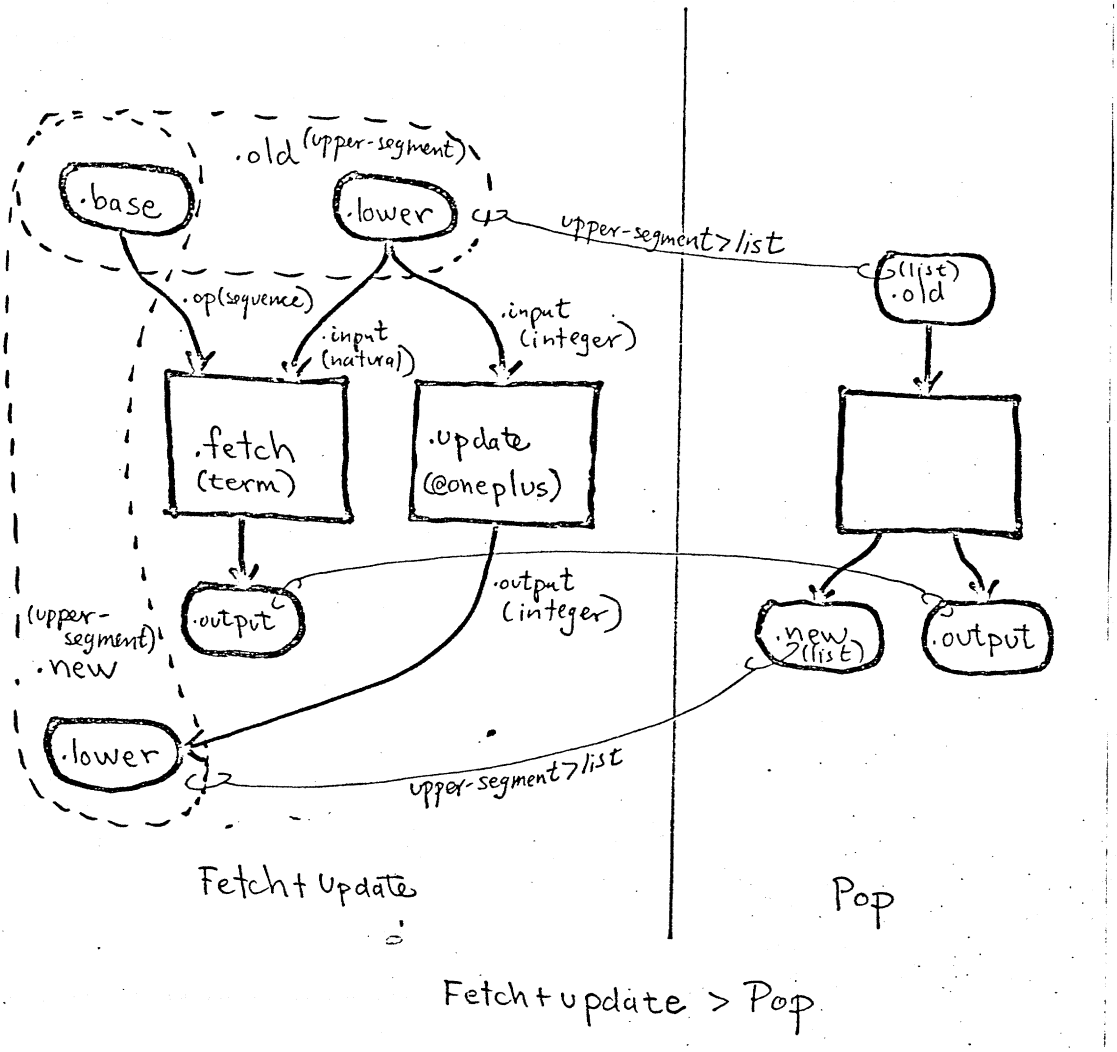


Figure 4-9. Implementation of Pop.

many questions unanswered here concerning how the search and matching is performed and how the instantiated plan is hooked up with the existing plan structure.

In bottom-up analysis, overlays are used in a similar way to build up more abstract descriptions of the program under analysis. The first step is to recognize known plans in the surface plan translation of the program. This may involve deduction, since some of the required constraints may not yet be explicit assertions. Furthermore, this recognition process can be made more hypothesis driven by first matching against explicit assertions and then either trying to derive the rest of the required constraints, or assuming them in order to accumulate more evidence for and against the hypothetical analysis. Once a plan has been recognized, we seek to overlay it (again, pun is intended) with another equivalent or more abstract plan. This is achieved by searching the library as above for overlays which have the given plan on one side. Having found one, an instance of the plan on other side is made and integrating into the evolving analysis.

Finally, we come to the use of overlays in verification. Recall that for verification we have in mind a very rich structure. First of all, there is the layered decomposition of plans and sub-plans. Plans at different layers are connected by overlays. Furthermore, each overlay is tied together by a network of dependencies which summarizes its verification. Whether we start by analyzing an existing program or with initial specifications for a new program to be synthesized, the final state of description is this top to bottom decomposition. From this standpoint, overlays are pre-verified modules which include both plans and teleology. Some of these overlays may be quite difficult to derive from first principles. However, once this has been done, they can be used over and over again. One of the goals of this thesis to compile enough of these pre-verified overlays so that the verification of routine[1] programs becomes mostly a matter of combining these pieces with very little difficult deduction remaining.

---

1. There is an intended circularity here. I believe that what makes certain programs "routine" is that they are a straightforward combination of familiar chunks.

CHAPTER FIVE

ANALYSIS BY INSPECTION

This chapter presents a scenario of the automated analysis of part of a symbol table program similar to the example introduced in Chapter Two. The input to this analysis is the Lisp code in Table I together with a comment that this program deals with a set of entries implemented as a hash table on keys. The output of this analysis is a hierarchy of plans which describe the computations performed by the given program at various levels of abstraction. The topmost plans in this hierarchy describe these computations in very abstract terms, i.e. in terms of set operations. The bottommost plans are very close to the code. They describe the computations in terms of the primitive data structures and operations of Lisp, such as dotted pairs, CAR and CDR. Connections between these different levels of description are represented using overlays.

The type of analysis shown in this chapter can be construed as a reconstruction of the top-down design of a program. This does not mean that the given program was actually designed that way, or that programs should be designed top-down. It only means that a top-down account is a useful way of understanding an existing program.

Table I. Lisp Code to be Analyzed.

```
; A SET OF ENTRIES IS IMPLEMENTED AS
;    A HASH TABLE ON KEYS.

; THE BUCKETS ARE IMPLEMENTED AS LISTS.

(SETQ TBL (ARRAY TBLSIZE))

(DEFINE LOOKUP
   (LAMBDA (KEY)
       (PROG (BKT ENTRY)
             (SETQ BKT (ARRAYFETCH TBL (HASH KEY)))
          LP (COND ((NULL BKT)(RETURN NIL)))
             (SETQ ENTRY (CAR BKT))
             (COND ((EQ (CAR ENTRY) KEY)
                    (RETURN ENTRY)))
             (SETQ BKT (CDR BKT))
             (GO LP))))

(DEFINE HASH
   (LAMBDA (KEY)
      (REMAINDER (MAKNUM KEY) TBLSIZE)))
```

5.1 Why Analysis?

In a programmer's apprentice system, a complete reconstruction of the abstract structure of a program as illustrated in this chapter would seldom be required, since the intermediate levels of description would be built up incrementally as part of the development process. There are, however, other reasons for studying this type of analysis. As a practical matter, automated analysis will be useful in converting from the present programming technology, which deals exclusively with code, to future technologies, which will involve many levels of computation description. Furthermore, for the foreseeable future the only common medium for transfer of programs between different systems will likely be code written in a standard programming language. In both of these situations, it is necessary to reconstruct a plausible design from given code in order to assimilate already written programs into new systems.

More fundamentally, many of the capabilities required for program analysis are important in other parts of the programming process as well. For example, the ability to recognize standard computations (analysis by inspection) at various levels of abstraction is important for automating both synthesis and verification, even in an incremental system. This is because there are often several different, but equally intuitive, ways of abstracting a given computation. For example, the symbol table LOOKUP routine can be abstracted either as associative retrieval (i.e. finding an entry in the set satisfying a given predicate), or as the application of a (partial) function from keys to entries. A programmer may be developing a program along one of these viewpoints, but the system may have to reanalyze it in a different way in order to bring the power of the plan library to bear. Furthermore, in an interactive program development system, this reanalysis need not wait until the plans involved are specific enough to be translated into code -- reanalysis can be useful at all levels of abstraction.

Others have also studied program analysis as a way of gaining insight into the programming process and the knowledge involved in programming.

5.2 Overview

The overall goal of the analysis described in this chapter is to decompose a given program into parts which may be recognized from the plan library. This is done in formal language which four major steps. The first two steps are basically algorithmic and have been implemented. The second two steps are of a more heuristic nature, and have not yet been implemented. In summary, while this chapter gives a fairly complete account of what constitutes the analysis of a program, it only goes part way towards automating the process of constructing one.

The first step in analyzing an already written program is to translate the computation description from the given program programming language into the plan calculus. This step is viewed as a translation because it does not involve any programming knowledge other than the semantics of the given programming language. The plans which are are the output of this translation step are called *surface plans*. The purpose of this translation step is to insulate the rest of the analysis process from the syntactic differences between various programming languages. Surface plans resulting from the translation of Lisp code were discussed briefly in Chapter Four. Code to surface plan translation has also been implemented for Fortran [60] and Cobol.

The second step of analysis described in this chapter is loop analysis. The purpose of this step is to decompose loops and recursions in a way which makes producer-consumer relationships explicit. Furthermore, the producer and consumer components resulting from this decomposition are often specializations of standard plans in the library. For example, temporal analysis decomposes the loop in LOOKUP roughly into three parts: CDR generation, iterative application of CAR, and iterative testing for an entry with the given key. These components are connected by data streams which represent the history of values taken on by the loop variables BKT and ENTRY. The idea for this type of loop analysis using the plan calculus was developed and has been implemented by Waters.

The final two steps of analysis in this chapter are less well worked out. The basic idea is to try to recognize known plans, first working bottom-up and then top-down. Working bottom-up entails regrouping parts of the surface plan and the temporal analysis in various ways so as to match plans in the library. One method of controlling this process is to use the *types* of the various descriptions involved (such as list, number, test, or loop) as a first filter on the grouping and matching. Also, not all plans in the library are considered in this first bottom-up matching phase. For example, with the current library, bottom-up goes as far as recognizing plans which have distinctive control flow and data flow features, but does not include recognizing the program structure having to do with the hash table. How far bottom-up methods can proceed with a larger plan library is an issue for further study.

The final step of plan recognition in this scenario is top-down analysis by synthesis. I assume that we are given a high level description of the program to start with. For example, for the symbol table program we are told that "a set of entries is implemented as a hash table on keys", and that "the buckets are implemented as lists". The concepts of set, hash table, key, bucket and list are all known in the current library. Furthermore, the names of the Lisp functions in Table I, HASH and LOOKUP, and the names of their arguments, KEY and ENTRY, are taken as part of the program documentation indicating that these routines implement a hashing function and associative retrieval from the set of entries, respectively.

The basic idea of analysis by synthesis is to use the plan library to generate possible implementations of the given high high level description until we find one which matches the existing bottom-up analysis. With the current library and the symbol table example, this technique appears to be feasible with simple breadth-first search through the space of possible implementations. With a larger library, some additional control mechanisms will need to be developed. Fickas has done some initial work in this direction.

The approach of dividing plan recognition into a bottom-up phase and a top-down phase has the feature that programs for which the appropriate higher level plans are not in the library can still be partially analyzed at the lower levels. For example, if the methods described in this chapter, together with the current plan library, were applied to analyzing an associative retrieval data base implemented entirely with linked lists, the top-down part of recognition would fail, but we would still succeed in analyzing the structure of the program at the level of search loops and list manipulations.

The next four sections present the four steps of analysis illustrated using LOOKUP. Note that there is not much to say about the analysis of the first two s-expressions in Table I by themselves. These expressions simply create a Lisp vector (TBL) of a specified size and define a numerical function (HASH), both of which are used later.

5.3 Surface Plans

In this section, we go over the diagram of the surface plan of LOOKUP in detail, explaining both the specifics of this example, and some points about surface plans in general.

A plan is a description which specifies a set of parts (or steps) with constraints between them. A computation is an instance of a plan if its parts satisfy the constraints of the plan. For surface plans, this is any computation which is the result of executing the corresponding code (e.g. with various inputs).

For example, the surface plan translation of LOOKUP is shown in Fig. 5-1 and Fig. 5-2. At the top level, this plan has has three steps: application of the hashing function, fetching from the hash table, and a loop with two exits. This structure is shown in Fig. 5-1 as a dashed outline with four boxes inside labelled One, Two, Loop and End. (The fourth box, End, is required to join the two cases of the loop). The names One, Two, Loop and End are called role names, which are local to the plan. The name of the whole plan is Lookup-surface. These particular names are generated by the translation process based on some simple conventions.

The Loop role of Lookup-surface is further described by another plan, which is shown partially in Fig. 5-1, and in full in Fig. 5-2. Role names can be composed using periods (read as "point") to form path expressions, such as Lookup-surface.One and Lookup-surface.Loop.If-one, meaning respectively role One of Lookup-surface, and role If-one of the Loop role of Lookup-surface.

Note in these figures that inputs and outputs that are not constrained by data flow are usually either unconstrained (as far as the larger plan is concerned) or fixed to some constant. Unconstrained inputs and outputs are labelled with the appropriate role names in ovals, just as in the defining diagram. Roles that are fixed to constants are indicated by writing the constant inside the corresponding oval. Constants can be distinguished from role names by the absence of the point prefix. All of these notations are illustrated by Lookup-surface.One in Fig. 5-1. The function being applied (Lookup-surface.One.Op) is a constant, Hash1, which is the mathematical function defined by HASH. The argument to the function (Lookup-surface.One.Input), which corresponds to the variable KEY in the code, is unconstrained. Finally, there is a data flow link between Lookup-surface.One.Output and the second input of Two.

The input-output specification for role One of Lookup-surface is @Function, the application of a given function (Op) to a given domain element (Input) to compute the correspnding range element (Output). In the case of Lookup-surface.One, the function applied is Hash1, the numerical function implemented by the HASH procedure.

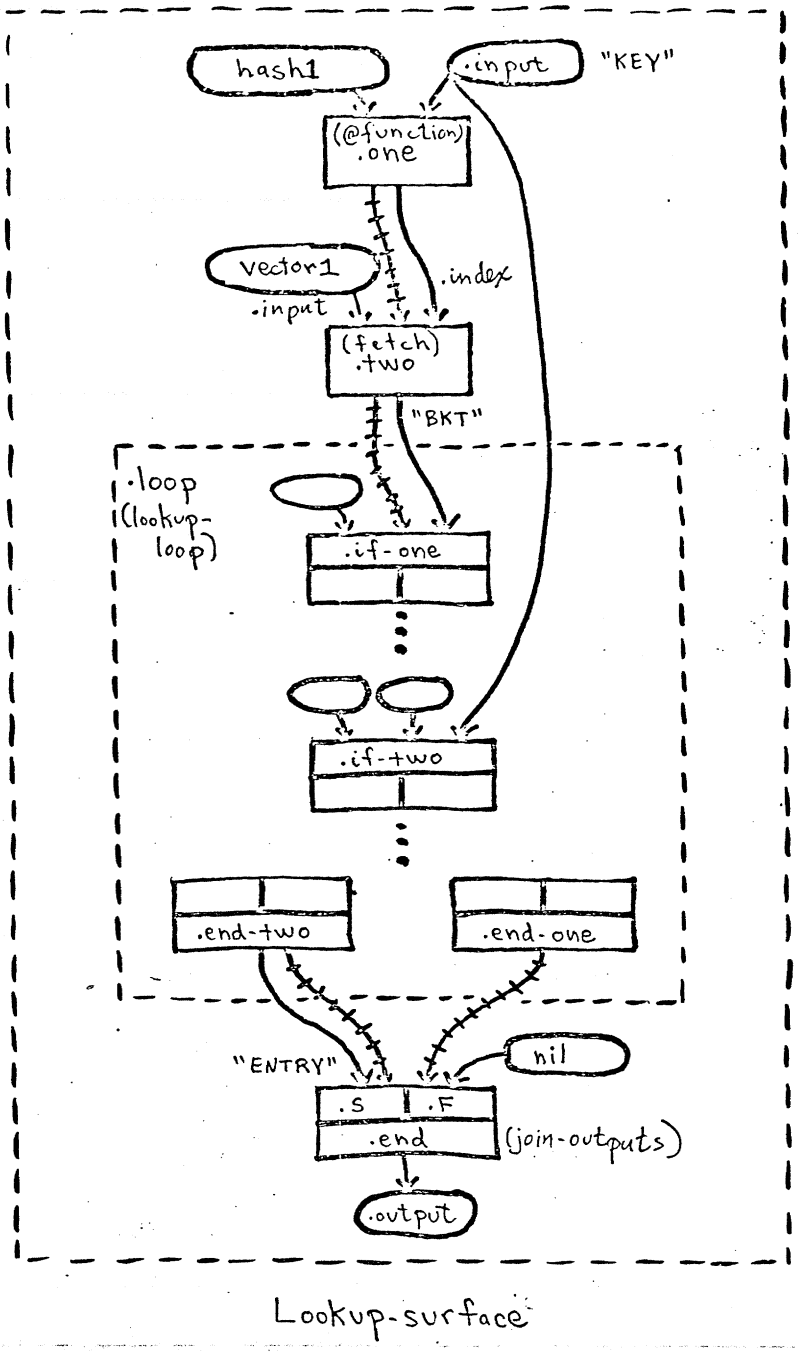


Figure 5-1. Toplevel Surface Plan for Lookup.

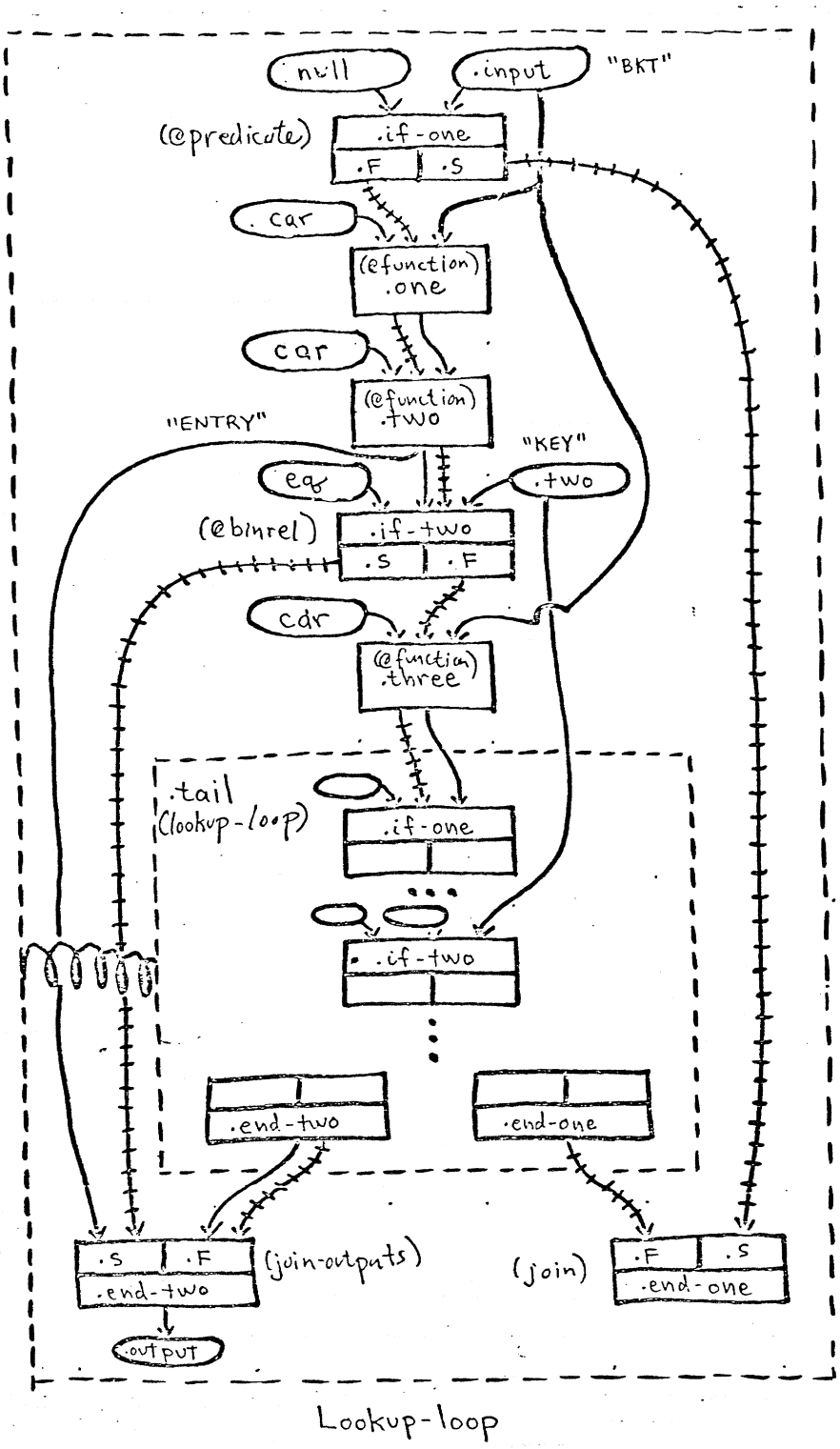


Figure 5-2. Surface Plan for Loop in Lookup.

The input-output specification for role Two of Lookup-surface is **Fetch.** he input roles are, in order from left to right: Input, a Lisp vector, and Index, a valid numerical index for that vector. The output role, Output, is the corresponding element of the vector. Lookup-surface.Two.Input is constrained to be Vector1, the Lisp vector created in the first line of the symbol table listing.

After Lookup-surface.Two, control flows into Lookup-surface.Loop. As can be seen in Fig. 5-1, control exits from the loop at two different locations. These two exits correspond to the two instances of RETURN in the code. In one case, ((RETURN ENTRY)) there is also data flow out of the loop.

The surface plan for the looping part of LOOKUP is shown in Fig. 5-2. The most prominent feature of this plan is that it is recursively defined. In the plan calculus, loops are represented using recursive definition, as suggested by the following code.

```
(DEFINE LOOKUP
   (LAMBDA (KEY)
       (PROG (BKT ENTRY)
             (SETQ BKT (ARRAYFETCH TBL (HASH KEY)))
             (LP))))

(DEFINE LP
   (LAMBDA ()
             (COND ((NULL BKT)(RETURN NIL)))
             (SETQ ENTRY (CAR BKT))
             (COND ((EQ (CAR ENTRY) KEY)
                    (RETURN ENTRY)))
             (SETQ BKT (CDR BKT))
             (LP))))
```

This turns out to be the most convenient representation for many purposes, especially for making inductive arguments in program verification.[1] In plan diagrams, recursive definition is indicated by a curly line, as in the lower left of Fig. 5-2. This notation means that the Tail role of Lookup-loop is defined to have the same plan as Lookup-loop. Enough of the Tail is expanded in this diagram to specify the connections between one repetition of the loop and the next.

Lookup-loop has seven other roles, in addition to Tail.[2] Three of these (One, Two and Three) are applications of the primitive Lisp functions, Car and Cdr. These are the translations of (CAR BKT), (CAR ENTRY) and (CDR BKT) in the code. The other four roles in Lookup-loop are various kinds of tests and joins.

A test specification has two cases: a test either succeeds or fails, depending on some some specified conditions on the inputs. Two particular kinds of tests used in Lookup-loop are @Predicate and @Binrel. @Predicate tests whether or not a given unary relation (Criterion) is true of given object

---

1. There are interpreters, and certainly compilers, which execute tail recursive code as efficiently as code with loops in control flow, essentially making these syntactic variants.
2. As in Lookup-surface, the role names in this plan are chosen by the translation process based on some simple conventions.

(@Predicate.Input). Similarly, @Binrel tests whether two given objects satisfy a given binary relation (Criterion).

The first test in Lookup-loop, If-one, is constrained to be a instance of @Predicate in which the Criterion is Null, a primitive Lisp predicate. This role is the translation of the code

```
(COND ((NULL BKT)...)) .
```

When this test succeeds, control exits from the loop, as can be seen by the control flow arrow from the "S" side of If-one which bypasses the Tail. When this test fails, control passes to One and then Two, which are the translation of the following portion of the loop code.

```
(SETQ ENTRY (CAR BKT))
...(CAR ENTRY)...
```

The output of Two feeds into input One of If-two, which is an instance of @Binrel. The criterion in this test is the primitive binary relation, Eq. Input Two (KEY in the code below) is unconstrained as far as the Lookup-loop plan is concerned, except that it doesn't change on successive repetitions of the loop. The fact that input Two of this test is the same as the argument to the hashing function is reflected in the constraints of Lookup-surface, as can be seen in Fig. 5-1.

```
(COND ((EQ ... KEY) ...))
```

If this test succeeds, control exits the loop through End-two making One.Output (ENTRY) available outside the loop. Otherwise, Cdr is applied to If-one.Input (BKT), with the result feeding into the recursive invocation.

Lookup-surface.End, Lookup-loop.End-one and Lookup-loop.End-two are joins, the complementary construct to tests. Joins have two input cases (similarly labelled "S" and "F"), indicated in plan diagrams by dividing the top part of the box in half. There are two possible ways for control to flow into a join, and one way out.[1] Joins are also used to represent the effect of control flow on data flow. For example, the pattern of data flow and control flow through Lookup-surface.End in Fig. 5-1 indicates that in one case the value returned by LOOKUP is the entry that satisfied the second exit test of the loop, and in the other case it is the constant, Nil.

---

1. Note that this is not a parallelism construct. In any given computation, only one or the other branch of a conditional is taken.

5.4 Loop Analysis

The goal of analysis in this chapter is to decompose a program into recognizable parts. In other words, we want to figure out how the surface plan for a program could be built up out of standard plans in the library. This section in particular is concerned with the analysis of singly recursive surface plans, such as Lookup-loop, which represent the looping parts of a program. It is important to note, however, that none of the analysis in this section is particular to whether a surface plan is the translation of code written in Lisp versus some other standard programming language. Although appropriate specializations for Lisp will be emphasized for the purpose of this example, the plans and overlays introduced in this section are all quite general.

The analysis of loops takes place in two steps. In the first step, a loop is decomposed into standard recursively defined fragments. In the second step, the behavior of these fragments is abstracted in such a way that a loop can be represented by a non-recursive plan. This allows further analysis to treat the looping and non-looping parts of programs uniformly.

Loop Augmentations

The natural building blocks for non-recursive plans are typically input-output specifications, which are composed using control flow, data flow and tests. The plan library contains many standard input-output specifications and their implementations in terms of compositions of others. For recursively defined plans, however, a different notion of composition is needed in order to make a library of standard building blocks. Loops are viewed here as being built up by a process of *augmentation*.[1] For example, the loop of LOOKUP can be built up starting with just the part that does the CDRing, as suggested by the following code.

```
(PROG (BKT)
      (SETQ BKT ...)
   LP ...
      (SETQ BKT (CDR BKT))
      (GO LP))
```

This pattern of looping, in which a given function is repeatedly applied to the output of the preceding application of that function, is called iterative generation. Iterative generation using Cdr is a common building block of many loops in Lisp.

This loop can be augmented by adding the code underlined below.

---

1. This view of loops is taken from Waters [60]. In this reference, Waters also goes into a more lengthy justification of why a different analysis method is required for loops as compared to straight-line code.

```
(PROG (BKT ENTRY)
      (SETQ BKT ...)
   LP ...
      (SETQ ENTRY (CAR BKT))
      ...
      (SETQ BKT (CDR BKT))
      (GO LP))
```

The basic idea of augmentation is that the augmented loop does everything the unaugmented loop does, plus something extra. For example, the augmented loop above makes available in ENTRY the CAR of each successive value of BKT computed by the generation part of the loop. This pattern of augmentation is called iterative application; the function being applied in this case is Car.

Two other kinds of augmentation, which are not illustrated in the symbol table example, are filtering and accumulation. These will be covered in Chapter Nine.

The addition of an exit test to a loop, as shown underlined below, is a kind of augmentation which violates the general rule that an augmentation must not disturb the behavior of the unaugmented loop.

```
(PROG (BKT ENTRY)
      (SETQ BKT ...)
   LP (COND ((NULL BKT)(RETURN NIL)))
      (SETQ ENTRY (CAR BKT))
      ...
      (SETQ BKT (CDR BKT))
      (GO LP))
```

The reason an exit test is treated as an exceptional case of augmentation is because, as will be seen in the next section, its effect is modelled similarly. In general, the effect of an augmentation is to create a new sequence of data objects (such as the values of ENTRY) in the augmented loop which is related in some way to a sequence of objects (such as the values of BKT) in the unaugmented loop. The effect of adding an exit test to a loop is modelled as creating truncated versions of the (potentially infinite) sequences which would be generated by the loop without the exit test. This also applies for adding more than one exit test, as shown below.

```
(PROG (BKT)
      (SETQ BKT ...)
   LP (COND ((NULL BKT)(RETURN NIL)))
      (SETQ ENTRY (CAR BKT))
      (COND ((EQ (CAR ENTRY) KEY)
             (RETURN ENTRY)))
      (SETQ BKT (CDR BKT))
      (GO LP))
```

Waters has implemented a system which automatically decomposes loops according to this idea of augmentation. The basic algorithm his system uses is to iteratively remove parts of a loop which do not produce data objects required by the remaining parts. For example, for the loop of LOOKUP,, the effect of this algorithm is to undo the augmentation steps above in the reverse order. The

plan library contains plans for many standard augmentations. The rest of this section shows some of these which are used in LOOKUP and how they are represented in the plan calculus.

The first augmentation recognized in the LOOKUP loop is shown in Fig. 5-3. On the left hand side of this figure we have the surface plan for the loop, Lookup-surface. On the right hand side is a plan from the library called **Terminated-iterative-search**. This plan captures the idea of a search loop with two exits, without specifying how the sequence of objects being searched is produced. Role If-two of this plan is a test which applies a given criterion (the same on each iteration) to the current input (provided by the rest of the loop). When this test succeeds, the current input is made available outside the loop (as End-two.Output). The other exit test (If-one) is for terminating the loop when there are no more objects in the search space.

Note that the role names of a plan in the library, such as Terminated-iterative-search, are fixed at the time the plan is catalogued. In general, role names have been chosen to have some mnemonic value relative to the given plan, but this strategy is somewhat restricted by the fact that specialized plans inherit their role names from their generalizations. For example, the most general plan for a two exit loop, of which Terminated-iterative-search is a specialization, is Cascade-iterative-termination. At the level of generality of Cascade-iterative-termination, it is not possible to give any better names to the two exit test roles than If-one and If-two.

The hooked lines between the left and right hand sides of Fig. 5-3 indicate how the Terminated-iterative-search plan is matched against Lookup-surface: Lookup-loop.If-one corresponds to Terminated-iterative-search.If-one; Lookup-loop.If-two corresponds to Terminated-iterative-search.If-two;[1] and there is a correspondence between the joins, End-one and End-two. The fact that the corresponding roles have the same names is a coincidence. The hooked line between Lookup-loop.Tail and Terminated-iterative-search.Tail indicates that the match is made recursively.

Fig. 5-3 is an example of an *overlay*. The basic idea of overlays is re-description. The plan on the left describes a set of computations -- the instances of the plan. The correspondences in the figure indicate how to re-describe (part of) any such computation as an instance of the plan on the right, in this case a standard plan from the library. In order for this re-description to be possible, the constraints of the right hand plan must logically follow from the constraints of the left hand plan, substituting appropriately for the corresponding parts. It can be seen in Fig. 5-3 that this condition is met for control flow and data flow constraints (control flow is transitive).

---

1. A detail is being skipped here, which is covered in the appendix. Lookup-loop.If-two is a test in which a binary relation (Eq) is applied to two inputs. Terminated-iterative-search.If-two is a test involving a predicate (unary relation). In order to recognize Terminated-iterative-search as indicated, an intermediate step is required in which Lookup-loop.If-two is grouped together with Lookup-loop.Two and these are viewed as the implementation of testing a composite predicate of the form (EQ (CAR ...) KEY).

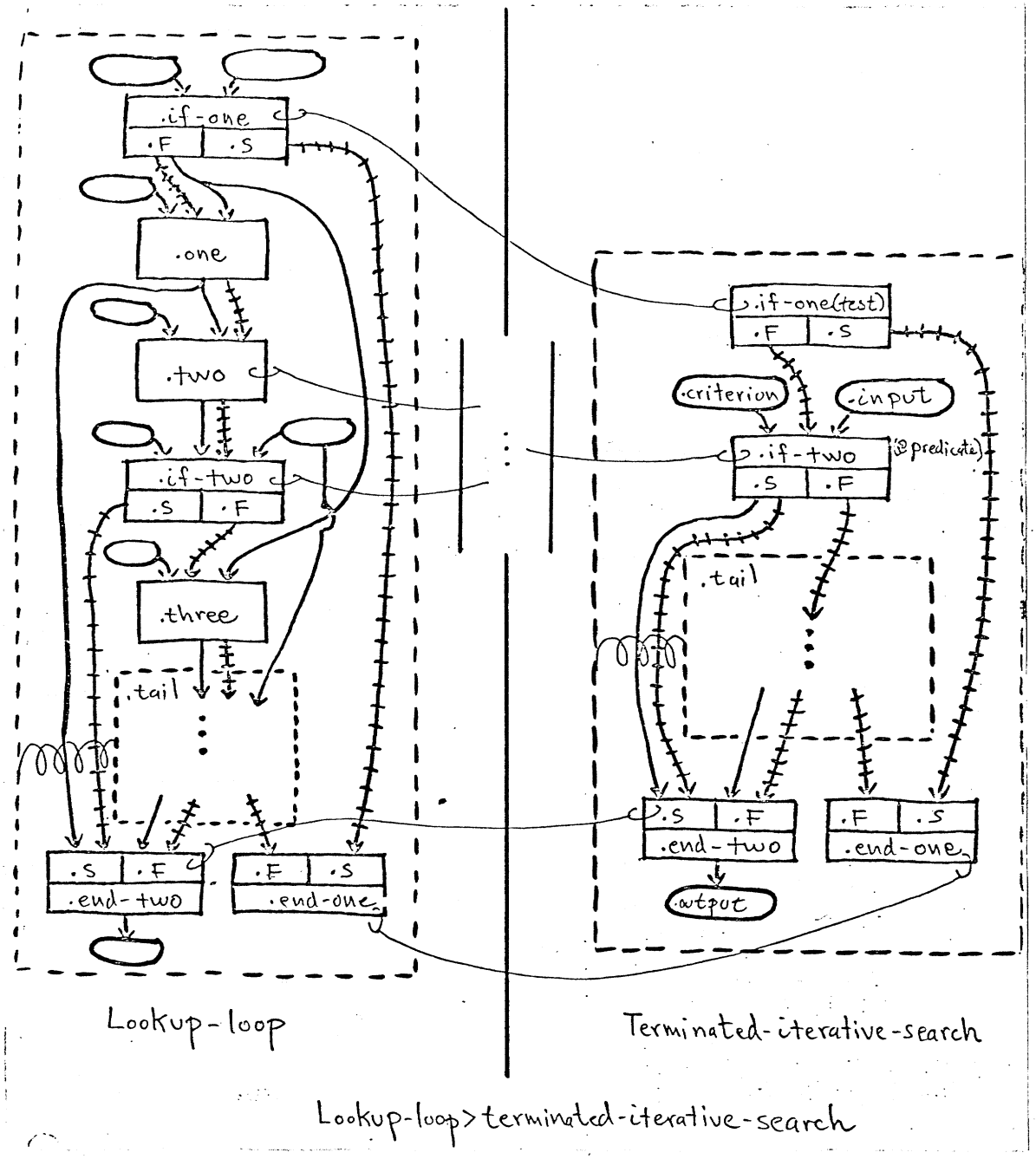


Figure 5-3. Terminated Search in LOOKUP Loop

Overlays are the mechanism used to relate various various levels of description in the analysis of a program. The origin of the term "overlay" is in the idea of having each plan drawn on a transparent slide and laying one on top of the other, lining up the corresponding parts.[1] Some overlays, such as the one in Fig. 5-3, are particular to the analysis of one specific program; others, of which we will see examples later, express re-descriptions of general applicability and are therefore catalogued in the plan library together with the plans to which they apply.

Recognition of the other augmentations in Lookup-loop takes place in a model of the loop in which the exit tests are assumed to always fail. This is what I call the *steady state* model. The relationship between the surface plan and the steady state model is also represented using an overlay which is explained in more detail in Chapter Nine.

The first augmentation recognized in the steady state model of Lookup-loop is the iterative application of Car, shown in Fig. 5-4. On the right hand side of this overlay is the plan from the library, **Iterative-application**, which represents the general idea of repeatedly applying a given function (Action.Op) to an input provided by the rest of the loop (Action.Input) to produce an output (Action.Output), which may be used by the rest of the loop. The correspondences between this plan and Lookup-loop on the left indicate that Lookup-loop.One in the steady state matches this description. Similarly, Fig. 5-5 shows the how Lookup-loop.Three is re-described as Iterative-generation.

Temporal Abstraction

Given that we have decomposed a loop plan into these standard augmentations, the question remains of how to represent the connection between, say, the generation and the application. Temporally, the components of each computation are interleaved, but it seems more logical to view the generation and application as being composed in some way. This section shows how to construct this viewpoint.

The basic idea of temporal abstraction is to view all the objects which fill a given role in a recursively defined plan as a single data structure.[2] In terms of Lisp code, this often corresponds to having an explicit representation for the sequence of values taken on by a particular variable at a particular point in a loop. For example, in the LOOKUP loop we would like to talk about the sequence of objects iteratively generated by Cdr, i.e.

---

1. Sussman uses the term "slice" for a similar concept in the analysis of electronic circuits.
2. Both Shrobe [56] and Waters [60] use the idea of temporal abstraction, but with slightly different formalizations than presented here.

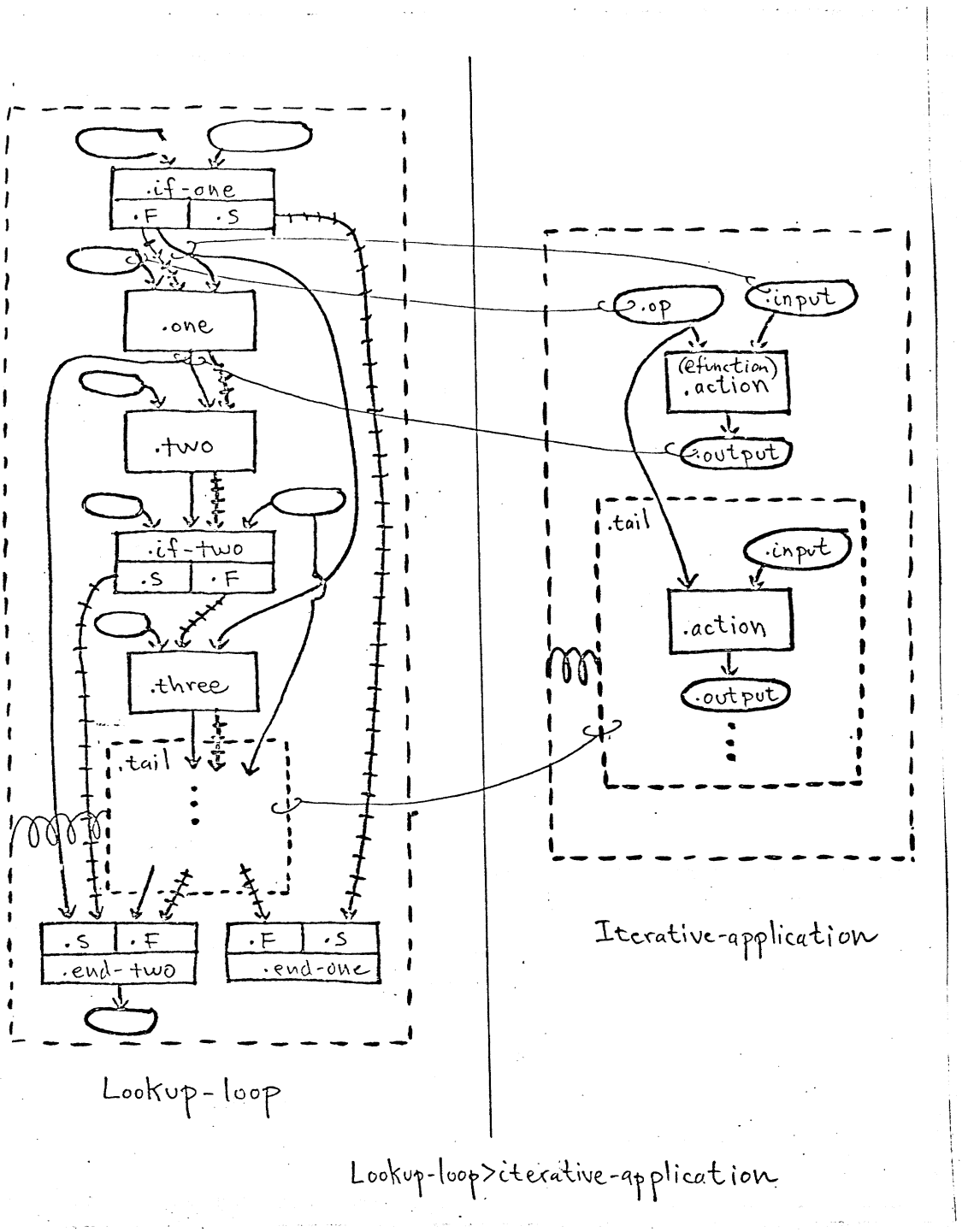


Figure 5-4. Iterative Application in LOOKUP Loop.

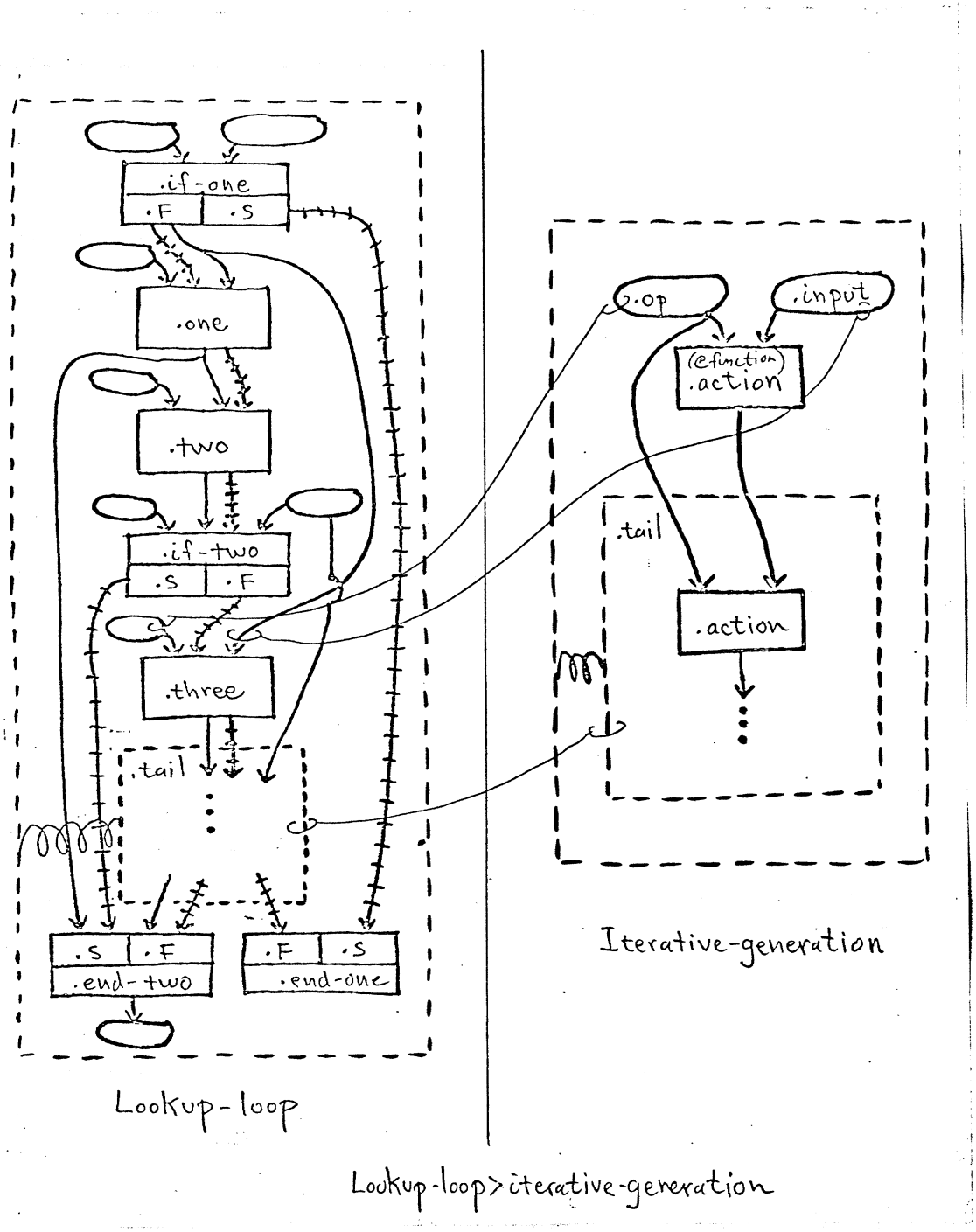


Figure 5-5. Iterative Generation in LOOKUP Loop

Iterative-generation.Action.Input,
Iterative-generation.Tail.Action.Input,
Iterative-generation.Tail.Tail.Action.Input,
...

which corresponds to the values of BKT at the point underlined below each time around the loop (in the steady state).

```
(PROG (BKT)
      (SETQ BKT ...)
   LP ...
      (SETQ BKT (CDR BKT))
      (GO LP))))
```

The bottom overlay in Fig. 5-6 shows how this abstraction is made in terms of the input-output specification, Iterate, which takes as input a data structure called an iterator (which is the linear specialization of a generator), and gives as output the generated sequence. An iterator has two parts: the Seed, the starting value which will be the first term of the generated sequence; and the Op, the function which maps from one term to the next. As shown by the hooked lines in Fig. 5-6, the Iterate.Input.Seed corresponds to Iterative-generation.Action.Input, and Iterate.Input.Op corresponds to Iterative-generation.Action.Op. Iterate.Output then represents the sequence of inputs to Action on each iteration, as described above.

Iterator is an example of a data plan -- the plan for a data structure. This plan, together with Iterate and the overlays in Fig. 5-6, are part of the current library. An important feature of the plan calculus is that it allows the hierarchical description of data structures and temporal computations (and mixtures of the two) in a single formalism.

The top overlay in Fig. 5-6 makes the same sort of abstraction for Iterative-application. In this overlay from the library, Iterative-application is viewed as the input-output specification, Map, which takes a sequence and a function (Op) as inputs, and has a sequence as output. The terms of the output sequence are the result of applying the given function to the terms of the input sequence. In this temporal overlay, the input sequence of Map is the abstraction of the inputs to the Action of Iterative-application on each iteration, and the output sequence is the abstraction of the outputs of Action; and, of course, Map.Op corresponds to Action.Op. In terms of the code of LOOKUP, Map.Input represents the values of BKT at the point underlined below and Map.Output represents the values of ENTRY (in the steady state).

```
(PROG (BKT ENTRY)
      (SETQ BKT ...)
   LP ...
      (SETQ ENTRY (CAR BKT))
      ...
      (SETQ BKT (CDR BKT))
      (GO LP))))
```

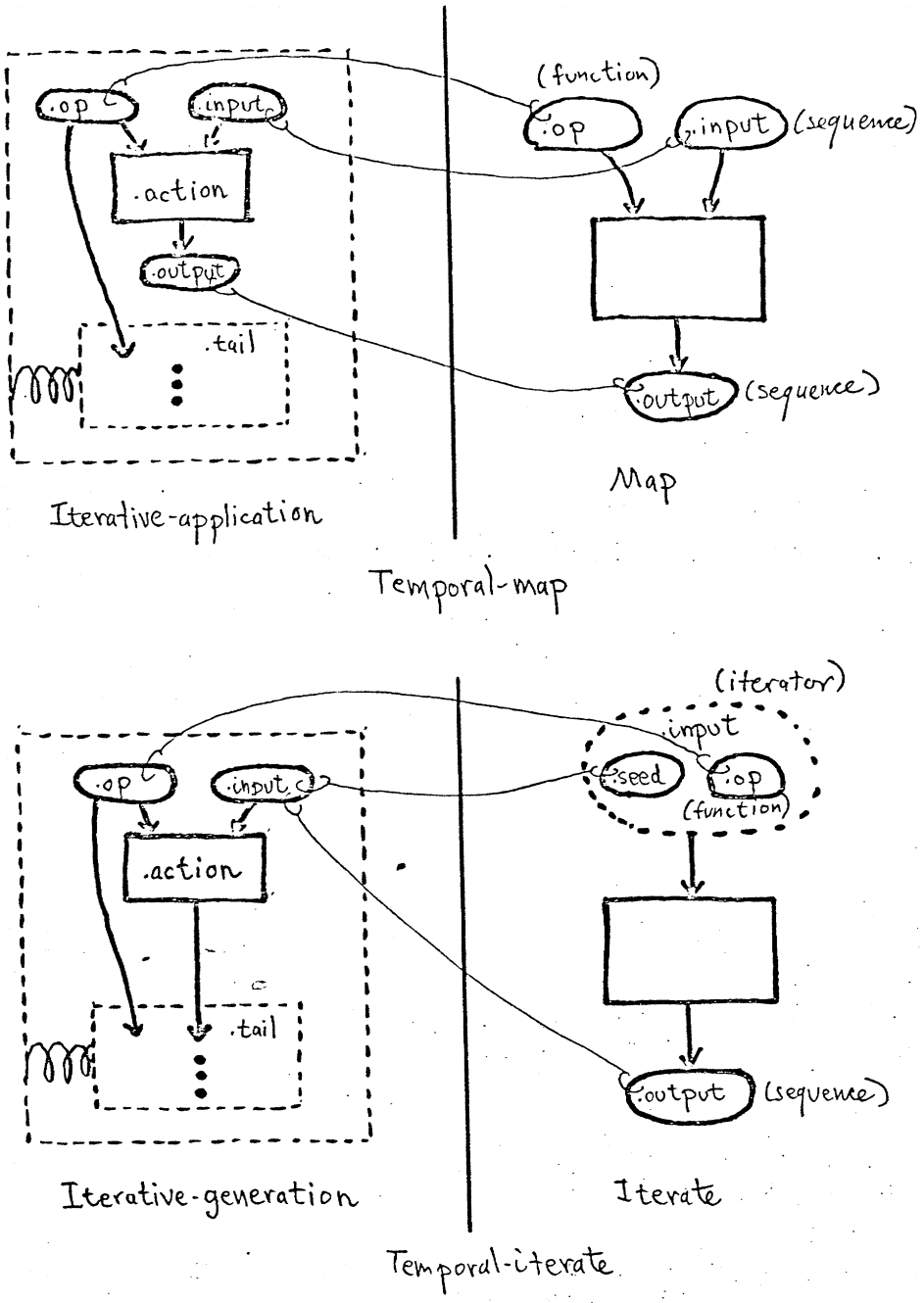


Figure 5-6. Temporal Overlays

Notice in this code that the value of BKT is the same at the underlined point as at the input to CDR. This means that in the temporal abstraction of Lookup-loop, the output sequence of Iterate is the same as the input sequence of Map. In the next section we will see how this pattern, together with the temporal abstraction of the NULL test, is recognized as the standard plan for generating a list in Lisp.

Exit test augmentations are also temporally abstracted. Fig. 5-7 shows an overlay from the library which abstracts Terminated-iterative-search as an input-output specification on sets. This overlay also illustrates a second kind of temporal abstraction, in which we talk about the set of objects filling a given role in a recursive plan, ignoring their temporal order.[1] As we shall see, this turns out to be the appropriate level of abstraction for this example.

The pattern of two exit tests with an output from the second one, which has been recognized in Lookup-loop, can be viewed as the temporal implementation of a standard test on sets called Any. Given a set (Any.Universe) and a predicate (Any.Criterion), this test succeeds if there is a member of the set which satisfies the predicate, and returns such an object (Any.output); otherwise it fails. In the temporal overlay of Terminated-iterative-search as Any shown in Fig. 5-7, Any.Universe corresponds to the set of inputs to the second exit test of the search.[2] In the code of LOOKUP, this is the set of values of ENTRY at the point underlined below.

```
(PROG (BKT ENTRY)
      (SETQ BKT ...)
   LP (COND ((NULL BKT)(RETURN NIL)))
      (SETQ ENTRY (CAR BKT))
      (COND ((EQ (CAR ENTRY) KEY)
             (RETURN ENTRY)))
      (SETQ BKT (CDR BKT))
      (GO LP))))
```

Finally, the effect of the NULL test in the code above is modelled in the temporal view by an input-output specification called **Cotruncate.** Cotruncate takes as input two sequences (Cotruncate.Input and Cotruncate.Co-input) and a predicate (Cotruncate.Criterion). Its output is the truncation of the second sequence at the earliest term for which the corresponding term of the first sequence satisfies the given predicate. This may sound like a somewhat obscure specification, but the idea of two parallel sequences is in fact quite basic. For example, the standard plan for computing the length of a Lisp list can be naturally viewed in terms of two parallel temporal sequences: the natural numbers, and the sequence of CDRs of the list. In the code above, the sequence of values of

---

1. Formally this abstraction is done in two steps: first a temporal sequence (not exactly) abstraction is made; and then this ordered structure is viewed as the implementation of a set. This will be explained more fully in Chapter Nine.

2. More precisely, Any.Universe corresponds to the set of objects which would be searched if there were no member satisfying the predicate. This abstraction involves forming a steady state model in which exit Two always fails.

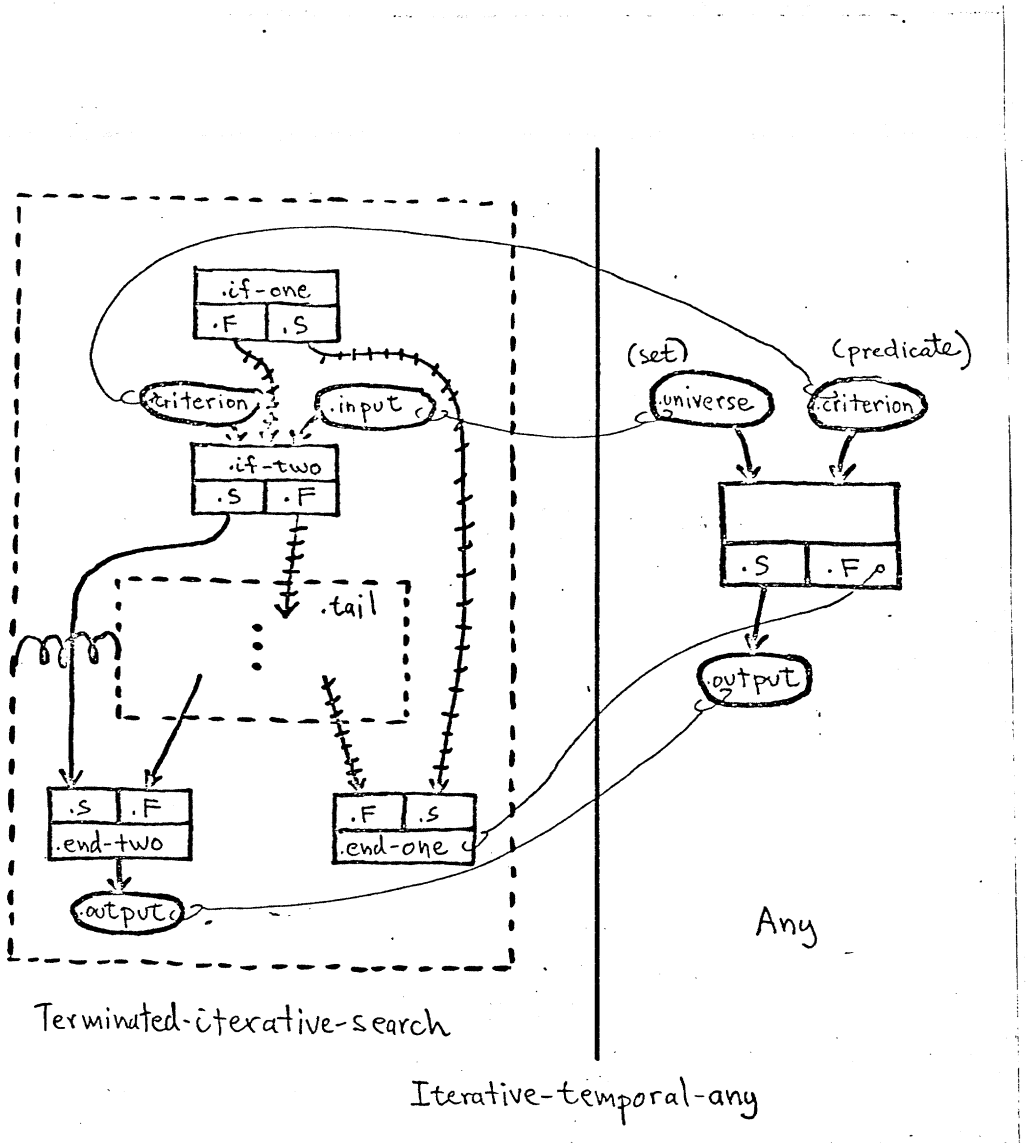


Figure 5-7. A Temporal Set Overlay.

ENTRY (the output of Map) is truncated according to the occurrence of NIL in the sequence of values of BKT (the output of Iterate).

The relationship between the temporal abstractions of the various parts of Lookup-loop is illustrated in Fig. 5-8. This figure shows all four overlays discussed in this section applied to Lookup-loop simultaneously. In order to reduce clutter, only the data flow constraints in Lookup-loop and the correspondences which involve temporal abstraction are drawn. Notice that many of the temporal sequences on the right are the abstraction of roles of Lookup-loop which are constrained by data flow to be the same. In particular, Iterate.Output is the same sequence as Map.Input and Cotruncate.Input, and Map.Output is the same sequence as Cotruncate.Co-input.

Some of the temporal correspondences in Fig. 5-8 involve different steady state models. For example, Cotruncate.Input is the temporal abstraction of Lookup-loop.One.Output in the steady state model with no exit tests; Cotruncate.Output is the sequence which includes the effect of the Null test. This detail cannot be shown conveniently in this figure, but is explained in the next section.

The relationship between the output sequence of Cotruncate and Any.Universe is represented by an overlay, **Sequence>set,** which expresses in general how to view a sequence as a set. Such overlays between abstract descriptions are typical as analysis progresses beyond the surface plan.

In summary, Fig. 5-9 shows an overview of the plans and overlays used in the loop analysis thus far. The names of the plans are arranged in a hierarchy which reflects the order in which they must be recognized.[1] Each plan depends on the recognition of the plans below it as indicated by the vertical lines in the figure. Plans at the same level in the hierarchy may be recognized in any order. Overlays from the library used in this analysis are drawn as vertical lines with arrow heads to suggest that once the lower plan is recognized, the library is searched to suggest a more abstract description. The other lines represent pattern matching that is done specifically for this example. Notice that the analysis of a program is not a strict hierarchy. Distinct nodes at one level may share parts of the same plan at a level below. For example, the recognition of both Iteration and Iterative-application share the Iterative-steady-state plan. Conversely, the fact that a given plan or role has been used in one overlay, does not make it ineligible for use in others.

---

1. Some of these steps have been skipped over in this initial exposition, but are included here for future reference.

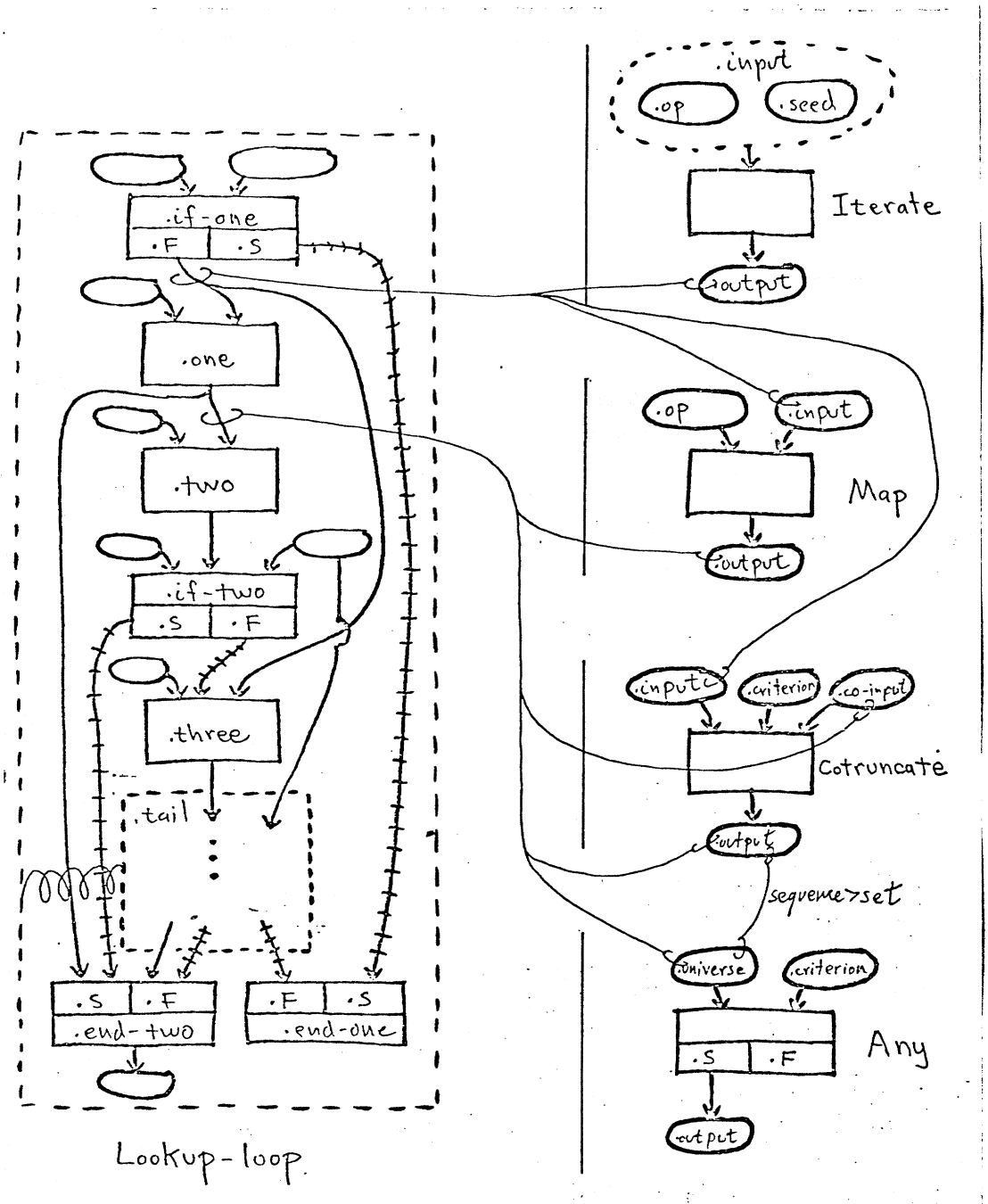


Figure 5-8. Temporal Abstractions of LOOKUP Loop.

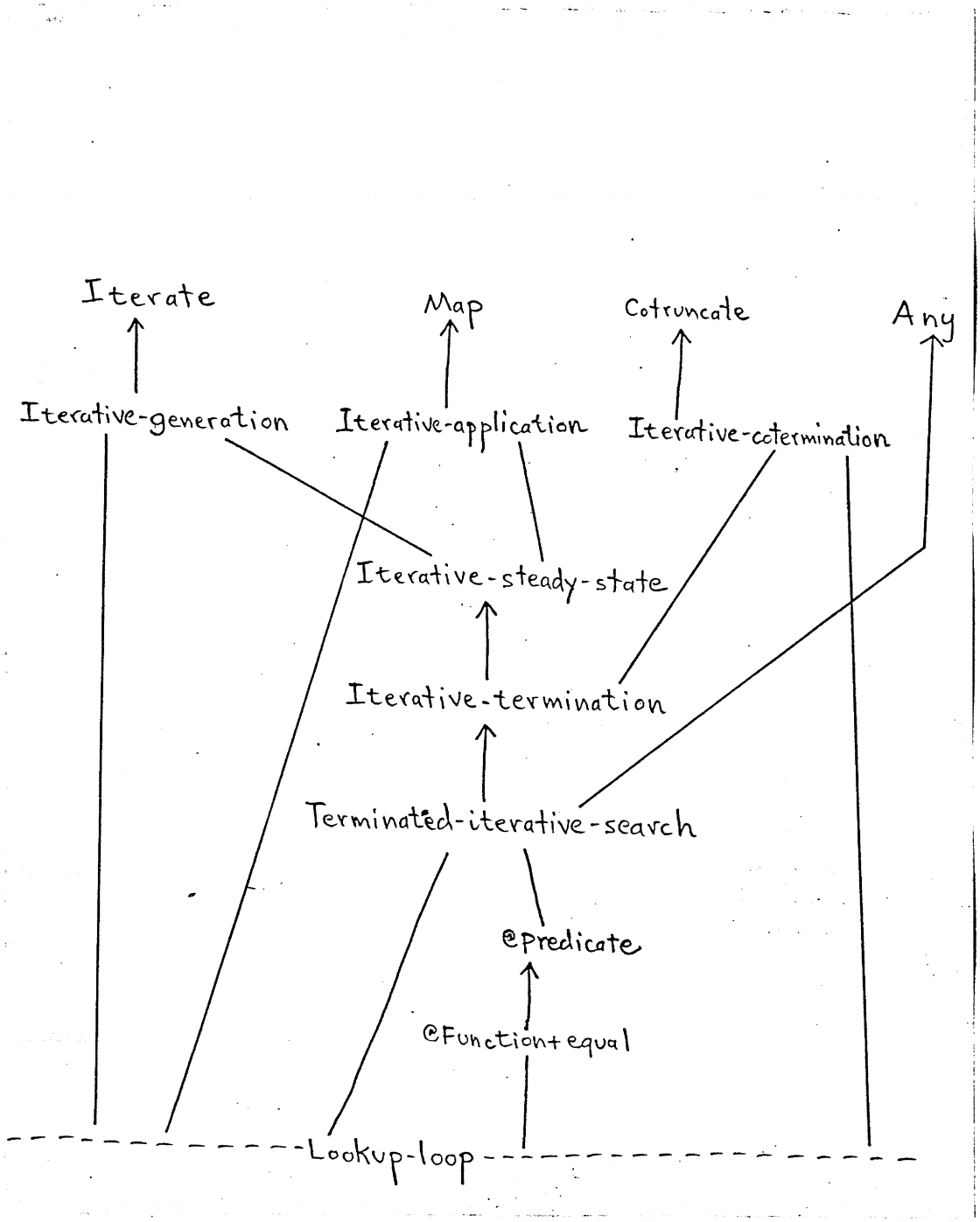


Figure 5-9. Overview of Analysis of LOOKUP Loop.

5.5 Bottom-up Recognition

It is natural to divide the analysis of LOOKUP roughly into three layers, as shown in Fig. 5-10. The bottom layer is loop analysis, as described in the preceding section. The middle and top layers are distinguished mostly by the complexity of the data structures involved. The plans in the middle layer involve only basic data structures such lists, sequences and sets. The effect of temporal abstraction, which is the final step of loop analysis, is to re-describe looping computations in terms of these basic data structures. The top layer of analysis in this example involves the relatively complex and specialized hash table data structure.

My intuition is that these general layers of abstraction are not specific to this example, though in larger programs there would be more upper layers. This means that the plan library itself can be roughly divided into layers. Most of the plans in the current library are in the middle layer involving lists, sequences, sets, and also directed graphs. Presently the only more complicated data plan is the hash table.

The three layers of knowledge in this example also suggest a three stage strategy for automating analysis. The first stage is the specialized algorithm for loop analysis described in the preceding section. The second stage can be thought of as bottom-up pattern recognition, in which the standard plans involving basic data structures familiar to every experienced programmer are recognized. The final stage of analysis in this example depends on being given some high level description of what the whole program is trying to do, so that top-down analysis by synthesis can be used. An alternative scenario, in which no top level description is given, is not considered in this thesis.[1] These three stages agree with my own introspection and experience in analyzing previously unseen programs. It would be interesting to conduct some experiments to verify the psychological validity of this model.

The rest of this section describes the particular plans in middle layer of the analysis of LOOKUP, which are recognized bottom-up. The next section describes the plans in the top layer, which are recognized top-down, and how the two layers are connected.

As we can see in Fig. 5-10, there are several plans in the middle layer which may be recognized in any order. We begin with the plan **Car+cdr+null**, shown in Fig. 5-11, which has three steps: One, an instance Iterate; Two, an instance of Map; and Three, an instance of Cotruncate. The data flow between the roles in this plan is the same as between the overlays of Iterate, Map and Cotruncate on Lookup-loop described in the preceding section and shown in Fig. 5-8. This plan in general is called **Truncated-list-generation.** Car+cdr+null is a specialization in which the generating

---

1. Such a scenario would presumeably involve a much stronger control structure for hypothesis formation and testing.

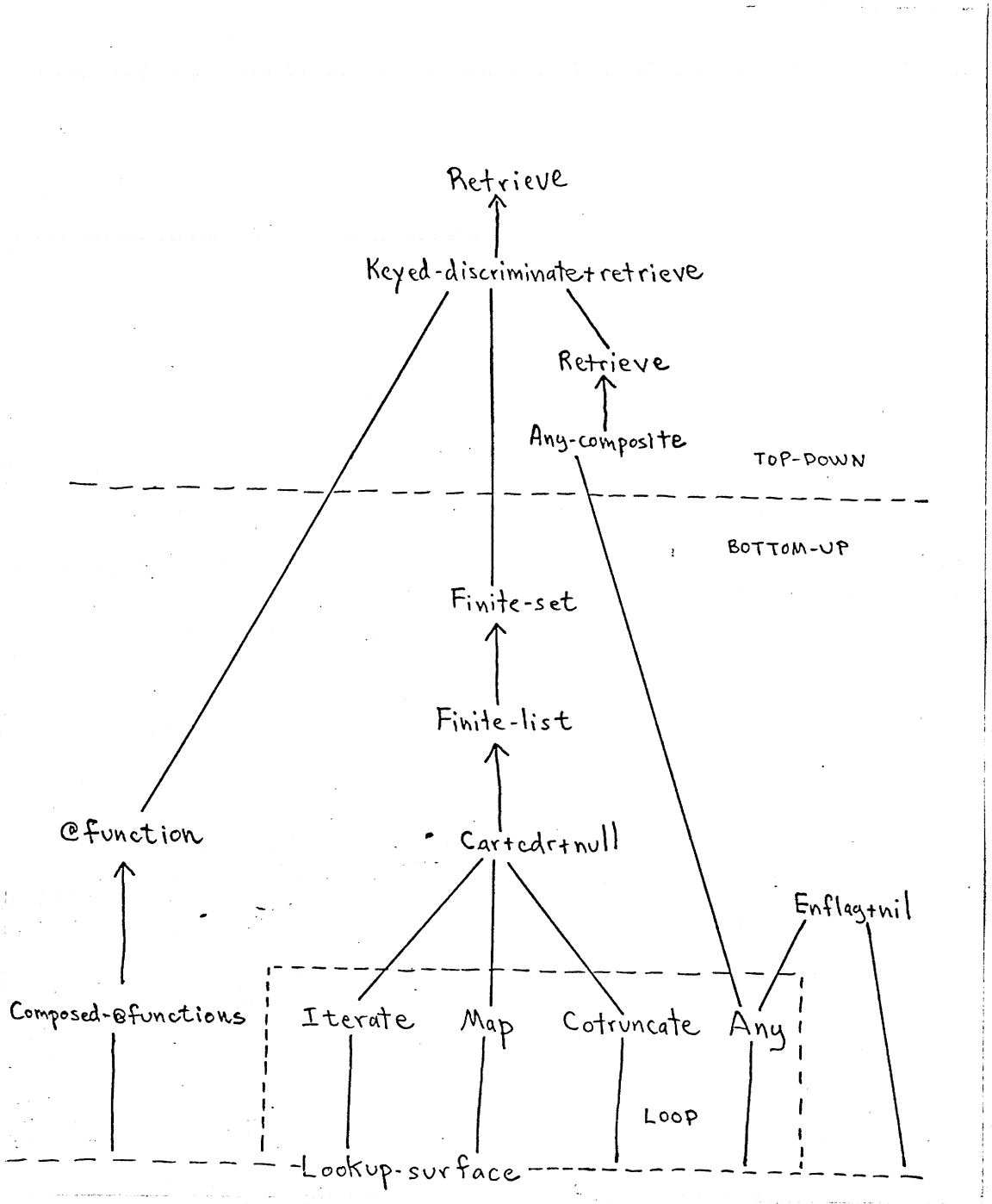


Figure 5-10. Layers in Analysis of LOOKUP.

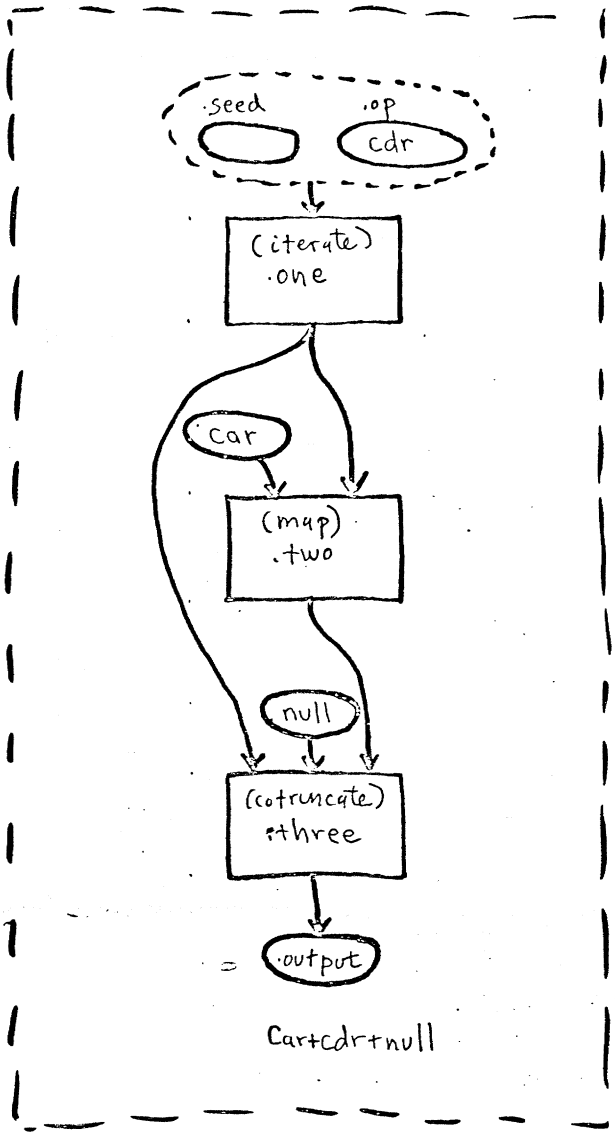


Figure 5-11. Plan for Generating a Lisp List.

function is Cdr, the function being applied by Map is Car, and the criterion of Cotruncate is Null.[2]

Returning to LOOKUP, we have now come as far as recognizing that the initial input to the loop (the initial value of BKT) is a Lisp list. The temporal abstraction of the second exit from the loop as Any goes one step further and views this list as the implementation of a set. From the analysis of LOOKUP alone, it is not clear whether or not this list may contain duplicates. In the plan library, the implementation of sets as irredundant lists is represented as a specialization of the overlay used here.

There are two more small points to be covered. The first two steps of Lookup-surface (please refer back to Fig. 5-1) need to be analyzed as the application of a functional composition, Composed-@functions, as described in Chapter Four. This is a common cliche which is needed here to put the surface plan in a form which will connect well with the top-down recognition stage of the next section.

Another feature of the surface plan of LOOKUP to be recognized bottom-up is that the final output object (Lookup-surface.End.Output), which in the code is the value returned by the LAMBDA expression, can be viewed as a flag. Flags are a minor programming technique which is formalized in the appendix. The basic idea is that the result of a test (in this case Any) is encoded in a data object and a given predicate (in this case Null) so that the information discovered by the test can be recovered after the join. (In this case control is joined because Lisp does not allow multiple return points). The information encoded in the flag is recovered later by testing the object with the given predicate.

---

2. We always try to recognize the most specialized version of a plan where possible.

5.6 Top-down Recognition

The main point of this section to illustrate how a (moderately) complex data structure, such as a hash table, is decomposed in terms of the plan library. This section also introduces an important heuristic principle which is applicable to both top-down recognition and the standard synthesis scenario.

The comment at the beginning of the code for the symbol table program in Table I reads: "a set of entries is implemented as a hash table on keys". In this analysis scenario, we make use of this comment to retrieve the top few plans in the analysis of LOOKUP from the library.

At the highest level of abstraction, we are dealing with the implementation of a set. This set is implemented as a hash table on keys. In the analysis presented here, this implementation is decomposed into three basic ideas: discrimination, hashing and keys.

According to the plan library, a discrimination is a function which maps some domain (in this example, "entries") onto a set of sets, called buckets. Such a function can be viewed as implementing a set wherein a given object is a member if and only if it is a member of the bucket obtained by applying the discrimination function to that object. Operations on a set implemented this way reduce to operations on a single bucket, which is often more efficient, especially in the case of operations which involve search. This idea is also part of many other data structures, such as discrimination nets.[1]

The basic idea of hashing is to implement a discrimination function as the composition of two functions. The first function, called the hash function, maps the domain of the discrimination onto the set of valid indices for a sequence, called the table, which is the second function. The utility of this decomposition is that modifications to the discrimination function may be achieved by modifying only the table.

Discrimination on keys is also an implementation idea involving functional decomposition. In a keyed discrimination, each member of the implemented set has an associated key. In the symbol table example, the function from entries to keys is Car. The discrimination function in this implementation is the composition of two functions: the first function is the key function; the second function maps from the set of keys to the buckets. The point of this decomposition is that for certain operations, such as associative retrieval, we are given only the key of an entry, rather than the entry itself.

---

1. The relationship between hash tables and discrimination nets is pursued further in Chapter Four.

To summarize, all three of these ideas are combined in the symbol table example as follows. At the top level we have a set implemented as a keyed discrimination. The key function is Car, and the function from keys to buckets is implemented as a hash table. The hash function of the hash table is Hash1 (HASH); the table is an abstraction of Vector1 (TBL), in which it is viewed as a function from natural numbers to sets implemented as Lisp lists.

Being able to formally analyze a data structure design in this way is a new and important result of this thesis. This analysis gives a deep insight into the logical structure of this implementation and captures what it has in common with other implementations. It also decomposes the verification of the design, since each component can be separately verified. This aspect of this thesis is a contribution towards current efforts in computer science to develop an "algebra" of practical programming constructs. Others in this movement thus far have either concentrated on the composition of procedural constructs, similar to the ideas described in the loop analysis section, or have worked only with simpler data structures [43].

The Maximal Sharing Heuristic

There are several plausible accounts of how the analysis described above could be derived automatically, given the code and comments in Table I, the bottom-up analysis described in the preceding section, and the current plan library. All of these accounts involve using what I call the *maximal sharing heuristic*. The origin of this heuristic is in program synthesis, but it also turns out to provide an elegant solution to the problem in program analysis of connecting bottom-up recognition with top-down analysis by synthesis.

The maximal sharing heuristic is applied at each point in synthesis when an implementation step is made. The basic idea of the heuristic is, rather than always adding new structure for an implementation, to reuse as many parts as possible of other plans in the current design which satisfy the constraints of the current implementation plan. The effect of this heuristic is to cause there to be a (locally) maximal amount of sharing in the analysis hierarchy. The motivations for this heuristic, and its application in a synthesis scenario are elaborated in Chapter Six.

The way to apply this heuristic in analysis is to view the parts of the bottom-up analysis as parts of the current design which are available for reuse. Whenever a part of the bottom-up analysis gets used in a top-down synthesis step, a connection has been achieved between the two directions of analysis. This holds out the promise that a module written for automated synthesis which obeys this heuristic may be used without change in automated analysis.

Another nice feature of this approach is that it suggests two fairly intuitive notions of partial analysis. One situation is when you can't find parts of the program you expect. This corresponds to when parts of the top-down synthesis never get connected with the bottom-up analysis. In an

interactive system, this could signal a potential bug or at least a request for further explanation from the user. The complementary situation is when parts of the bottom-up analysis are never used by the top-down phase. The most natural interpretation of this situation is that the programmer is using plans which are not in the current library. An interesting topic for future research is the possibility of isolating and generalizing these novel parts of a program so that new plans can automatically be added to the library.

Returning now to LOOKUP, let me give one account of how the final steps of analysis might proceed. It may help to refer to Fig. 5-10 to follow this explanation.

The first step in the top-down analysis is to conclude that the set operation implemented by LOOKUP is associative retrieval. This could be deduced from the name of the routine, or by looking at the types of its inputs and outputs and the fact that it has two cases.

The library overlay for implementing associative retrieval from a keyed discrimination is shown in Fig. 5-12. The input-output specification for associative retrieval on the right hand side is called **Retrieve**. It is a test with three inputs, a set (Universe), the key function (Key), and an input key (Input), and one output. If there exists a member of the set with the given key, then the test succeeds and returns such a member; otherwise it fails. On the left hand side of the overlay we have the typical two step plan for implementing a set operation on a discrimination: apply the discrimination function to fetch the appropriate bucket, and then perform the same operation on the bucket. This general implementation works for adding and removing a member, and certain kinds of retrieval. It does not work for other operations such as union or intersection.

The first step (Discriminate) of the plan on the left of Fig. 5-12 is thus constrained to be an instance of @Function, in which the function being applied is the discrimination function from keys to buckets. The maximal sharing heuristic suggests using the @Function recognized bottom-up (see Fig. 5-10) in this role. Recall that this @Function is itself implemented as the composition of two instances of @Function,

```
(ARRAYFETCH TBL (HASH KEY)),
```

from which we can conclude that the hashing function is Hash1 and the table is Vector1.

The second step (If) of the plan on the left of Fig. 5-12 is constrained to be an instance of Retrieve, applied to the bucket fetched in step one. According to the second comment at the front of the code (see Table I), "the buckets are implemented as lists". The only implementation in the library for Retrieve on sets other than than discriminations is as Any (an input-output specification introduced earlier in this chapter) in which the criterion is a composite predicate. The form of this predicate is to test whether the key of a given object is equal to some constant. If the key function is Car, this comes down to Lisp code like the following test in LOOKUP.

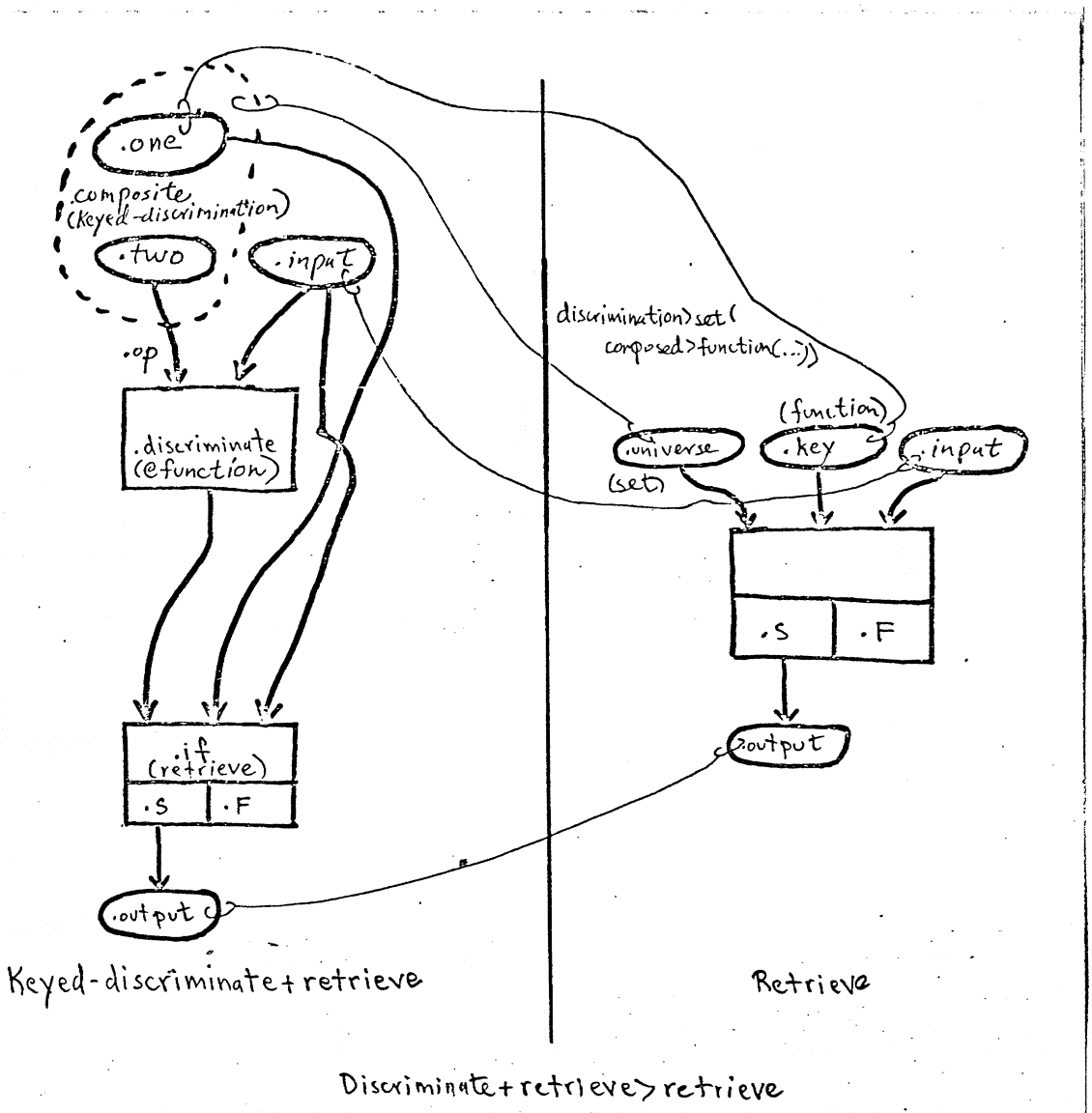


Figure 5-12. Associative Retrieval from a Keyed Discrimination.

```
(COND ((EQ (CAR ...) KEY)
       ...))
```

The sharing heuristic suggests recognizing the bottom-up Any specification as implementing the bucket Retrieve in this way. In order for this to be the case, the key function of the implementation must be Car.

This completes the analysis of LOOKUP. Let me emphasize that the last few paragraphs are only one of many possible accounts of how the top-down recognition could be accomplished. There are many other strong clues in this program, particularly in the types of objects. For example, the only candidate for the table part of the hash table, by virtue of being a vector, is Vector1; the only candidate for the hash function, by virtue of being a numerical function, is Hash1.

CHAPTER SIX

SYNTHESIS BY INSPECTION

6.1 Introduction

A library of plans, such as presented in Chapter Three, opens up many new possibilites for what an interactive program development system can do to help a user synthesize programs. This chapter is an exploration into some of these new possibilities. In particular, this chapter highlights the use of the plan calculus as a design language.

In broad terms, the plan library represents a significant body of knowledge about programming which is shared between the user and the system, which has never been the case before. The most advanced current program development systems [8] have some built-in knowledge of programming language syntax and type restrictions, but none include the range and kind of knowledge represented in the plan library.

This chapter presents a simple scenario of interactive program synthesis in which the working medium is the plan calculus rather than Lisp code. Code is generated only as the final translation of the synthesized surface plan. Also, the order of development in this scenario is top-down. The user the progressive refinement of an initial abstract specification by application of overlays from the plan library. This scenario is thus restricted to programs which can be completely analyzed using plans in the library alone. This scenario also portrays an expert user who is familiar with the plan library.

This chapter also picks up where Chapters Two and Five leave off in showing the details of how the plans in the library are used together in a complete example.

The fundamental interaction between the system and the user in this scenario is for the system to propose a *menu* of overlays from the library which are applicable to the current design plan, and for the user to choose between them. In this way, the user guides the synthesis in top-down fashion. The user also intervenes at certain crucial points in the development to introduce new plans from the library, and to suggest reanalysis of the current design which leads to a more efficient implementation. In addition to retrieving overlays from the library, the system also needs to be able to spontaneously propagate some information and construct specializations of library plans appropriate to the current design. This implies a deductive component in the system, whose operation will not be discussed here, since it is part of related research reported elsewhere [56].

Deductive capabilities are also required to apply the maximal sharing heuristic. The basic idea of this heuristic, as described in Chapter Five, is to build plans which share as much structure as possible.[1] The motivation for this heuristic is that it often leads to more efficient programs. It is applied in synthesis each time an overlay is used to further implement some part of the current design. To apply the heuristic, the system needs to know whether a given subset of the roles on the left hand side of the overlay can be identified with roles of other existing plans in the current design, while being consistent with both the constraints of the plan on the left hand side of the overlay and the existing constraints on the other roles.

With the maximal sharing heuristic in operation, synthesis using overlays becomes a mixture of progressive refinement and constraint. In the refinement steps, an overlay is used to expand the current design in a tree-like fashion, by adding more detail at one of the terminal nodes. Alternatively, whenever sharing is established in the application of an overlay, the effect to is add further constraints to the current design.

The synthesis scenario in this section is divided into three distinct phases: data structure design, procedure implementation, and code generation. In the first phase, the user lays out the implementation of the hash table data structure using overlays between data plans. The second phase, procedure implementation, involves refining the abstract plans (e.g. input-output specifications) for associative retrieval, addition, and deletion on the hash table down to the level of Lisp surface plans. The final stage is the generation of Lisp code from surface plans. As in the first scenario in this chapter, the system prompts for variable and procedure names where needed in the coding. Furthermore, in this particular scenario the implementation of the retrieval, addition and deletion procedures are each carried out fully, including the coding phase, before moving on to the next one.

I do not, however, make any claims for the particular order of synthesis depicted in this scenario. The major purpose of the scenario is to demonstrate what it could be like to develop programs interactively with a system that had significant programming knowledge in the form of a plan library. In reality, any such system will have to be based on a mixed initiative model which allows the user to tailor the order of development to the particular programming task at hand. As in the scenario of Chapter Two, lines typed by the system are shown in upper case; lines typed by the user are shwon in lower case.

---

1. Sacerdoti, in his work on general problem solving [50] uses a similar heuristic of the form "use existing objects whenever possible".

6.2 Data Structure Design

In this section, the user designs the main data structure of the symbol table program, starting with a description of it as a set of entries, and culminating with its implementation as a keyed discrimination in which the function from keys to buckets is implemented by hashing.

```
> let an "entry" be a data structure.
> "symbol" (an atom) is part of an entry.
> "info" is part of an entry.
```

The user begins by defining a new data plan which is particular to the programming task at hand. This definition becomes a part of the permanent documentation of the program. An "entry" is defined as a data structure with two fields, named Symbol and Info. The Symbol field contains a Lisp atom; nothing is said about the Info field.

```
> let a "symbol table" be a finite set.
> for all "x" if x belongs to a symbol table, then x is an entry.
```

The user now defines a symbol table as finite set of entries. The rest of this section of the scenario is concerned with implementing this set.

```
> )
APPLICABLE IMPLEMENTATIONS FOR: SYMBOL-TABLE (A FINITE SET)
      1. LIST
      2. SEQUENCE
      3. LABELLED-THREAD
      4. DISCRIMINATION FUNCTION
? 4
```

The user transfers initiative temporarily to the system by typing an empty command line. The basic action of the system in this situation is retrieve from the library all overlays which are applicable to the current design. In general, this includes all overlays whose right hand sides (or specializations or extensions of them) appear in the current design. Furthermore, the system in general first proposes overlays for a whole plan before trying to find overlays for any of its roles. In this case, the system retrieves four overlays: the first three are for implementing sets as lists, sequences, and labelled threads, respectively; the fourth is for implementing a finite set as a discrimination. The user chooses the fourth overlay as the one to apply, and the system responds by instantiating the left hand side of the overlay, propagating the appropriate information through the correspondences.

```
LET "DISCRIMINATION1" BE THE IMPLEMENTATION OF SYMBOL-TABLE AS A
DISCRIMINATION FUNCTION.
```

Thus Discrimination1 is a discrimination function on entries, which is the implementation of Symbol-table. Fig. 6-1 shows the complete implementation of Symbol-table, as it will be developed in this section. Implementation relationships, such as between Discrimination1 and Symbol-table, are shown in this figure as vertical arrows pointing up from the instance of the left

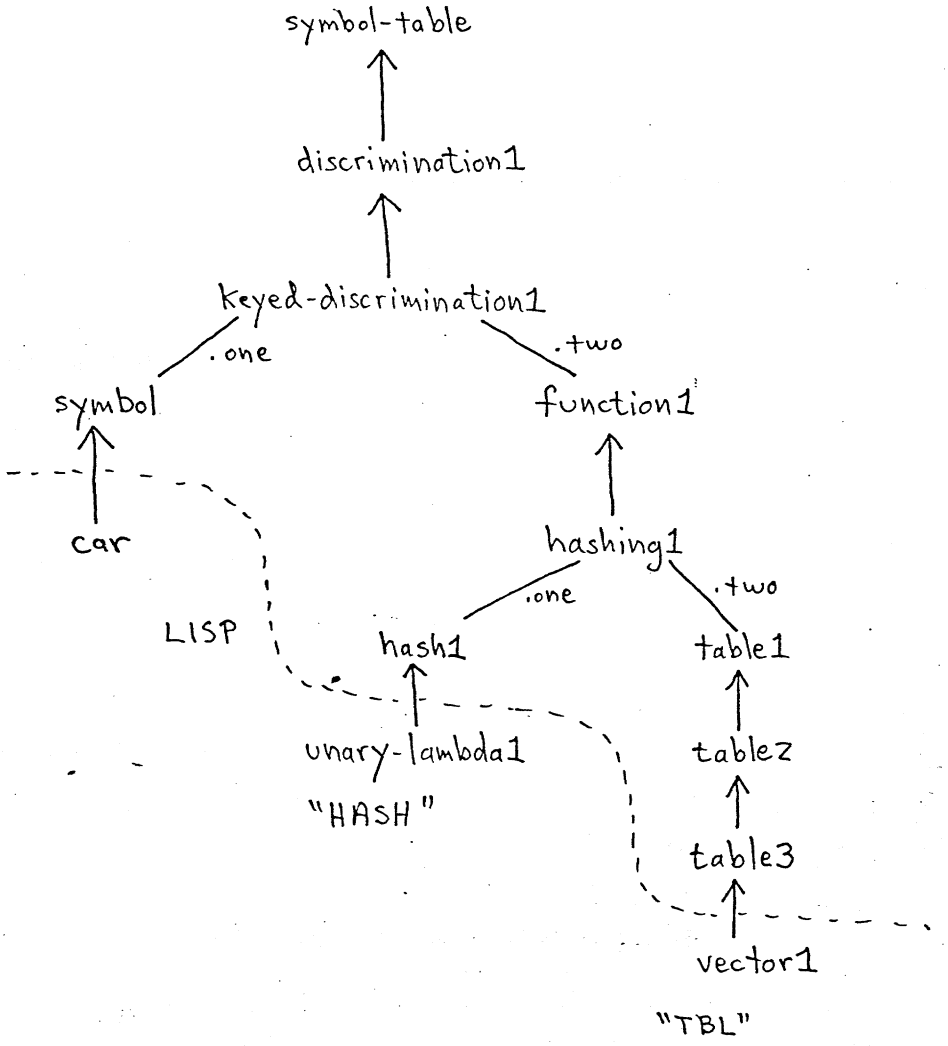


Figure 6-1. implementation of the symbol table.

hand side to the instance of the right hand side. (This is the same convention as was used in Chapter Three.) In this figure, and the other similar one is this chapter, vertical lines without arrow heads denote part-whole relationships, with the role names indicated.

```
> )
APPLICABLE IMPLEMENTATIONS FOR: DISCRIMINATION1 (A DISCRIMINATION FUNCTION)
      1. COMPOSED-FUNCTIONS
      2. UNARY-LAMBDA
? 2
```

The scenario continues with the system proposing applicable overlays for Discrimination1. In this case, the only applicable overlays are for functions in general, of which there are two in the current library: a function can either be implemented as the composition of two other functions, or it can be implemented directly in Lisp as a lambda expression. The user chooses the first option.

```
SPECIALIZATIONS OF: COMPOSED-FUNCTIONS
      1. HASHING
      2. KEYED-DISCRIMINATION
? 2

LET "KEYED-DISCRIMINATION1" BE THE IMPLEMENTATION OF DISCRIMINATION1 AS A
KEYED-DISCRIMINATION.
```

Before applying an overlay, the system checks in the library for any specializations or extensions of the plan on the left hand side. If any exist, the user is given a chance to choose one. In this case, the system finds two specializations of Composed-functions, namely Hashing and Keyed-discrimination. The user chooses the second, causing the system to instantiate Keyed-discrimination1 as the implementation of Discrimination1. (The user could have responded "no", causing the system to instantiate an instance of Composed-functions with no further constraints.)

```
> )
ELIGIBLE SHARING FOR: KEYED-DISCRIMINATION1.ONE ("THE KEY FUNCTION")
      1. SYMBOL
      2. INFO
? 1

LET SYMBOL BE THE KEY FUNCTION OF KEYED-DISCRIMINATION1.
LET "FUNCTION1" THE BUCKET FUNCTION OF KEYED-DISCRIMIANTION1.
```

The instantiation of Keyed-discrimination1 gives us our first opportunity to see the maximal sharing heuristic in action. The system above has searched for existing objects in the current design which could fill the roles of Keyed-discrimination1 and satisfy the constraints of the Keyed-discrimination plan. The first filter on this search can be the object *types* -- roles One and Two of a keyed discrimination must be functions. There are three functions in the current design: Symbol,[1] Info and Discrimination1. Discrimination1 can immediately be eliminated from consideration because it is above Keyed-discrimination1 in the tree, so that sharing with it would lead to a

1. Role names are formally functions.

meaningless circularity. Symbol and Info can be rejected for role Two of Keyed-discrimination1, since the range of this function is constrained to be finite sets. This leaves the possibility of Symbol or Info filling role One of Keyed-discrimination, which the system proposes as shown above. The user chooses a keyed discrimination on the symbol field of entries. The system completes this frame of the interaction by instantiating Function1, a function from Lisp atoms to finite sets, to fill role Two. (Again, the user could have responded "no" to the question above, in which case a new object would be instantiated for role One as well as for role Two.)

```
> )
APPLICABLE IMPLEMENTATIONS FOR: FUNCTION1 (A FUNCTION)
      1. COMPOSED-FUNCTIONS
      2. UNARY-LAMBDA
? 1

SPECIALIZATIONS OF: COMPOSED-FUNCTIONS
      1. HASHING
      2. KEYED-DISCRIMINATION
? 1

LET "HASHING1" BE THE IMPLEMENTATION OF FUNCTION1 AS A HASHING.
```

In this next frame, Function1 is implemented as a hash table, Hashing1.

```
> )
LET "HASH1" BE THE HASH FUNCTION OF HASHING1.
LET "TABLE1" BE THE TABLE OF HASHING1.
```

Since there are no existing objects which can fill the roles of Hashing1, the system instantiates Hash1 and Table1.

```
> implement the buckets of table1.
```

After letting the system carry the initiative for a few steps, the user intervenes here with a command to retrieve overlays from the library for implementing the buckets of the discrimination (the range elements of Table1).

```
APPLICABLE IMPLEMENTATIONS FOR: BUCKETS OF TABLE1 (A FINITE-SET)
      1. LIST
      2. SEQUENCE
      3. LABELLED-THREAD
      4. DISCRIMINATION
? 1

SPECIALIZATIONS OF: LIST
      1. IRREDUNDANT-LIST
? 1

LET "TABLE2" BE THE IMPLEMENTATION OF TABLE1 SUCH THAT
FOR ALL "I" AND "X", IF TERM I OF TABLE2 EQUALS X, THEN X IS THE
IMPLEMENTATION OF TERM I OF TABLE1 AS AN IRREDUNDANT-LIST.
```

The user above chooses to implement the buckets as lists, and in particular lists without duplicates. The system responds by instantiating Table2 which is related to Table1 an overlay in which each term of Table2 (an irredundant list) is viewed as the corresponding term of Table1 (a finite set).

This completes the part of the data structure design that is programming language independent (see Fig. 6-1).

Lisp-Specific Data Structure Implementation

The remaining few steps in this section are involving with bringing the data structure design down to the level of Lisp data objects.

```
> implement an entry as a dotted pair, such that
  entry.symbol=dotted-pair.car and entry.info=dotted-pair.cdr
```

Here the user specifies the implementation of entries as dotted pairs such that the Symbol function is implemented by Car and the Info function by Cdr.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: HASH1 (A HASHING)
        1. COMPOSED-FUNCTIONS
        2. UNARY-LAMBDA
  ? 2
```

Hash1 is to be implemented as a Lisp procedure, which will be coded later.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: BUCKETS OF TABLE2 (IRREDUNDANT-LIST)
        1. UPPER-SEGMENT
        2. DOTTED-PAIR
        3. LIST-GENERATION
  ? 2

  LET "TABLE3" BE THE IMPLEMENTATION OF TABLE2 SUCH THAT
  FOR ALL "I" AND "X", IF TERM I OF TABLE3 EQUALS X, THEN X IS THE
  IMPLEMENTATION OF TERM I OF TABLE2 AS A DOTTED-PAIR.
```

The range elements of Table2 are not yet implemented as Lisp data objects. The system suggests three overlays for implementing lists. The user chooses the direct route of implementing lists as dotted pairs.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: TABLE3 (A SEQUENCE)
        1. SEGMENT
        2. VECTOR
        3. LIST

  ? 2

  LET "VECTOR1" BE THE IMPLEMENTATION OF TABLE3 AS A VECTOR.
```

Finally we have Table3, which is a sequence of Lisp data objects (dotted pairs), which can be implemented as a Lisp vector.

6.3 Procedure Synthesis

The user now moves on to the implementation of some procedures which access the symbol table data structure. The first procedure is to retrieve the entry associated with a given symbol. Fig. 6-2 gives an overview of this implementation. Down the left side of this figure is the data structure implementation developed in the preceding section. As in Fig. 6-1, arrows in this figure denote overlays, and roles names are labelled. In this figure, however, many roles are left out in order to make it more readable. The names in parentheses are the types of the roles.

```
> let "symbol table retrieve" be a specialization of retrieve,
  such that the universe is a symbol table, and the key function is symbol.
```

The starting point for the program development which culminates in the LOOKUP procedure a specialization of Retrieve, in which the Universe is the symbol table designed in the preceding section, and the key function is the function which extracts the symbol component of an entry.

In traditional terms, Symbol-table-retrieve would be called the specifications for LOOKUP. In the framework of this thesis, however, the usual distinction between specifications and programs as separate formalisms does not exist. What we have in general is plans at various levels of abstraction. The topmost plan often amounts to what would normally called a specification, and the bottommost (surface) plan is certainly what would be called an implementation. All of these descriptions are in the same language, and there are implementation relationships between the intermediate plans also. Furthermore, in this framework there is no reason to restrict a user's starting plan to being an input-output specification or test with one input situation and one or two output situations. The most natural top level description of a program may also be a multi-step plan.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: SYMBOL-TABLE-RETRIEVE (RETRIEVE)
        1. ANY-COMPOSITE
        2. DISCRIMINATE+RETRIEVE
  ? 2
```

In the interaction above, the system has searched the plan library for ways of implementing Symbol-table-retrieve (i.e. for overlays with Retrieve as their right hand side). In the current library, there are two: the default implementation as Any, and the implementation in which the universe is implemented as a keyed discrimination (see Fig. 6-3). These are presented as options to the user, who chooses the second.

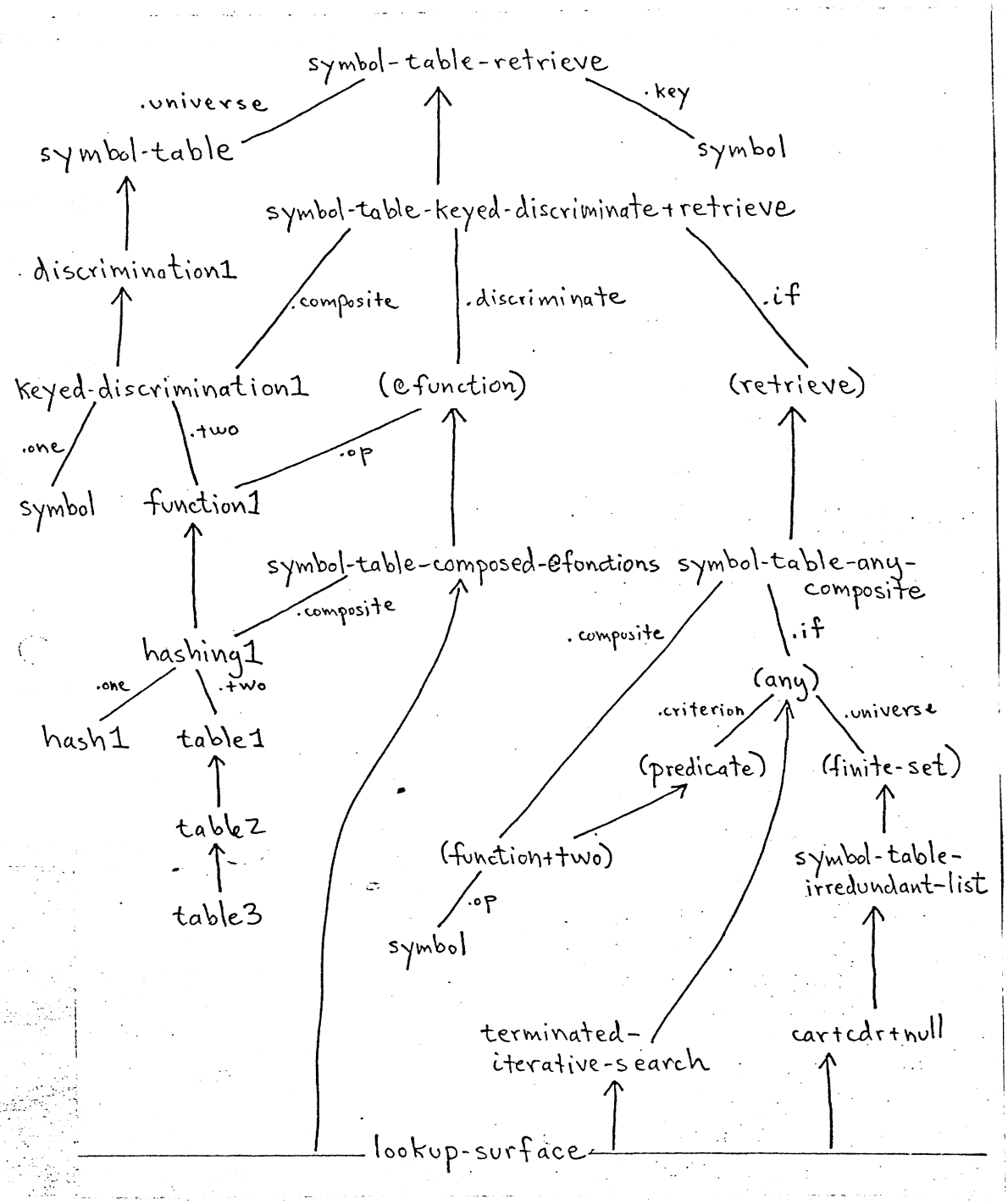


Figure 6-2. Implementation of Symbol Table Retrieval.

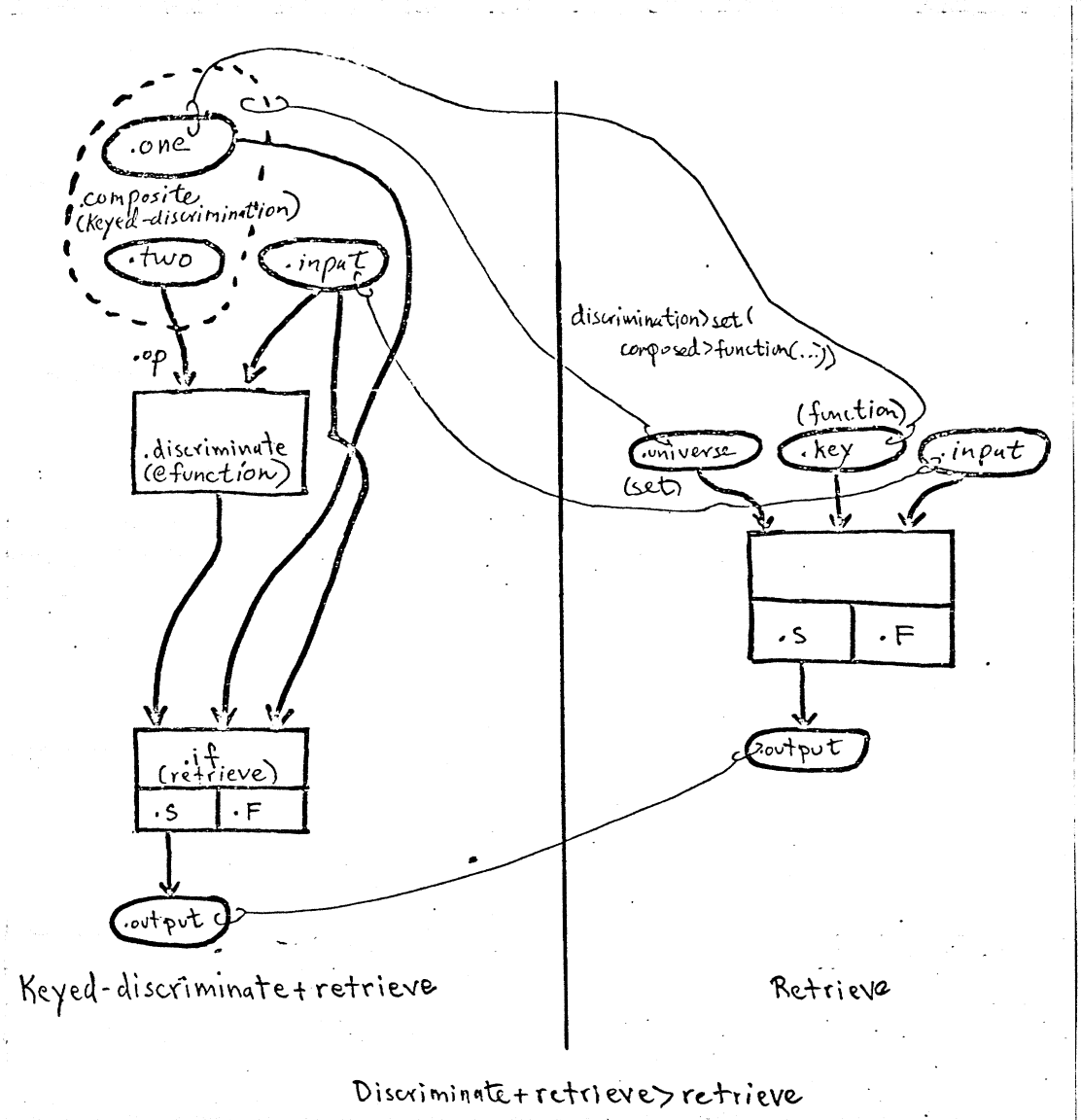


Figure 6-3. Associative Retrieval from a Keyed Discrimination.

The system at this point could be more clever and conclude that since Symbol-table has already been implemented as Keyed-discrimination1, therefore the second choice was indicated. However, this degree of automation in general may be more difficult, particularly in the presence of multiple views. In any case, once option two is chosen, either by the system or the user, the maximal sharing heuristic makes sure that Keyed-discrimination1 does become part of the implementation plan, as shown below.

```
LET "SYMBOL-TABLE-KEYED-DISCRIMINATE+RETRIEVE" BE THE IMPLEMENTATION
OF SYMBOL-TABLE-RETRIEVE AS KEYED-DISCRIMINATE+RETRIEVE.

ELIBIBLE SHARING FOR: SYMBOL-TABLE-KEYED-DISCRIMINATE+RETRIEVE.COMPOSITE
                                        (KEYED-DISCRIMINATION)
      1. KEYED-DISCRIMINATION1
? )
```

The system has created a specialized version of the plan Keyed-discriminate+retrieve, (wherein the keyed discrimination is Keyed-discrimination1) which implements Symbol-table-retrieve (see Fig. 6-2).

```
> )
APPLICABLE IMPLEMENTATIONS FOR:
              SYMBOL-TABLE-KEYED-DISCRIMINATE+RETRIEVE.DISCRIMINATE(@FUNCTION)
      1. COMPOSED-@FUNCTIONS
? )

LET "SYMBOL-TABLE-COMPOSED-@FUNCTIONS" BE THE IMPLEMENTATION OF
SYMBOL-TABLE-KEYED-DISCRIMINATE+RETRIEVE.DISCRIMINATE AS COMPOSED-@FUNCTIONS.

ELIGIBLE SHARING FOR:
              SYMBOL-TABLE-COMPOSED-@FUNCTIONS.COMPOSITE (COMPOSED-FUNCTIONS)
      1. HASHING1
? )
```

Since there are no overlays for implementing Symbol-table-keyed-discriminate+retrieve as a whole, the system proposes applicable overlays for the roles, beginning with the Discriminate role, which is constrained to be an instance of @Function. There is only one plan in the library for implementing @Function, that is as a composition of two other instances of @Function. Using the maximal sharing heuristic again, these become the application of the hash function, Hash1, followed by fetching from the hash table, Table1.

```
> )
APPLICABLE IMPLEMENTATIONS FOR:
              SYMBOL-TABLE-KEYED-DISCRIMINATE+RETRIEVE.IF (RETRIEVE)
      1. ANY-COMPOSITE
      2. DISCRIMINATE+RETRIEVE
? 1

LET "SYMBOL-TABLE-ANY-COMPOSITE" BE THE IMPLEMENTATION OF
SYMBOL-TABLE-KEYED-DISCRIMINATE+RETRIEVE.IF AS ANY-COMPOSITE.
```

Implementation of the other role (If) of Symbol-table-keyed-discriminate+retrieve is shown above. This role is an instance of Retrieve applied to the bucket obtained by Discriminate. As before, the system presents two options for implementing Retrieve. This time the user chooses the first option: retrieval from the bucket is implemented as Any in which the criterion is a composite (see Function+two in Chapter Four) of the key function (Symbol) and the Input to Retrieve.

The plan **Any-composite** is shwon in Fig. 6-4. An instance of Retrieve can be implemented as an instance of Any in which the Criterion predicate has a definiton of the following form.

$$P(x) \equiv F(x,K)$$

This way in general of constructing a predicate, P, for a given function, F, and a value, K, is formalized as the overlay Function+value>predicate in the appendix. In the implementation of Retrieve as Any, F corresponds to the key function of Retrieve and K is the input key. This overlay gives a default way of implementing Retrieve in terms of Any. For example, if the Universe set is implemented as a list, Retrieve can be implmented as a CAR-CDR search loop.

Loop Synthesis

We now come to the point in the synthesis where we begin to introduce loops into the design. The job of making sure that the loop implementations of different parts of a program are combined into a single loop when possible may require more specific expertise than is currently conceived of as part of the maximal sharing heuristic. A temporal synthesis expert module will therefore likely need to be written to realize this part of scenario.[1]

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: SYMBOL-TABLE-ANY-COMPOSITE.IF (ANY)
        1. TERMINATED-ITERATIVE-SEARCH
  ? )
```

To begin, Any is implemented as Terminated-iterative-search (see figure in Chapter Five).

```
> )
  APPLICABLE IMPLEMENTATIONS FOR:
                SYMBOL-TABLE-ANY-COMPOSITE.IF.UNIVERSE (FINITE-SET)
        1. LIST
        2. SEQUENCE
        3. LABELLED-THREAD
        4. DISCRIMINATION
  ? 1
```

1. Waters has written a module which does part of this work.

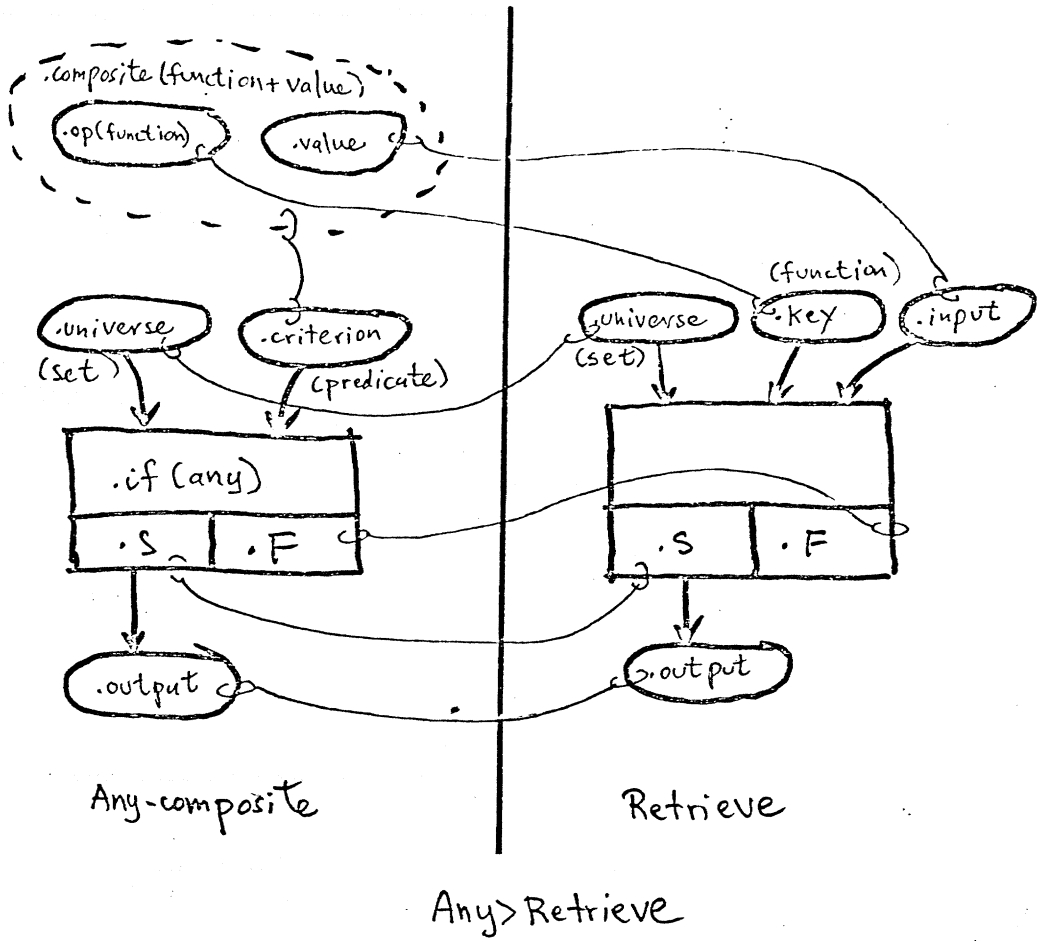


Figure 6-4. Default Implementation of Associative Retrieval.

```
SPECIALIZATIONS OF: LIST
      1. IRREDUNDANT-LIST
? )

LET "SYMBOL-TABLE-IRREDUNDANT-LIST" BE THE IMPLEMENTATION OF
SYMBOL-TABLE-ANY-COMPOSITE.IF.UNIVERSE AS AN IRREDUNDANT-LIST.
```

The Universe of Any must then be implemented as a loop augmentation which generates the inputs to the second exit test of this search loop. This takes place in two steps under user guidance. In the first step above the set is implemented as a list (without duplicates). As earlier in this scenario, the system could also be more clever here and save the user some effort. In particular, the system should realize that the finite sets which fill Symbol-table-any-composite.If.Universe are the buckets of Table1, which were implemented earlier as irredundant lists.

```
> )
APPLICABLE IMPLEMENTATIONS FOR:
              SYMBOL-TABLE-IRREDUNDANT-LIST (IRREDUNDANT-LIST)
      1. UPPER-SEGMENT
      2. DOTTED-PAIR
      3. LIST-GENERATION
? 3

EXTENSIONS OF: LIST-GENERATION
      1. TRUNCATED-LIST-GENERATION
? )

SPECIALIZATIONS OF: TRUNCATED-LIST-GENERATION
      1. CAR+CDR+NULL
? )
```

Finally, the irredundant lists of entries are implemented as the standard Lisp CAR, CDR and NULL loop.

Code Generation

The final phase of the top-down synthesis scenario is code generation. The user has guided the implementation of all parts of the program down to the level of applications of the primitive functions of the programming language and tests of the primitive predicates, connected by data flow and control flow. All that remains now is for a special purpose procedure, the surface plan synthesizer, to gather up the constraints between all the terminal nodes of Fig. 6-2 into the form of a single surface plan. A lot of detail is suppressed in this figure, but the result in the end is essentially the same surface plan for LOOKUP as shown in Chapter Five. This surface plan is then turned over to a code generator, which implements the data and control flow using the available connective primitives in the programming language. A code generator for Lisp has been implemented by Waters as part of the Programmer's Apprentice project.

```
> show code for symbol-table-retrieve

        (DEFINE SYMBOL-TABLE-RETRIEVE
           (LAMBDA (TABLE INPUT)
               (PROG (BUCKET OUTPUT)
                     (SETQ BUCKET (ARRAYFETCH TABLE (HASH INPUT)))
                  LP (COND ((NULL BUCKET)(RETURN NIL)))
                     (SETQ OUTPUT (CAR BUCKET))
                     (COND ((EQ (CAR OUTPUT) INPUT)
                            (RETURN OUTPUT)))
                     (SETQ BUCKET (CDR BUCKET))
                     (GO LP))))
```

At the end of all of this the code as shown in Chapter Two and Chapter Five is produced, perhaps with some minor syntactic variations due to the stylistic biases of the code generatyion algorithm.

Synthesis of Insert

This section shows the synthesis of a procedure to add entries to the symbol table. Two new points are introduced in this example. First, the plans in this implementation involve side effects. Second, the user intervenes at a key point in the development in order to suggest a reanalysis which leads the system to the desired program. An overview of the implementation is shown in Fig. 6-5.

```
> let "symbol table add" be a specialization of set add by side effect
  such that the old set is a symbol table, and the input does not belong to
  the old set.
```

The starting plan for this synthesis is a specialization of the input-output specification #Set-add,[1] in which the old set is a symbol table implemented earlier. #Set-add is a specialization of Set-add in which the new set has the same identity (but different members) as the old set. The role names "old" and "new" then refer to the state of the same object before and after the side effect operation, rather than to different objects. The representation of side effects will be specified in more detail in Chapters Eight and Ten. Note that the user above has also specified as a precondition that the entry to be added is not already in the table, which simplifies the implementation.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: #SYMBOL-TABLE-ADD (SET-ADD)
        1. PUSH
        2. INTERNAL-THREAD-ADD
        3. DISCRIMINATE+ACTION+UPDATE
  ? 3

  LET "#SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE" BE THE IMPLEMENTATION OF
  #SYMBOL-TABLE-ADD AS DISCRIMINATE+ACTION+UPDATE, SUCH THAT
  #SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE.ACTION IS SET ADDITION,
  AND #SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE.UPDATE IS BY SIDE EFFECT.
```

---

1. The character "#" is intended to be read as "impure". Thus #Set-add is "impure set add" or "set add by side effect".

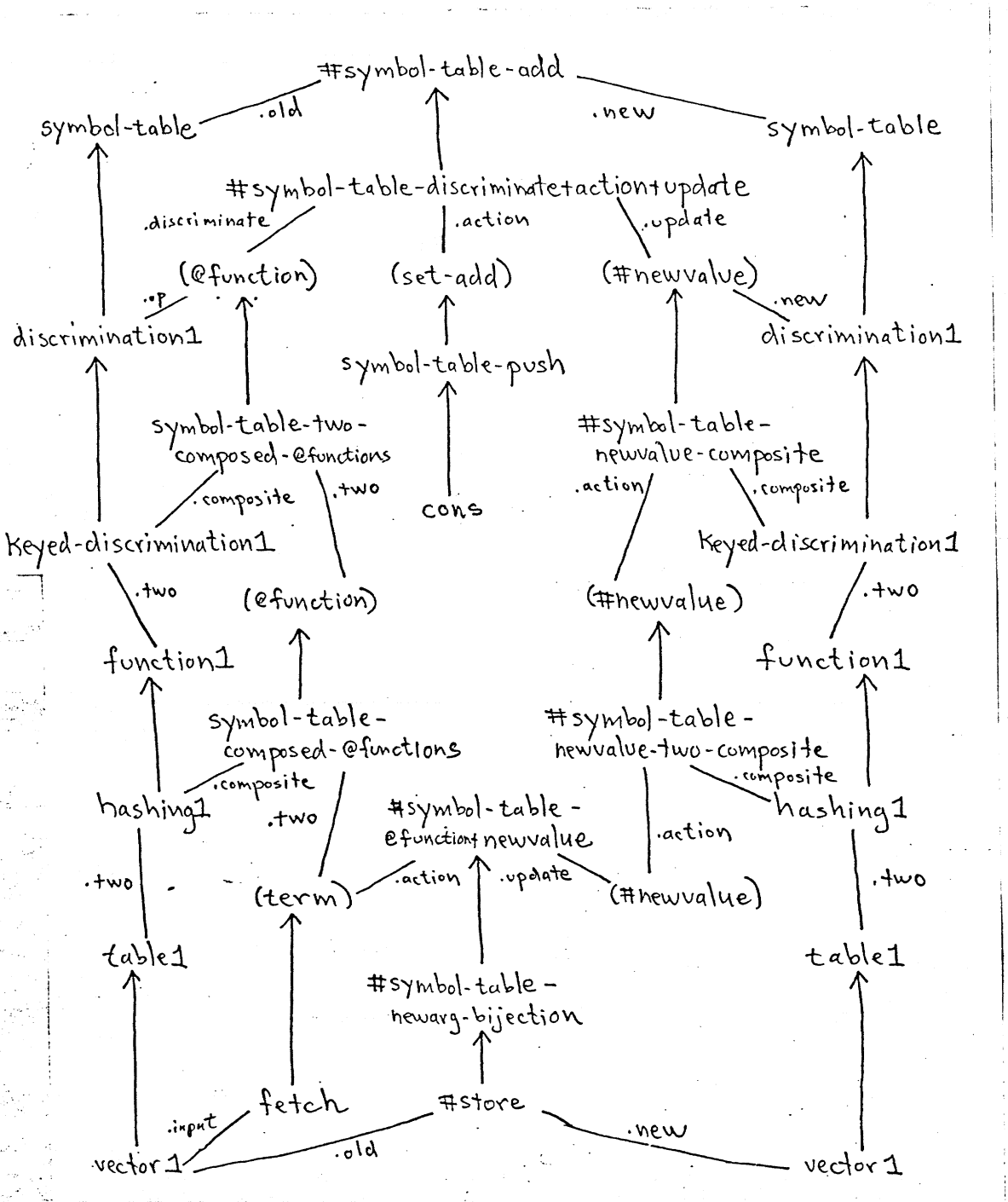


Figure 6-5. Implementation of Addition to Symbol Table.

The system begins by retrieving three possible implementations for #Symbol-table-add. The first two are implementations of Set-add for sets implemented as lists or labelled threads; the third is the implementation of Old+input+output-set (of which Set-add is a specialization) for sets implemented as discriminations, shown in Fig. 6-6. The user chooses the third option,[1] and the system responds, as usual by specializing the left hand side plan appropriately.

Notice that the overlay in Fig. 6-6 is between two plans in which no commitment has yet been made as to whether or not side effects are involved. One of the pre-computed properties of this overlay, however, is that an Update by side effect on the left hand side (i.e. #Newvalue) corresponds to a Set-add or Set-remove by side effect on the right hand side.

Since there are no overlays for #Symbol-table-discriminate+action+update as a whole, we move on to implementing the roles separately. The Discriminate role is an instance of @Function, in which Discrimination1 is function applied (Op). The further implementation of this role is simply a two level composition of instances of @Function which mirrors the decomposition of Discrimination1 into Symbol, Hash1 and Table1. This is shown in Fig. 6-5, but omitted from the scenario transcript here.

```
> )
APPLICABLE IMPLEMENTATIONS FOR:
                #SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE.ACTION (SET-ADD)
      1. PUSH
      2. INTERNAL-THREAD-ADD
      3. DISCRIMINATE+ACTION+UPDATE
? 1

LET "SYMBOL-TABLE-PUSH" BE THE IMPLEMENTATION OF
#SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE.ACTION AS PUSH.
```

Set-add operations on the buckets (which are implemented as irredundant lists) are implemented by Push operations.

```
> )
APPLICABLE IMPLEMENTATIONS FOR:
                #SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE.UPDATE (#NEWVALUE)
      1. NEWVALUE-COMPOSITE
? )

LET "NEWVALUE-COMPOSITE" BE THE IMPLEMENTATION OF
#SYMBOL-TABLE-DISCRIMINATE+ACTION+UPDATE.UPDATE AS NEWVALUE-COMPOSITE
BY SIDE EFFECT.
```

---

1. As discussed earlier, if the system assumes the same set is not being implemented two different ways, it could choose this option on its own initiative. However, some clever implementations actually do involve implementing the same abstract data structure simultaneoulsy two different ways.

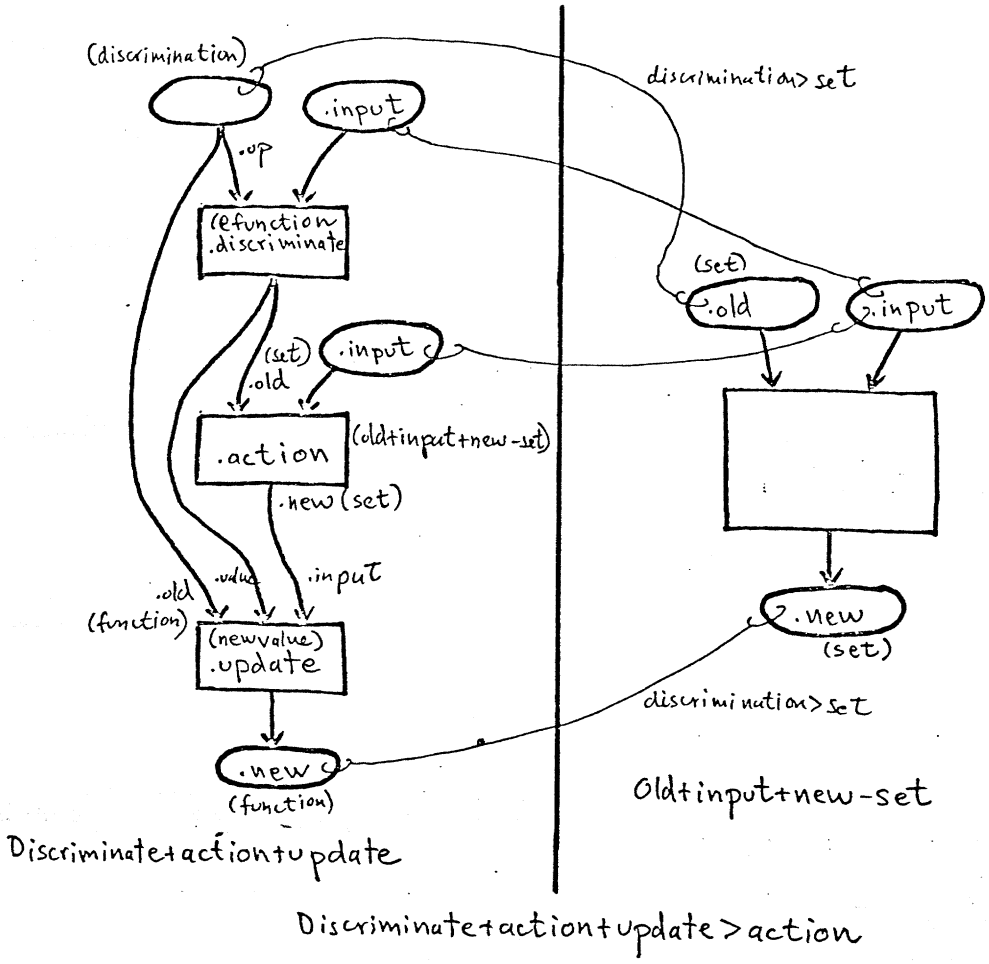


Figure 6-6. Adding and Removing Members from a Discrimination.

As discussed in Chapter Four, Newvalue operations on a function implemented as a composition can be implemented by Newvalue operations on the second component only. Furthermore, a property of this implementation, is that if the operation on the second component is by side effect equivalent, then in effect the composed function has been modified. In this case, #Newvalue operations on Discrimination1 are implemented as #Newvalue operations on Function1. Similarly (see Fig. 6-5, but not shown here), #Newvalue operations on Function1 are implemented as #Newvalue operations on Table1. This completes implementation of all roles of #Symbol-table-discriminate+action+update.

```
> )
  APPLICABLE OVERLAYS FOR: SYMBOL-TABLE-PUSH (PUSH)
        1. BUMP+UPDATE
        2. CONS
  ? 2

> )
  APPLICABLE OVERLAYS FOR: SYMBOL-TABLE-COMPOSED-@FUNCTIONS.TWO (TERM)
        1. FETCH
  ? )
```

Prompted by the user, the system continues to suggest overlays for implementing the parts of the design which are not yet down to the level of Lisp primitives. The two simple steps shown above are: to implement Symbol-table-push as CONS, corresponding to the implementation of the buckets of the table as Lisp lists; and to implement Term applied to Table1 as ARRAYFETCH, corresponding to the implementation of Table1 as Vector1.[1] Other simple steps, omitted here, are the implementation of the application of Symbol as CAR, and the application of Hash1 as a procedure call. This leaves only #Newvalue applied to Table1 (see Fig. 6-5) to be implemented further.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR:
                #SYMBOL-TABLE-NEWVALUE-TWO-COMPOSITE.ACTION (#NEWVALUE)
        1. NEWVALUE-COMPOSITE
  ? no

> )
  NO APPLICABLE OVERLAYS.
```

Unfortunately, the only implemention in the current library for #Newvalue[2] is for a function implemented as a composition of two functions, which is not what we want for Table1. At this point the simple refinement strategy used by the system thus far is stymied. The problem is that, in order to implement #Newvalue as the simpler #Newarg (which then becomes ARRAYSTORE for Lisp vectors), the system must recognize that the function involved is one-to-one (a **Bijection**) and

---

1. This is skipping the intermediate steps of Table2 and Table3, as discussed earlier.
2. Recall that Newvalue is the specification for updating a function such that all arguments that used to map to a given value, map to a new given value.

that the argument which maps to the old value has already been computed. This is the plan **@Function+newvalue**, shown on the right hand side of Fig. 6-7.

The basic idea of the overlay in Fig. 6-7 is that in the special case of one-to-one functions, an instance of @Function followed by Newvalue, as in the Discriminate+action+udpate plan, can be implemented simply by an instance of Newarg. In other words, if you know that there is only one domain element which maps to a given range element, then updating all domain elements which map to that range element (i.e. the specifications of Newvalue) degenerates into changing the value associated with that one domain element (i.e. Newarg). Furthermore, in terms of side effects, an impure Update operation (#Newvalue) in @Function+newvalue corresponds to #Newarg.

```
> recognize @function+newvalue.

  LET "#SYMBOL-TABLE-@FUNCTION+NEWVALUE" BE A SPECIALIZATION OF
  @FUNCTION+NEWVALUE SUCH THAT
  #SYMBOL-TABLE-@FUNCTION+NEWVALUE.ACTION.OP=TABLE1 .
```

The user guides the system at this point by advising it to try to recognize an instance of the plan @Function+newvalue somewhere in the current design. Given the focus of trying to recognize only one particular plan, the system succeeds in noticing that Symbol-table-composed-@functions.Two (see Fig. 6-5 together with the #Symbol-table-newvalue-two-composite.Action satisfy the constraints of @Function+newvalue.[1] What has happened here is that parts of two different branches of the tree have been grouped together to recognize a plan which has a known implementation. This is a novel feature of this sythesis scenario as compared to the standard top-down refinement approach.

```
> )
  APPLICABLE IMPLEMENTATIONS FOR:
                #SYMBOL-TABLE-@FUNCTION+NEWVALUE (@FUNCTION+NEWVALUE)
        1. NEWARG-BIJECTION
  ? )

  LET "#SYMBOL-TABLE-NEWARG-BIJECTION" BE THE IMPLEMENTATION OF
  #SYMBOL-TABLE-@FUNCTION+NEWVALUE AS NEWARG-BIJECTION.
```

Now the overlay can be applied which implements #Newvalue as #Newarg, or in the case of a sequence, #Newterm,

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: #SYMBOL-TABLE-NEWARG-BIJECTION (#NEWTERM)
        1. #STORE
  ? )
```

and finally, as #Store.

---

1. Table1 is an irredundant sequence, which means it is a one-to-one function.

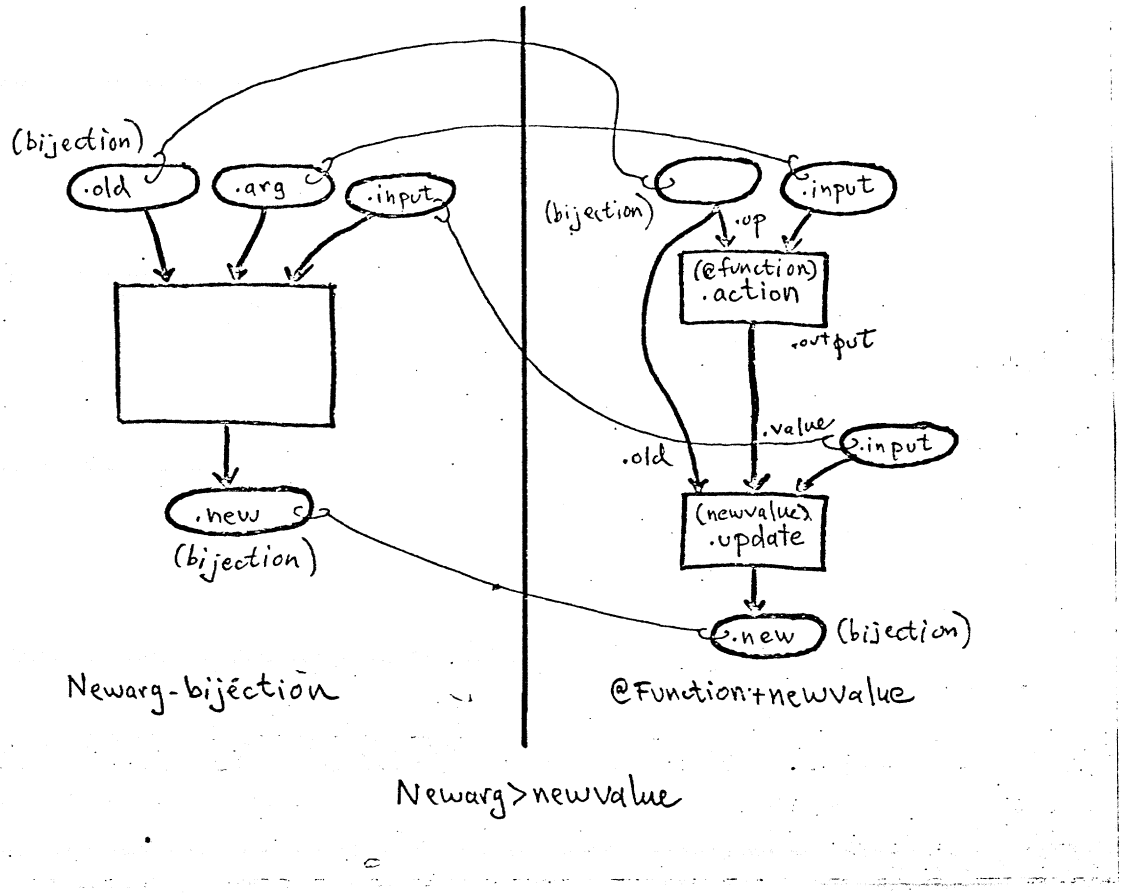


Figure 6-7. Updating a Bijection.

Code generation follows in a similar fashion to before.

```
> show code for #symbol-table-add

        (DEFINE SYMBOL-TABLE-ADD
           (LAMBDA (TABLE INPUT)                ;MODIFIES TABLE.
              (PROG (INDEX)
                    (SETQ INDEX (HASH (CAR INPUT)))
                    (ARRAYSTORE TABLE INDEX
                                (CONS (ARRAYFETCH TABLE INDEX)
                                      INPUT)))))
```

Synthesis of Delete

The last procedure to be synthesized is for associative deletion of entries in the symbol table. This procedure and its development share many features with LOOKUP and INSERT. This part of the scenario will therefore be brief and will for the most part rely on Fig. 6-8 rather than showing the detailing system-user interactions, as in the preceding sections.

```
> let "symbol table expunge" be a specialization of expunge by side effect such
  that the old set is a symbol table and the key function is symbol;
  and such that there exists a unique "x" such that x belongs to the old set
  and the key function applied to x equals the input.
```

These are the starting specifications. Notice that the deletion is by side effect, and that there is expected to be exactly one entry in the table whose Symbol is the given Lisp atom. This precondition is a standrad specialization of Expunge in the library called **Expunge-one.**

```
> )
  APPLICABLE IMPLEMENTATIONS FOR: #SYMBOL-TABLE-EXPUNGE-ONE (EXPUNGE-ONE)
        1. RESTRICT-COMPOSITE
        2. KEYED-DISCRIMINATE+EXPUNGE+UPDATE
  ? 2

  LET "#SYMBOL-TABLE-KEYED-DISCRIMINATE+EXPUNGE+UPDATE" BE THE IMPLEMENTATION
  OF #SYMBOL-TABLE-EXPUNGE-ONE AS KEYED-DISCRIMINATE+EXPUNGE+UPDATE.
```

For sets implemented as keyed discriminations, Expunge is implemented by the three step plan Discriminate+expunge+update, shown in Fig. 6-9 which is similar to Discriminate+action+update in the implementation of SYMBOL-TABLE-ADD. Like Discriminate+action+update the standard pre-compiled side effect analysis of this plan says that the side effect implementation is achieve by specializing the Update step to #Newvalue. Part of the cleverness in this synthesis example involves avoiding this step by performing the Action by side effect instead.

The first step, Discriminate, is an instance of @Function which computes the appropriate bucket from the given key. The implementation of this step uses the plan Symbol-table-composed-@functions, which was developed in the synthesis of LOOKUP (see Fig. 6-8).

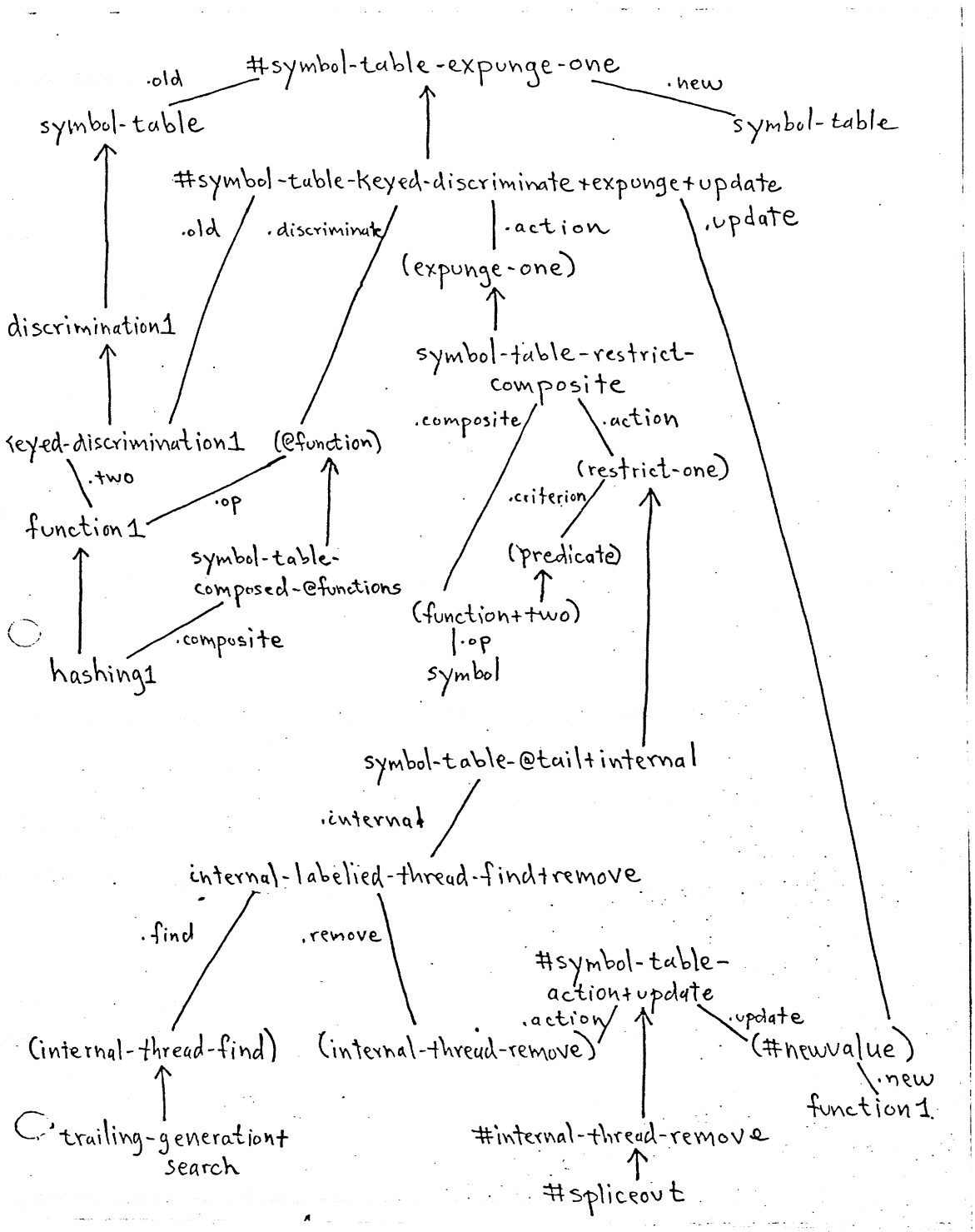


Figure 6-8. Implementation of Associative Deletion from Symbol Table.

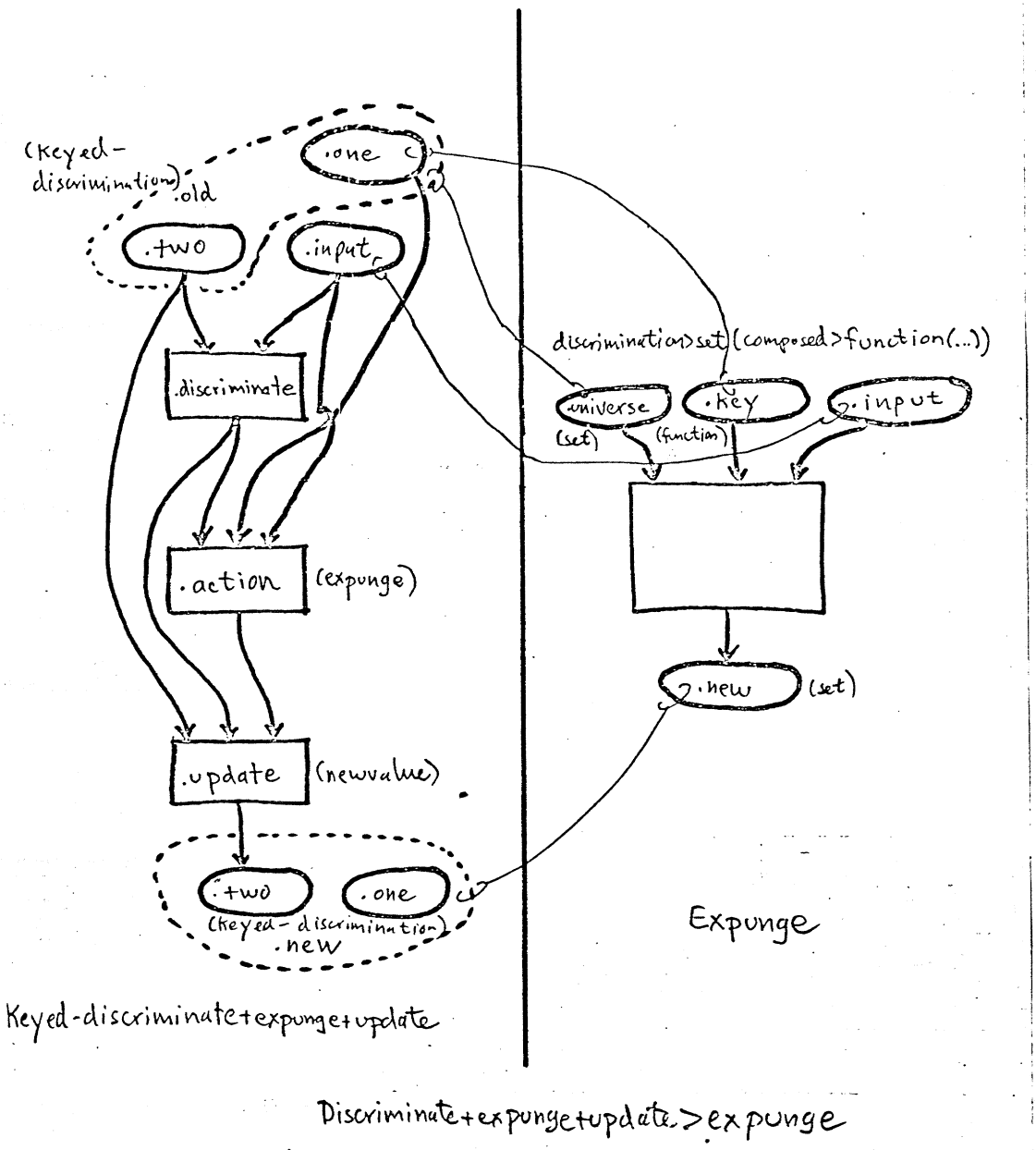


Figure 6-9. Associative Deletion from Keyed Discrimination.

```
> )
 APPLICABLE IMPLEMENTATIONS FOR:
          #SYMBOL-TABLE-KEYED-DISCRIMINATE+EXPUNGE+UPDATE.ACTION (EXPUNGE-ONE)
      1. RESTRICT-COMPOSITE
      2. KEYED-DISCRIMINATE+EXPUNGE+UPDAT
 ? 1

 LET "SYMBOL-TABLE-RESTRICT-COMPOSITE" BE THE IMPLEMENTATION OF
 #SYMBOL-TABLE-KEYED-DISCRIMINATE+EXPUNGE+UPDATE.ACTION AS RESTRICT-COMPOSITE.
```

The Expunge-one action on the buckets is implemented in the default way as Restrict, in which the criterion is a composition of the Symbol function and #Symbol-table-expunge-one.Key. This overlay is shown in Fig. 6-10. It is similar to the implementation of Retrieve as Any-composite in SYMBOL-TABLE-RETRIEVE. Furthermore, it is a property of this overlay that if the right hand side is specialized to Expunge-one, then the Action on the left hand side is correspondingly specialized to **Restrict-one**, in which there is expected to be only one member of the Universe set which satisfies the given Criterion.

```
> )
APPLICABLE IMPLEMENTATIONS FOR:
          SYMBOL-TABLE-RESTRICT-COMPOSITE.ACTION (RESTRICT-ONE)
      1. ITERATIVE-FILTERING
      2. @TAIL+INTERNAL
? 2

LET "SYMBOL-TABLE-@TAIL+INTERNAL" BE THE IMPLEMENTATION OF
SYMBOL-TABLE-RESTRICT-COMPOSITE.ACTION AS @TAIL+INTERNAL.
```

Restrict can be implemented either as a filtering loop, or as the plan **@Tail+internal**, shown in Fig. 6-11. This plan removes a member from a set implemented as an irredundant **list.**

Removing a member from a set implemented as an irredundant list breaks down into two cases: if it happens that the member to be removed is the head of the list, then removal is achieved simply by a taking the tail of the list; otherwise, viewing the list as a labelled thread, the internal node of the spine which is labelled with the given member must be found and removed. These two cases will eventually manifest themselves in the code for SYMBOL-TABLE-DELETE as follows:

```
(PROG (... BUCKET PREVIOUS)
      ...
      (COND ((EQ (CAAR PREVIOUS) INPUT)
             (... (CDR PREVIOUS))
             (RETURN NIL)))
   LP ...
      (COND ((EQ (CAAR BUCKET) INPUT)
             (RPLACD PREVIOUS (CDR BUCKET))
             (RETURN NIL)))
      ...
      (GO LP))
```

Let us first consider the overlay @Tail+internal>restrict in Fig. 6-11, which formalizes the breakdown into two cases described above. On the right hand side of this overlay we have Restrict-one, which specifies the removal of the (unique) member of a set which does not satisfy a given

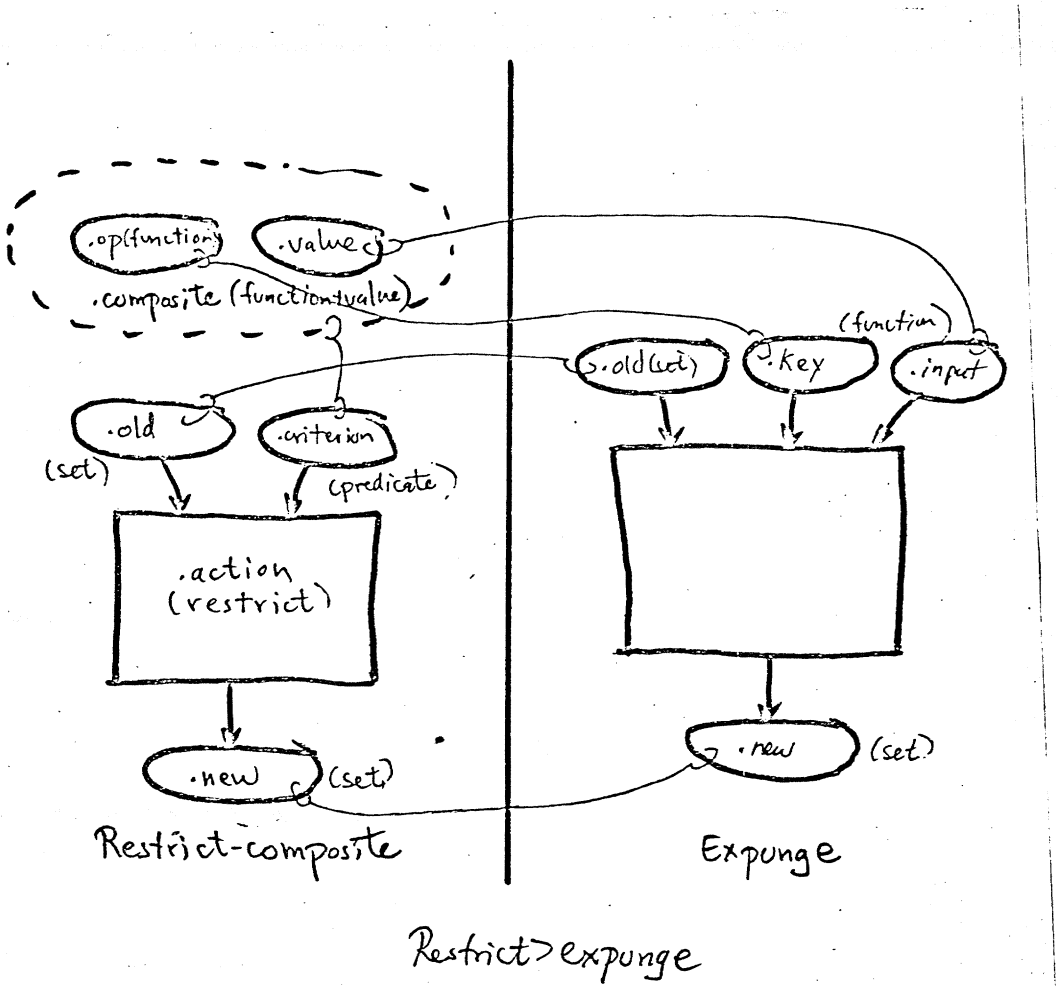


Figure 6-10. Default Implementation of Associative Deletion.

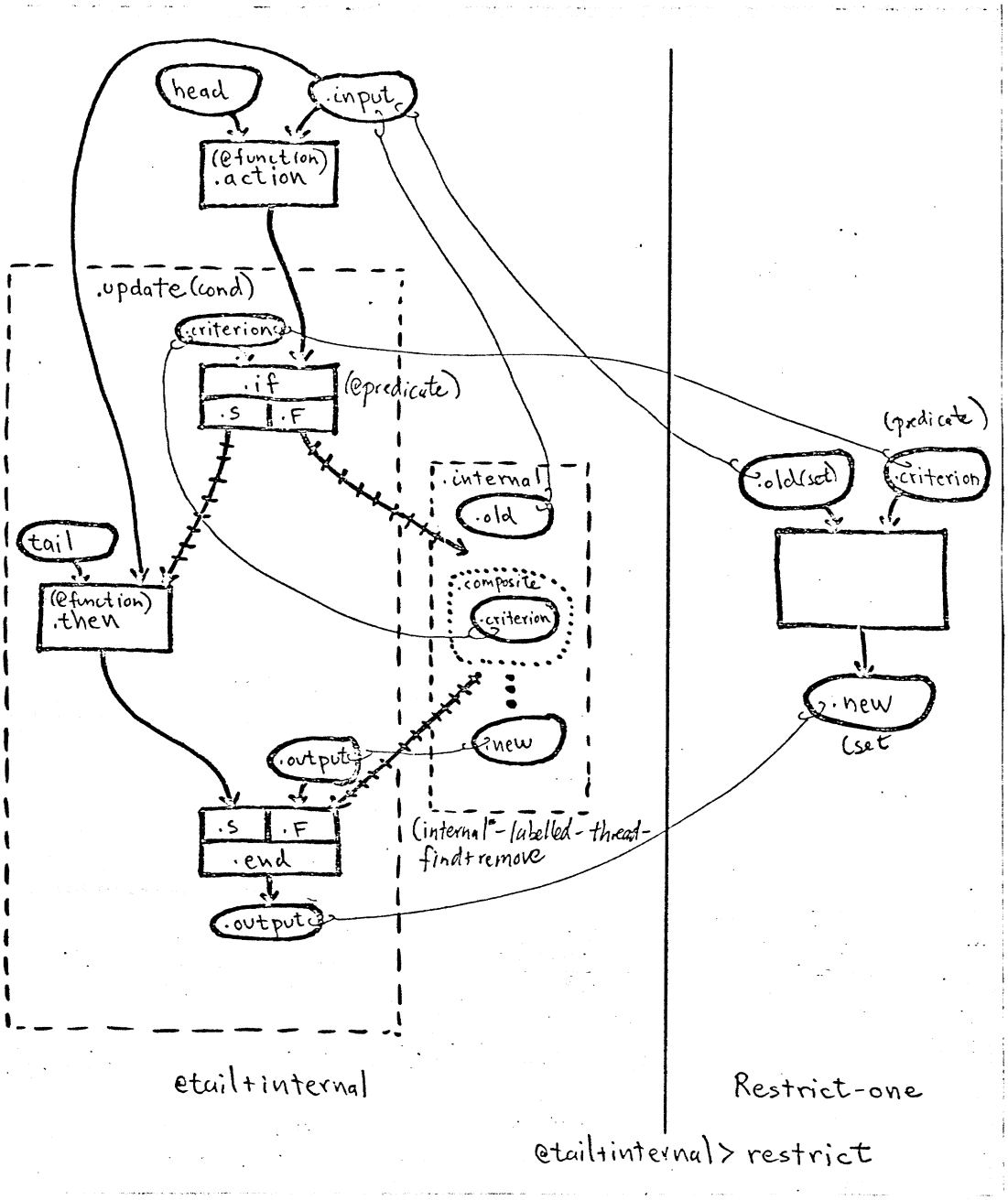


Figure 6-11. Set Removal for Irredundant Lists.

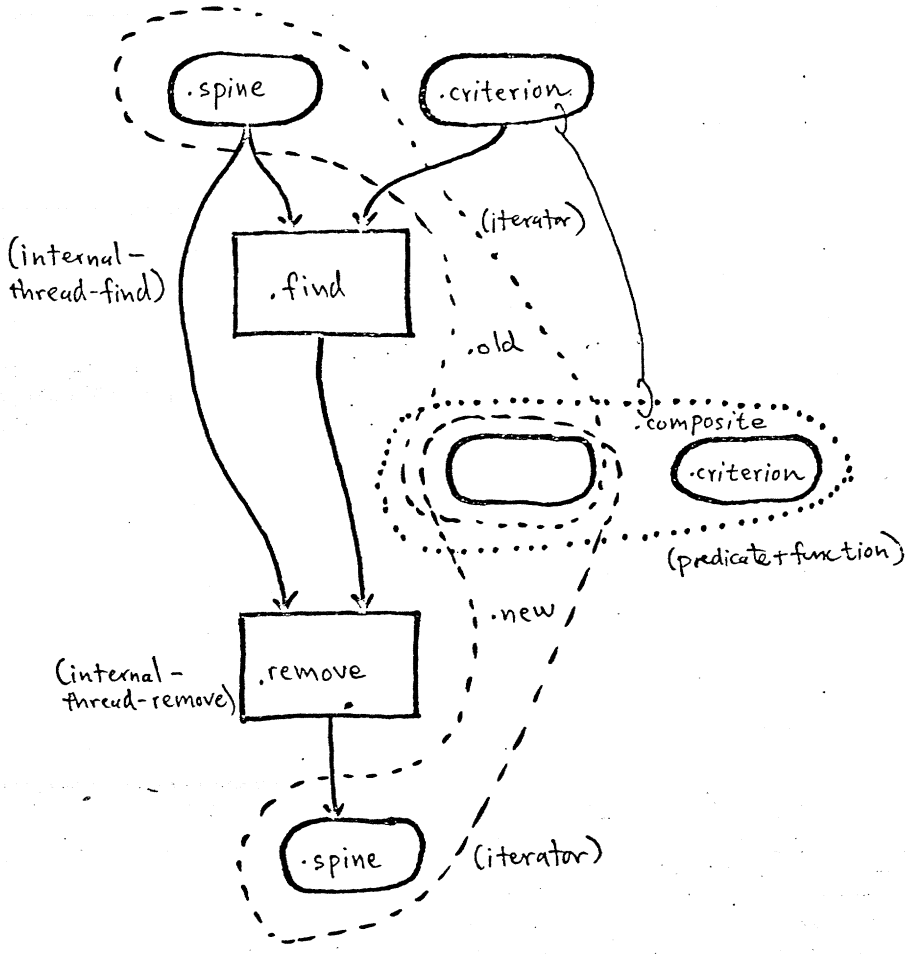


Figure 6-12. Internal Labelled Thread Find and Remove.

criterion. The essence of the plan on the left, which implements these specifications, is a conditional (Cond). The Input to the test of this conditional is the head of the irredundant list which implements the Old set; the criterion is the complement of the criterion of Resrict-one. The output of this conditional (End.output) is the irredundant list which implements the New set. In the Succeed case (i.e. when the head of the input list satisfies the given criterion), this output is the resulting of taking the tail of the input list. In the Fail case, the new list corresponds to the New labelled thread of Internal, an instance of Internal-labelled-thread-find+remove.

Internal-labelled-thread-find+remove, shown in Fig. 6-12, is an extension of Internal-thread-find+remove. In this plan, the old and new lists are thought of as labelled threads. Internal-labelled-thread-find+remove removes an internal node from the spine of a labelled thread (Old), the label of which satisfies a given predicate, resulting in a (New) labelled thread. As used in @Tail+internal, the criterion applied by Find to each node in the spine of the labelled list is composed from Update.if.criterion and the label function of the list viewed as a labelled thread, according to the overlay Predicate+function>predicate, given in the appendix. he basic idea of this construction is to test the label of each node, rather than the node itself. Thus for example, if the label function is Car (as in the case of Lisp lists), and Update.if.criterion is P, then the criterion of the Find step is Q defined as follows:

$$Q(x) \equiv P(\mathrm{car}(x))$$

```
> )
  APPLICABLE IMPLEMENTATIONS FOR:
                SYMBOL-TABLE-@TAIL+INTERNAL.INTERNAL.FIND (INTERNAL-THREAD-FIND)
        1. TRAILING-GENERATION+SEARCH
  ? )
```

Internal-labelled-thread-find+remove has two roles. The first role, which is an instance of Internal-thread-find, is implemented as a Trailing-generation+search loop, as shown in Fig. 6-14. The Universe of Internal-thread-find is the thread generated by the trailing generation, and the two outputs of the loop correspond to the two output of Internal-thread-find. This plan will eventually appear in the code for SYMBOL-TABLE-DELETE as follows, in which the function being applied by the action is Cdr, the Current object is hold in BUCKET and the Previous object in PREVIOUS.

```
(PROG (... BUCKET PREVIOUS)
      ...
   LP (SETQ BUCKET (CDR PREVIOUS))
      (COND ((...BUCKET...)
             ...PREVIOUS...BUCKET...
             (RETURN NIL)))
      (SETQ PREVIOUS BUCKET)
      (GO LP))
```

The system then proposes to implement Internal-thread-remove by splicing out, but the user intervenes to suggest a reanalysis.

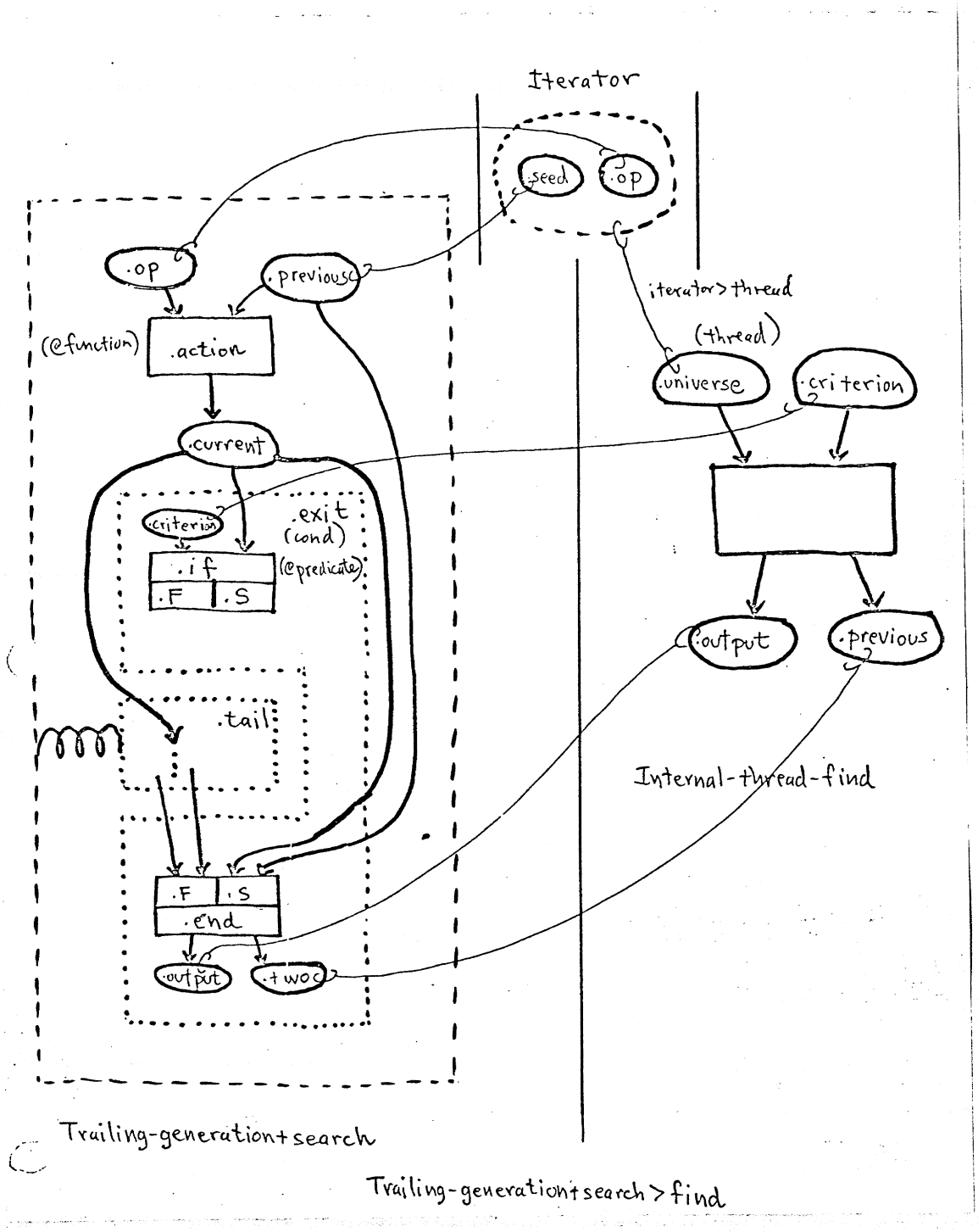


Figure 6-13. Loop to Find Location in Thread.

This page intentionally left blank.

See Fig. 6-13.

Figure 6-14. Loop to Find Location in Thread.

```
> )
APPLICABLE IMPLEMENTATIONS FOR: SYMBOL-TABLE-@TAIL+INTERNAL.INTERNAL.REMOVE
                                        (INTERNAL-THREAD-REMOVE)
      1. SPLICEOUT
? no

> recognize #action+update.

LET #SYMBOL-TABLE-ACTION+UPDATE BE A SPECIALIZATION OF #ACTION+UPDATE
SUCH THAT #SYMBOL-TABLE-ACTION+UPDATE.UPDATE.OLD=FUNCTION1 .
```

In contrast to SYMBOL-TABLE-ADD, where advice from the user was crucial to completing the synthesis, this intervention is merely to cause the system to come out with a more efficient program. In particular we want the system to realize that, if Internal-thread-remove is implemented by side effect, then in the case when the member of the bucket to be deleted is not the Head, the operation to update the table is not necessary. This piece of implementation knowledge is represented in the library by the overlay #Old+input+new>action+update, which will be discussed further in Chapter Ten. The basic idea of this overlay, however, is that in general, modifying a range element amounts to modifying the function. In order to apply this overlay, however, the system must first group together parts of plans on different branches of the tree (see Fig. 6-8), as was the case in the synthesis of SYMBOL-TABLE-ADD.

Thus the system implements the Internal.Remove step of Symbol-table-@tail+internal as #Internal-thread-remove, which is further implemented as #Spliceout, as shown in Fig. 6-16.

Spliceout has four roles: Old and New, which are iterators with the same seed; Bump, which is an instance of Apply; and Splice, which is an instance of Newarg. The purpose of Bump is to get the successor of the node to be removed, which becomes the Input of Splice. The Arg of Splice is the predecessor of the Input of Bump (which typically comes from an instance of Internal-thread-find). The Op of the old iterator (e.g. Cdr for Lisp lists) is the Op input to both Bump and Splice; the Op of the new iterator is the output of Splice. This plan will eventually emerge as the following code in SYMBOL-TABLE-DELETE.

```
(RPLACD PREVIOUS (CDR BUCKET))
```

Spliceout implements Internal-thread-remove as described by the overlay **Spliceout>remove**, shown in Fig. 6-16. The old iterator implements the old thread, and the new iterator implements the new thread. The node being deleted is the Input of Bump. Notice that the Arg input to Splice in the Spliceout plan (the predecessor of the node deleted) has no corresponding object on the right hand side of the overlay. This means that as far as this overlay is concerned, some other part of the program surrounding an instance of the left hand side (e.g. the Internal-thread-find) must provide an Arg input to Splice which satisfies the successor constraint. In other words this is an implementation of Internal-thread-remove for the case when we already know the location of the node to be removed.

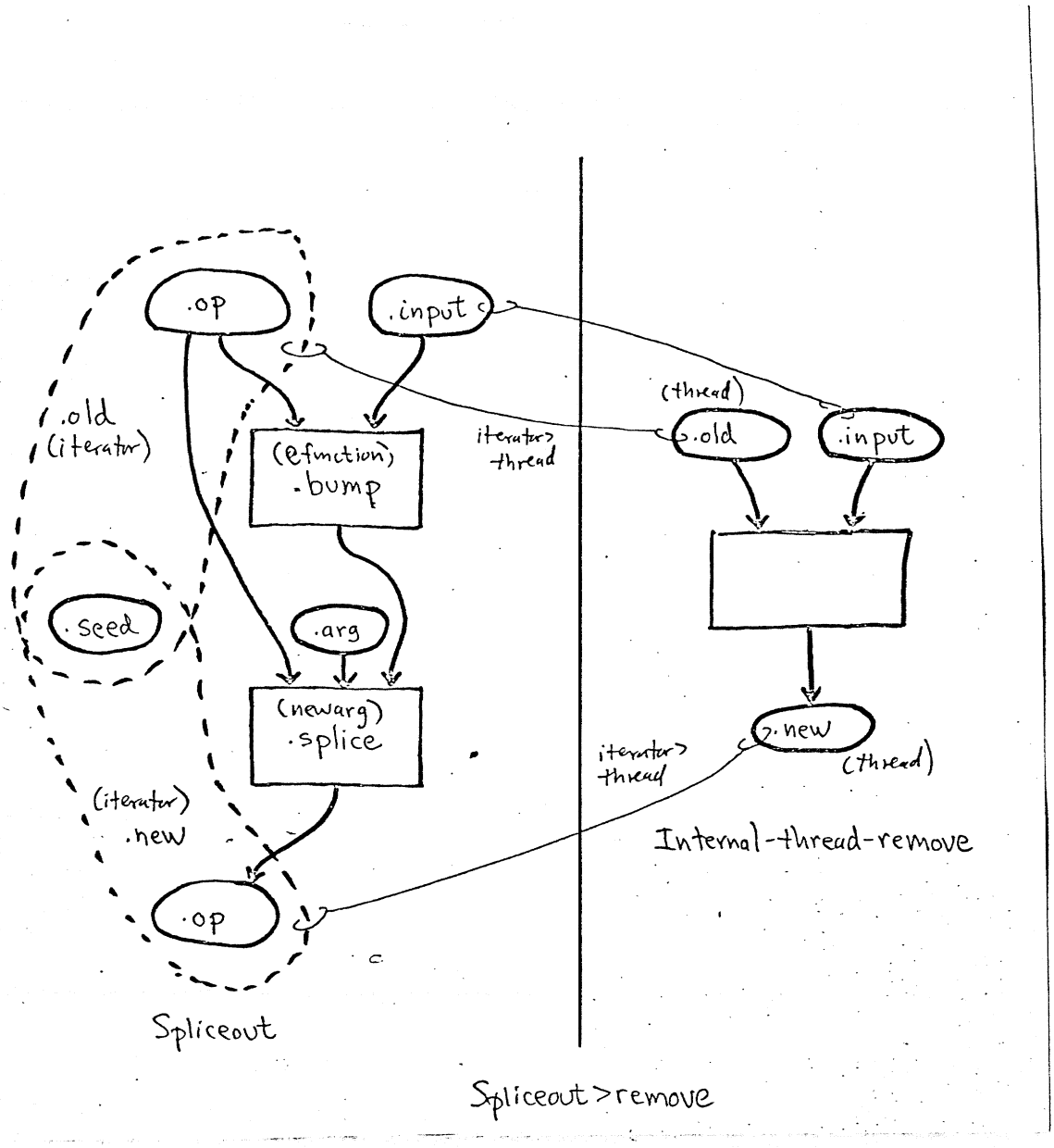


Figure 6-15. Removing from a Thread by Splicing Out

This page is intentionally left blank.

See Fig. 6-15

Figure 6-16. Removing from a Thread by Splicing Out

By the further rearrangement and straightforward implementation steps, we arrive finally at a surface plan which can then be turned over to the code generator. The resulting code is essentially the same as in the scenario of Chapter Two.

```
> show code for #symbol-table-expunge-one.

        (DEFINE SYMBOL-TABLE-DELETE
           (LAMBDA (TABLE INPUT)
              (PROG (INDEX BUCKET PREVIOUS)
                    (SETQ INDEX (HASH INPUT))
                    (SETQ PREVIOUS (ARRAYFETCH TABLE INDEX))
                    (COND ((EQ (CAAR PREVIOUS) INPUT)
                           (ARRAYSTORE TABLE INDEX (CDR PREVIOUS))
                           (RETURN NIL)))
                 LP (SETQ BUCKET (CDR PREVIOUS))
                    (COND ((EQ (CAAR BUCKET) INPUT)
                           (RPLACD PREVIOUS (CDR BUCKET))
                           (RETURN NIL)))
                    (SETQ PREVIOUS BUCKET)
                    (GO LP))))
```

CHAPTER SEVEN

VERIFICATION BY INSPECTION

Verification has two aspects: verifying correctness and detecting errors. Inspection methods are applicable in both of these areas.

In the area of verifying correctness, the basic idea is to use only implementation steps that are already known to be correct. The overall correctness of a program then follows directly from the correctness of the building blocks. This implies that the maintainer of the plan library must convince themself that each overlay in the library is correct.

Verifying Overlays

As will be explained further in Chapter Eight, an overlay is formally a function from instances of the plan on the left hand side to instances of the plan on the right hand side. For an overlay to be correct means formally that this function and its inverse must be *total* (i.e. it and its inverse must be defined on all elements of the domain and range). Practically speaking, the effect of this definition of correctness is to force all of the conditions required for the correct use of an overlay to be explicitly stated in the constraints of the plans on both sides.

The verification of overlays is important for this thesis only in that it is possible. Chapter Eight specifies the logical foundations of the plan calculus within which it is possible to verify overlays to whatever degree of rigor is warranted, up to and including a step-by-step formal proof in first order logic (which might be mechanically checked). Note however that these proofs can be quite difficult and idiosyncratic, depending as they do on the mathematical properties of the various programming domains involved; but this is as expected. The basic idea of inspection methods is to take advantage of this effort by re-using these standard forms in new situations.

Near-Miss Recognition

An inspection method for error detection is called *near-miss* recognition. In near-miss recognition, most but not all of the constraints of a plan are satisfied. If part of a user's design almost matches a plan in the library, the discrepancy between the two descriptions can be brought to the user's attention as a potential error. This method of error detection thus makes use of the standard correct plans in the library to detect errors, rather than explicitly adding a taxonomy of errors to the "grammar" as in Ruth.

Like all inspection methods, error detection by inspection is not as powerful as more general methods. However, it has the advantage that potential errors are characterized in terms which are closer to the engineering vocabulary of the user's design. The remainder of this chapter gives an example in detail. The method described in this example has not yet been implemented, however the implementation of an algorithm for near-miss pattern matching using the plan library of this thesis is currently in progress by Brotsky.

In the scenario of Chapter Two, the user typed in the following code for finding an element in a list satisfying a given criterion, and splicing it out.

```
(DEFINE BUCKET-DELETE
   (LAMBDA (BUCKET INPUT)                ;MODIFIES BUCKET.
      (PROG (P Q)
            (SETQ P BUCKET)
         LP (COND ((EQUAL (CAAR P) INPUT)
                   (RPLACD Q P)          ;SPLICE OUT.
                   (RETURN BUCKET)))
            (SETQ Q P)
            (SETQ P (CDR P))
            (GO LP))))
```

There were two errors detected in this code: one in the loop that finds the element, and one in the splicing out. This section discusses only the detection of the first error.

The first step in detecting an error is to translate the code above into the plan calculus as discussed in Chapters Four and Five. The surface plan for the loop part of BUCKET-DELETE (not including the splice out after the loop) is shown on the left of Fig. 7-1. To make this example easier to follow, the surface plan shown in the figure has been simplified by omitting the control flow arcs (since the important recognition in this example depends on the data flow), and by assuming that the code (EQUAL (CAAR P) INPUT) has already been analyzed as the testing of P by a composite predicate made up out of the Eq relation, the Caar function and INPUT.[1]

The detection of the error in this loop plan has three steps. In the first two steps, the plans Trailing-search and Iterative-generation are recognized (see Chapter Five for a discussion of recognition). After a successful recognition, the system in general attempts to recognize any specializations or extensions of the recognized plan in the library. In this example, Trailing-generation+search is an extension of both Trailing-search and Iterative-generation in which a data flow arc is added between the output of the Action step of Iterative-generation and the input of the Exit test of Trailing-search.

---

1. See Binrel+two>predicate and Predicate+function>predicate in appendix.

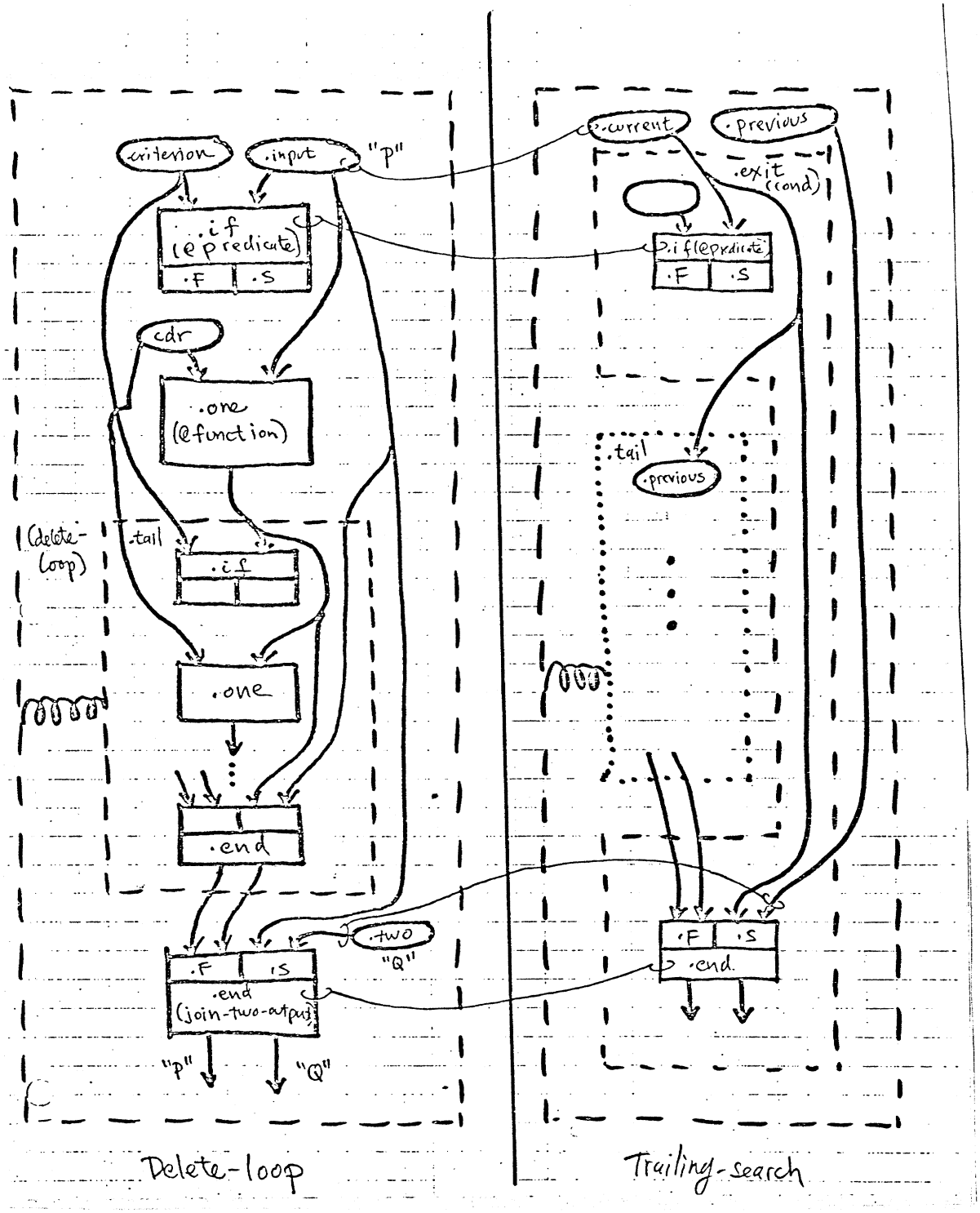


Figure 7-1. Recognizing Trailing Search in Bucket-Delete.

In the third step, the system tries to recognize Trailing-generation+search in the surface plan -- and finds that the required data flow is missing. The system then decides to consider this as a near-miss (rather than just a failure to match) and warns the user. The exact criteria for distinguishing in general between near-misses and failures to match will have to be determined experimentally.

Fig. 7-1 illustrates the recognition of Trailing-search in the surface plan for BUCKET-DELETE. **Trailing-search,** shown on the right hand side of the figure, is a loop plan with four roles: Exit, Tail, Current and Previous. As in all loop plans,[1] the recursively defined role is called Tail. The Exit role is a conditional plan which groups together the exit test (Exit.If) of the loop and the join (Exit.If) "on the way up". If the exit test succeeds, the loop terminates (and the input to the test is available through the join as an output of the loop); otherwise it continues. The Current and Previous roles are what make this plan a *trailing* loop. The Current object on each iteration is the same as the Previous object on the next iteration (Tail.Previous). The Current object in a trailing search loop is the input to the test; and both the Current and the Previous object are available through the join as outputs of the loop.

Delete-loop can be analyzed as Trailing-search by corresponding Delete-loop.If with Trailing-search.Exit.If (in which case P in the code corresponds to the Current object), and corresponding Delete-loop.End with Trailing-search.Exit.End (in which case Q in the code corresponds to the Previous object).

Fig. 7-2 illustrates the recognition of Iterative-generation in the surface plan for BUCKET-DELETE. Iterative-generation is the plan for repeatedly applying a given function (the same function each time) to the output of the preceding application of that function. This plan has two roles: Action and Tail. Action is an instance of @Function, in which the function is applied; Tail is the standard recursive invocation. Delete-loop can be analyzed as Iterative-generation by corresponding Delete-loop.One with Iterative-generation.Action, as shown in the figure.

In the third and final step of this scenario, the system attempts to recognize Trailing-generation+search. This plan as five roles: Exit, Action, Tail, Current Previous. Exit, Current and Previous are constrained as in Trailing-search; Action is constrained as in Iterative-generation; Tail is recursively defined. In addition, there is a data flow constraint between Action.Output and Exit.Test.Input. This data flow is not found in Delete-loop, leading to the following error message (from the scenario in Chapter Two.)

---

1. A detailed taxonomy of loop plans is given in Chapter Nine.

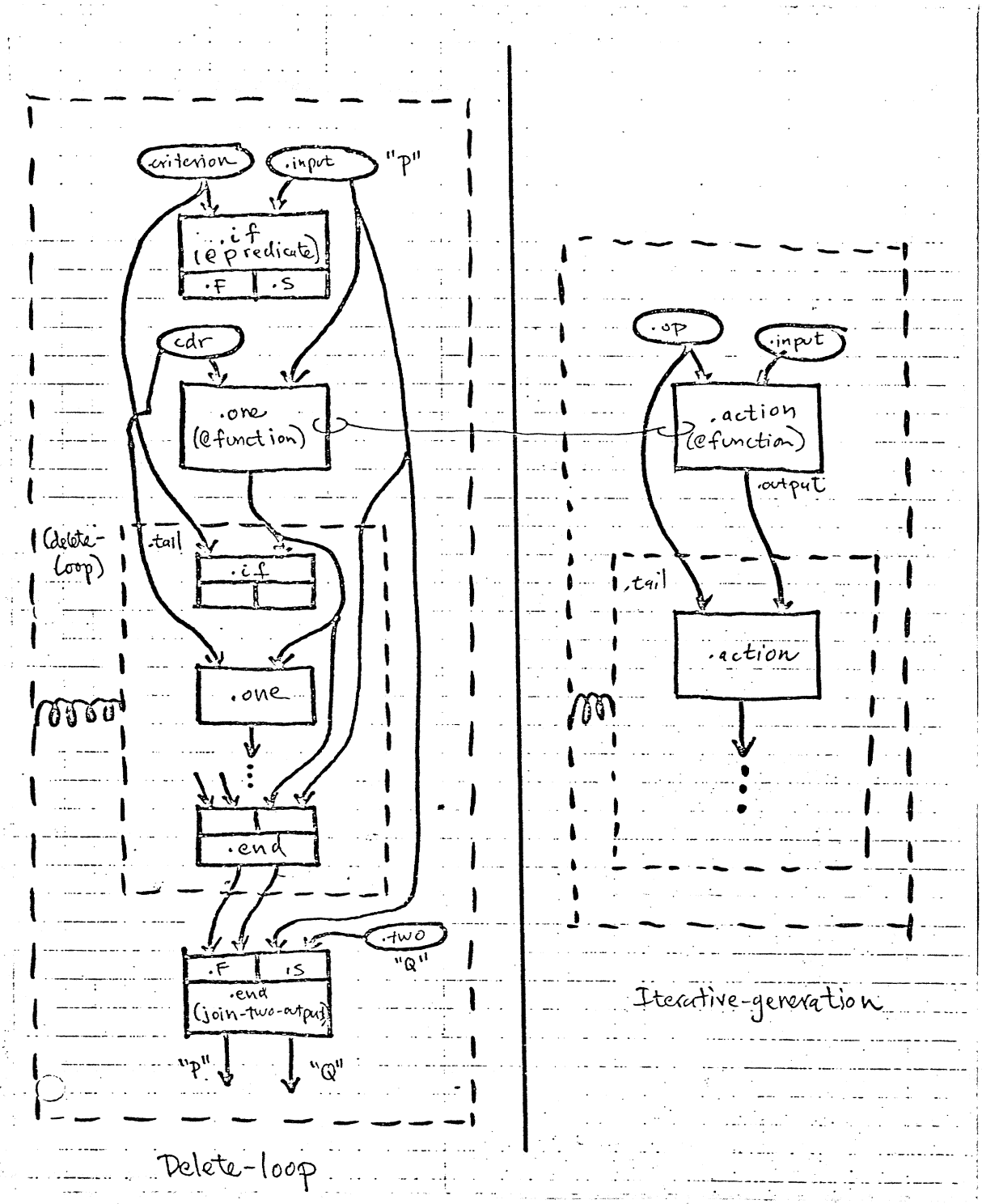


Figure 7-2. Recognition of Iterative-generation in Bucket-Delete.

WARNING! THE LOOP IN BUCKET-DELETE IS ALMOST A
TRAILING GENERATION AND SEARCH,
CURRENT: P
PREVIOUS: Q
EXIT: (COND ((EQUAL (CAAR P) ...)))
ACTION: (CDR P)
EXCEPT THAT THE OUTPUT OF THE ACTION IS NOT EQUAL TO THE
INPUT OF THE EXIT TEST.

CHAPTER EIGHT

LOGICAL FOUNDATIONS

8.1 Introduction

The preceding chapters take a practical approach to the issue of knowledge representation. This has meant focussing on how the plan calculus is used in a practical program understanding system, such as a programmer's apprentice. This chapter takes a more semantic and formal approach to plans. We begin by defining a formal logical language, called the *situational calculus*, similar to the situational calculus of McCarthy and Hayes [42]. This language is adequate for expressing the fundamental computational concepts we are interested in.

We use the situational calculus to provide a semantic foundation for the plan calculus by giving rules for translating plans into sets of axioms in the situational calculus. The presentation of these rules will be done in two stages. First we will develop enough of the situational calculus to support the semantics of data plans and overlays. We will then add a notion of temporal order and give the translation rules for temporal plans and overlays.

The reason for doing this is to be more precise about what the rules of inference are for plan calclus. For example, we need to answer questions like whether one plan subsumes another, or whether one plan is a correct implementation of another. This is particularly important in order to pre-verify plans in the plan library.

In most of this chapter, examples of Lisp computations will be used to motivate various aspects of the formalism. However, the logical framework developed here is equally applicable to other conventional sequential programming languages.

8.2 Mutable Objects and Side Effects

The everyday world of physical objects is a system with mutable objects and side effects. For example, if I drill a hole in my dining room table, I normally choose to think of it as the same object even though it now behaves differently (i.e. has different properties). Similarly for a hierarchically structured object, such as an automobile, changing some of the parts (for example replacing the brake linings) is normally viewed as a side effect, rather than resulting in a new automobile which has many of the same parts as the original.

The question of side effects is tied up with the phenomenon of naming.[1] As observers of the system, we choose to use the same name for the dining room table and the automobile at two different points in time, despite the fact that they have been modified. The notion of mutable objects thus involves two aspects: *identity* and *behavior*. The identity of a mutable object is unchangeable. Its behavior can change over time.

Syntax

The language we will use to express these ideas formally is called the situational calculus. Syntactically, the situational calculus is a standard first order logical language with constants, variables, function and relation symbols, logical connectives ($\wedge$, $\vee$ and $\neg$), quantifiers ($\forall$ and $\exists$), and equality ($=$ and $\neq$). Set theory ($\in$ and $\notin$) and integer arithmetic (Plus, Times, Gt, etc.) are taken for granted.

Basic Semantic Domains

The identity of a mutable object is embodied in its *name*. The set of names is called **P**. If p is a name, we will commonly say "the object p", rather than more precisely "the object named by p". Names are similar to what are called pointers in computer science.

The universe of possible behaviors is called **U**. Think of **U** as a universe of mathematical entities which are used to describe the properties of objects at given points in time. A nice feature of this approach is that **U** can be treated strictly as formal domain (with an equality relation). For example, suppose we want to talk about mutable sets; **U** would then be the universe of mathematical sets.

Time is represented as a set of situations, **S**. Situations are denoted in the language by constant symbols such as s and t. In this section, we are interested only in a notion of time for

---

1. Sussman and Steele give a very good illustration of this point in the context of programming language interpreters in [58].

distinguishing different behaviors of mutable objects. In a later section, a primitive order relation on situations will be introduced in order to specify the flow of control in computations.

Behavior Functions

The behavior of an object at a given point in time is expressed by a *behavior function*, which maps from a name and a situation to a behavior.

$$B: P \times S \rightarrow U$$

The term B(p,s), where B is a behavior function, may be thought of as expressing the "state of object p at s". The notion of whether or not an object p exists at time s is represented by mapping the behavior of p in some situations to a distinguished element of U called Undefined.

Generally speaking, a computing system provides the user with a set of primitive names and one or more primitive behavior functions, out of which all other mutable objects are built. For example, in Lisp the primitive names are the pointers (addresses) of CONS cells, arrays, and atoms. The primitive behavior functions specify the dotted pair behavior (i.e. the CAR and CDR) of CONS cells at given points in time; the array behavior (i.e. the current function from indices to objects) of array pointers; and the property list of atom pointers. The complete details of the model of Lisp used in this thesis will be given later.

Equality

Equality in P, S and U is denoted by "=", with the usual rules of substitution. We first consider the intuitive meaning of equality in these three domains, and then discuss how the notion of side effect is represented using equality.

Intuitively, p=q, for two names, p and q, means that p and q are different names for the same object. This could be the case if we introduced two anonymous objects named p and q, and then wanted to consider what would happen if they were the same object.

Intuitively, s=t, for two situations, s and t, means that the behavior of all mutable objects is the same in s and t. We express this formally as the *Axiom of Extensionality for Situations*, which has the following form (where B,C,... are behavior functions).

$$\forall st\,[\forall p\,[B(p,s)=B(p,t) \wedge C(p,s)=C(p,t) \wedge ...] \supset s=t]$$

For a given computing system it is adequate to include only the primitive behavior functions in this axiom. For example, for Lisp, two situations are equal in which all CONS cells have the same CAR and CDR, all arrays have the same items, and all atoms have the same property lists.

Later in this chapter, we will extend this axiom to distinguish situations which are temporally distinct, but in which the behavior of all objects is the same.

Equality in U is the equality relation for the particular mathematical domain used to represent behavior. For example, if U is sets, then normal set equality is used.

Side Effects

We speak of a side effect having occured when an object behaves differently in two situations. Formally this is when for some behavior function, B,

$$B(p,s) \neq B(p,t) \qquad \text{where } s \neq t.$$

We say here that p has been *modified.* For example, to describe the side effect in which the integer 3 is added to the mutable set p which originally contains the integers 1 and 2, we write the following.

$$set(p,s) = \{1,2\}$$
$$set(p,t) = \{1,2,3\} \quad \text{where } s \neq t$$

To say that the behavior of an object p is the same in two situations, s and t, we write $B(p,s) = B(p,t)$.

Behavior Types

In practice, we want to use many different mathematical domains, such as pairs, sequences, sets, integers, lists, etc., to specify the behavior of mutable objects. These sub-domains of U are called behavior *types.*

The details of how a behavior type is specified are not important for this level of discussion. For now we can think of a type as providing two things: a predicate on elements of U which distinguishes behaviors of that type from other behaviors, and a rule for determining equality

---

Table I. Axioms for Dotted Pairs.

Axiom of Extensionality

$$\forall xy\,[\,[\text{dotted-pairp}(x) \wedge \text{dotted-pairp}(y) \wedge \text{car}(x) = \text{car}(y) \wedge \text{cdr}(x) = \text{cdr}(y)] \supset x = y]$$

Axiom of Comprehension

$$\forall xy\,[\,[x \neq \text{undefined} \wedge y \neq \text{undefined}] \supset \exists z\,[\text{dotted-pairp}(z) \wedge \text{car}(z) = x \wedge \text{cdr}(z) = y]\,]$$

between behaviors of that type. For example, for dotted pairs, the type predicate is Dotted-pairp, and the rule for equality is an axiom which says that two dotted pairs are equal if their CAR and CDR are equal, as shown in Table I.

Associated with each behavior type we usually define a behavior function which maps to elements of that type. For example, Dotted-pair is the primitive behavior function of Lisp which specifies the dotted pair behavior of a CONS cell at a given point in time. This function thus has the following relationship to the type predicate Dotted-pairp. (In the following local context Greek letters will be used for elements of U.)

$$\forall ps\,[\alpha = \text{dotted-pair}(p,s) \supset [\text{dotted-pairp}(\alpha) \vee \alpha = \text{undefined}]\,]$$

Alternatively, we can (and will) take the approach of considering the behavior function rather than the type predicate as primitive. For example, for dotted pairs, we can define the type predicate in terms of the behavior function as follows.

$$\text{dotted-pairp}(x) \equiv [\exists ps\ \text{dotted-pair}(p,s) = x \wedge x \neq \text{undefined}]$$

In general, for type T (formally a behavior function), we can always write

$$[\exists ps\ \text{T}(p,s) = x \wedge x \neq \text{undefined}]\,.$$

where we need to assert a type predicate on x. Furthermore this will be abbreviated[1]

$$\text{instance}(\text{T},x)\,.$$

Function Objects

Many plans in this thesis are parameterized with respect to functions and relations. For example, a directed graph is modelled as a set of nodes and an edge relation. The accumulation loop plan abstracts away from which particular aggregative function (e.g. Plus, Times, Union) is used. We also need to talk about functions as mutable objects. For example, splicing operations are viewed as side effects to the edge relation of a graph.

In order to formalize such plans, we introduce functions as a behavior type in U. The standard technique for doing this is in a first order language is to introduce the function symbol, Apply (and Apply2 for functions of two arguments, etc.) which is axiomatized as shown in Table II. For basic functions, such as Plus, Times, etc. which we want to use both as first order function symbols and as elements of U, we introduce corresponding underlined symbols such as Plus, Times, etc. with axioms such as the following.

---

1. Instance must be formally treated as a syntactic abbreviation in order to keep the language first order.

Table II. Functions and Sequences.

Axiom of Extensionality

$\forall fg$ [[instance(function,f) $\wedge$ instance(function,g) $\wedge$ $\forall x$ apply(f,x) = apply(g,x)] $\supset$ $f = g$]

Sequences

$\forall lf$[[instance(function,f) $\wedge$ ge(l,0) $\wedge$ $\forall i$ [apply(f,i) $\neq$ undefined $\leftrightharpoons$ [ge(i,1) $\wedge$ le(i,l)]]]
$\leftrightharpoons$ [instance(sequence,f) $\wedge$ length(f) = l]

$\forall x$ apply(<u>oneplus</u>,x) = oneplus(x)

Furthermore, given this convention, we can omit the underlining since the underlined symbols can appear only as terms, which are syntactically distinct from function symbols in a first order language.

Relation objects are modelled as boolean valued functions. For example, the element of U which corresponds to the arithmetic binary relation, Gt, is axiomatized as follows.

$\forall xy$ [apply2(gt,x,y) = true $\leftrightharpoons$ gt(x,y)]

Sequences are treated as functions on a range of integers. This makes it convenient to model vectors in Lisp as mutable sequence objects. For example, to describe a STORE operation in which the first item of a vector p is changed from 3 to 4, we write the following.

apply(sequence(p,s),1) = 3
apply(sequence(p,t),1) = 4 where s $\neq$ t

Sequences are introduced formally as a subtype of functions by the last axiom in Table II, which states basically that a sequence is a function defined for all integers between 1 and the length of the sequence.

As a final example of mutable function objects, consider a view of Lisp in which CAR and CDR are the names of mutable function objects, whose domains are CONS cells. In this view, RPLACA and RPLACD are modelled as modifying the function behavior of CAR and CDR, rather than modifying the dotted pair behavior of a given CONS cell. The relationship between these two views is expressed in the following axioms.

$\forall ps$ apply(function(car,s),p) = car(dotted-pair(p,s))
$\forall ps$ apply(function(cdr,s),p) = cdr(dotted-pair(p,s))

8.3 Multiple Points of View

The ability to view the behavior of an object in different ways is fundamental to this thesis. We also need to represent objects whose behavior at a given time depends on the behaviors of other objects at the same time. (This is similar to the notion of implementation in computer science.)

As a simple example, suppose we are using a computing system in which mutable sets are not provided as primitives. If mutable sequences are available (either as primitives or themselves built out of some other mutable objects), we can in effect implement a mutable set by viewing a sequence as the set of its range elements. This point of view is defined formally as follows.

$$\sigma = \text{sequence}{>}\text{set}(p,s) \equiv [\text{instance}(\text{set},\sigma) \wedge \forall \alpha[(\alpha \in \sigma) \leftrightharpoons \exists i\,[\text{apply}(\text{sequence}(p,s),i) = \alpha \wedge \alpha \neq \text{undefined}]\,]]$$

Sequence>set is a behavior function for sets. Notice that the form of this definition is to construct a set behavior function using a sequence behavior function, as highlighted below.

$$\sigma = \text{sequence}{>}\text{set}(p,s) \equiv [\ldots\text{sequence}(p,s)\ldots]$$

We make other definitions of this form to describe how to implement set behavior in terms of list behavior,

$$\sigma = \text{list}{>}\text{set}(p,s) \equiv [\ldots\text{list}(p,s)\ldots]$$

and sequence behavior in terms of list behavior,

$$\theta = \text{list}{>}\text{sequence}(p,s) \equiv [\ldots\text{list}(p,s)\ldots]$$

and so on.

This way of defining behavior functions in terms of other behavior functions is the key idea for representing the implementation of mutable objects. However, one could argue that this approach misses the more fundamental implementation relationships between elements of U, e.g. from mathematical sequences to mathematical sets. This anomaly is solved by extending the functionality of behavior functions so that their first argument may be either a name or an element of U.

$$\mathrm{B}: (\mathrm{P} \cup \mathrm{U}) \times \mathrm{S} \rightarrow \mathrm{U}$$

For each primitive behavior function, such as Set, Sequence and List, we define an additional axiom which says in effect that behaviors are immutable objects which behave like

themselves.[1] For example, for Sequence we have the following axiom.

$$\forall \theta \, [\text{instance}(\text{sequence},\theta) \supset \forall s \, \text{sequence}(\theta,s) = \theta]$$

Given this convention, definitions like that of Sequence>set express both the implementation relationship between mathematical sequences and mathematical sets and between mutable sequences and mutable sets.

Sharing

Related to the notion of mutable objects and multiple points of view is the fact that two objects can share structure. The significance of sharing is that side effects on an object propagate to become side effects on other objects with which it shares structure. For example, in Lisp, a single RPLACA can modify the behavior of several different list objects. This is similar to what is called "aliasing" in computer science.

Sharing arises out of implementations which involve names. In order to describe such implementations, we need to make names part of **U**.

$$\mathbf{P} \subset \mathbf{U}$$

In other words we can have pairs of names, sets of names, sequences of names, etc. Given the convention introduced above that behaviors name themselves, this means that the functionality of behavior functions is now simply

$$\mathbf{B}: \mathbf{U} \times \mathbf{S} \rightarrow \mathbf{U}.$$

Since we still want to distinguish those elements of **U** which are not names; we define the set of *values*, **V**, as

$$\mathbf{V} = \mathbf{U} - \mathbf{P}.$$

(In the following local context, Greek letters will now be used to denote values.)

The easiest way to explain how shared structure and the propagation of side effects arises from the use of names is by an example. Consider implementing a (mutable) set as a (mutable) sequence of (mutable) sets, such that an object is a member of the implemented set iff it is a member of one of the sets in the sequence. This is part of the idea of hash tables, in which the sets in the sequence are called "buckets". This implementation can be defined formally as follows.

---

1. This is similar to the idea in Lisp that constants such as T, NIL and integers, evaluate to themselves.

$$\sigma = \text{sequence-of-sets>set}(p,s) \equiv [\text{instance}(\text{set},\sigma) \wedge \forall x\,[(x \in \sigma) \leftrightharpoons \exists i\,(x \in \text{set}(\text{apply}(\text{sequence}(p,s),i),s))]\,]$$

Notice, as highlighted below, that the Set behavior function is used to obtain the set behavior of terms in the sequence.

$$\sigma = \text{sequence-of-sets>set}(p,s) \equiv [\ldots\text{set}(\text{apply}(\text{sequence}(p,s),\ldots),s)\ldots]$$

This means that the terms in the sequence may be names. By always using behavior functions this way, we provide for the mutability of objects.

Now let us see how this implementation leads to sharing. In particular, let us see how a side effect to any bucket amounts to a side effect to the implemented set. Consider a sequence named H which is viewed as implementing a set according to the technique of Sequence-of-sets>set. Furthermore, suppose B is some bucket of H at some particular time s,

$$\text{apply}(\text{sequence}(H,s),i) = B$$

and that the sequence H is not modified between s and t.

$$\text{sequence}(H,s) = \text{sequence}(H,t)$$

However, if B is modified between s and t, i.e.

$$\text{set}(B,s) \neq \text{set}(B,t) \quad \text{where } s \neq t,$$

it follows from the definitions above that the Sequence-of-sets>set behavior of H is also modified.

$$\text{sequence-of-sets>set}(H,s) \neq \text{sequence-of-sets>set}(H,t)$$

The general point illustrated by this example is that the potential for structure sharing and the propagation of side effects is introduced whenever you start to manipulate names (pointers) as behaviors. It is usual to think of sharing at the lowest level of implementation, such as at the machine language level, or at the CONS cell level in Lisp. This example demonstrates that it may enter in at any level of abstraction.

Sharing does not always arise when pointers are used. For example, suppose we simultaneously view the sequence H above as implementing a set another way, e.g. according to Sequence>set. In this view, for the same situations s and t, no side effect has occurred.

$$\text{sequence>set}(H,s) = \text{sequence>set}(H,t)\,,$$

This is because Sequence>set(H,s) is the set of bucket *names*, which doesn't change even though one of the buckets has been modified. We could give separate names to these two set views of H, as follows.

$$\forall s\ \text{set}(M,s) = \text{sequence-of-sets>set}(H,s)$$
$$\forall s\ \text{set}(K,s) = \text{sequence>set}(H,s)$$

The set M can be thought of as the set of members of the hash table, and K the set of buckets.

Shared List Structure in Lisp

As a second example of sharing, we show how to represent a kind of sharing which should be very familiar to Lisp programmers -- shared list structure. This example is more complicated than the hash table example mostly because of the recursive nature of the definition of list behavior. The axioms for lists are given in Table III.

Lists in Lisp are built out of dotted pairs whose CDR is either NIL (a distinguished element of U) or the name of (pointer to) another such dotted pair. This is often called the "linked list" implementation. It is defined formally in terms of behavior functions as follows.

$$\lambda = \text{dotted-pair>list}(p,s) \equiv [\text{instance}(\text{list},\lambda) \wedge \text{head}(\lambda) = \text{car}(\text{dotted-pair}(p,s)) \wedge \text{tail}(\lambda) = \text{dotted-pair>list}(\text{cdr}(\text{dotted-pair}(p,s)),s)]$$

Notice that this is a recursive definition. The tail of the implemented list is the list implemented by the CDR of the dotted pair (in the same way).

To demonstrate how this implementation of lists in Lisp entails structure sharing we show an example of how side effects are propagated. Consider three CONS cells C, D, and E, such that in situation s the CDR of both C and D is E.

Table III. Axioms for Lists.

Axiom of Extensionality

$$\forall \alpha\beta pqs\,[\,[\alpha = \text{list}(p,s) \wedge \beta = \text{list}(q,s) \wedge \text{head}(\alpha) = \text{head}(\beta) \wedge \text{list}(\text{tail}(\alpha),s) = \text{list}(\text{tail}(\beta),s)] \supset \alpha = \beta]$$

Axiom of Comprehension

$$\forall xys\,[\,[x \neq \text{undefined} \wedge [y = \text{nil} \vee \text{list}(y,s) \neq \text{undefined}]\,] \leftrightharpoons \exists \alpha p\,[\alpha = \text{list}(p,s) \wedge \alpha \neq \text{undefined} \wedge \text{head}(\alpha) = x \wedge \text{tail}(\alpha) = y]\,]$$

cdr(dotted-pair(C,s)) = E
cdr(dotted-pair(D,s)) = E

If we view C, D and E as implementing lists according to Dotted-pair>list, then by the definitions above, C and D share tails in s, i.e.

tail(dotted-pair>list(C,s)) = tail(dotted-pair>list(D,s))

If we now modify E (e.g. by RPLACA), without changing C and D, so that

dotted-pair(C,s) = dotted-pair(C,t)
dotted-pair(D,s) = dotted-pair(D,t)
dotted-pair(E,s) ≠ dotted-pair(E,t) for s≠t,

it follows that

dotted-pair>list(E,s) ≠ dotted-pair>list(E,t) .

Furthermore, since they share structure with E viewed as a list, it follows that the list behaviors of C and D have both been modified.

dotted-pair>list(C,s) ≠ dotted-pair>list(C,t)
dotted-pair>list(D,s) ≠ dotted-pair>list(D,t) .

8.4 Data Plans

We are now in a position to explain the meaning of data plans in terms of the formal framework developed in the preceding sections. The basic idea is that a data plan defines a new type and an associated behavior function. We will first present an example, and then outline the general rules for how to translate from the data plan formalism to a set of axioms in the situational calculus.

Consider the data plan, Segment, shown in Fig. 8-1, which consists of a sequence (the Base) and two natural numbers (Upper and Lower), with the constraint that Upper and Lower are valid indices for the Base, and Lower is less than or equal to Upper.

In terms of the formal framework developed in the preceding sections, the meaning of this plan is to define a new behavior type with the two axioms shown in Table IV. The first axiom says that two segments are equal iff their base sequences and upper and lower indices are the same. The second axiom says that for any sequence and two numbers which are valid indices for that sequence, there exists a segment with that sequence as the base and the two numbers as the upper and lower indices; and conversely, that the upper and lower indices of any segment are valid indices for the base sequence and the lower index is less than the upper.

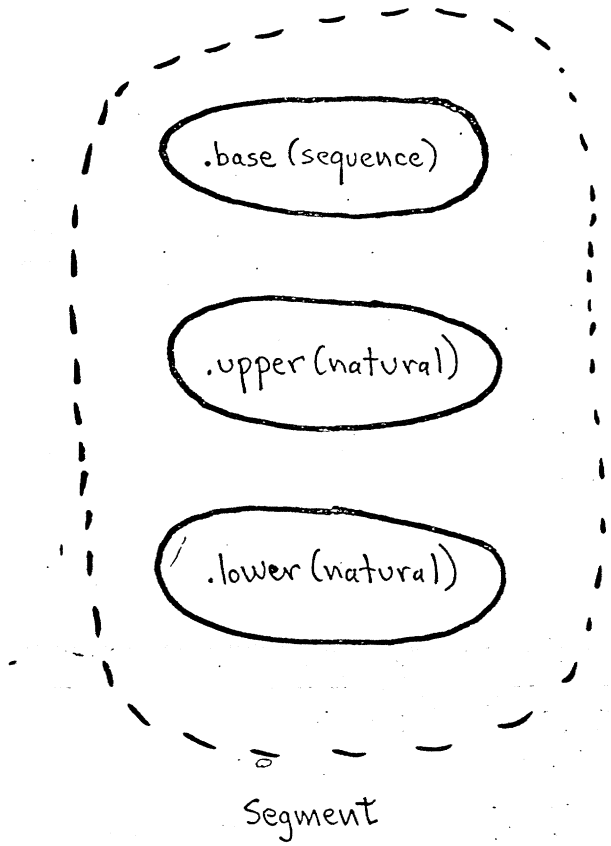


Figure 8-1. Segment Data Plan.

Table IV. Segment Data Plan.

Axiom of Extensionality

$$\forall \alpha\beta pqs\,[\,[\alpha = \text{segment}(p,s) \wedge \beta = \text{segment}(q,s)$$
$$\wedge\ \text{sequence}(\text{base}(\alpha),s) = \text{sequence}(\text{base}(\beta),s)$$
$$\wedge\ \text{natural}(\text{lower}(\alpha),s) = \text{natural}(\text{lower}(\beta),s)$$
$$\wedge\ \text{natural}(\text{upper}(\alpha),s) = \text{natural}(\text{upper}(\beta),s)]$$
$$\supset \alpha = \beta]$$

Axiom of Comprehension

$$\forall xyzs\,[\,[\text{sequence}(x,s) \neq \text{undefined} \wedge \text{natural}(y,s) \neq \text{undefined} \wedge \text{natural}(z,s) \neq \text{undefined}$$
$$\wedge\ \text{le}(\text{natural}(y,s),\text{natural}(z,s))$$
$$\wedge\ \text{le}(\text{natural}(y,s),\text{length}(\text{sequence}(x,s)))$$
$$\wedge\ \text{le}(\text{natural}(z,s),\text{length}(\text{sequence}(x,s)))]$$
$$\leftrightharpoons \exists \alpha p\,[\alpha = \text{segment}(p,s) \wedge \alpha \neq \text{undefined}$$
$$\wedge\ \text{base}(\alpha) = x \wedge \text{lower}(\alpha) = y \wedge \text{upper}(\alpha) = z]\,]$$

DataPlan Segment
roles .base(sequence) .lower(natural) .upper(natural)
constraints le(.lower,.upper)
$\wedge$ le(.lower,length(.base)) $\wedge$ le(.upper,length(.base))

Notice that behavior functions are used throughout these axioms to refer to the behavior of the parts of a segment. This is necessary to allow for shared structure at any level. For example this means that the Base of a segment can be either a sequence of the name of a sequence.

The general rule for translating data plans into a set of axioms in the situational calculus has two steps. First, the name of the plan formally becomes a behavior function, and the roles of the plan become functions on behaviors of that type. Second, two axioms are written involving these functions.

The first axiom defines equality on the new behavior type in terms of equality of the appropriate behaviors of the roles. So for data plan D with n roles f,g,..., restricted to behavior types T,U,..., respectively, the following axiom (called the axiom of extensionality) is written.

$$\forall \alpha\beta pqs\,[\,[\alpha = \text{D}(p,s) \wedge \beta = \text{D}(q,s)$$
$$\wedge\ \text{T}(\text{f}(\alpha),s) = \text{T}(\text{f}(\beta),s) \wedge \text{U}(\text{g}(\alpha),s) = \text{U}(\text{g}(\beta),s) \wedge \ldots]$$
$$\supset \alpha = \beta]$$

The second axiom involves the type restrictions on roles of a data plan and the constraints between roles. Formally the constraints are an *n*-ary relation, where each argument position corresponds to a role, with an extra role for the situation argument to the behavior functions for each role type. So for the same data plan D as above, with constraint relation, C, the following axiom (called the axiom of comprehension) is written.

$$\forall sxy\ldots\,[\,[T(x,s)\neq \text{undefined} \wedge U(y,s)\neq \text{undefined} \wedge \ldots \wedge C(s,x,y,\ldots)] \\ \Leftrightarrow \exists \alpha p\,[\alpha = D(p,s) \wedge \alpha \neq \text{undefined} \wedge f(\alpha)=x \wedge g(\alpha)=y \wedge \ldots]\,]$$

This axiom specifies that instances of the plan D exist, and that all instances satisfy the role type restrictions and constraints.

Finally, the information in the axioms for a data plan can be written in more compact tabular form as shown at the bottom of Table IV. This is the notation that will be used in the remainder of this thesis for formal logical plan definitions. In this notation, the definition of the constraint relation is made easier to read by using the role names preceded with a leading point (such as ".base") instead of quantified variables corresponding to roles, as appear in the fully written out axioms. Remaining points in constraint formulae are interpreted as normal function application. For example, a path name like ".f.r.s", where f is a role in the plan being defined, is formally equivalent to "s(r(.f))", since r and s are other role functions.

An additional abbreviation used in writing constraints in data plans is to make the behavior functions applied to role functions implicit when the behavior function is the same as the type restriction on the role. For example, at the bottom of Table IV,

le(.upper,length(.base))

is an abbreviation for

le(natural(.upper,*s*),length(sequence(.base,*s*))) .

The type restriction on each role of a data plan is indicated in the compact notation in parentheses following each role name. For example, the axioms for lists can be rewritten using this notation as follows.

DataPlan List
  roles .head(object) .tail(list+nil)

The type List+nil is defined by the behavior function shown below.

$$\lambda = \text{list+nil}(p,s) \equiv [\lambda = \text{list}(p,s) \vee \lambda = \text{nil}]$$

The absence of type restriction (other than being defined) is indicated by the keyword "object" after the role name. For example, the axioms for dotted pairs can be rewritten using this notation as follows.

DataPlan Dotted-pair
roles .car(object) .cdr(object)

8.5 Data Overlays

Intuitively, a data overlay is a many-to-one mapping from one behavior type to another. Formally, a data overlay is a behavior function which is defined in terms of another behavior function. For example, Sequence>set is a data overlay for viewing a (mutable) sequence as the implementation of a set. Furthermore, because of the way overlays are used in analysis and synthesis, the mapping must be "total" in both directions. For example, for the Sequence>set overlay this means that given any sequence, there exists a set which it implements in this way; and conversely, given any set, there is at least one sequence which implements it in this way. These properties are written formally as the two totality axioms shown in Table V. The definition of Sequence>set is also repeated in this table for reference.

As in the case of data plans, it is more convenient to use a compact tabular notation than to write out the definition and axioms for a data overlay as in Table V. The tabular notation that will be used in the rest of this thesis for data overlays is shown at the bottom of the table. The general rules for recovering the fully written out formal logical definition and axioms are as follow. In general, the definition of an overlay V from behavior type T to behavior type U,

DataOverlay V: T → U

Table V. Sequence as Set Overlay.

Totality Axioms

$\forall xs\,[\text{sequence}(x,s)\neq\text{undefined} \supset \exists y\,[y=\text{sequence>set}(x,s)]\,]$

$\forall ys\,[\text{set}(y,s)\neq\text{undefined} \supset \exists x\,[y=\text{sequence>set}(x,s)]\,]$

Definition

$y=\text{sequence>set}(p,s) \equiv [\text{instance}(\text{set},y) \wedge \forall\alpha\,[(\alpha \in y) \leftrightharpoons \exists i\,\text{apply}(\text{sequence}(p,s),i)=\alpha]\,]$

DataOverlay Sequence>set: sequence → set
definition $y=\text{sequence>set}(p,s) \leftrightharpoons \forall\alpha[(\alpha \in y) \leftrightharpoons \exists i\,\text{apply}(\text{sequence}(p,s),i)=\alpha]$

is of the following form.

$$y = V(p,s) \equiv [\text{instance}(U,y) \wedge ...y...T(p,s)...]$$

The content of this definition is in the formula relating y and $T(p,s)$ above. The standard prefix, Instance(U,y), is omitted in the tabular notation. Furthermore, two totality axioms are written from the type information in the header of the tabular notation. These axioms have the following form.

$$\forall xs\,[T(x,s) \neq \text{undefined} \supset \exists y\,[y = V(x,s)]]$$
$$\forall ys\,[U(y,s) \neq \text{undefined} \supset \exists x\,[y = V(x,s)]]$$

8.6 Computations

In this thesis, computations are thought of as structures some of whose parts are situations (elements of **S**) and some of whose parts are objects (elements of **U**). In order to formally describe computations in the situational calculus, we introduce a new basic domain, **C**, of computations. **C** is divided into types which are specified by axioms similar to those used to specify behavior types in **U**. In the rest of this section, after some formal preliminaries, we present axioms for various computation types. In the next section we use these foundations to the specify the semantics of the temporal plan formalism.

Temporal Order

Thus far we have been using situations only as arguments to behavior functions to distinguish the different states of objects. In order to represent temporal order in computations we introduce a new primitive relation, called Precedes, which is formally a total order on **S**. Intuitively, this relation captures the notion of states occuring "before" or "after" other states. This relation also makes it possible to talk about cyclic computations in which all objects return to the same state as at some earlier time. Formally, this is achieved by extending the Axiom of Extensionality for Situations as follows.

$$\forall st\,[[\,\forall p\,[B(p,s) = B(p,t) \wedge C(p,s) = C(p,t) \wedge ...]$$
$$\wedge\ \forall u\,[\text{precedes}(s,u) \leftrightharpoons \text{precedes}(t,u)]\,] \supset s = t]$$

B,C,... here are the appropriate primitive behavior functions as before. What this axiom says is that two situations are identical if the behavior of all objects is the same and they are indistinguishable in the temporal order.

Note that Precedes is a *total* order. This is because we are formally dealing with sequential computations. As we will see shortly, however, in specifying computation *types* we will often leave the order between two steps unconstrained.

Termination

Another basic feature of computations we need to deal with is termination. In order to talk about this formally, we introduce a bottom element in **S**, i.e.

$$\forall s \text{ precedes}(s,\perp) .$$

Intuitively, $\perp$ represents a computation step which is never reached. As we shall see in the following sections, $\perp$ appears in the axioms for elementary computation types, such as operations and tests. The termination properties of composite types, such as loops, are then derived from the axioms of the components and their connections. Important termination properties are whether or not a given step is reached in all instances of a computation type, and whether there exists an instance of a computation type in which a given step is reached. Formally these properties amount to whether or not specified situations are equal to $\perp$.

Operations

The most basic computation types are operations. Operations in general involve two situations, one of which precedes the other, and some number of input and output objects. An example of such a type is set addition operations. Intuitively, a set addition computation is an operation involving three objects: the old set, the new set, and the member added. This is specified formally by the two axioms shown in Table VI. These axioms involve the type predicate, Set-add, and the functions In, Out, Old, New, and Input, on elements of the type which act like the part functions (e.g. Head and Tail for lists) of a data structure. For example, consider two situations, s and t, and mutable sets A and B, such that the following statements hold.

$$\begin{aligned} &\text{precedes}(s,t)\\ &\text{set}(A,s)=\{1,2\}\\ &\text{set}(B,t)=\{1,2,3\} \end{aligned}$$

Formally, what we have here is a computation, α , such that

$$\begin{aligned} &\text{set-add}(\alpha)\\ &\text{in}(\alpha)=s\\ &\text{out}(\alpha)=t\\ &\text{old}(\alpha)=A\\ &\text{input}(\alpha)=3\\ &\text{new}(\alpha)=B . \end{aligned}$$

In the following local context Greek letters will be used to denote elements of **C**. Note that we will also informally refer to elements of **C** as *instances* of a computation type, T. Formally, this just means $T(\alpha)$.

Table VI. Set Addition Operations.

Axiom of Extensionality

$$\forall \alpha\beta\ [\ [\text{set-add}(\alpha) \wedge \text{set-add}(\beta) \wedge \text{in}(\alpha)=\text{in}(\beta) \wedge \text{out}(\alpha)=\text{out}(\beta) \wedge \text{old}(\alpha)=\text{old}(\beta) \wedge \text{input}(\alpha)=\text{input}(\beta) \wedge \text{new}(\alpha)=\text{new}(\beta)] \supset \alpha=\beta]$$

Axiom of Comprehension

$$\forall xyzst\ [\ [\text{precedes}(s,t) \wedge [s\neq\perp \supset [t\neq\perp \wedge \text{set}(x,s)\neq\text{undefined} \wedge \text{set}(y,t)\neq\text{undefined} \wedge z\neq\text{undefined} \wedge (z \in \text{set}(y,t)) \wedge \forall w\,[w\neq z \supset [(w \in \text{set}(y,t)) \leftrightharpoons (w \in \text{set}(x,s))]\ ]\ ]\ ]\ ] \leftrightharpoons \exists\alpha\ [\text{set-add}(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{out}(\alpha)=t \wedge \text{old}(\alpha)=x \wedge \text{new}(\alpha)=y \wedge \text{input}(\alpha)=z]\ ]$$

IOspec Set-add / .old(set) .input(object) $\Rightarrow$ .new(set)
postconditions (.input $\in$ .new)
$\wedge\ \forall x\,[x\neq$.input $\supset$ [(.input $\in$ .new) $\leftrightharpoons$ (.input $\in$ .old)]]

The first axiom in Table VI defines equality of set addition operations in terms of equality of the situations and objects involved. The second axiom specifies a necessary and sufficient condition between the objects and situations of set addition operations. These axioms amount to what would be called an *input-output specification* in standard software engineering terminology.

Let us now pay attention to the details of the second axiom in Table VI. Part of the necessary and sufficient condition deals with the temporal order and termination properties of set addition operations, as shown below. (This pattern of specification is followed for operations in general).

$$[\text{precedes}(s,t) \wedge [s\neq\perp \supset [t\neq\perp \wedge \ldots]\ ]\ ]$$

Thus the In situation precedes the Out situation. Furthermore, if the In situation is reached, it follows that the Out situation is reached, i.e. the operation always terminates. Notice that it follows from this axiom and the definition of $\perp$ that, if the In situation is never reached (i.e. $s=\perp$), then the Out situation is never reached ($t=\perp$).

The remainder of the condition part of the second axiom specifies that the members of the New set in the Out situation are exactly the members of the Old set in the In situation, with the sole addition of the Input object. This relationship is conditionalized inside $t\neq\perp$ to avoid contradiction in the case when neither situation is reached, i.e. $s=t=\perp$.

Notice that this specification uses the Set behavior function in referring to the Old and New objects. This means that that instances of this computation type include both by operations in which the input and output sets are distinct objects, and those which involve a side effect (e.g. suppose $old(\alpha)=new(\alpha)$ in the example above). More will be said about plans involving side effects in a later section.

A more compact tabular notation for writing input-output specification is shown at the bottom of Table VI. The first line of this notation lists the name of the operation type (formally a predicate on computations), separated by a slash from the input roles, separated by a double arrow from the output roles. Type restrictions are indicated in parentheses following the role names, as in the compact data plan notation. Roles are formally functions on computations. To recover the formal axioms from this notation for a general input-output specification, P, with input roles f,g,..., and output roles m,n,..., we first write an axiom of extensionality of the following form.

$$\forall\alpha\beta\,[\,[P(\alpha)\wedge P(\beta)\wedge in(\alpha)=in(\beta)\wedge out(\alpha)=out(\beta)$$
$$\wedge f(\alpha)=f(\beta)\wedge g(\alpha)=g(\beta)\wedge\ldots\wedge m(\alpha)=m(\beta)\wedge n(\alpha)=n(\beta)\wedge\ldots]$$
$$\supset \alpha=\beta]$$

The constraint between roles in an input-output specification is made easier to read in the compact tabular notation by using the role names preceded with a leading point instead of quantified variables, similar to the constraint notation for data plans. Embedded points are interpreted as normal functional nesting.

Like the compact notation for data plans, the application of behavior functions corresponding to role types is also made implicit in compact input-output specifications. For example, at the bottom of Table VI

(.input $\in$ .new)

is an abbreviation for

(.input $\in$ set(.new,.in)) .

By convention, the situational argument to such implicit behavior function applications is either ".in" or ".out", depending on whether the role involved is an input or an output. No behavior function is supplied for roles, such as Input, without type restriction (indicated by the keyword type "Object" as in data plans).

After expanding all abbreviations as outlined above, the constraint relation is formally a relation, C, where each argument position corresponds to a role, plus two situational arguments which correspond to In and Out. In general for an input-output specification P with input roles f,g,...,

with type restrictions T,U,..., and output roles m,n,..., with type restrictions Λ,B,...,we then write the following axiom.

$$\forall stxy...vw... [\,[\text{precedes}(s,t) \wedge [s \neq \perp \supset$$
$$[t \neq \perp \wedge T(x,s) \neq \text{undefined} \wedge U(y,s) \neq \text{undefined} \wedge ...$$
$$\wedge \Lambda(v,t) \neq \text{undefined} \wedge B(w,t) \neq \text{undefined} \wedge ...$$
$$\wedge C(s,t,x,y,...,v,w,...)]\,]\,]$$
$$\leftrightharpoons \exists \alpha\, [P(\alpha) \wedge \text{in}(\alpha) = s \wedge \text{out}(\alpha) = t$$
$$\wedge f(\alpha) = x \wedge g(\alpha) = y \wedge ... \wedge m(\alpha) = v \wedge n(\alpha) = w \wedge ...]\,]$$

Finally, note that as a matter of convention the constraint clauses in an input-output specification are divided into those which involve only input roles (called preconditions), and those which involve both input and output roles (called postconditions). For example, the following is the compact specification of @Function, the operation of applying a function to an argument to get the corresponding range element.

IOspec @Function / .op(function) .input(object) ⇒ .output(object)
preconditions $\exists x\, [x = \text{apply}(.\text{op},.\text{input})]$
postconditions apply(.op,.input) = .output

Tests

The second basic computation type used in this thesis is tests. Tests in general have three situational roles: an input situation, In, and two alternative following situations, Succeed and Fail, only one of which is reached in any instance.

An example of a type of test, membership tests, is shown specified formally in Table VII. The first axiom is the usual axiom of extensionality which defines equality on a computation type in terms of equality of its roles. The roles of a membership test are the three situational roles, In, Succeed and Fail, and two object roles, Universe (a set) and Input.

The second axiom in Table VII says roughly that membership tests succeed if the Input is a member of the Universe; otherwise they fail. This is expressed formally by specifying the conditions under which the Succeed and Fail roles are equal to ⊥, as shown below.

$$[\text{precedes}(s,t) \wedge \text{precedes}(s,u) \wedge [t = \perp \vee u = \perp]$$
$$\wedge [s \neq \perp \supset [\,[t \neq \perp \vee u \neq \perp] \wedge ... \wedge [t \neq \perp \leftrightharpoons ...]\,]\,]\,]$$

This is the pattern of specification used in general for tests. At most one of either Succeed or Fail is reached in any instance. If the condition of the test is true in the In situation, then the Succeed situation is reached; if it is false, then the Fail situation is reached. If the In situation is never reached, it follows that neither Succeed nor Fail are reached.

Table VII. Membership Tests.

Axiom of Extensionality

$$\forall \alpha\beta\ [\,[\text{member?}(\alpha) \wedge \text{member?}(\beta) \wedge \text{in}(\alpha)=\text{in}(\beta) \wedge \text{succeed}(\alpha)=\text{succeed}(\beta) \wedge \text{fail}(\alpha)=\text{fail}(\beta) \wedge \text{universe}(\alpha)=\text{universe}(\beta) \wedge \text{input}(\alpha)=\text{input}(\beta)] \supset \alpha=\beta]$$

Axiom of Comprehension

$$\forall xystu\ [\,[\text{precedes}(s,t) \wedge \text{precedes}(s,u) \wedge [t=\perp \vee u=\perp] \wedge [s\neq\perp \supset [\,[t\neq\perp \vee u\neq\perp] \wedge x\neq\text{undefined} \wedge \text{set}(y,s)\neq\text{undefined} \wedge [t\neq\perp \leftrightharpoons (x \in \text{set}(y,s))]\,]\,]\,] \leftrightharpoons \exists\alpha\ [\text{member?}(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{succeed}(\alpha)=t \wedge \text{fail}(\alpha)=u \wedge \text{universe}(\alpha)=y \wedge \text{input}(\alpha)=x]\,]$$

Test Member? / .universe(set) .input(object)
condition (.input ∈ .universe)

In the next section, we will see how tests specified this way can be combined with other computations, via the notion of control flow, to construct specifications for larger conditional computations.

Finally, Table VII shows an example of the compact notation for tests. The header line lists the name of the computation type followed by the object role names with type restrictions, similar to the input-output specification notation introduced in the preceding section. The axiom of extensionality which follows from this notation in general is obvious. The axiom of comprehension for a test P? with object roles f,g,..., and type restrictions T,U,..., is of the following form.

$$\forall stuxy...\ [\,[\text{precedes}(s,t) \wedge \text{precedes}(s,u) \wedge [t=\perp \vee u=\perp] \wedge [s\neq\perp \supset [\,[t\neq\perp \vee u\neq\perp] \wedge T(x,s)\neq\text{undefined} \wedge U(y,s)\neq\text{undefined} \wedge ... \wedge [t\neq\perp \leftrightharpoons C(s,x,y,...)]\,]\,]\,] \leftrightharpoons \exists\alpha\ [P?(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{succeed}(\alpha)=t \wedge \text{fail}(\alpha)=u \wedge f(\alpha)=x \wedge g(\alpha)=y \wedge ...]\,]$$

The relation C above is derived by expanding abbreviations in the "***condition***" part of the compact test notation in the same way abbrevations are expanded in the preconditions and postconditions of an input-output specification, supplying ".in" as the situational argument to implicit behavior functions where required.

8.7 Temporal Plans

In this section we extend **C** by allowing parts of computations to be not only situations and names, but also other computations. This gives us the ability to combine already defined computation types, such as operations and tests, into the specification of larger computations. For example, we can define a computation type which has two steps. The first step is an instance of @Discrimination;[1] the second step is a membership test. The temporal plan representation of this computation type is shown in Fig. 8-2. The axioms which are the formal translation of this plan are given in Table VIII.

Notice that the name of the plan, Discriminate+member?, is formally a predicate on computations. The roles of the plan, Discriminate[2] and If, are formally functions on computations, like Old, In, Input, New, etc. in the preceding section. The ranges of these role functions, however, are computations, as can be seen in the second axiom of Table VIII highlighted below.

$$\forall \alpha\beta\ [\ [@\text{discrimination}(\alpha) \wedge \text{member?}(\beta)\ ...]$$
$$\leftrightharpoons \exists\delta\ [\text{discriminate+member?}(\delta) \wedge \text{discriminate}(\delta)=\alpha \wedge \text{if}(\delta)=\beta]\ ]$$

Table VIII. Discriminate and Member Plan.

Axiom of Extensionality

$$\forall \alpha\beta\ [\ [\text{discriminate+member?}(\alpha) \wedge \text{discriminate+member?}(\beta)$$
$$\wedge\ \text{discriminate}(\alpha)=\text{discriminate}(\beta) \wedge \text{if}(\alpha)=\text{if}(\beta)] \supset \alpha=\beta]$$

Axiom of Comprehension

$$\forall \alpha\beta\ [\ [@\text{discrimination}(\alpha) \wedge \text{member?}(\beta) \wedge \text{cflow}(\text{out}(\alpha),\text{in}(\beta))$$
$$\wedge\ \text{set}(\text{output}(\alpha),\text{out}(\alpha))=\text{set}(\text{universe}(\beta),\text{in}(\beta))$$
$$\wedge\ \text{input}(\alpha)=\text{input}(\beta)]$$
$$\leftrightharpoons \exists\delta\ [\text{discriminate+member?}(\delta) \wedge \text{discriminate}(\delta)=\alpha \wedge \text{if}(\delta)=\beta]\ ]$$

TemporalPlan discriminate+@member
roles .discriminate(@discrimination) .if(member?)
constraints cflow(.discriminate.out,.if.in)
$\wedge$ .discriminate.output=.if.universe $\wedge$ .discriminate.input=.if.input

---

1. @Discrimination is a specialization of @Function in which the Output role is a set. Specialization and extension will be discussed later in this chapter.
2. The name of this role is due to the fact that this plan is the implementation of membership tests on a set implemented as a discrimination function.

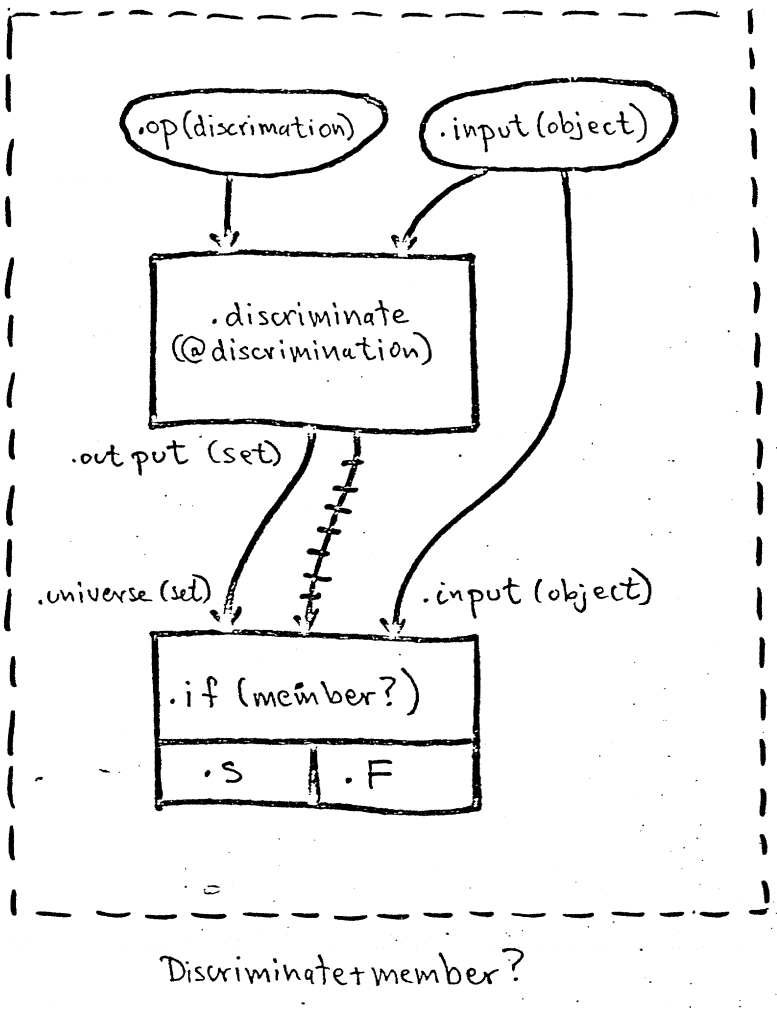


Figure 8-2. A Temporal Plan

Table IX. Bump and Update Plan.

Axiom of Extensionality

$$\forall \alpha\beta\ [\ [\text{bump+update}(\alpha) \wedge \text{bump+udpate}(\beta) \wedge \text{bump}(\alpha)=\text{bump}(\beta) \wedge \text{update}(\alpha)=\text{update}(\beta) \wedge \text{old}(\alpha)=\text{old}(\beta) \wedge \text{new}(\alpha)=\text{new}(\beta)] \supset \alpha=\beta]$$

Axiom of Comprehension

$$\begin{aligned}\forall xy\alpha\beta\ [\ [&\text{@oneminus}(\alpha) \wedge \text{newterm}(\beta)\\ &\wedge \text{upper-segment}(x,\text{in}(\alpha)) \neq \text{undefined}\\ &\wedge \text{upper-segment}(y,\text{out}(\beta)) \neq \text{undefined}\\ &\wedge \text{cflow}(\text{out}(\alpha),\text{in}(\beta))\\ &\wedge \text{upper-segment}(x,\text{in}(\alpha)) = \text{upper-segment}(x,\text{in}(\beta))\\ &\wedge \text{upper-segment}(y,\text{out}(\alpha)) = \text{upper-segment}(y,\text{out}(\beta))\\ &\wedge \text{integer}(\text{input}(\alpha),\text{in}(\alpha)) = \text{natural}(\text{lower}(\text{upper-segment}(x,\text{in}(\alpha))),\text{in}(\alpha))\\ &\wedge \text{sequence}(\text{old}(\beta),\text{in}(\beta)) = \text{sequence}(\text{base}(\text{upper-segment}(x,\text{in}(\beta))),\text{in}(\beta))\\ &\wedge \text{integer}(\text{output}(\alpha),\text{out}(\alpha)) = \text{natural}(\text{arg}(\beta),\text{in}(\beta))\\ &\wedge \text{integer}(\text{output}(\alpha),\text{out}(\alpha)) = \text{natural}(\text{lower}(\text{upper-segment}(y,\text{out}(\alpha))),\text{out}(\alpha))\\ &\wedge \text{sequence}(\text{new}(\beta),\text{out}(\beta)) = \text{sequence}(\text{base}(\text{upper-segment}(y,\text{out}(\beta))),\text{out}(\beta))]\\ &\quad \leftrightharpoons \exists\delta\ [\text{bump+update}(\delta)\\ &\qquad \wedge \text{bump}(\delta)=\alpha \wedge \text{update}(\delta)=\beta \wedge \text{old}(\delta)=x \wedge \text{new}(\delta)=y]\ ]\end{aligned}$$

TemporalPlan Bump+update
roles .bump(@oneminus) .update(newterm) .old(upper-segment) .new(upper-segment)
constraints cflow(.bump.out,.update.in)
$\wedge$ $.\text{old}_{.\text{bump.in}} = .\text{old}_{.\text{update.in}}$
$\wedge$ $.\text{new}_{.\text{bump.out}} = .\text{old}_{.\text{update.out}}$
$\wedge$ .bump.input = .old.lower
$\wedge$ .update.old = .old.base
$\wedge$ .bump.output = .update.arg
$\wedge$ .bump.output = .new.lower
$\wedge$ .update.new = .new.base

In general, the type restriction on a role in a temporal plan is either a behavior type (formally a behavior function) or a computation type (formally a predicate on computations). An example of a temporal plan which has some of both kinds of roles is shown in Fig. 8-3. The Old and New roles are restricted to being instances of the Upper-segment data plan;[1] Bump and Update are operations.[22] The axioms for this plan are shown in Table IX. The axiom of comprehension in this

---

1. Upper-segment is a specialization of Segment in which the Upper index is equal to the length of the sequence.
2. The specifications for @Oneminus and Newterm can be found in the appendix.

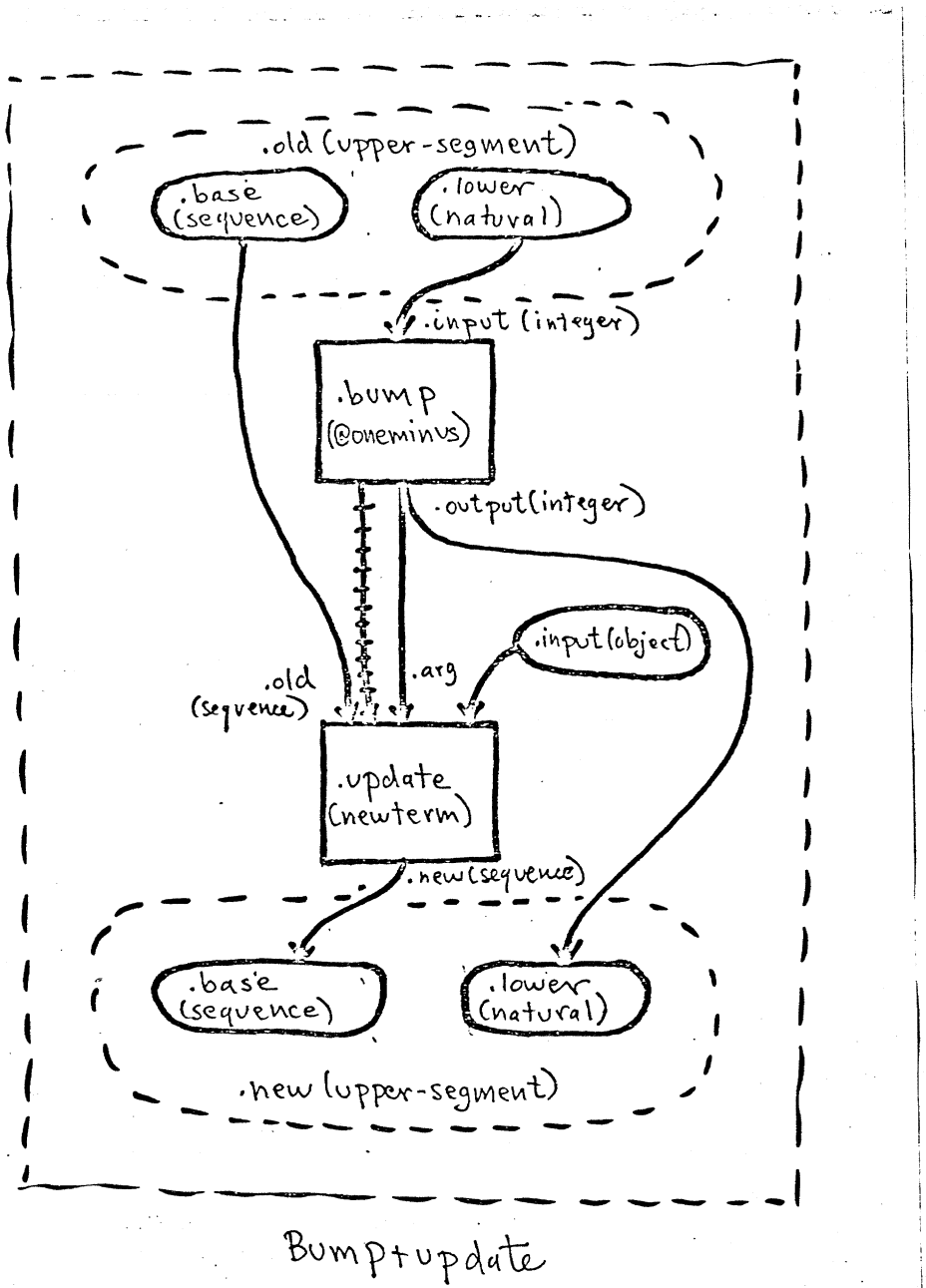


Figure 8-3. Another Temporal Plan.

table is quite long, but is of the same general form as the axiom of comprehension for Discriminate+ member?. The first three lines stipulate type restrictions. For temporal roles, these are assertions of the appropriate computation type predicates, e.g.

$$@\text{oneminus}(\alpha) \wedge \text{newterm}(\beta)\,.$$

For behavior type roles, the assertion of a type restriction has to include the situation in which it is used, e.g.

$$\text{upper-segment}(x,\text{in}(\alpha)) \neq \text{undefined}$$
$$\text{upper-segment}(y,\text{out}(\beta)) \neq \text{undefined}\,.$$

For data roles that are used in more than one place, additional equalities are added to guarantee that the data object is the same in all situations of use. For example, the two lines following the control flow constraint in the comprehension axiom of Bump+update are for this purpose.

$$\text{upper-segment}(x,\text{in}(\alpha)) = \text{upper-segment}(x,\text{in}(\beta))$$
$$\text{upper-segment}(y,\text{out}(\alpha)) = \text{upper-segment}(y,\text{out}(\beta))$$

The remaining equalities have to do with data flow, which will be discussed later in this section.

Control Flow

Control flow constraints (hatched arrows in the plan calculus) are formalized in the situational calculus as follows.

$$\text{cflow}(s,t) \equiv [\text{precedes}(s,t) \wedge [s=\perp \leftrightharpoons t=\perp]]$$

In other words, control flow entails temporal order and termination is preserved. However, the two situations do not have to be equal.

Each control flow arc in a temporal plan becomes a Cflow clause in the axiom of comprehension for the computation type. The terms in this clause are the appropriate In, Out, Succeed or Fail roles, as read from the diagram. For example, the control flow arc in Fig. 8-2 becomes the following clause in the comprehension axiom of Table VIII.

$$\text{cflow}(\text{out}(\alpha),\text{in}(\beta))$$

Data Flow

A second kind of "glue" in temporal plans is data flow. Data flow arcs in general are translated into equalities between names and values in different situations. The details of this translation, however, depend on whether the data flow is between operations and tests, or whether it also involves data plan roles, such as Old and New in Bump+update.

We start with the simple case of the data flow arc in Discriminate+member (Fig. 8-2) from Discriminate.Input to If.Input. This arc is translated into the following clause in the comprehension axiom for this plan.

$$\text{input}(\alpha) = \text{input}(\beta)$$

This is an example of data flow between the untyped roles of two operations. In other words, what is being passed between these two operations is being treated as a name. The other data flow arc in Fig. 8-2 is between Discriminate.Output (a set) and If.Universe (a set). For typed roles, the rules is to write the equality in terms of the behavior function and the appropriate situational role, such as In or Out, e.g.

$$\text{set}(\text{output}(\alpha),\text{out}(\alpha)) = \text{set}(\text{universe}(\beta),\text{in}(\beta)).$$

The distinction between whether or not a data flow equality involves a behavior function is similar to the distinction between "call by name" and "call by value" in some programming languages.

Fig. 8-3 shows data flow involving data plan roles. In particular, different parts of Old and New are inputs and outputs of Bump and Update. These data flows are translated into the equalities listed on separate lines of the comprehension axiom in Table IX. The first of these is

$$\text{integer}(\text{input}(\alpha),\text{in}(\alpha)) = \text{natural}(\text{lower}(\text{upper-segment}(x,\text{in}(\alpha))),\text{in}(\alpha)).$$

This is the translation of the arc from Old.Lower to Bump.Input in the plan representation. Notice how the behavior functions have been supplied on both sides above,[1] and that the situational arguments are the In situation of the consuming operation.

$$\text{sequence}(\text{old}(\beta),\text{in}(\beta)) = \text{sequence}(\text{base}(\text{upper-segment}(x,\text{in}(\beta))),\text{in}(\beta))$$

The next data flow arc, shown above, is from Old.Base to Update.Old. Here again we have behavior functions on both sides, with the same situational argument, namely Update.In. The translation of the two data flow arcs involving New are similar, as shown below.

---

1. The input and output of @Oneminus are of type integer. Natural is a specialization of Integer.

integer(output(α),out(α)) = natural(lower(upper-segment(y,out(α))),out(α))
sequence(new(β),out(β)) = sequence(base(upper-segment(y,out(β))),out(β))

Finally, examples of the compact notation for writing the axioms for temporal plans are shown at the bottom of Table VIII and Table IX. In general for a temporal plan P with roles f,g,..., we write the following axiom of extensionality.

$$\forall \alpha\beta\ [\ [P(\alpha) \wedge P(\beta) \wedge f(\alpha) = f(\beta) \wedge g(\alpha) = g(\beta) \wedge ...] \supset \alpha = \beta]$$

The axiom of comprehension is of the following form, where f,g,..., are temporal roles with types T,U,... and k,l,... are data roles with types A,B,... .

$$\forall xy...vw...\ [\ [T(x) \wedge U(y) \wedge ...$$
$$\wedge A(v,...) \neq \text{undefined} \wedge B(w,...) \neq \text{undefined} \wedge ...$$
$$\wedge C(x,y,...,v,w,...)]$$
$$\leftrightharpoons \exists \alpha\ [P(\alpha) \wedge f(\alpha) = x \wedge g(\alpha) = y \wedge ... \wedge k(\alpha) = v \wedge l(\alpha) = w \wedge ...]\ ]$$

The constraint relation C above is derived by expanding abbreviations in the constraints of the compact notation in a similar manner to the way abbreviations are expanded in compact input-output specifications. In particular, implicit applications of behavior functions with appropriate situational arguments is provided for path names which terminate in roles typed by behavior functions. For example,

.if.universe

in the constraints of Discriminate+member? is expanded to

set(.if.universe,.if.in) .

C also includes constraints that guarantee an object used in more than one situation is the same in all situations of use. Table IX illustrates all these conventions. To facilitate comparison, the constraints of Bump+update in the compact notation are written in the same order line by line as in the fully written out axiom above in the table. The first two lines of the compact notation in Table IX following the control flow constraint illustrate how situational arguments can be explicitly indicated in the compact notation by subscripts.

Conditional Plans

Fig. 8-4 is an example of a conditional plan which computes absolute value.[1] The formal axioms for this plan, written in compact notation, are as follow.

TemporalPlan Abs
roles .if(lt-zero?) .then(@negative) .end(join-outputs)
constraints cflow(.if.succeed,.then.in)
∧ cflow(.then.out,.end.succeed)
∧ cflow(.end.fail,.end.fail)
∧ .if.input = .then.input
∧ .then.output = .end.succeed-input
∧ .if.input = .end.fail-input

This plan has two key features which are typical of conditional plans in general. First, notice the control flow arc from If.Succeed to Then.In. The intuitive meaning of this arc is that the @Negative operation is to be performed only if the test succeeds. This is expressed formally as the following property of the Abs plan, which follows from the way tests, operations and control flow have been axiomatized.

$$\forall \alpha\ [\mathrm{abs}(\alpha) \supset [\mathrm{in}(\mathrm{then}(\alpha)) \neq \bot \leftrightharpoons \mathrm{lt}(\mathrm{input}(\mathrm{if}(\alpha)),0)]]$$

Second, notice the data flow and control flow arcs involving the join (End). The meaning of these arcs is that the output of the join is either If.Input or Then.Output, depending on whether the test succeeds or fails. Stated formally, we want the Abs plan to have the following property.

$$\forall \alpha\ [\mathrm{abs}(\alpha) \supset [[\mathrm{lt}(\mathrm{input}(\mathrm{if}(\alpha)),0) \supset \mathrm{output}(\mathrm{end}(\alpha)) = \mathrm{negative}(\mathrm{input}(\mathrm{if}(\alpha)))] \wedge [\neg \mathrm{lt}(\mathrm{input}(\mathrm{if}(\alpha)),0) \supset \mathrm{output}(\mathrm{end}(\alpha)) = \mathrm{input}(\mathrm{if}(\alpha))]]]$$

This is achieved by axiomatizing joins (with one output) as shown in Table X. Joins are like the mirror images of tests. Joins have three situational roles, Succeed, Fail, and Out. Like tests, only one of either Succeed or Fail is reached in any instance. Unlike tests, however, joins do not represent any real computation, since the Out situation is always equal to either the Succeed or Fail situation, depending on which is reached. The purpose of the join is to state, in a modular fashion, the connection between which way a test goes and which of two possible outputs is made available for further computation. The two possible outputs are the Succeed-input and Fail-input roles of the join. One of these is equal to the Output role (which one depends on whether Succeed or Fail is reached), from which data flow arcs to following computations emanate.

---

1. Lt-zero? is the test for less than zero. @Negative computes the negative of an integer.

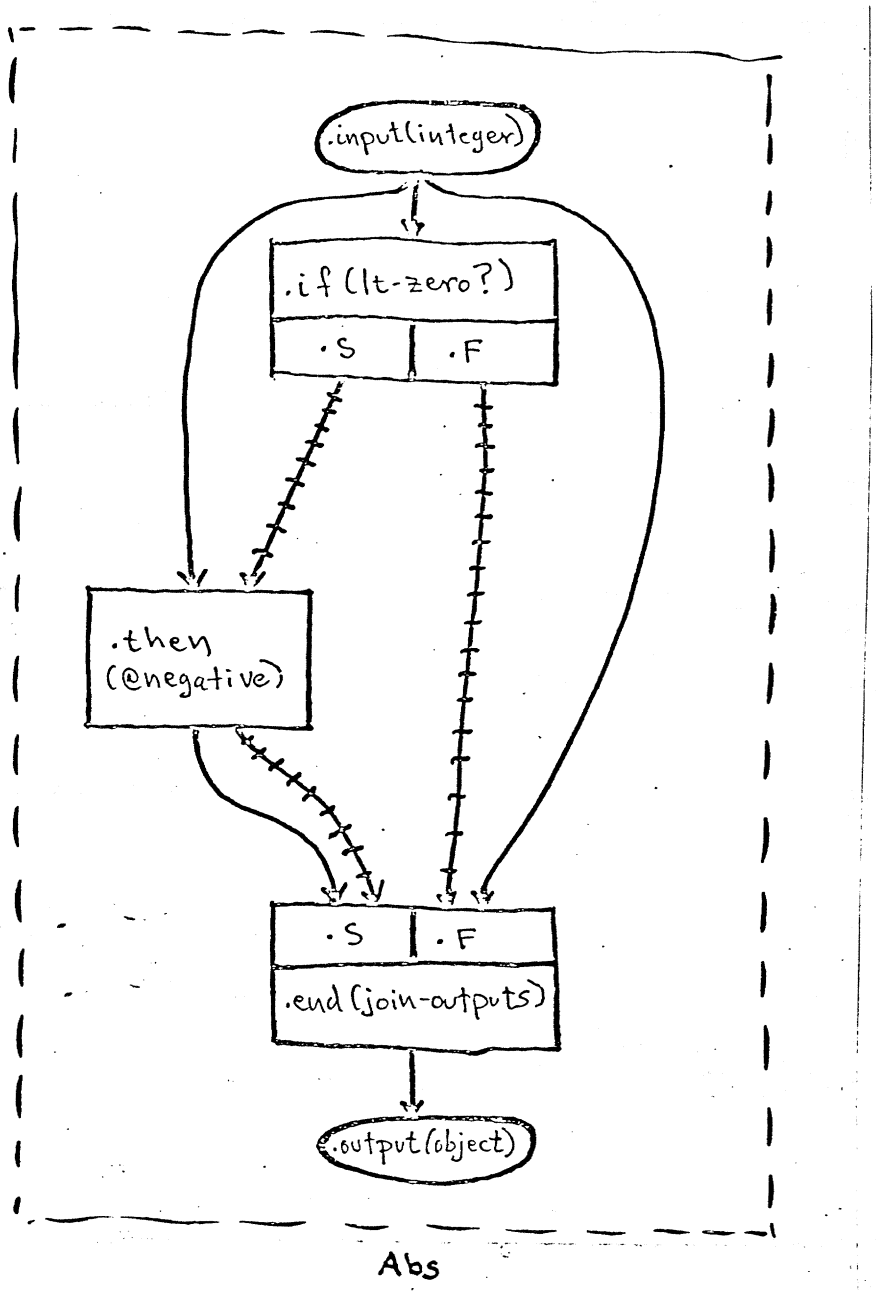


Figure 8-4. A Conditional Plan.

Table X. Joining Outputs.

Axiom of Extensionality

$$\forall \alpha\beta\ [\,[\text{join-outputs}(\alpha) \wedge \text{join-outputs}(\beta) \wedge \text{succeed}(\alpha)=\text{succeed}(\beta) \wedge \text{fail}(\alpha)=\text{fail}(\beta) \wedge \text{out}(\alpha)=\text{out}(\beta) \wedge \text{succeed-input}(\alpha)=\text{succeed-input}(\beta) \wedge \text{fail-input}(\alpha)=\text{fail-input}(\beta)] \supset \alpha=\beta]$$

Axiom of Comprehension

$$\forall stuxyz\,[\,[\,[t=\perp \vee u=\perp] \wedge [t=\perp \supset [s=u \wedge x=z]\,] \wedge [u=\perp \supset [s=t \wedge x=y]\,]\,] \leftrightharpoons \exists\alpha\,[\text{join-outputs}(\alpha) \wedge \text{out}(\alpha)=s \wedge \text{succeed}(\alpha)=t \wedge \text{fail}(\alpha)=u \wedge \text{output}(\alpha)=x \wedge \text{succeed-input}(\alpha)=y \wedge \text{fail-input}(\alpha)=z]\,]$$

8.8 Temporal Overlays

A temporal overlay is formally a function from one computation type to another. Furthermore, like data overlays, this function must be total in both directions. For example, consider the temporal overlay shown in Fig. 8-5, which expresses how to view instances of the temporal plan Discriminate+member? as implementing membership tests in a set implemented as a discrimination function.

The formal definition and totality axioms for this overlay are given in Table XI. Each correspondence in the figure becomes an equality in the formal definition. Unlabelled correspondences, such as between Discriminate+member?.Discriminate.Input on the left and Member?.Input on the right become simple equalities such as

$$\text{input}(\beta)=\text{input}(\text{discriminate}(\alpha))\,.$$

Since overlays can be used in defining other overlays, some correspondences in temporal overlays are labelled with the names of other overlays. For example, the correspondence between Discriminate+member?.Discriminate.Op on the left and Member?.Universe on the right is labelled with the Discrimination>set overlay.[1] Intuitively, this means that Discriminate+member?.Discriminate.Op is viewed as implementing Member?.Universe according to Discrimination>set. This is written formally in the definition of Discriminate+member?>member? as follows.

---

1. This is a data overlay similar to Sequence-of-sets>set introduced earlier in this chapter.

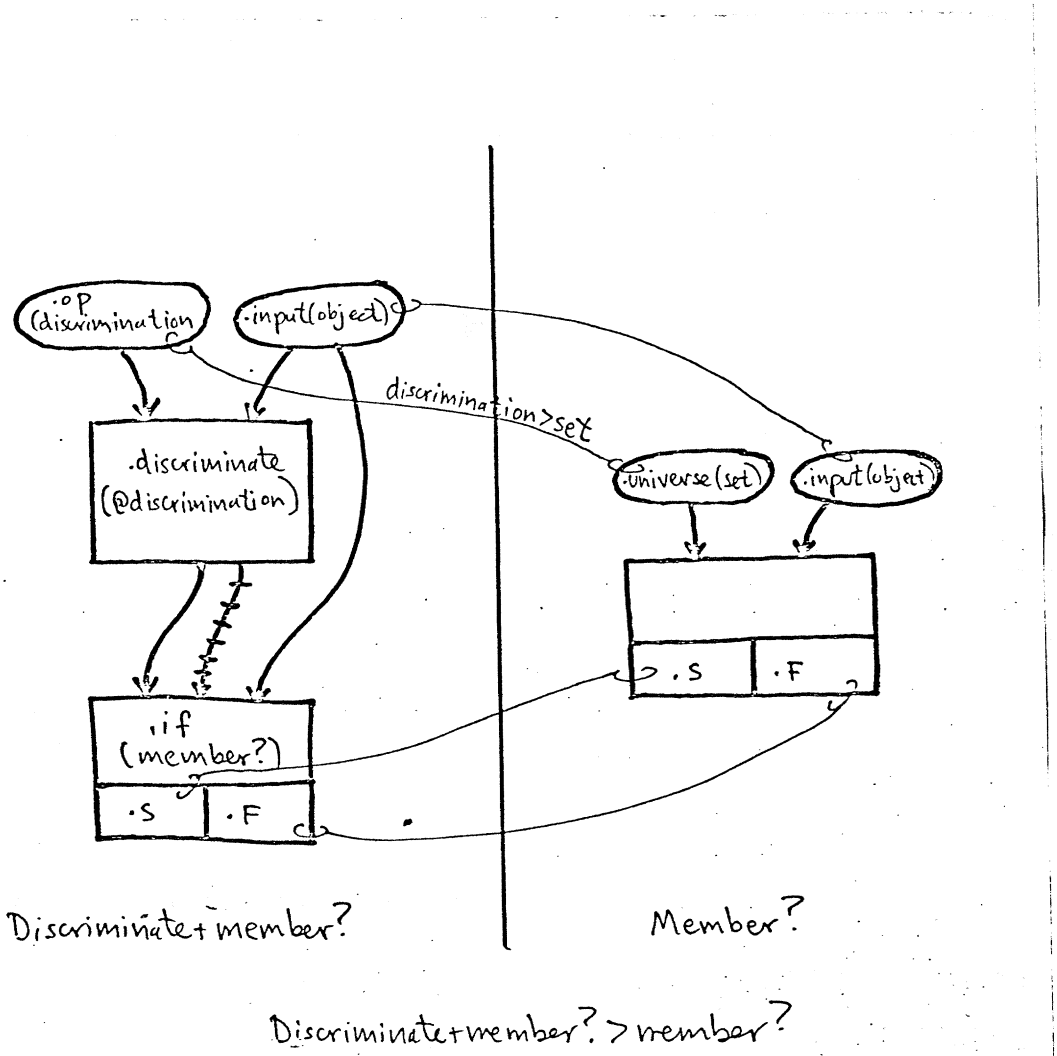


Figure 8-5. A Temporal Overlay.

Table XI. Implementing Membership in a Discrimination

Totality Axioms

$\forall\alpha$ [discriminate+member?(α) $\supset$ $\exists\beta$ [member?(β) $\wedge$ β = discriminate+member?>member?(α)]]

$\forall\beta$ [member?(β) $\supset$ $\exists\alpha$ [discriminate+member?(β) $\wedge$ β = discriminate+member?>member?(α)]]

Definition

β = discriminate+member?>member?(α) $\equiv$ [member?(β)
$\wedge$ set(universe(β),in(β)) = discrimination>set(op(discriminate(α)),in(discriminate(α)))
$\wedge$ input(β) = input(discriminate(α))
$\wedge$ in(β) = in(discriminate(α))
$\wedge$ fail(β) = fail(if(α))
$\wedge$ succeed(β) = succeed(if(α))]

TemporalOverlay Discriminate+member?>member?: discriminate+member? → member?
correspondences
member?.universe = discrimination>set(discriminate+member?.discriminate.op)
$\wedge$ member?.input = discriminate+member?.discriminate.input
$\wedge$ member?.in = discriminate+member?.discriminate.in
$\wedge$ member?.fail = discriminate+member?.if.fail
$\wedge$ member?.succeed = discriminate+member?.if.succeed

---

set(universe(β),in(β)) = discrimination>set(op(discriminate(α)),in(discriminate(α)))

Notice that behavior functions are supplied for typed roles with the appropriate situational arguments as usual. In general, the definition of an overlay V from computation type T to computation type U, where f,g,... are the role functions of U, is of the following form.

$$\beta = V(\alpha) \equiv [U(\beta) \wedge f(\beta) = ...\alpha... \wedge g(\beta) = ...\alpha... \wedge ...]$$

In other words, there is an equality for each role of β in terms of some function of α. This form, together with the extensionality axiom of U, guarantees the uniqueness property of the function V.

As with data overlays, it is more convenient to use a compact tabular notation than to write out the definiton and axioms for a temporal overlay as in Table XI. An example of the tabular notation is shown at the bottom of the table. In general, from the header line

TemporalOverlay V: T → U

the following two totality axioms are written.

$$\forall\alpha\,[\Gamma(\alpha) \supset \exists\beta\,[U(\beta) \wedge \beta = V(\alpha)]\,]$$
$$\forall\beta\,[U(\beta) \supset \exists\alpha\,[\Gamma(\alpha) \wedge \beta = V(\alpha)]\,]$$

The definition of the overlay function is abbreviated in the tabular notation by listing only the equalities and leaving behavior functions and situational arguments implicit in the usual way.

8.9 Specialization and Extension

In this section we discuss two additional ways of making use of already defined plans in defining new ones: specialization and extension.

Specialization

The basic idea of specialization is to define a type whose instances are a subset of another type. A common motivation for doing this is to exploit the properties of the subtype in some particular implementation. For example, we have earlier in this chapter defined a general data plan, Segment, involving an upper and lower index to a base sequence. One way of implementing a mutable stack is to use an instance of Segment in which only the lower index is varied -- the upper index is always equal to the length of the base sequence. We called this plan Upper-segment. The formal relation between Upper-segment and Segment is captured by the following statement.

$$\sigma = \text{upper-segment}(p,s) \equiv [\sigma = \text{segment}(p,s) \wedge \text{natural}(\text{upper}(\sigma),s) = \text{length}(\text{sequence}(\text{base}(\sigma),s))]$$

Thus Upper-segment is Segment with additional constraint. In tabular notation, this will be written as follows.

DataPlan Upper-segment ***specialization*** segment
roles .base(sequence) .lower(natural) .upper(natural)
constraints .upper = length(.base)

Notice that a specialization has the same roles as the more general plan, and that the application of behavior functions of the appropriate type for each role is abbreviated in the constraints in usual manner.

The specialization of computation types is similar. For example, the following is the general input-output specifications for finding a node in a directed graph (Digraph), which satisfies a given predicate.

IOspec Digraph-find / .universe(digraph) .criterion(predicate) ⇒ .output(object)
preconditions $\exists x$ [node(.universe,x) ∧ apply(.criterion,x) = true]
postconditions node(.universe,.output) ∧ apply(.criterion,.output) = true

An important special case of directed graphs is threads, in which each node has a unique successor and there are no cycles. Finding nodes in threads is considerably simpler than the general case. The computation type of such operations is specified formally as follows.

$$\text{thread-find}(\alpha) \equiv [\text{digraph-find}(\alpha) \wedge \text{thread}(\text{old}(\alpha),\text{in}(\alpha)) \neq \text{undefined}]$$

Thus the additional constraint here is an additional type restriction on the Old role. (The behavior function Thread is the appropriate specialization of Digraph.) This is written in the compact tabular notation as follows.

IOspec Thread-find / .universe(thread) .criterion(predicate) ⇒ .output(object)
specialization digraph-find

Of course, computation types can also be specialized by additional constraints between roles. For example, in this thesis set addition by side effect, #Set-add, is viewed as a specialization set addition in general. This is expressed formally as follows.

$$\#\text{set-add}(\alpha) \equiv [\text{set-add}(\alpha) \wedge \text{old}(\alpha) = \text{new}(\alpha)]$$

In other words, instances of #Set-add are those instances of Set-add in which the Old and New set objects are identical. In tabular notation, this will be written as follows.

IOspec #Set-add / .old(object) .input(object) ⇒ .new(object) ***specialization*** set-add
postconditions .old = .new

Notice that the type restrictions on Old and New above are Object rather than Set, as in Set-add. This usage is essentially a syntactic trick to control the abbreviation that will be applicable in the postcondition above. Logically, an Object restriction is weaker than a Set restriction, so no information is added.

Extension

The basic idea of extension is to define a new type with an additional role function, such that instances of the new type have the same constraints as the old type between those roles which are in common. The formalization of extension is more complicated than the formalization of specialization in the preceding section. In the case of specialization, the new behavior function or predicate on computations can be simply defined in terms of the old one. For extension, however, new extensionality and comprehension axioms need to be written for the new type. However, these new axioms are related to those of the old type in a systematic way.

Table XII. Internal Thread Find.

Axiom of Extensionality

$$\forall \alpha\beta\ [\,[\text{internal-thread-find}(\alpha) \wedge \text{internal-thread-find}(\beta) \wedge \text{in}(\alpha)=\text{in}(\beta) \wedge \text{out}(\alpha)=\text{out}(\beta)$$
$$\wedge\ \text{universe}(\alpha)=\text{universe}(\beta) \wedge \text{criterion}(\alpha)=\text{criterion}(\beta) \wedge \text{output}(\alpha)=\text{output}(\beta)$$
$$\underline{\wedge\ \text{previous}(\alpha)=\text{previous}(\beta)}]$$
$$\supset \alpha=\beta]$$

Axiom of Comprehension

$$\forall wxyzst\ [\,[\exists\alpha\ [\text{thread-find}(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{out}(\alpha)=t$$
$$\wedge\ \text{universe}(\alpha)=w \wedge \text{criterion}(\alpha)=x \wedge \text{output}(\alpha)=y]$$
$$\underline{\wedge\ z\neq\text{undefined}}$$
$$\underline{\wedge\ \text{apply}(\text{predicate}(x,s),\text{root}(\text{thread}(w,s)))=\text{false}}$$
$$\underline{\wedge\ \text{successor}(\text{thread}(w,s),z,y)}]$$
$$\leftrightharpoons \exists\beta\ [\text{internal-thread-find}(\beta) \wedge \text{in}(\beta)=s \wedge \text{out}(\beta)=t$$
$$\wedge\ \text{universe}(\beta)=w \wedge \text{criterion}(\beta)=x \wedge \text{output}(\beta)=y \wedge \text{previous}(\beta)=z]\,]$$

IOspec Internal-thread-find / .universe(thread) .criterion(predicate)
⇒ .output(object) .previous(object)

extension thread-find
preconditions apply(.criterion,root(.universe))=false
postconditions successor(.universe,.previous,.output)

∧ common use of extension is to add an additional output to an input-output specification. For example, when Thread-find operations are used in conjunction with other plans, such as splicing, it is convenient to output not only the node found, but also the previous node in the thread. We call this extra role Previous, and the extension type Internal-thread-find.

The axioms for Internal-thread-find are shown in Table XII. They are derived from the axioms of Thread-find by adding the underlined portions. In the axiom of extensionality, an additional equality is added for the Previous role. The axiom of comprehension specifies the constraints on the new type by first referring to the corresponding instance of the old type,

$$\exists\alpha\ [\text{thread-find}(\alpha) \wedge \text{in}(\alpha)=s \wedge \ldots]$$

and then specifying the added underlined constraints, which include the type restriction on the new role and some additional constraints between this role and the others. One could think of this as an extension step, followed by a specialization, but in practice one almost never adds a new role without relating it to the old roles. As usual the more compact tabular notation is shown at the bottom of the table.

CHAPTER NINE

LOOPS AND TEMPORAL ABSTRACTION

9.1 Introduction

Like many other formal languages, the plan calculus uses self-referential (i.e. recursive) definitions to represent unbounded structures. This chapter concentrates on the special case of singly recursive plans, and loops in particular. The generalization of these ideas to multiple recursion will be discussed briefly at the end of the chapter.

We begin in Table I with a minimal plan, Single-recursion, which says nothing more than that there is a role, Tail, constrained to be either an instance of Nil or itself a Single-recursion. A finite single recursion is defined as one whose tail is nil or eventually has a nil tail. "Eventually" is defined by the transitive closure tail relation, Tail^*, which is in turn defined in terms of the n-th tail relation, Tail^n. The Null predicate is introduced as a shorthand for saying that an object is an instance of Nil.

Table I. Single Recursion.

DataPlan **single-recursion**
  roles .tail(single-recursion+nil)

Type **single-recursion+nil** *uniontype* single-recursion nil

DataPlan **finite-single-recursion** *specialization* single-recursion
  roles .tail(single-recursion+nil)
  constraints .tail = nil $\vee$ $\exists x$ (tail*(.tail,x) $\wedge$ x = nil)

Function **tail*** : single-recursion $\rightarrow$ object
  properties $\forall R$ [instance(single-recursion,tail*(R)) $\vee$ tail*(R) = nil]
  definition x = tail*(R) $\equiv$ [$\exists n$ tail^n(n,R) = x]

Binfunction **tail^n** : natural $\times$ single-recursion $\rightarrow$ single-recursion+nil
  definition x = tail^n(n,R) $\equiv$ [(n = 1 $\wedge$ R.tail = x)
    $\vee$ tail^n(oneminus(n),R.tail) = x)]

The most common singly recursive data plan, List, has already been discussed in Chapter Eight. The next section in this chapter will concentrate on how loops, the most common singly recursive temporal plans, are represented in the plan calculus. The section following then shows how to represent the relationship between singly recursive temporal plans (loops) and and singly recursive data plans (lists) using overlays. Finally, note that the taxonomy of loops discussed in this chapter covers only loops with a single jump from the end of the loop to the beginning (i.e. interleaved loops are not included).

9.2 Loops

Since the temporal order relation on situations is not allowed to have any cycles, loops are represented in the plan calculus as singly recursive plans where the jump from the end of the loop to the beginning is viewed as a recursive invocation. For example, Fig. 9-1 is a diagram of the SEARCHLIST program below.

```
(DEFINE SEARCHLIST
   (LAMBDA (L P)
      (PROG (ENTRY)
         LP (SETQ ENTRY (CAR L))
            (COND ((FUNCALL P ENTRY)(RETURN ENTRY)))
            (SETQ L (CDR L))
            (GO LP))))
```

The Tail role, which represents the recursive invocation of the loop body, is constrained to be an instance of the same plan as the outside plan. This is indicated in plan diagrams by a spiral line from the outside plan box to the recursive role. The operation boxes in the diagram are instances of @Function; the test boxes are instances of @Predicate; and the join boxes are instances of Join-outputs. Thus we are viewing the program as if it were coded as follows.

```
(DEFINE SEARCHLIST
   (LAMBDA (L P)
      (PROG (ENTRY)
            (LP))))

(DEFINE LP
   (LAMBDA ()
            (SETQ ENTRY (CAR L))
            (COND ((FUNCALL P ENTRY)(RETURN ENTRY)))
            (SETQ L (CDR L))
            (LP)))
```

In fact, recent work on Lisp interpreters and compilers suggests that the distinction between loops and single recursions where the recursive call is the last step of the program (so called "tail recursions") can be considered only a superficial syntactic variation. In Scheme, a dialect of Lisp developed by Sussman and Steele, the PROG with GO construction is provided as a macro which expands into a single recursion similar to the example above. The Scheme interpreter executes tail recursions without accumulating list depth. The compiler for Scheme also views looping constructs

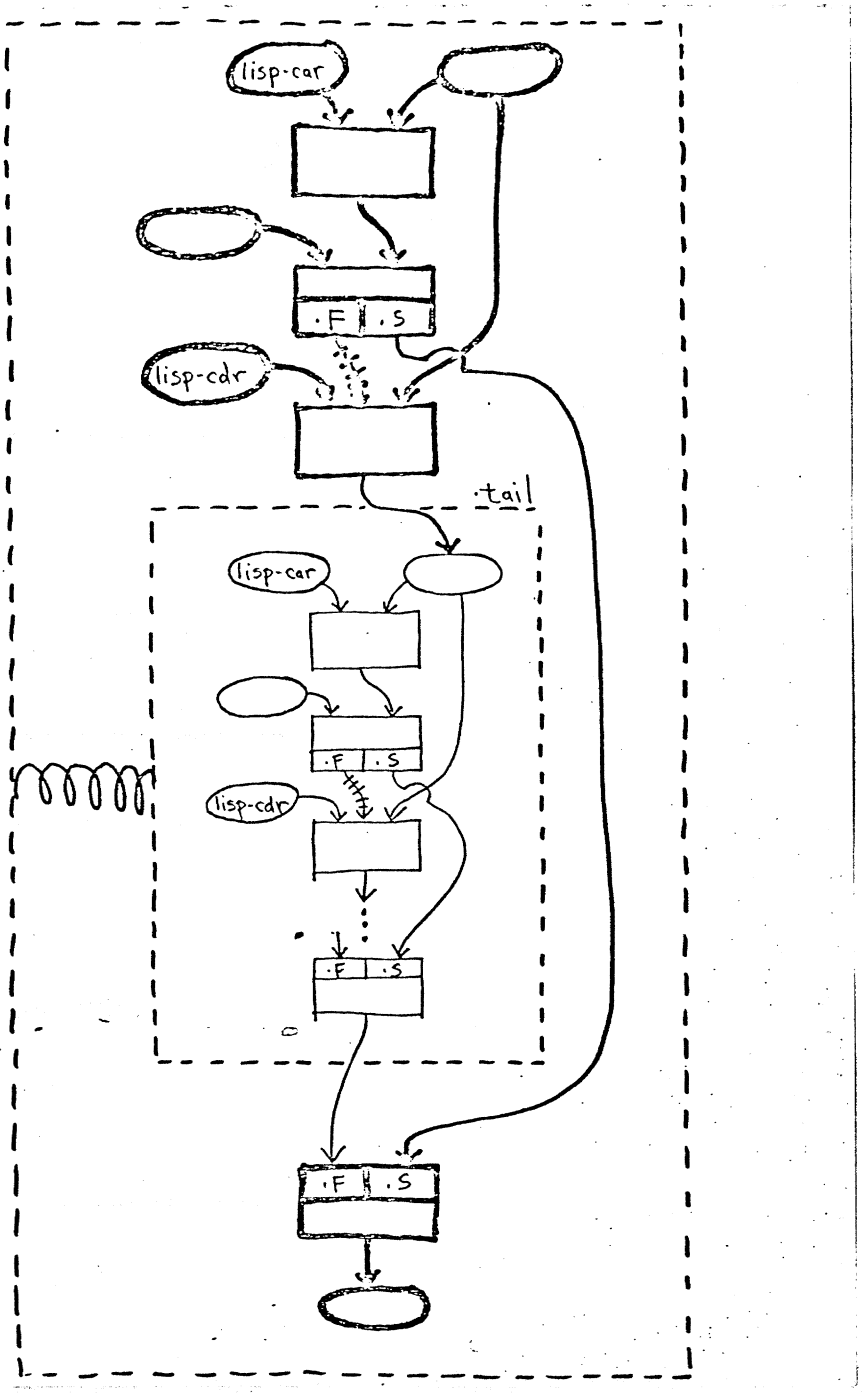


Figure 9-1. A Tail Recursive Temporal Plan.

as macros which expand into singly recursive structures. This special case of tail recursive temporal plans is often referred to as the "iterative" case.

Given this representation for loops, it is possible to formalize a small set of the basic plans which decompose many loops into intuitively meaningful parts. The remainder of this section will present these plans, along with explanations and some typical fragments of code which are instances. The taxonomy of loops presented here is an extension of the work of Waters [60].

Steady State Plans

To begin let us ignore any exits from a loop and the question of termination. This is what I call the "steady state" model. This viewpoint will be formalized later as an overlay which explicitly assumes that the loop does not exit.

One of the most common computations in a loop is to repeatedly apply a given function (the same function each time) to the output of the preceding application of that function. This pattern of application is in general (i.e. for multiply recursive plans) called *generation*. The special case for loops is called iterative generation, as shown in Table II and Fig. 9-2. Notice that the starting value for the generation is Action.Input, the input to the first application. The SEARCHLIST example contains an instance of Iterative-generation, as shown below, where the function being applied is Cdr and the variable L holds the successive values generated.

```
(PROG (...)
   LP ...
      (SETQ L (CDR L))
      (GO LP))))
```

Table II. Iterative Steady State Plans.

TemporalPlan **iterative-application** *extension* single-recursion
roles .action(@function) .tail(iterative-application)
constraints .action.op = .tail.action.op ∧ cflow(.action.out,.tail.action.in)

TemporalPlan **iterative-generation** *specialization* iterative-application
roles .action(@function) .tail(iterative-generation)
constraints .action.output = .tail.action.input

TemporalPlan **iterative-filtering** *extension* single-recursion
roles .filter(cond) .tail(iterative-filtering)
constraints instance(@predicate,.filter.if)
∧ .filter.if.criterion = .tail.filter.if.criterion
∧ cflow(.filter.end.out,.tail.filter.if.in)

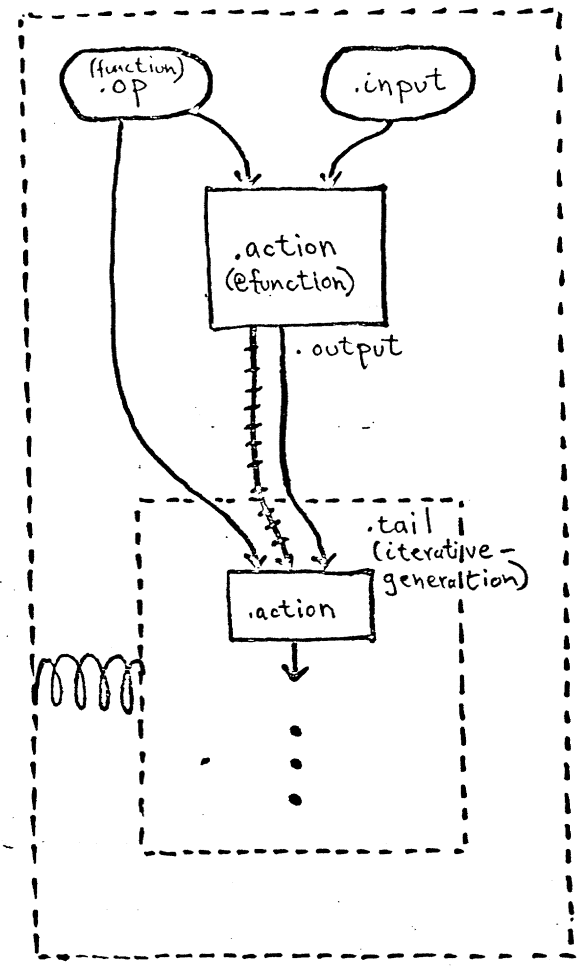


Figure 9-2. Iterative Generation Plan.

Using SETQ this way is the most common way of coding iterative generation in Lisp, but there are other ways of achieving the necessary data flow, as illustrated below.

```
(DEFINE LP
   (LAMBDA (L)
      ...
      (LP (CDR L))))
```

A particulary common special case of iterative generation is Counting, where the function applied is Oneplus and the initial input is 1.

TemporalPlan **counting** *specialization* iterative-generation
roles .action(@function) .tail(counting)
constraints .action.op = oneplus ∧ .action.input = 1

The Iterative-application plan shown in Table II and Fig. 9-3, captures the idea of repeatedly applying a given function to an input which is generated by some other part of the loop. The output of this application may then be the input to some other repeated computation. The application role in this plan is called the Action. For example in SEARCHLIST, the function CAR is applied to current value of L to get a new ENTRY each time around the loop.[1]

```
(PROG (...)
   LP (SETQ ENTRY (CAR L))
      ...
      (GO LP))
```

These two simple plans, Iterative-generation and Iterative-application, together with a number of common specializations, such as counting and CDRing, form the backbone of many common computations. For example, we have just analyzed the CAR-CDRing in SEARCHLIST as iterative generation and application. A similar programming cliche is looping through an array:

```
(PROG (I)
      ...
      (SETQ I 1)
   LP ...
      ...(ARRAYFETCH A I)...
      (SETQ I (1+ I))
      (GO LP)
      ...)
```

Here the iterative generation is counting and the application is fetching from the array.

The final iterative plan in Table II is Iterative-filtering, also shown in Fig. 9-4. Typically it is used to select some subset of the values of a loop variable for special processing, as suggested by the following code.

---

1. Iterative-generation is in fact a specialization of Iterative-application, as can be seen in Table II.

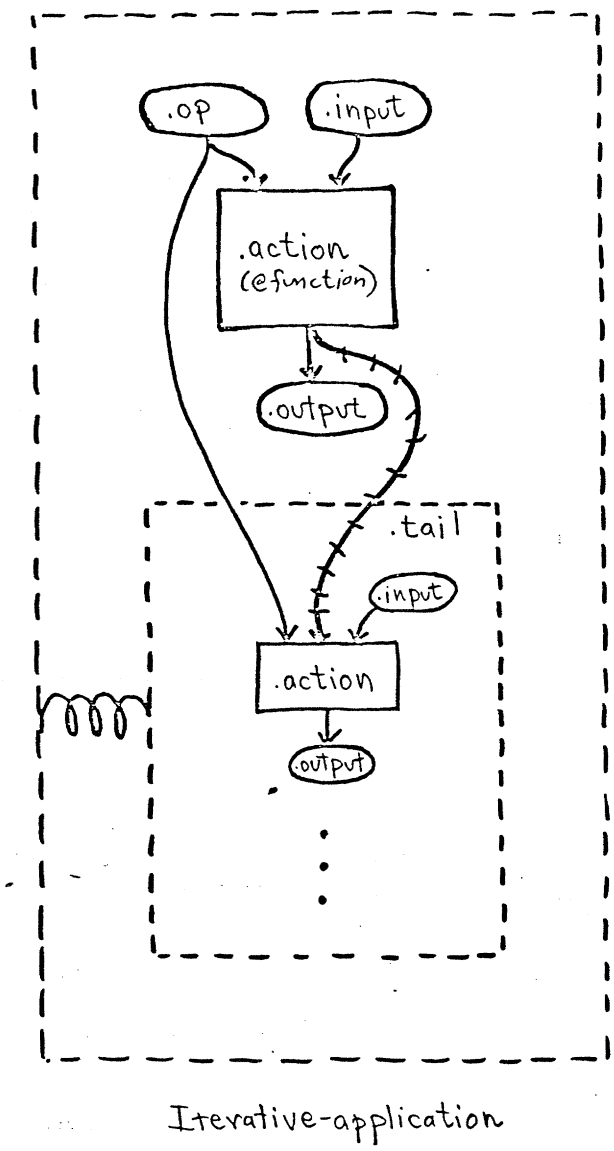


Figure 9-3. Iterative Application Plan.

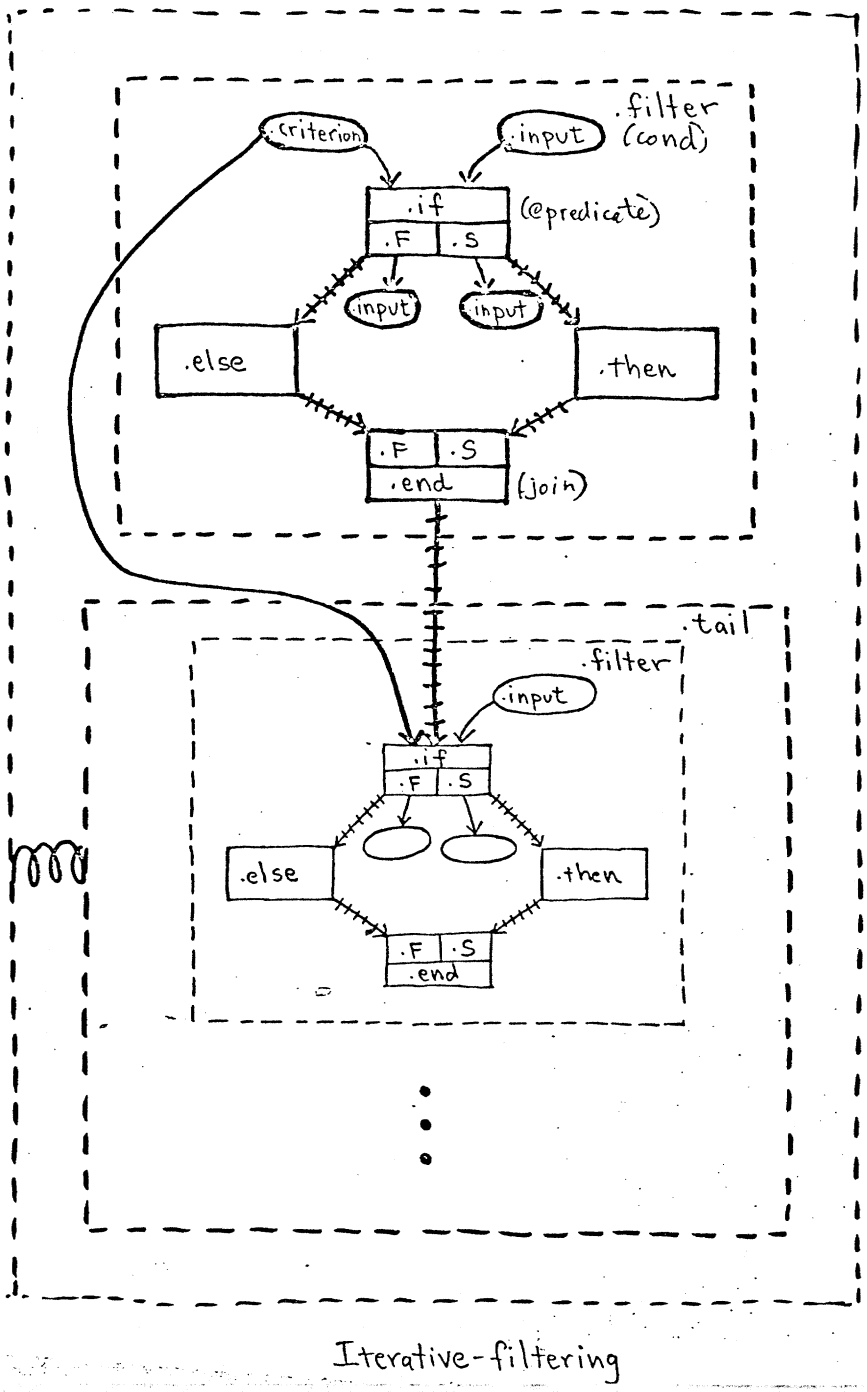


Figure 9-4. Iterative Filtering Plan.

```
(PROG (A)
      ...
   LP ...
      (COND ((P A) ...A...))
      ...
      (GO LP))
```

The non-recursive role in this plan, Filter, is a conditional structure (Cond). Each time around the loop, a given predicate (the same one each time) is used to test some object provided by the rest of the loop. Based on the results of this test, either the Then or the Else wings of the conditional will be executed.

Table III. Iterative Termination.

TemporalPlan **iterative-termination** *extension* single-recursion
roles .exit(cond) .tail(iterative-termination+nil)
constraints (.tail = nil ⇋ .exit.if.succeed≠⊥
∧ cflow(.exit.if.fail,.tail.exit.if.in)
∧ cflow(.tail.exit.end.out,.exit.end.fail)

TemporalPlan **iterative-termination-predicate** *specialization* iterative-termination
roles .exit(cond) .tail(iterative-termination-predicate+nil)
constraints instance(@predicate,.exit.if)
∧ .exit.if.criterion = .tail.exit.if.criterion

TemporalPlan **iterative-termination-output** *specialization* iterative-termination
roles .exit(cond) .tail(iterative-termination-output+nil)
constraints instance(join-outputs,.exit.end)
∧ .tail.exit.end.output = .exit.end.fail-input

TemporalPlan **iterative-cotermination** *extension* iterative-termination-predicate
roles .exit(cond) .co-iterand(object) .tail(iterative-cotermination+nil)
constraints .co-iterand≠.exit.if.input

Type **iterative-termination+nil** *uniontype* iterative-termination nil

Type **iterative-termination-predicate+nil** *uniontype* iterative-termination-predicate nil

Type **iterative-termination-output+nil** *uniontype* iterative-termination-output nil

Type **iterative-cotermination+nil** *uniontype* iterative-cotermination nil

Termination Plans

Let us now consider loops that have exits. The minimal plan for a loop with exits is Iterative-termination, shown in Table III and Fig. 9-5. This plan describes loops with a single exit which are expected to terminate by that exit. It is similar to Iterative-filtering in that the non-recursive role, called Exit here, is an instance of Cond. In this plan, however, the recursive invocation (Tail) is constrained to occur *between* the test and the join. The succeed case of the test exits by bypassing the recursive invocation; if the exit test fails, then the exit test of the tail must occur. Furthermore if the tail of the loop exits through the end join, the whole loop ends. These control flow constraints, together with the constraints of Cond, prohibit other exits from the loop and require that the loop eventually terminates, i.e. it follows from the constraints on Iterative-termination that

cflow(.exit.if.in,.exit.end.out) .

Given a loop with an instance of Iterative-termination, the exit can be ignored for the purpose of steady state analysis by assuming that the exit test always fails. This modelling assumption is formalized by the overlay shown in Table IV. The Iterative-steady-state plan is introduced to represent a non-terminating loop, i.e. there is control flow from each Step of the iteration to the next. According to the overlay Iterative-termination>steady-state, an instance of Iterative-termination is viewed as an instance of Iterative-steady-state in which the current Step corresponds to the input situation of the exit test. The control flow constraint in the Iterative-steady-state plan thus amounts to assuming the exit test always fails.

The two specializations of Iterative-termination in Table III concern what kind of test is performed and whether a final value is output. Iterative-termination-predicate is the plan for the common case of loops where the exit test is a predicate that does not change as the computation proceeds. Iterative-termination-output is a fragment (used to build up other plans) which expresses the pattern of data flow needed in a loop to make the final value of some iterand available as an output of the entire loop when it is done. For such loops, the End join is an instance of Join-outputs,

---

Table IV. Steady State Model.

TemporalOverlay **iterative-termination>steady-state:** iterative-termination → iterative-steady-state
correspondences
iterative-termination.exit.if.in = iterative-steady-state.step
∧ iterative-termination>steady-state(iterative-termination.tail)
= iterative-steady-state.tail

TemporalPlan **iterative-steady-state**
roles .step(situation) .tail(iterative-steady-state)
constraints cflow(.step,.tail.step)

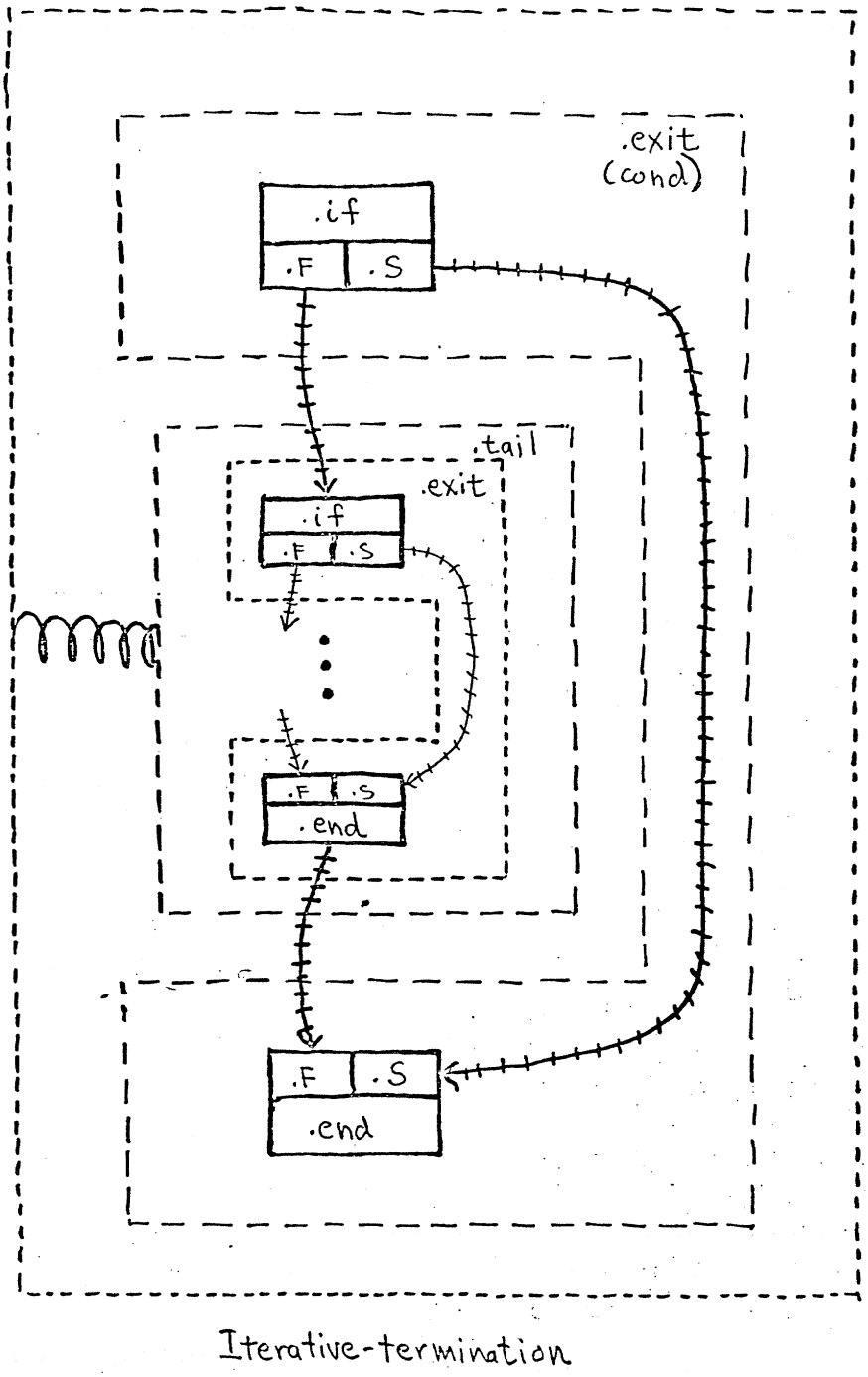


Figure 9-5. Iterative Termination Plan.

and the failure case of each join has data flow from the output of the End join of the tail. The final plan in this table, Iterative-coterminaton, is also a fragment used to build up other plans. In this case, a Co-iterand role is added to Iterative-termination-predicate, which identifies an object of interest in the loop other than the input to the test.

Given these fragments, Iterative-search, shown in Table V and Fig. 9-6, can be defined simply as a specialization of both Iterative-termination-predicate and Iterative-termination output. The only additional constraint added to express the idea of a search loop is that the output object is the final object that satisfied the predicate of the exit test. For example, in the SEARCHLIST program the value of ENTRY on the last repetition is returned.

```
(PROG (ENTRY)
   LP ...
      (COND ((FUNCALL P ENTRY)(RETURN ENTRY)))
      ...
      (GO LP))
```

A closely related kind of search loop is one in which the object returned is not the same as the object tested in the exit test, as for example the program below, which calculates the length of a Lisp list.

```
(DEFINE LENGTH
   (LAMBDA (L)
      (PROG (N)
            (SETQ N 0)
         LP (COND ((NULL L)(RETURN N)))
            (SETQ L (CDR L))
            (SETQ N (1+ N))
            (GO LP))))
```

Table V. Iterative Search.

TemporalPlan **iterative-search**
specialization iterative-termination-predicate iterative-termination-output
roles .exit(cond) .tail(iterative-search+nil)
constraints .exit.if.input = .exit.end.succeed-input

TemporalPlan **iterative-cosearch** *specialization* iterative-cotermination
extension iterative-termination-output
roles .exit(cond) .co-iterand(object) .tail(iterative-cosearch+nil)
constraints $.\text{co-iterand}_{.\text{exit.if.in}} = .\text{exit.end.succeed-input}$

Type **iterative-search+nil** *uniontype* iterative-search nil

Type **iterative-cosearch+nil** *uniontype* iterative-cosearch nil

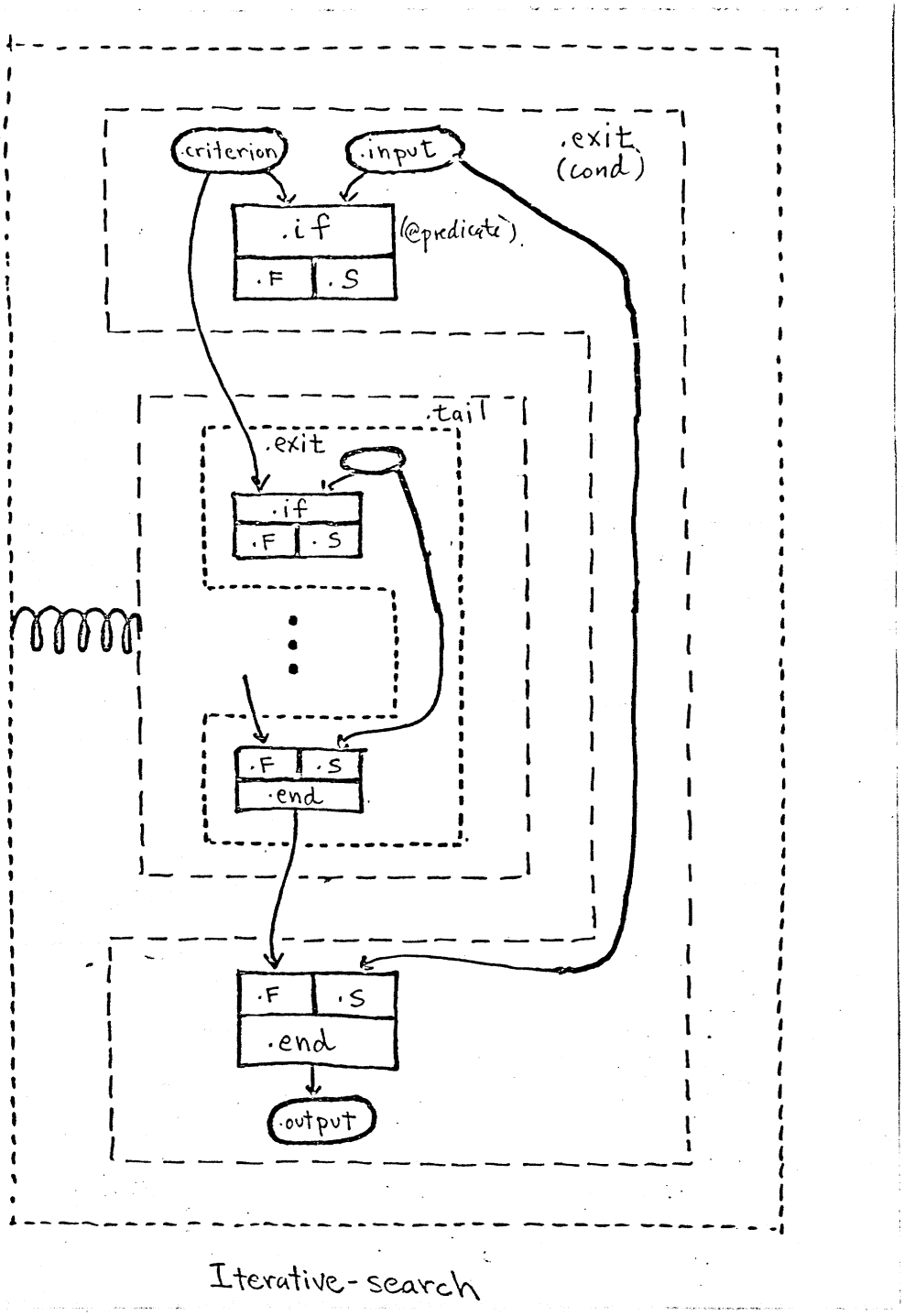


Figure 9-6. Iterative Search Plan.

Here consecutive values of L (generated by CDR) are tested for NULL, while at the same time N is counting. When the NULL test finally succeeds, the current value of N is returned as the output of the loop. This pattern of loop termination is formalized as the Iterative-cosearch plan shown in Table V. This plan is a specialization of Iterative-cotermination and an extension of Iterative-termination-output (adding the Co-iterand role), in which the output object is constrained to be the final co-iterand when the loop exits.

A third basic loop plan which returns an output is Iterative-accumulation, shown in Table VI and Fig. 9-7. On each repetition of an accumulation loop, an operation (Add) is performed which takes an Old object and another input, and returns a New object of the same type. Set-add and Push are examples of such operations for sets and lists. The Old input on the first repetition is called the Init; on successive repetitions of the loop, the Old input of each Add is the same as the New output of the previous Add. The Input of Add on each repetition is provided from the rest of the loop. For example, the following is code for a typical accumulation loop.

```
(PROG (ACCUM ...)
      (SETQ ACCUM ...)
      ...
   LP (COND ((...)(RETURN ACCUM)))
      ...
      (SETQ ACCUM (CONS ... ACCUM))
      ...
      (GO LP))
```

Here the Add operations are instances of Push (implemented as CONS for Lisp lists). The value of ACCUM returned from the loop (Exit.End.Output) is the same as the New output of the last Push, except when the loop exits the first time, when the value of ACCUM is determined by the initial SETQ (Init).

Specializations of Iterative-accumulation for different types of Add and Init correspond to common programming idioms. If the Add roles are filled by instances of Push and the Init is an instance of Nil, then we are accumulating the Input's as a list. If the Add roles are filled by instances of Set-add and the Init is an empty set, then we are accumulating the Input's as a set. Applications of

---

Table VI. Iterative Accumulation.

TemporalPlan **iterative-accumulation** *extension* iterative-termination-output
roles .exit(cond) .init(object) .add(old+input+new) .tail(iterative-accumulation)
constraints .init=.add.old ∧ .init=.exit.end.succeed-input
∧ .add.new=.tail.init ∧ cflow(.exit.if.fail,.add.in)
∧ cflow(.add.out,.tail.exit.if.in) ∧ ∧ cflow(.tail.end.out,.end.fail)

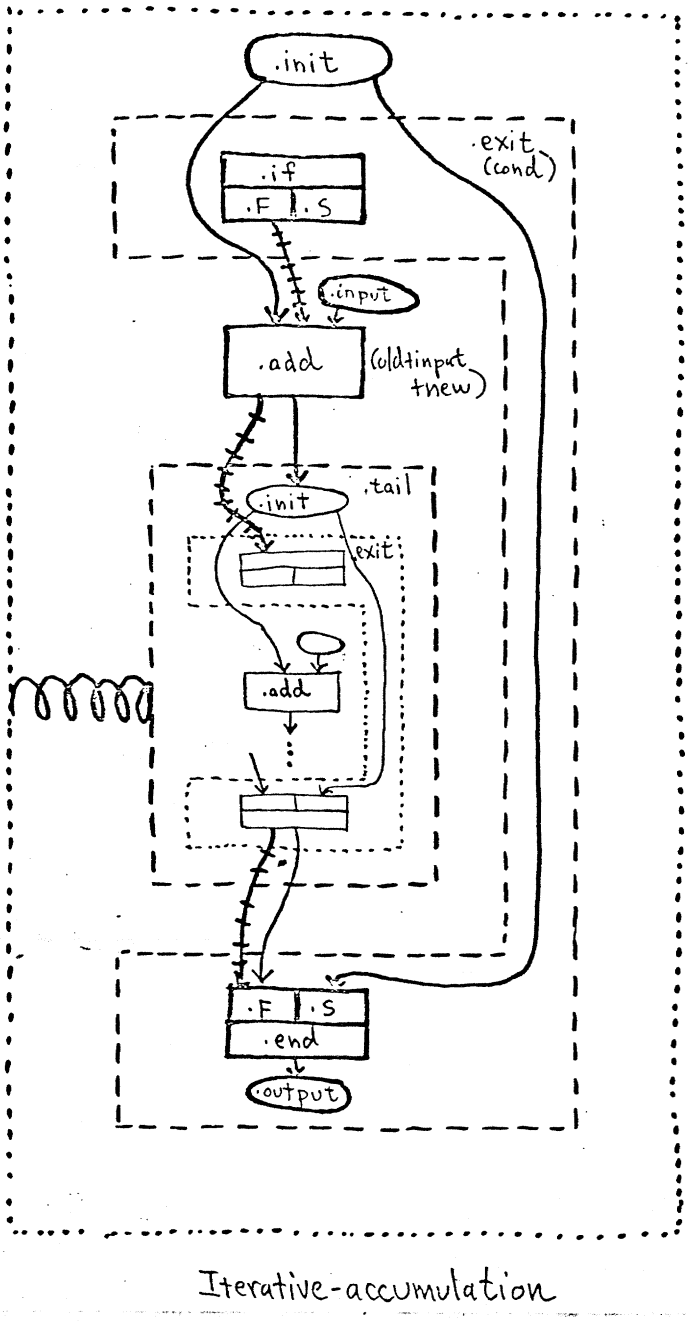


Figure 9-7. Iterative Accumulation Plan.

Plus and Times can also be viewed as instances of Old+input+new.[1] With appropriate Init's, 0 and 1, these accumulation loops compute the summation and product of the Input's. These ideas about accumulation will be formalized further later in this chapter.

The minimal plan for two exits from a loop is Cascade-iterative-termination, shown in Table VII and Fig. 9-8. For example, the loop in the LOOKUP routine of Chapter Five had two exits as shown below.

```
(PROG (...)
      ...
   LP (COND ( ... (RETURN ...)))
      ...
      (COND (... (RETURN ...)))
      ...
      (GO LP))
```

The plan for this code has two tests, If-one and If-two, and two corresponding joins, End-one and End-two. Each time around the loop, If-one is performed first; if it fails, then If-two is performed. If the second test fails, the loop is repeated. If either test succeeds, the loop is terminated by control flow to the corresponding join which bypasses the recursive invocation (Tail). The final constraint on Cascade-iterative-termination says that if the input situation of the first test occurs, then

---

Table VII. Two Exit Loops.

TemporalPlan **cascade-iterative-termination** *extension* single-recursion
roles .if-one(test) .if-two(test) .end-one(join) .end-two(join)
.tail(cascade-iterative-termination)
constraints cflow(.if-one.fail,.if-two.in) ∧ cflow(.if-one.succeed,.end-one.succeed)
∧ cflow(.if-two.fail,.tail.if-one.in) ∧ cflow(.if-two.succeed,.end-two.succeed)
∧ cflow(.tail.end-one.out,.end-one.fail) ∧ cflow(.tail.end-two.out,.end-two.fail)
∧ (.tail = nil ⇋ (.if-one.succeed ≠ ⊥ ∨ .if-two.succeed ≠ ⊥))
∧ (.if-one.in ≠ ⊥ ⇋ (.end-one.out ≠ ⊥ ∨ .end-two.out ≠ ⊥))

TemporalOverlay **cascade>iterative-termination:**
cascade-iterative-termination → iterative-termination
correspondences
cascade-iterative-termination.if-one = iterative-termination.exit.if
∧ cascade-iterative-termination.end-one = iterative-termination.exit.end
∧ cascade>iterative-termination(cascade-iterative-termination.tail) =
iterative-termination.tail

---

1. See overlay @Binfunction>old+input+new in the appendix.

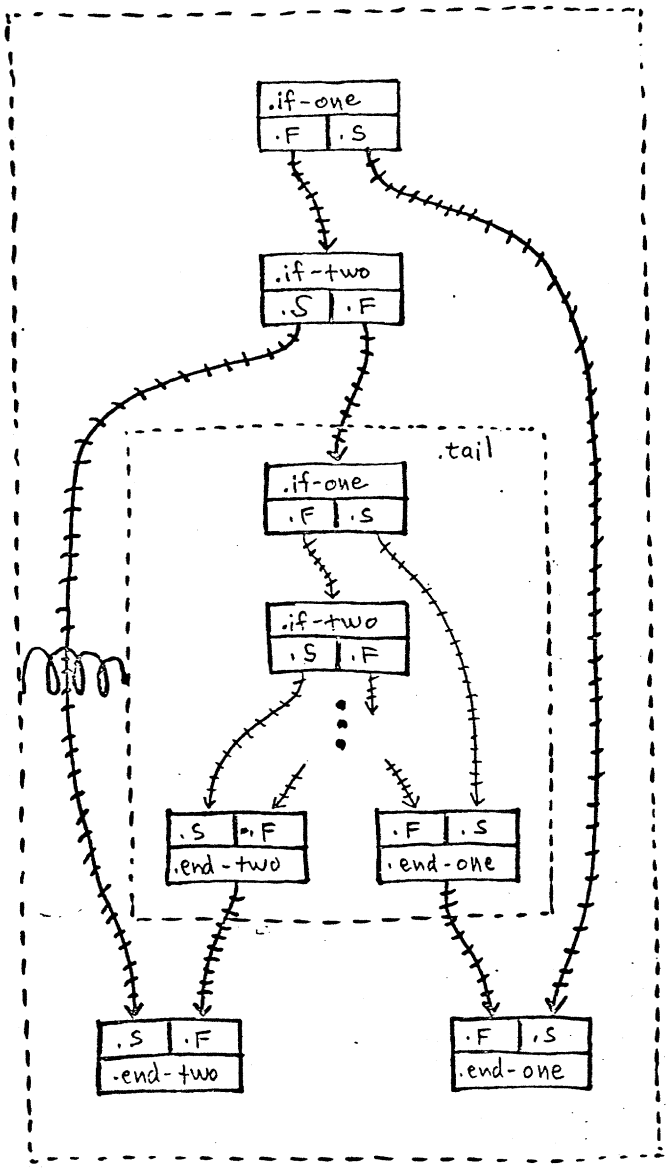


Figure 9-8. Two Exit Loop Plan.

one of the output situations of the joins will occur. This means that the loop is expected to eventually terminate on one of its exits.

Steady state analysis of two exit loops is achieved in two steps. First a two exit loop is viewed as a single exit loop by assuming the second exit is never taken (see overlay Cascade>iterative-termination in Table VII). The first exit can then be ignored using the Iterative-termination>steady-state overlay, as for any other single exit loop.

These few basic patterns of iterative computation, termination, application, generation, accumulation and filtering appear over and over again in routine programming. In fact, many recursive programs are built out of nothing but these plans. Waters [60] did an analysis of 44 programs chosen at random from the 220 programs which comprise the IBM Fortran Scientific Subroutine Package. All of the 164 loops contained in these programs could be analyzed solely in terms of these basic patterns.[1] Furthermore, most of these were instances of a small number of common specializations of the basic plans. Out of a total of 370 instances of generation and accumulation, 82 percent were either summing, product aggregation, maximum, minimum, or counting.[2] Out of a total of 186 loop exit tests, 89 percent were simple comparisons with a fixed number.[3]

Given that we have identified instances of these standard recursive plans in a program, the question remains of how to represent the connection between, say, an generation and an application. Temporally, the components of each are interleaved, but it seems more logical to view the generation and application as being composed in some way. The next section shows how to formalize this viewpoint.

---

1. Several loops had more than two exits, but this is a straightforward generalization of the one and two exit plans presented here.
2. Waters' analysis does not distinguish between generation and accumulation. They are both categorized as augmentations (application of a function or binary function) with feedback.
3. The input being tested in most of these exit tests was a simple sequence of numbers, most often just counting up from one. Thus for most of these loops termination is obvious. This is typical of routine programming -- in most cases the question of termination is settled by recognition of well-known patterns.

9.3 Temporal Abstraction

The basic idea of temporal abstraction is to view all the objects which fill a given role in a recursive temporal plan as a single data structure.[1] In terms of Lisp code, this often corresponds to having an explicit representation for the history of values taken on by a particular variable at a particular point in a loop or other recursive program. For example, in the example program for searching a list introduced earlier, we would like to talk about the values of L at the underlined point each time around the loop.

```
(DEFINE SEARCHLIST
   (LAMBDA (L P)
      (PROG (ENTRY)
         LP (SETQ ENTRY (CAR L))
            (COND ((FUNCALL P ENTRY)(RETURN ENTRY)))
            (SETQ L (CDR L))
            (GO LP))))
```

In general, temporal abstraction gives rise to tree structures. In this section, however, we will discuss only loops, which give rise to linear structures. Temporal abstraction is formalized using overlays. The left side of such an overlay is a recursive temporal plan; the right side is a recursive data plan of the same order (i.e. they are both singly recursive, or doubly, etc.). The definition of the overlay establishes a correspondence between roles in the recursive temporal plan and roles in the data plan, such that the time behavior of a computation which is an instance of the plan on the left is abstracted as a data structure which is an instance of the plan on the right.

Stream Overlays

In the case of loops, temporal abstraction amounts to thinking of a program in terms of *streams*. Streams at particular points in a loop are chosen for temporal abstraction based the analysis of the loop according the taxonomy of iterative temporal plans (generation, application, filtering, etc.) introduced earlier. For example, the temporal abstraction of the underlined values of L in the SEARCHLIST program above is the stream of objects generated by iterative CDR generation. The overlay below and in Fig. 9-9 shows how to express this abstraction formally.

TemporalOverlay **generation-stream:** iterative-generation → list
properties $\forall IS$ [generation-stream(I) = $S \supset$
(instance(thread,generator>digraph(temporal-iterator(I))
$\leftrightharpoons$ list>thread(S) = generator>digraph(temporal-iterator(I))))]
correspondences iterative-generation.action.input = list.head
$\wedge$ generation-stream(iterative-generation.tail) = list.tail

---

1. Both Shrobe [56] and Waters [60] use the idea of temporal abstraction, but with slightly different formalizations than presented here.

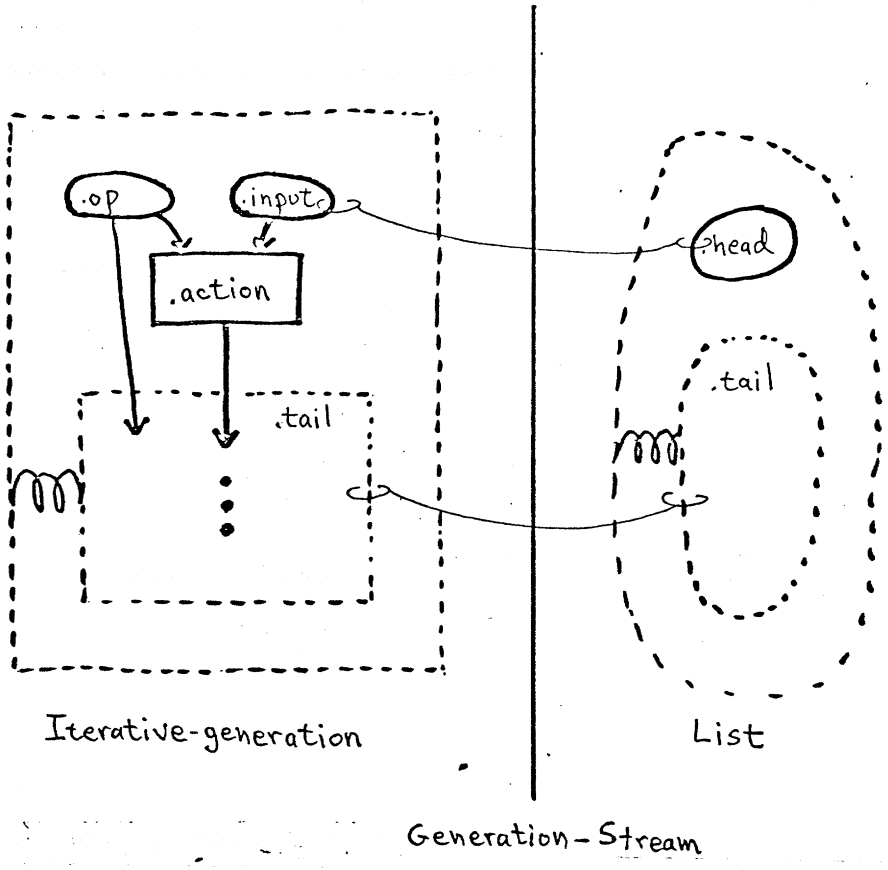


Figure 9-9. Stream Abstraction of Iterative Generation.

The head of the list corresponds to the input of the action of the iterative generation; the tail of the list is recursively defined as the temporal abstraction of the tail of the generation.

The property of Generation-stream stated above ties together two different viewpoints on generation loops that have been introduced in this chapter. In particular, if a generation loop, viewed as an iterator, generates a thread, then (and only then) is the temporally abstracted list of inputs to the action of that loop an irredundant list (which viewed as a thread is the same thread as generated by the iterator).

We next discuss how to abstract the temporal behavior of Iterative-application. For example, we would to express the relationship in the code below between the values of L and the values of ENTRY at the underlined points each time around the loop.

```
(DEFINE SEARCHLIST
   (LAMBDA (L P)
      (PROG (ENTRY)
         LP (SETQ ENTRY (CAR L))
            (COND ((FUNCALL P ENTRY)(RETURN ENTRY)))
            (SETQ L (CDR L))
            (GO LP))))
```

This is achieved by defining two overlays, shown in Table VIII and Fig. 9-11, which temporally abstract the input and output roles of the Action of the iterative application. The relationship between these two streams is then most conveniently expressed by viewing them as sequences, as explained in the next section.

Table VIII. Application Stream Overlays.

TemporalOverlay **application-in-stream:** iterative-application → list
correspondences iterative-application.action.input = list.head
∧ application-in-stream(iterative-application.tail) = list.tail

TemporalOverlay **application-out-stream:** iterative-application → list
correspondences iterative-application.action.output = list.head
∧ application-out-stream(iterative-application.tail) = list.tail

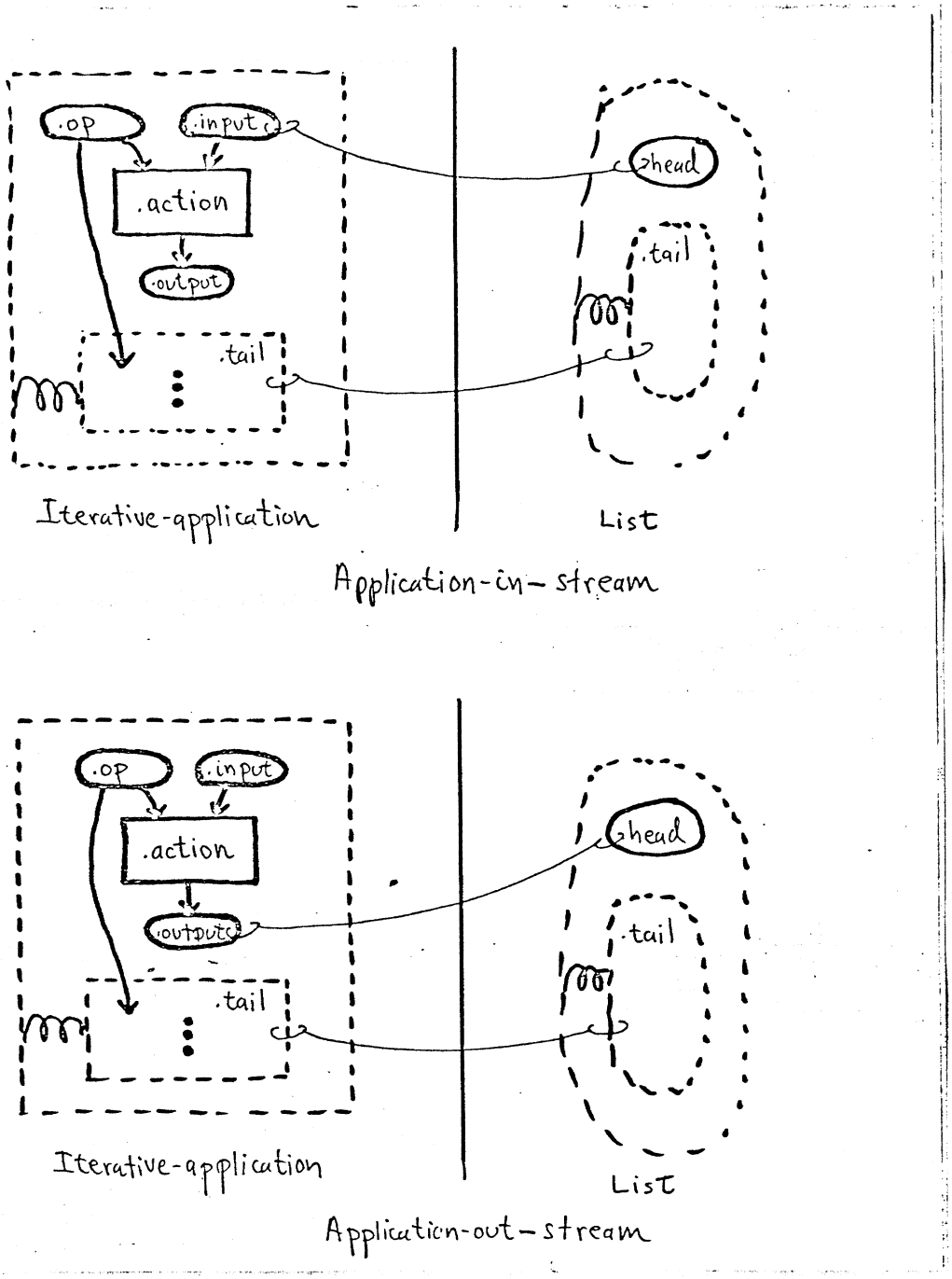


Figure 9-10. Stream Abstractions of Iterative Application.

This page intentionally left blank.

See Fig. 9-10.

Figure 9-11. Stream Abstractions of Iterative Application.

Temporal Sequences

Streams viewed as sequences are called temporal sequences. Making this abstraction step allows us to use the input-output specifications on sequences to describe the relationship between changing values in loops. In particular, iterative application can be thought of as implementing a Map operation from the stream of inputs of the Action role (viewed as a sequence) to the stream of outputs. This is expressed as the Temporal-map overlay shown in Table IX and in Fig. 9-12.

On the right side of this overlay is the Map plan. The inputs to the Action of the iterative application (e.g. the values of L above) are abstracted as the input to Map. The outputs of the Action (e.g. the values of ENTRY above) are modelled as the output of Map. The Op of Map corresponds to the Op of the Action. What we are doing in this overlay is modelling the time behavior of the recursive temporal plan on the left as a single step in some other time domain represented on the right.

Iterative generation can be similarly abstracted as an Iterate operation, as shown in Table IX. The input to the operation is an iterator whose Op is the Op of the Action of the generation and whose Seed is the initial Input. The output sequence is the generated stream, as defined in the preceding section, viewed as a sequence.

Table IX. Temporal Sequence Overlays.

TemporalOverlay **temporal-map:** iterative-application → map
correspondences
iterative-application.action.op = map.op
∧ list>sequence(application-in-stream(iterative-application)) = map.input
∧ list>sequence(application-out-stream(iterative-application)) = map.output

TemporalOverlay **temporal-iterate:** iterative-generation → iterate
correspondences
temporal-iterator(iterative-generation) = iterate.input
∧ list>sequence(generation-stream(iterative-generation)) = iterate.output
∧ iterative-generation.action.in = iterate.in

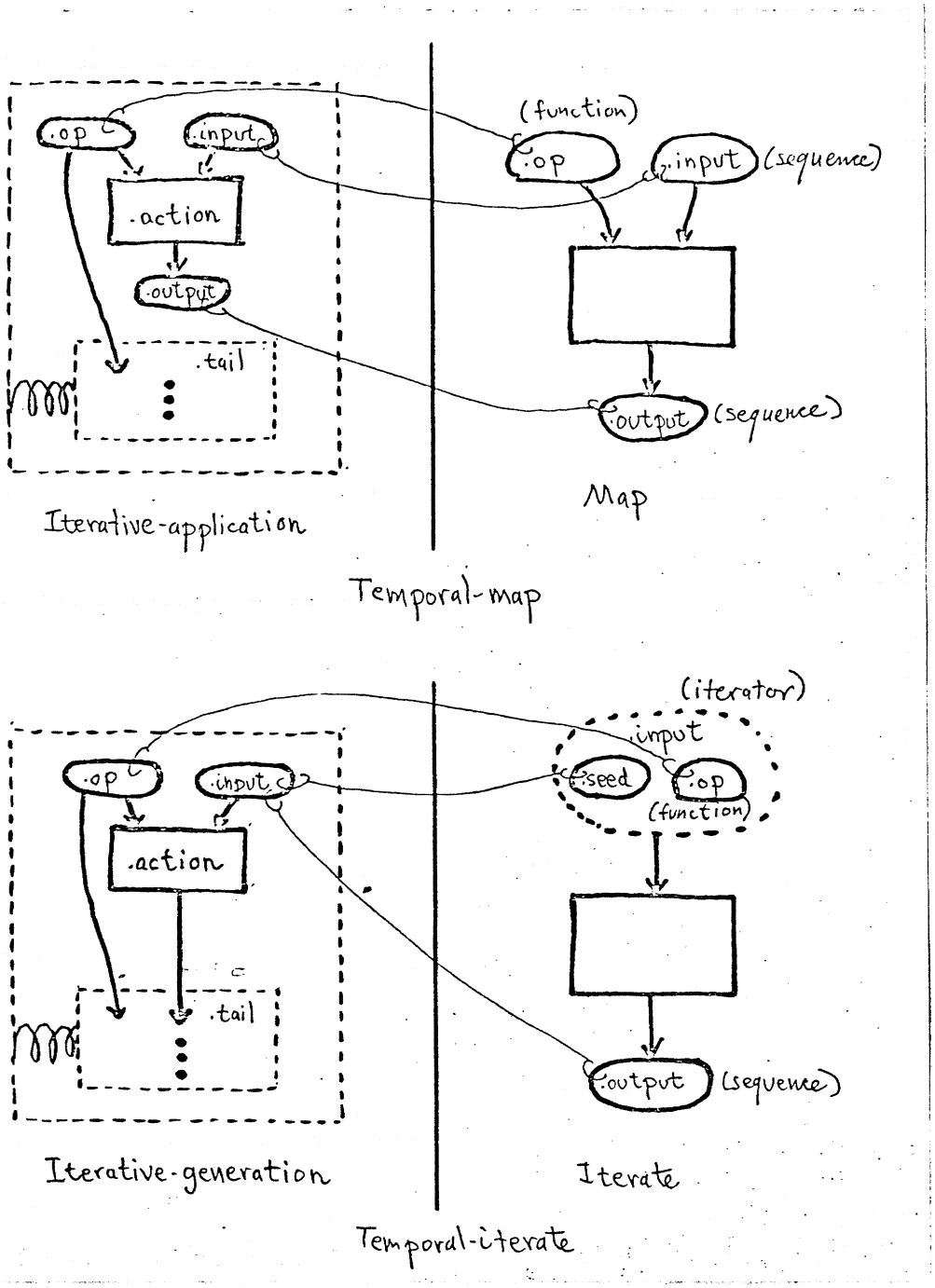


Figure 9-12. Temporal Overlays for Application and Generation.

Temporal Data Flow

In this section we further develop the notion of stream overlays in order to specify the connection between the temporal abstractions of different parts of a loop. For example, in the SEARCHLIST example, shown again below, the stream generated by the iterative CDR generation is the same as the input stream to the iterative CAR application.

```
(DEFINE SEARCHLIST
   (LAMBDA (L P)
      (PROG (ENTRY)
         LP (SETQ ENTRY (CAR L))
            (COND ((FUNCALL P ENTRY)(RETURN ENTRY)))
            (SETQ L (CDR L))
            (GO LP))))
```

This in turn means that data flow between operations in the recursive view implies data flow between operations in the temporally abstracted view.

For example, Fig. 9-13 shows the temporal sequence analysis of SEARCHLIST. On the left is the unanalyzed recursive computation. As has been pointed out before, this diagram contains an instance of Iterative-generation (the Action role corresponds to role Two of the surface plan), of Iterative-application (the Action role coresponds to role One of the surface plan), and of Iterative-search (the Exit.If role corresponds to the If role of the surface plan). The right side of the figure shows the plan after recognizing these iterative patterns and applying the Temporal-iterate, Temporal-map, and Temporal-earliest overlays.[1] Temporal sequence abstractions are labelled. The objects generated by Cdr are temporally abstracted as the output sequence of One (Iterate) on the right. Furthermore, since the input to the Action of the generation on each repetition is the same as the input to the Car application, this sequence is therefore also the temporal abstraction of the inputs to the iterative application. Similarly, data flow between the output of the action of iterative application and the input to the exit test of iterative search on each repetition in the surface plan implies data flow from the output of the Two (Map) to the input of Three (Earliest) in the temporal view.

Thus we see that temporal analysis leads to a viewpoint on loops in which there are producers, transducers and consumers of streams. An overlay like Temporal-iterate models a part of a loop which produces a stream; Temporal-map models a part of a loop which consumes one stream and produces another (i.e. a transducer); and Temporal-earliest is a stream consumer.

---

1. The overlay between Iterative-search and Earliest will be defined later in this section. To simplify the presentation, the figure omits several intermediate analysis steps.

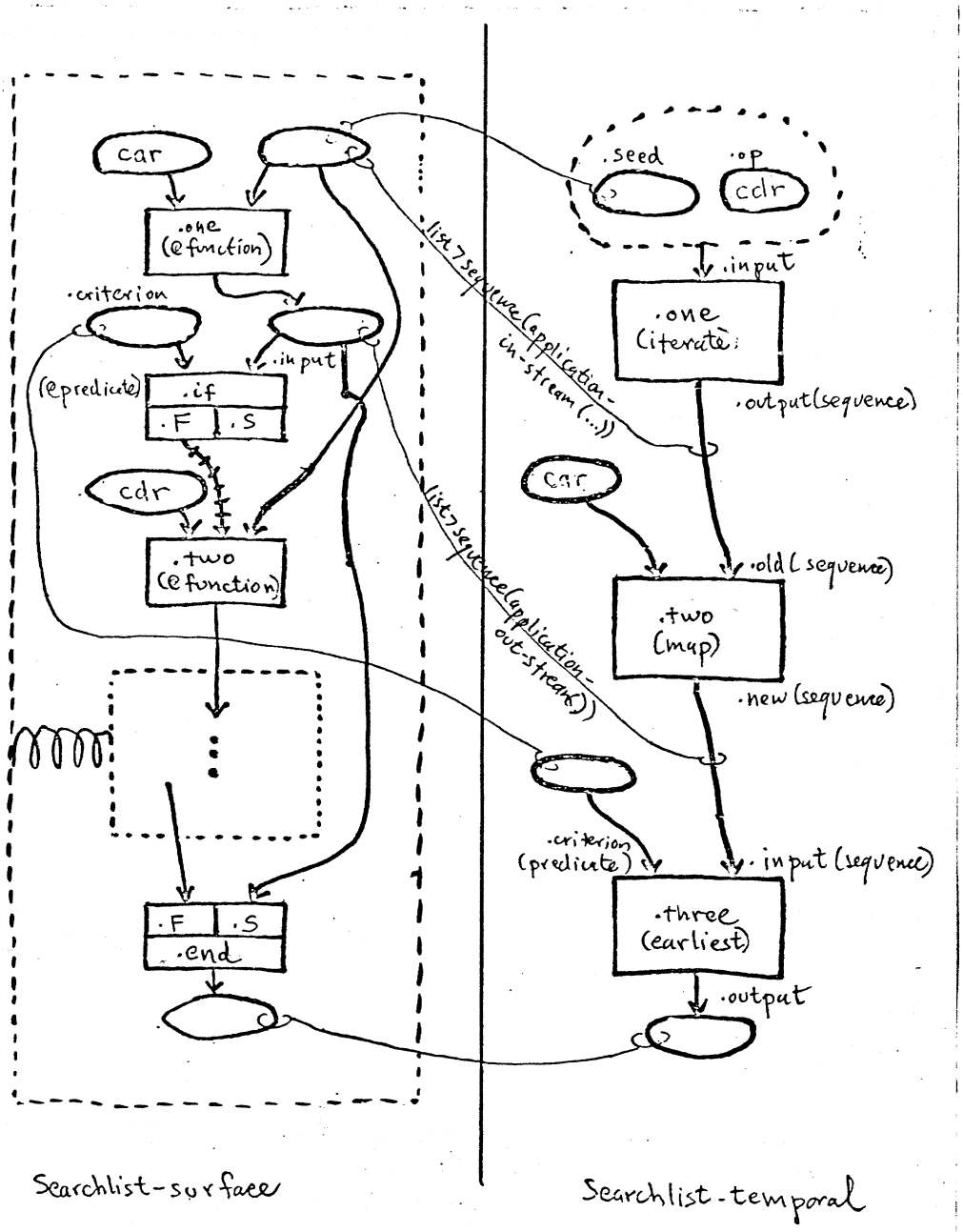


Figure 9-13. Temporal Analysis of SEARCHLIST.

Table X. List Generation.

TemporalPlan **list-generation**
properties $\forall P$ [instance(list-generation,P)
$\supset$ list>sequence(generation>list(P)) = P.two.output]
roles .one(iterate) .two(map)
constraints .one.output = .two.input $\wedge$ cflow(.one.out,.two.in)

TemporalOverlay **generation>list :** list-generation $\rightarrow$ list
definition L = list-generation>list(P) $\equiv$
[$\exists T$ (instance(labelled-thread,T)
$\wedge$ T.spine = generator>digraph(P.one.input)
$\wedge$ T.label = P.two.op)]

TemporalPlan **car+cdr** *specialization* list-generation
properties $\forall P$ [instance(car+cdr,P)
$\supset$ dotted-pair>list(P.one.input.seed) = generation>list(P)]
roles .one(iterate) .two(map)
constraints instance(cdr-iterator,.one.input) $\wedge$ .two.op = car

---

The pattern of Iterate and Map (particularly implemented temporally as iterative generation composed with application) is a common one. This plan is called List-generation, as shown in Table X, because the output sequence of the Map operation (role Two), viewed as a list, is the same as the labelled thread whose spine is generated by the input to the Iterate operation, and whose label is the function applied in the Map operation. This relationship is expressed by the overlay Generation>list, also shown in Table X.

Car+cdr is the common special case of in Lisp list generation, wherein the input to Iterate is an instance of Cdr-iterator (an iterator in which the Op is Cdr) and the function applied by Map is Car. It thus follows that the list generated by Car+cdr, according to the overlay Generation>list, is the same as the list implemented by the dotted pair which is the seed of the iterator input to Iterate, according to the overlay Dotted-pair>list.

Termination

We move on now to the temporal abstraction of termination plans, in particular the overlays in Table XI and Fig. 9-14, which express how to view the inputs to the exit test of Iterative-termination-predicate as a finite list.

The basic form of the Termination-in-stream overlay is the same as Application-in-stream, i.e. the head of the list is the input on the first repetition and the tail of the list is defined recursively. In this overlay however, the recursive definition is split into two cases: if the exit test succeeds, then

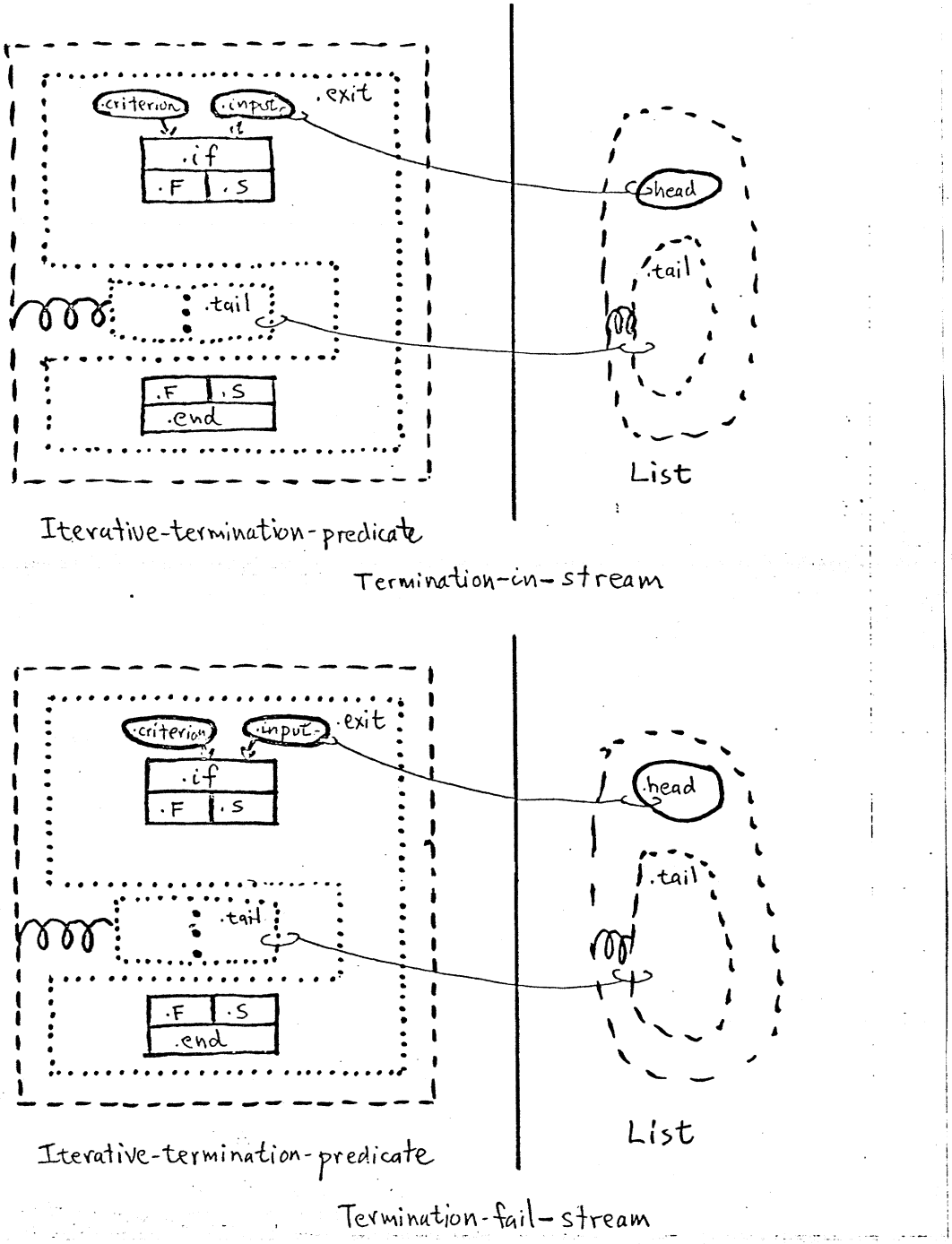


Figure 9-14. Stream Abstractions of Iterative Termination.

Table XI. Termination Stream Overlays.

TemporalOverlay **termination-in-stream:** iterative-termination-predicate → finite-list
correspondences
iterative-termination-predicate.exit.if.input = finite-list.head
∧ (*if* iterative-termination-predicate.tail = nil *then* nil
else termination-in-stream(iterative-termination-predicate.tail))
= finite-list.tail

TemporalOverlay **termination-fail-stream:** iterative-termination-predicate → finite-list+nil
definition S = termination-fail-stream(T) ≡ [list>sequence(S) = butlast(termination-in-stream(T))]

TemporalOverlay **steady-state-stream:** iterative-termination-predicate → list
correspondences
iterative-termination-predicate.exit.if.input = list.head
∧ steady-state-stream(iterative-termination-predicate.tail) = list.tail

---

the tail of the list is Nil; if it fails, then the tail of the list is the temporal abstraction of the tail of the termination plan. This means that the temporal abstraction of the inputs to a termination which exits on the first step is a singleton list (one with a Nil tail), and that for all uses of Termination-in-stream, the last object in the list (i.e. the head of the sublist with the Nil tail) satisfies the exit predicate.

Termination-fail-stream is the overlay which abstracts the inputs to the exit test of a loop as seen in an environment where the test is known to have failed. For example, the difference between Termination-in-stream and Termination-fail-stream is the difference between the stream of values of L seen at first underlined point below versus the second.

```
(PROG (L)
      ...
   LP ...L...
      (COND ((P L)(RETURN ...)))
      ...L...
      (GO LP))
```

The Termination-fail stream is defined in terms of the Termination-in stream, i.e. it is the same, viewed as a temporal sequence, as the Termination-in stream viewed as a sequence, except for being one term shorter (Butlast)

Like Iterative-application, Iterative-termination describes a fragment of a loop which has some of its inputs provided by other parts of the loop. The relationship between a termination test and the other parts of the loop which provide its inputs is most conveniently expressed in terms of the relationship between the Termination-in or Termination-fail streams and the stream of inputs to the test in the steady state analysis. The stream of inputs to the test in the steady state analysis is specified by the overlay Steady-state-stream in Table XI. It has a recursive definition similar to the other

stream overlays for iterative termination. In this case, however, we are talking about the stream of inputs which would be seen in the input situation of the abstraction of the exit test as a non-terminating loop.

The relationship between the steady state stream and the Termination-in and Termination-fail streams is most conveniently expressed in terms of Truncate and Truncate-inclusive operations on temporal sequences. For example, in the following loop the temporal sequence of values of N at the indicated point under the steady state assumption is 1,2,3,..., as generated by Naturals-iterator. The effect of adding the termination test is to truncate this sequence at 10 (inclusively at the point indicated).[1]

```
(PROG (N)
      (SETQ N 1)
   LP ...N...
      (COND ((= N 10)(RETURN ...)))
      (SETQ N (1+ N))
      (GO LP))
```

The overlays in Table XII formalize this analysis. Let us first consider Temporal-truncate-inclusive with reference to Fig. 9-15, which shows its application to the example program above. On the left is the unanalyzed symbolic computation. In the center is the steady state analysis in which an instance of Iterative-generation has been recognized. The temporal sequence generated by this

---

Table XII. Temporal Truncation.

TemporalOverlay **temporal-truncate-inclusive:** iterative-termination-predicate → truncate-inclusive
correspondences
iterative-termination-predicate.exit.if.criterion = truncate-inclusive.criterion
∧ list>sequence(steady-state-stream(iterative-termination-predicate)) = truncate.input
∧ list>sequence(termination-in-stream(iterative-termination-predicate))
= truncate-inclusive.output
∧ iterative-termination-predicate.exit.end.out = truncate-inclusive.out

TemporalOverlay **temporal-truncate:** iterative-termination-predicate → truncate
correspondences
iterative-termination-predicate.exit.if.criterion = truncate.criterion
∧ list>sequence(steady-state-stream(iterative-termination-predicate))
= truncate.input
∧ list>sequence(termination-fail-stream(iterative-termination-predicate))
= truncate.output
∧ iterative-termination-predicate.exit.end.out = truncate.out

---

1. In a later section, an overlay will be introduced which captures the relationship between using N at the point indicated and a loop that checks for (= N 11) and uses N after the test.

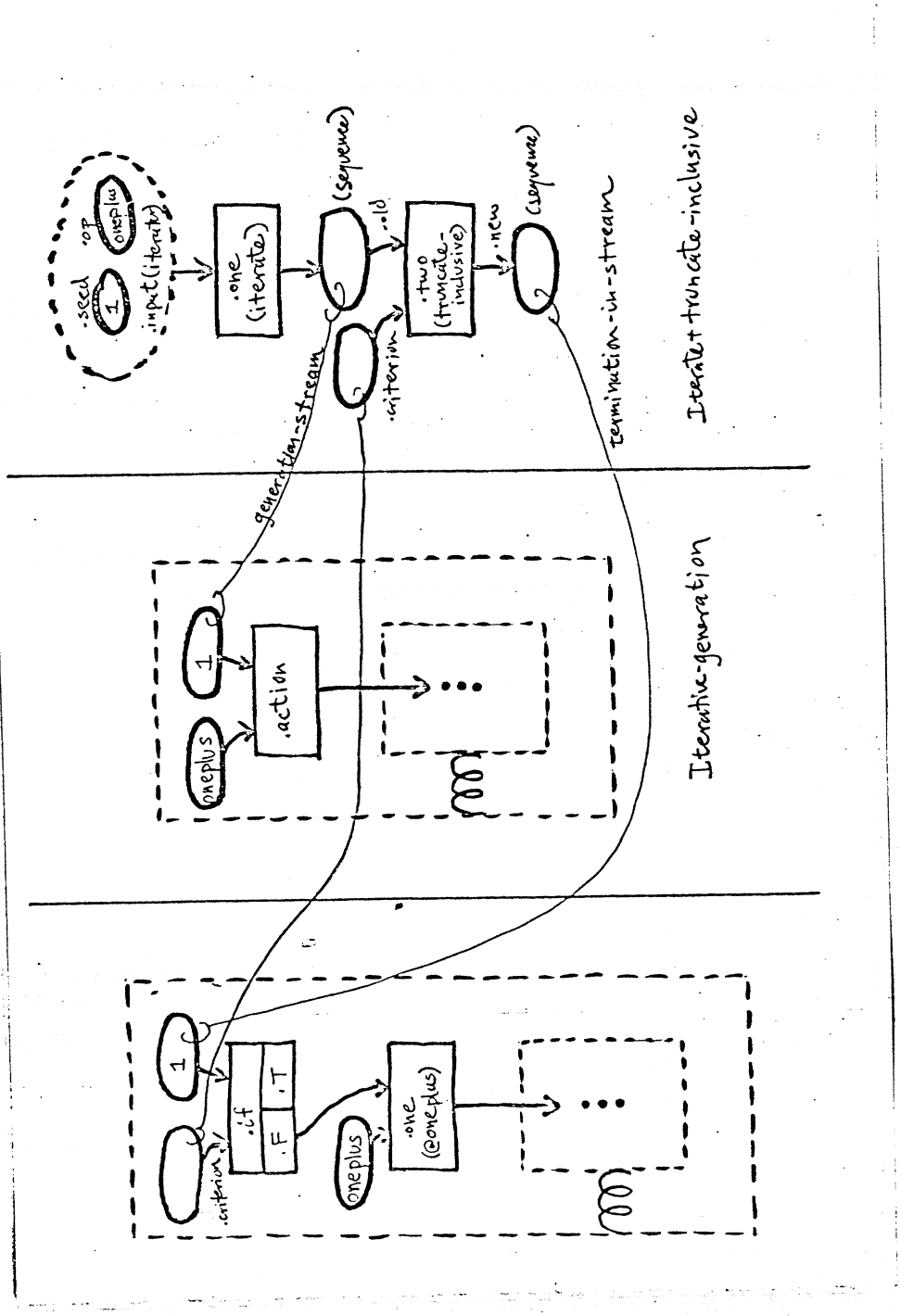


Figure 9-15. Symbolic Computations at Various Levels of Abstraction.

generation is abstracted on the right as the output of an Iterate operation. This sequence is then the input to a Truncate-inclusive operation whose output is the sequence of values of N actually seen temporally at the point indicated. The termination constraint on the Iterative-termination plan and its specializations is consistent with the precondition on Truncate that there exist a term of the sequence which satisfies the given criterion. The Temporal-truncate overlay is similar to Temporal-truncate-inclusive, except that the input sequence is the Termination-fail sequence, rather than the Termination-in sequence.

The pattern of Iterate and Truncate shown in Fig. 9-15, particularly implemented temporally as a loop in which the termination directly tests the current value of an iterative generation, is a common one. The Iterate+truncate and Iterate+truncate-inclusive plans shown in Table XIII express this in a data flow constraint between the output sequence of the Iterate operation (role One) and the input sequence of the Truncate(-inclusive) operation (role Two).

Table XIII. Iterate and Truncate.

TemporalPlan **iterate+truncate**
roles .one(iterate) .two(truncate)
constraints instance(irredundant-sequence,.one.output)
∧ .one.output = .two.input ∧ cflow(.one.out,.two.in)

TemporalOverlay **iterate+truncate>truncated-thread:** iterate+truncate → truncated-thread
properties
∀*I* truncated>digraph(iterate+truncate>truncated-thread(*I*),sequence>thread(*I*.two.output))
correspondences
sequence>thread(iterate+truncate.one.output) = truncated-thread.base
iterate+truncate.two.criterion = truncated-thread.criterion

TemporalPlan **iterate+truncate-inclusive**
roles .one(iterate) .two(truncate-inclusive)
constraints instance(irredundant-sequence,.one.output)
∧ .one.output = .two.input ∧ cflow(.one.out,.two.in)

TemporalOverlay **iterate+truncate-inclusive>truncated-thread:**
iterate+truncate-inclusive → truncated-thread
properties
∀*I* truncated>digraph-inclusive(iterate+truncate-inclusive>truncated-thread(*I*),
sequence>thread(*I*.two.output))
correspondences
sequence>thread(iterate+truncate-inclusive.one.output) = truncated-thead.base
iterate+truncate-inclusive.two.criterion = truncated-thread.criterion

Iterate+truncate(-inclusive) can be further abstracted as a truncated thread, as described by the overlays in Table XIII. The base of the truncated thread is the irredundant sequence output of the Iterate operation, viewed as a thread; the criterion is the criterion of the Truncate(-inclusive) operation. Furthermore, properties of these overlays connect this sequence view with the directed graph view described earlier in this chapter. In particular, the finite graph implemented (inclusively) by the truncated thread thus abstracted is the same as the output of the Truncate(-inclusive) operation, viewed as a thread.

Another temporal abstraction involving termination is to view an iterative search loop as the implementation of an Earliest operation, as shown in Table XIV and Fig. 9-16. In this overlay, the input sequence to the Earliest operation is the steady state stream of inputs to the test of the iterative search, viewed as a sequence. The output of the ending join of the search plan is the output of the Earliest operation. The termination constraint of Iterative-search is consistent with the precondition on Earliest which states that there exists a term of the input sequence which satisfies the given criterion.

Cotermination loops are temporally abstracted using overlays similar to those already presented for termination loops. The overlay Cotermination-in-stream, shown in Table XV abstracts the stream of co-iterands seen before the exit test, similar to Termination-in-stream. Cotermination-fail-stream abstracts the stream of co-iterands seen in an environment where the test is known to have failed, similar to Termination-fail-stream. Finally, the stream of co-iterands in the steady state is abstracted by the overlay Steady-state-costream.

Given these overlays, Iterative-cosearch, as in the LENGTH program below, can be modelled as the temporal implementation of a Co-earliest operation, as shown in Table XVI. The specifications of Co-earliest are similar to those of Earliest. Co-earliest takes as input two sequences (Input and Co-input), and returns as output the term of the Co-input sequence which corresponds to the term of the Input sequence which would be returned by Earliest.

Table XIV. Temporal Earliest.

TemporalOverlay **temporal-earliest:** iterative-search → earliest
correspondences
iterative-search.exit.if.criterion = earliest.criterion
∧ list>sequence(steady-state-stream(iterative-search)) = earliest.input
∧ iterative-search.exit.end.output = earliest.output
∧ iterative-search.exit.end.out = earliest.out

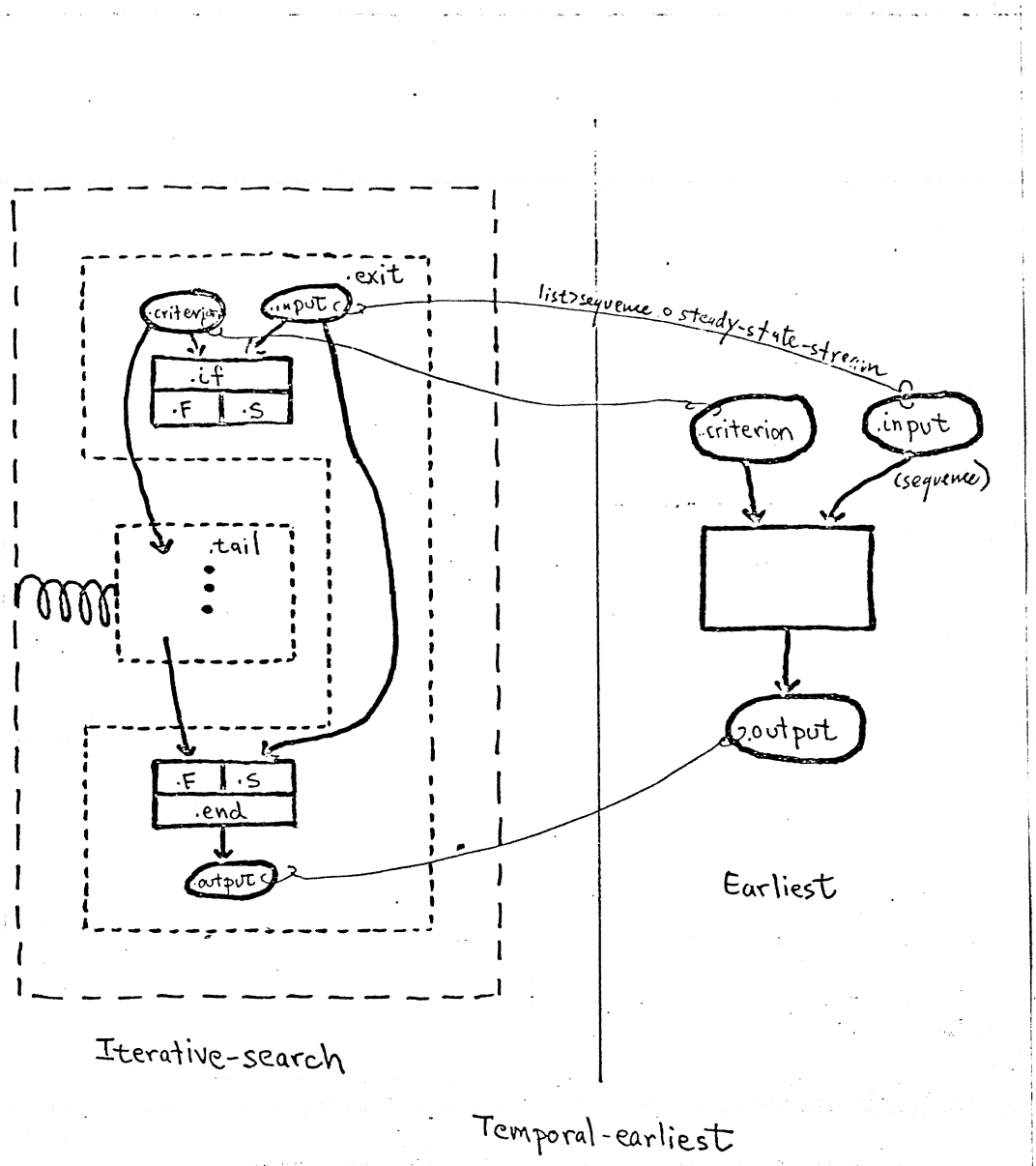


Figure 9-16. Temporal Abstraction of Search Loop.

Table XV. Cotermination Stream Overlays.

TemporalOverlay **cotermination-in-stream:** iterative-cotermination → finite-list
correspondences
iterative-cotermination.co-iterand = finite-list.head
∧ (*if* iterative-cotermination.tail = nil *then* nil
else cotermination-in-stream(iterative-cotermination.tail))
= finite-list.tail

TemporalOverlay **cotermination-fail-stream:** iterative-cotermination → finite-list+nil
definition S = cotermination-fail-stream(T) ≡
[list>sequence(S) = butlast(cotermination-in-stream(T))]

TemporalOverlay **steady-state-costream:** iterative-cotermination → list
correspondences
iterative-cotermination.co-iterand = list.head
∧ steady-state-costream(iterative-cotermination) = list.tail

Table XVI. Temporal Co-earliest.

IOspec **co-earliest** / .input(sequence) .criterion(predicate) .co-input(sequence)
⇒ .output(object)
preconditions le(length(.input),length(.co-input))
∧ ∃i apply(.criterion,apply(.input,i)) = true
postconditions ∃i (apply(.criterion,apply(.input,i)) = true
∧ apply(.co-input,i) = .output
∧ ∀j (lt(j,i) ⊃ apply(.criterion,apply(.input,j)) = false))

TemporalOverlay **temporal-co-earliest:** iterative-cosearch → co-earliest
correspondences
iterative-cosearch.exit.if.criterion = co-earliest.criterion
∧ list>sequence(steady-state-stream(iterative-cosearch)) = co-earliest.input
∧ list>sequence(steady-state-costream(iterative-cosearch)) = co-earliest.co-input
∧ iterative-cosearch.co-iterand = co-earliest.output
∧ iterative-cosearch.exit.end.out = co-earliest.out

```
(DEFINE LENGTH
   (LAMBDA (L)
      (PROG (N)
            (SETQ N 0)
         LP (COND ((NULL L)(RETURN N)))
            (SETQ L (CDR L))
            (SETQ N (1+ N))
            (GO LP))))
```

Thus this program can be temporally abstracted (as shown on the left of Fig. 9-17) as two instances of Iterate, one with a Cdr-iterator as input, and one with Natural-iterator (an iterator in which the Seed is 1 and the Op is Oneplus) as input, feeding into an instance of Co-earliest with a criterion of Null. What this figure also shows is how this plan implements @Length, the computation of the length of a sequence. If the Cdr-iterator input on the left hand side is the implementation of the spine of the sequence viewed as a labelled thread, then the output of Co-earliest is the length of the sequence.

The idea of truncating a sequence based on the occurence of a term satisfying a given predicate in another (parallel) sequence is expressed by the Cotruncate specification defined in Table XVII. This specification is similar to Truncate, except that the output sequence is some inital subsequence of a second input sequence, Co-input. Furthermore, it follows from these specifications that if an instance of Truncate and of Cotruncate have the same Input sequence and Criterion, the outputs are the same length. Cotruncate is implemented temporally by Iterative-cotermination, as shown in the overlay in Table XVII, which resembles the Temporal-truncate overlay.

One of the most common cliches in Lisp programming,, i.e. CDRing down a list until NULL, using the CARS, can be analyzed as the composition of an instance of Iterate, Map, and Cotruncate. The plan for this in general is called Truncated-list-generation, as shown in Table XVIII and Fig. 9-

---

Table XVII. Temporal Cotruncate.

IOspec **cotruncate** / .input(sequence) .criterion(predicate) .co-input(sequence)
⇒ .output(finite-sequence)
properties $\forall TC$ (instance(truncate,T) ∧ instance(cotruncate,C)
∧ T.input = C.input ∧ T.criterion = C.criterion)
⊃ length(T.output) = length(C.output)
preconditions le(length(.input),length(.co-input))
∧ $\exists i$ apply(.criterion,apply(.input,i)) = true
postconditions
$\forall i$ (index(.output,i) ⇋ $\forall j$ (le(j,i) ⊃
apply(.criterion,apply(.input,j)) = false))
∧ $\forall i$ (index(.output,i) ⊃ apply(.output,i) = apply(.co-input,i))
∧ apply(.criterion,apply(.input,oneplus(length(.output)))) = true

TemporalOverlay **temporal-cotruncate:** iterative-cotermination → cotruncate
correspondences
iterative-cotermination.exit.if.criterion = cotruncate.criterion
∧ list>sequence(steady-state-stream(iterative-cotermination)) = cotruncate.input
∧ list>sequence(steady-state-costream(iterative-cotermination)) = cotruncate.co-input
∧ list>sequence(cotermination-fail-stream(iterative-cotermination)) = cotruncate.output
∧ iterative-cotermination.exit.end.out = truncate.out

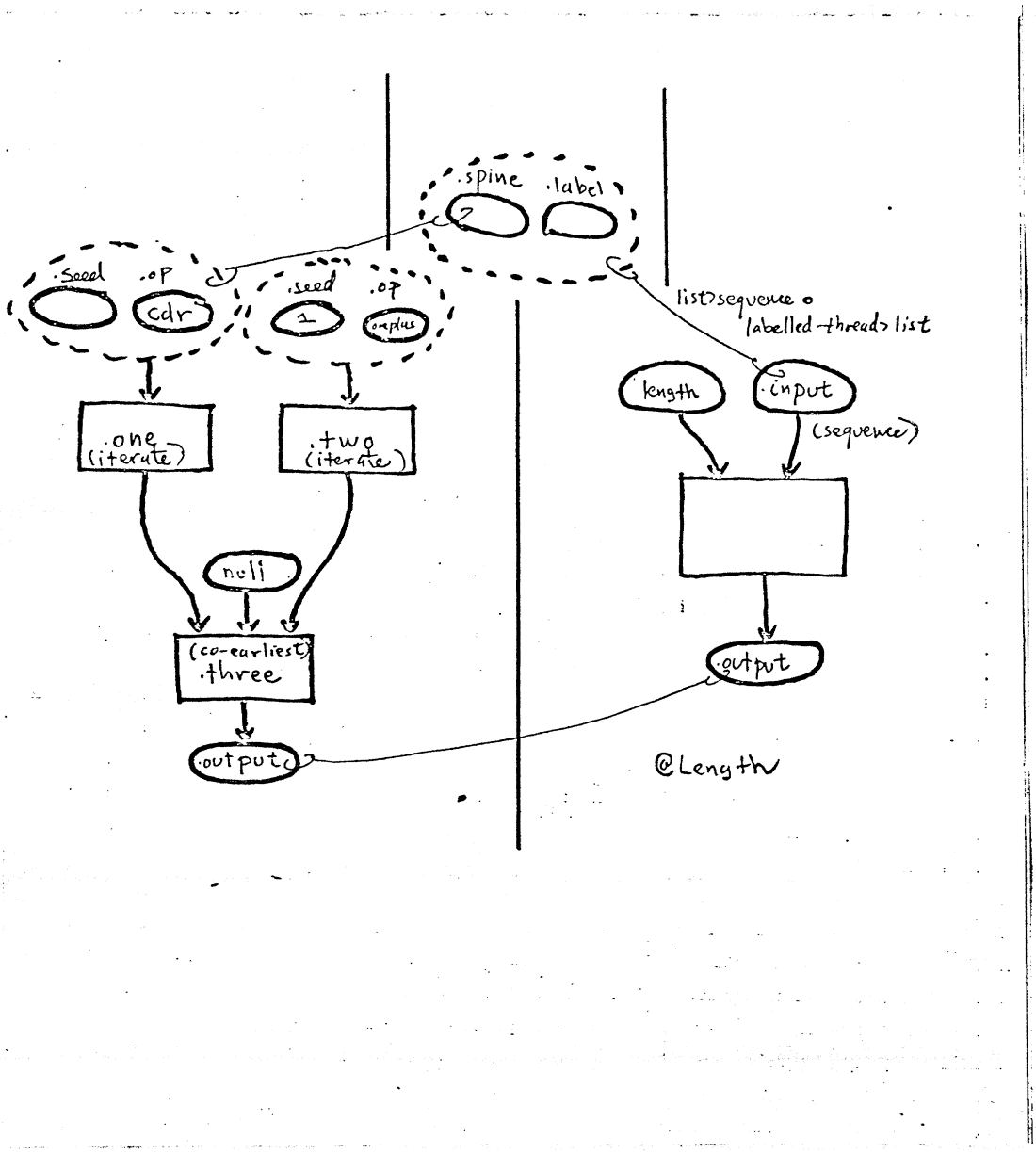


Figure 9-17. Computing the Length of a Sequence.

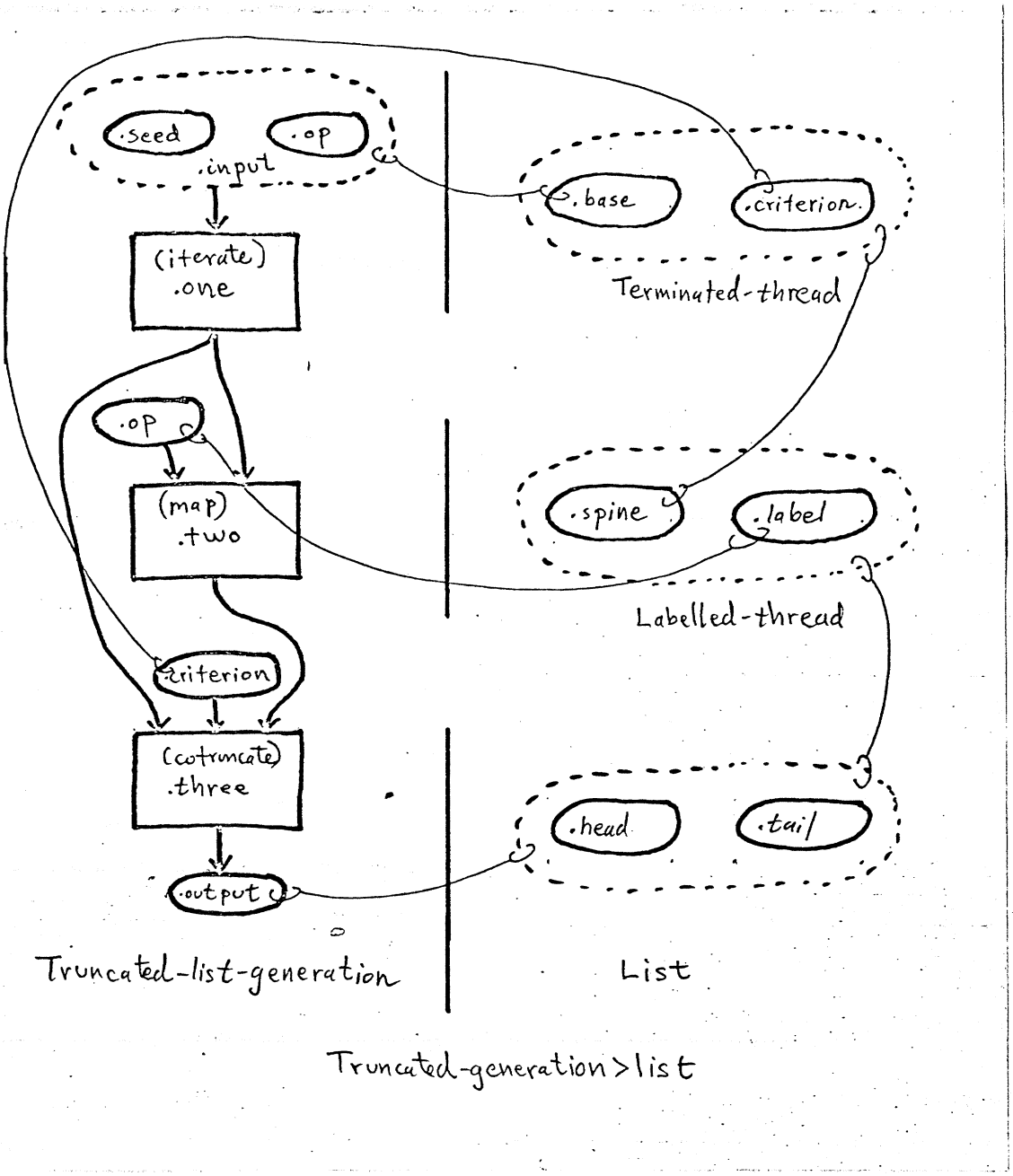


Figure 9-18. Truncated List Generation.

Table XVIII. Truncated List Generation.

TemporalPlan **truncated-list-generation** *extension* list-generation
properties $\forall P$ instance(truncated-list-generation,P)
$\supset$ list>sequence(truncated-generation>list(P),P.three.output)
roles .one(iterate) .two(map) .three(cotruncate)
constraints .one.output = .three.input $\wedge$ .two.output = .three.co-input
$\wedge$ cflow(.two.out,.three.in)

TemporalOverlay **truncated-generation>list :** truncated-list-generation $\rightarrow$ finite-list
definition L = truncated-generation>list(P,L) $\equiv$
$\exists TR$ [instance(labeled-thread,T) $\wedge$ instance(iterator,R)
$\wedge$ T.spine = truncated>digraph(R) $\wedge$ T.label = P.two.op
$\wedge$ R.base = generator>digraph(P.one.input)
$\wedge$ R.criterion = P.three.criterion]

TemporalPlan **car+cdr+null**
specialization truncated-list-generation
extension car+cdr
properties $\forall P$ instance(car+cdr+null,P) $\supset$
dotted-pair>list(P.one.input.seed) = truncated-generation>list(P)
roles .one(iterate) .two(map) .three(cotruncate)
constraints .three.criterion = null

---

18. This plan is an extension of List-generation, discussed earlier in this section. Similar to List-generation, the final output sequence of this plan (in this case the output of role Three), viewed as a list, is the same as the labelled thread whose spine is generated by the input to the Iterate operation and truncated by the criterion of the Cotruncate operation, and whose label is the function applied in the Map operation. This relationship is expressed by the overlay Truncated-generation>list in Table XVIII, and shown in Fig. 9-18.

The specialization of Truncated-list-generation for Lisp lists in particular, where the iterator function is Cdr, the Map function is Car, and the Cotruncate predicate is Null, is called Car+cdr+null.

```
(PROG (L)
      ...
   LP (COND ((NULL L)(RETURN ...)))
      ...(CAR L)...
      (SETQ L (CDR L))
      (GO LP))
```

The output of role Three (an instance of Cotruncate) in the Car+cdr+null plan corresponds to the sequence of values returned by CAR at the underlined point in the code above. This sequence, viewed as a list, is the same as the list implemented by the dotted pair which is the seed of the iterator input to Iterate, according to the overlay Dotted-pair>list.

Temporal Sets

In many programming applications, the order in which data is generated in a loop or recursion doesn't matter. In such cases it is appropriate to take temporal abstraction one step further and just talk about the *set* of objects which fill a given role in a recursive temporal plan.[1] This abstraction step is added to the existing temporal abstraction framework by using the List>set overlay.

For example, a generation loop can be thought of as the temporal implementation of a transitive closure operation, in which the binary relation being closed is a function. The overlay between these two views is shown in Table XIX and in Fig. 9-19. On the left of the overlay is a generation loop; on the right is the application of transitive closure to an iterator. The correspondence between the iterator input and the loop is the one already specified by Temporal-iterator, i.e. the Op of the Action is the Op of the iterator, and initial input to the Action is the Seed. The output set of the transitive closure operation is the stream generated at the input of the Action (as formalized by Generation-stream), viewed as a set.

Similar temporal overlays can be constructed for other input-output specifications with sets, such as Each, Set-find, Restrict, and Any. Iterative temporal overlays for Each and Set-find are shown in Table XX. Iterative-temporal-each is just like Temporal-map, except rather than abstracting the streams of inputs and outputs to an iterative application as sequences, they are abstracted as sets. Iterative-temporal-find is just like Temporal-earliest, except the steady state stream of inputs is also abstracted as a set.

Table XIX. Temporal Transitive Closure.

IOspec **@transitive-closure-function** / .op(function) .input(iterator) ⇒ .output(set)
specialization @function
preconditions .op = transitive-closure

TemporalOverlay **iterative-temporal-transitive-closure:**
iterative-generation → @transitive-closure-function
correspondences
temporal-iterator(iterative-generation) = @transitive-closure-function.input
∧ list>set(generation-stream(iterative-generation))
= @transitive-closure-function.output
∧ iterative-generation.action.in = @transitive-closure-function.in

---

1. More precisely, this abstraction is appropriate when order doesn't matter and either there are no duplicates or the occurence of duplicates doesn't matter.

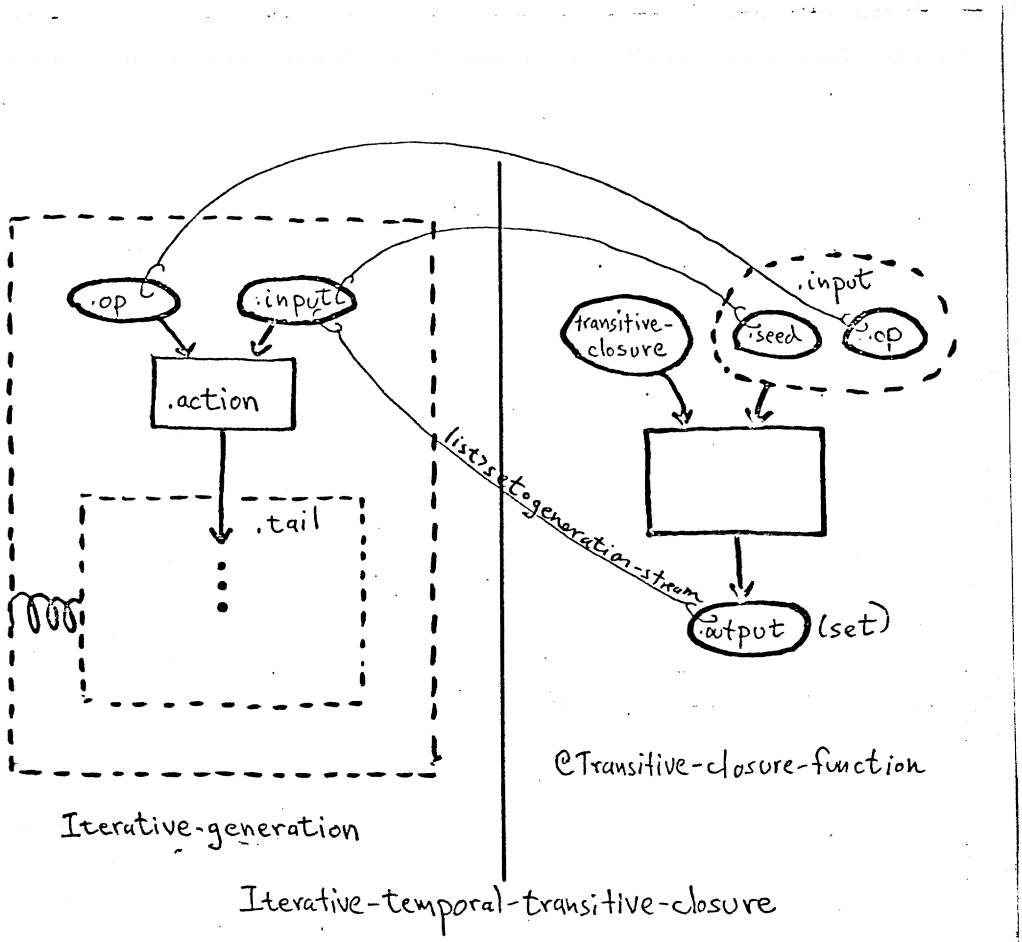


Figure 9-19. Iterative Generation as Transitive Closure.

Table XX. Temporal Each and Find

TemporalOverlay **iterative-temporal-each:** iterative-application → each
correspondences
iterative-application.action.op = each.op
∧ list>set(application-in-stream(iterative-application)) = each.old
∧ list>set(application-out-stream(iterative-application)) = each.new

TemporalOverlay **iterative-temporal-find:** iterative-search → set-find
correspondences
iterative-search.exit.if.criterion = set-find.criterion
∧ list>set(steady-state-stream(iterative-search)) = set-find.universe
∧ iterative-search.exit.end.output = set-find.output
∧ iterative-search.exit.end.out = set-find.out

Table XXI. Temporal Restrict.

TemporalOverlay **filtering-in-stream:** iterative-filtering → list
correspondences iterative-filtering.filter.if.input = list.head
∧ filtering-in-stream(iterative-filtering.tail) = list.tail

TemporalOverlay **filtering-succeed-stream:** iterative-filtering → list
definition S = filtering-succeed-stream(F) ≡
[[F.filter.if.fail ≠ ⊥ ⊃ S = filtering-succeed-stream(F.tail)]
∧ [F.filter.if.succeed ≠ ⊥ ⊃ [S.head = F.filter.if.input
∧ S.tail = filtering-succeed-stream(F.tail)]]]

TemporalOverlay **iterative-temporal-restrict:** iterative-filtering → restrict
correspondences
iterative-filtering.filter.if.criterion = restrict.criterion
∧ list>set(filtering-in-stream(iterative-filtering)) = restrict.old
∧ list>set(filtering-succeed-stream(iterative-filtering)) = restrict.new

---

Table XXI shows the temporal implementation of Restrict as a filtering loop. The overlays Filtering-in-stream and Filtering-succeed-stream (see Fig. 9-20) describe how to temporally abstract the stream of input values to the test of a filtering loop and the stream of values seen in an environment where the test has succeeded (i.e. in the Then role). The definition of Filtering-in-stream has the same recursive form as the temporal overlays for iterative generation and application introduced earlier. Filtering-succeed-stream is more complicated. The basic idea of this definition is to skip the inputs which do not satisfy the test predicate . This is done by defining the stream abstraction when the test fails to be the same as the stream abstraction of the next time around the

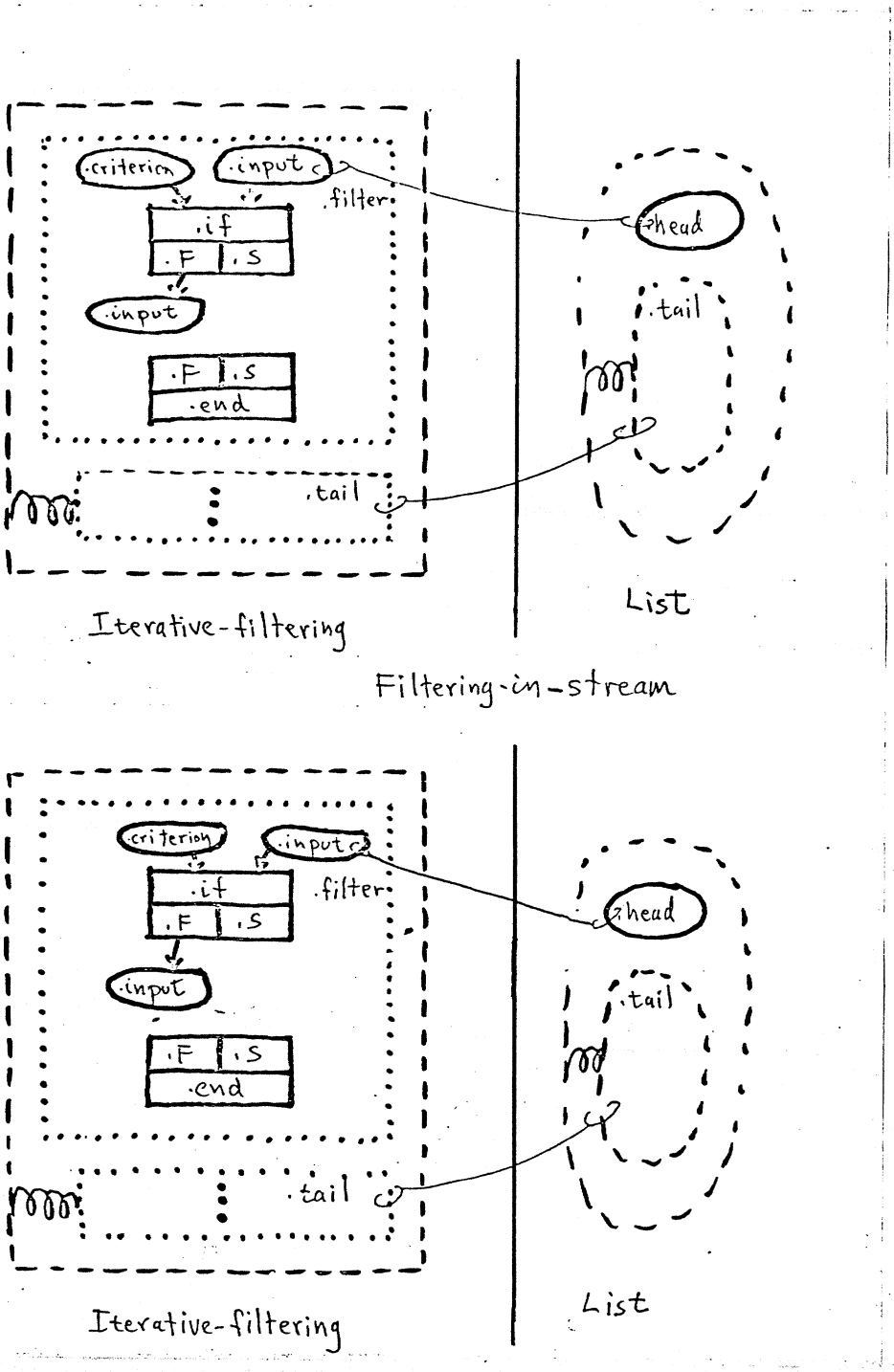


Figure 9-20. Stream Abstractions of Iterative Filtering.

loop (i.e. the tail of the recursive plan).[1]

The last overlay in Table XXI is Iterative-temporal-restrict, also shown in Fig. 9-21. This overlay has a similar structure to all the other temporal set overlays in this section. The Old set input to Restrict corresponds to the input values of the filter test. The New set output corresponds to the input values selected by the succeed case. The Criterion of Restrict is the same as the Criterion of the filter test.

The last temporal overlay in this section is an example of how to temporally abstract a loop with two exits (Cascade-iterative-termination). In particular, we consider here the plan Terminated-iterative-search, defined in Table XXII and shown on the left of Fig. 9-22, in which the second exit test (If-two) is performing a search. This means that this test is an instance of @Predicate, and when the test succeeds, the input tested becomes the output object of the corresponding join (End-two), just as in Iterative-search. When the first test (If-one) succeeds, it means that the search has failed. The following is an example of how this kind of loop might be coded for searching a finite Lisp list.

Table XXII. Temporal Any.

TemporalPlan **terminated-iterative-search**
  specialization cascade-iterative-termination
  roles .if-one(test) .if-two(@predicate) .end-one(join) .end-two(join-outputs)
      .tail(terminated-iterative-search)
  constraints .if-two.input = .end-two.succeed-input
      ∧ .tail.end-two.output = .end-two.fail-input

TemporalOverlay **iterative-temporal-any:** terminated-iterative-search → any
  correspondences
      list>set(termination-in-stream(cascade>iterative-termination(terminated-iterative-search))
                                  = any.universe
      ∧ terminated-iterative-search.if-two.criterion = any.criterion
      ∧ terminated-iterative-search.end-two.output = any.output
      ∧ terminated-iterative-search.end-one.out = any.fail
      ∧ terminated-iterative-search.end-two.out = any.succeed

---

1. This is a somewhat awkward construction, but I could not think of a better way of formally defining the idea of leaving out parts of the input stream. This way of modelling filtering is also motivated by considering the general case of tree recursion, where the structure of the selected inputs has to be like the structure of the input tree with chunks missing at various places in the middle.

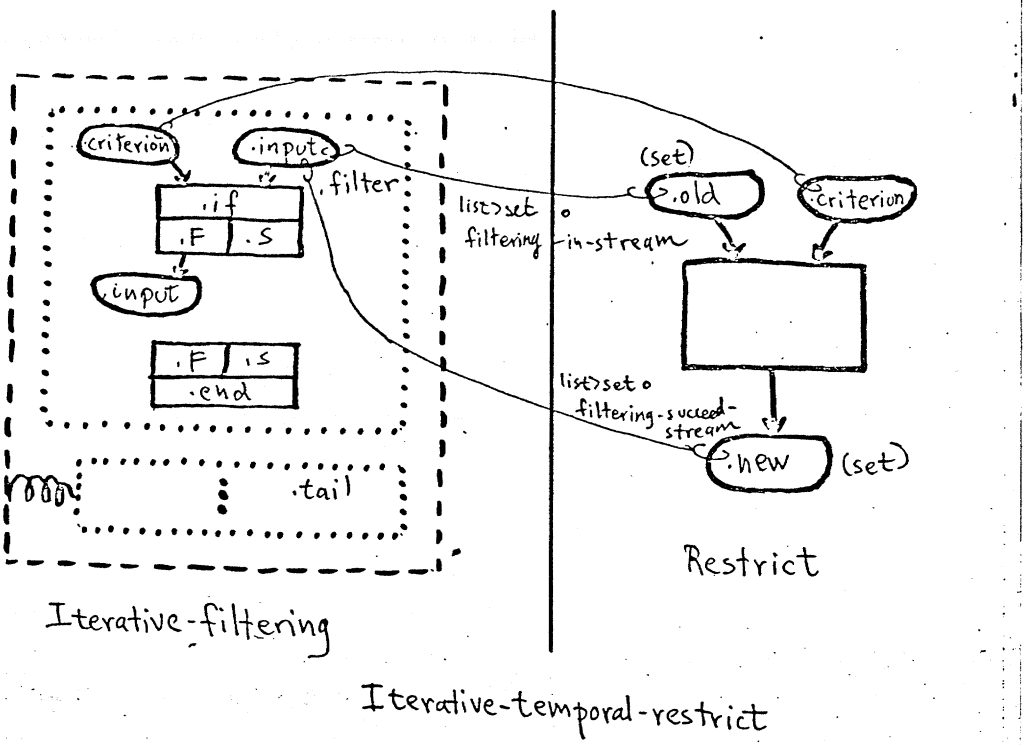


Figure 9-21. Temporal Set Abstraction of Iterative Filtering.

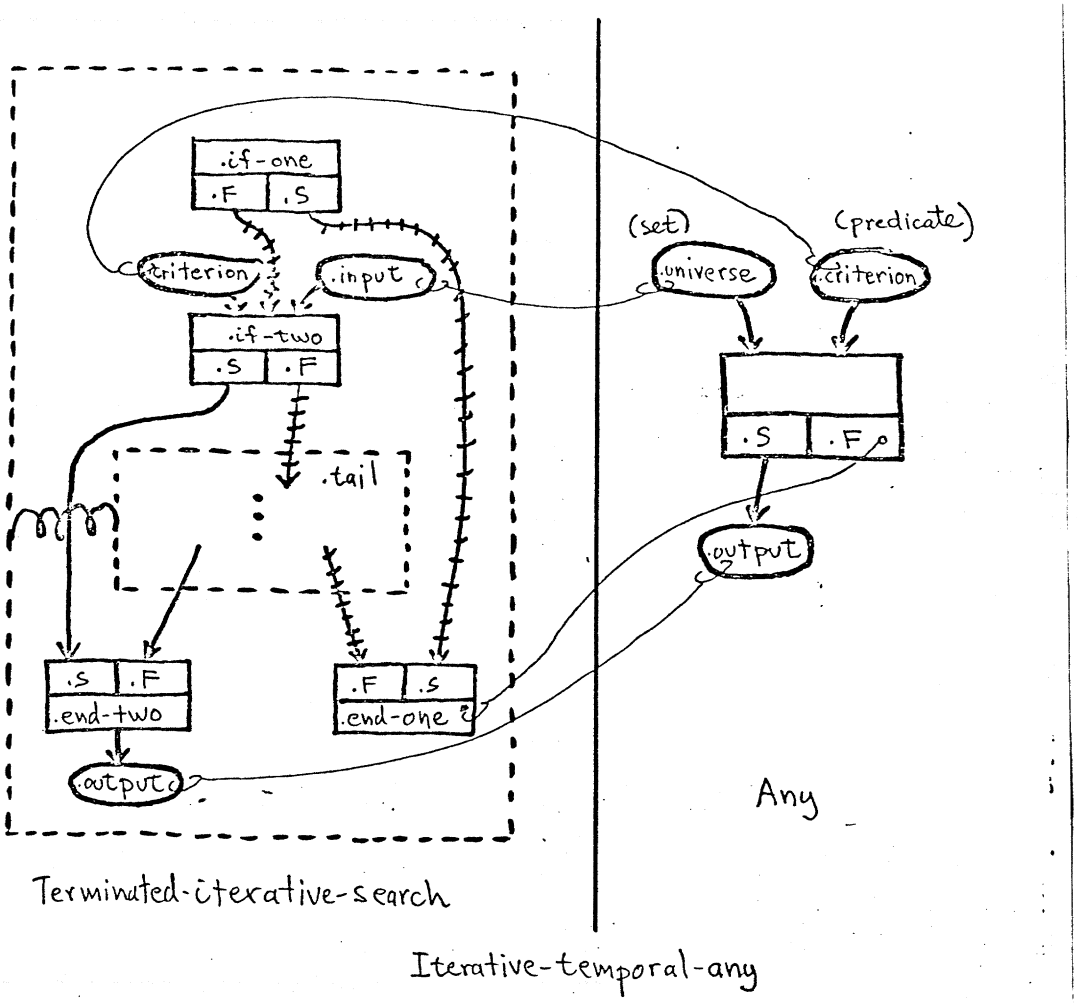


Figure 9-22. Temporal Set Abstraction of Terminated Iterative Search.

```
(PROG (L ENTRY)
      ...
   LP (COND ((NULL L)(GO FAIL)))
      (SETQ ENTRY (CAR L))
      (COND ((P ENTRY)
             (GO SUCCEED)))
      (SETQ L (CDR L))
      (GO LP)
SUCCEED ...ENTRY...
      ...
 FAIL ...)
```

However a more typical way of coding Terminated-iterative-search in Lisp is illustrated by the following code from the symbol table example.

```
(DEFINE LOOKUP
   (LAMBDA (...)
      (PROG (BKT ENTRY)
            ...
         LP (COND ((NULL BKT)(RETURN NIL)))
            (SETQ ENTRY (CAR BKT))
            (COND ((...ENTRY...) (RETURN ENTRY))
            (SETQ BKT (CDR BKT))
            (GO LP)))))
```

Here rather than maintaining two control flow paths to represent the succeed and fail cases, the result of the search is encoded in a flag[1] which is returned as the output of the loop. I believe this should be understood as an artifact of the restriction in Lisp that a subroutine can have only one return point. The original two control flow paths are typically recovered when the subroutine is invoked, as in the following code.

```
(SETQ FOUND (LOOKUP ...))
(COND (FOUND ...FOUND...)
      (T ...))
```

If the stream of inputs to the second test of Terminated-iterative-search is abstracted as a set, this plan can be viewed as the implementation of the Any specification, as shown in Table XXII and Fig. 9-22. Specifically, the Universe of Any is the (finite) set of inputs to If-two under the assumption that that exit is never taken. The criterion of If-two corresponds to the criterion of Any; the output of End-two corresponds to the output of Any; and the output situations of End-one and End-two correspond to the Fail and Succeed situations of Any, respectively.

---

1. This idea of a flag is formalized in the appendix.

Accumulation

This section discusses various ways to abstract iterative accumulation programs such as the following.

```
(PROG (ACCUM ...)
      (SETQ ACCUM ...)
      ...
   LP (COND ((...)(RETURN ACCUM)))
      ...
      (SETQ ACCUM (CONS ... ACCUM))
      ...
      (GO LP))
```

To begin, the following is the temporal overlay for viewing the Input's to the Add steps of an accumulation loop as a list.

TemporalOverlay **accumulation-stream:** iterative-accumulation → list+nil
definition S = accumulation-stream(A) ≡
[[A.exit.if.succeed ≠ ⊥ ⊃ S = nil]
∧ [A.exit.if.fail ≠ ⊥ ⊃ [S.head = A.add.input
∧ S.tail = accumulation-in-stream(A.tail)]]]

This definition breaks down into two cases: if the recursion terminates on the current level, then the temporal abstraction of the inputs is Nil; otherwise, the head of the list is the current Add.Input and the tail is defined recursively.

The special case of iterative accumulation in which the Add roles are filled by instances of Push, and the Init is an instance of Nil, is called Iterative-list-accumulation, as shown in Table XXIII. Fig. 9-23 show how Iterative-list-accumulation can be viewed as the operation of making the stream

---

Table XXIII. List Accumulation.

TemporalPlan **iterative-list-accumulation** *specialization* iterative-accumulation
roles .exit(cond) .init(nil) .add(push) .tail(iterative-list-accumulation)

TemporalOverlay **iterative-list-accumulation>@reverse:** iterative-list-accumulation → @reverse
correspondences
list>sequence(accumulation-stream(iterative-list-accumulation)) = @reverse.input
∧ list>sequence(iterative-list-accumulation.exit.end.output) = @reverse.output
∧ iterative-list-accumulation.in = @reverse.in
∧ iterative-list-accumulation.out = @reverse.out

IOspec **@reverse** / .op(function) .input(finite-sequence) ⇒ .output(finite-sequence)
specialization @function
preconditions .op = reverse

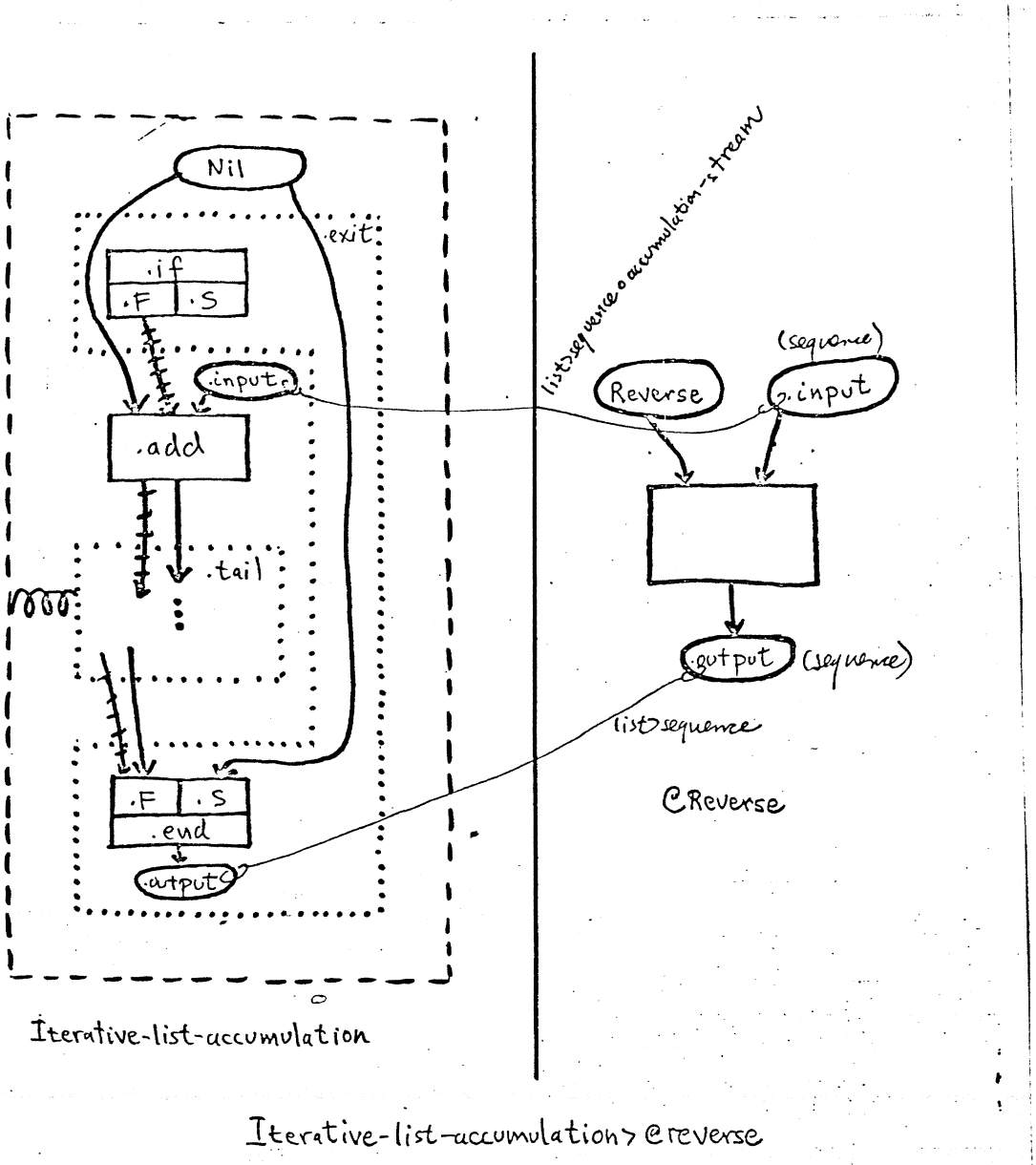


Figure 9-23. Accumulating a Stream as a List.

of Input's to Add in the temporal viewpoint available (in reverse order) as the output of an accumulation loop.[1]

Thus this overlay gives a crucial correspondence between the temporal viewpoint taken inside a loop and objects, such as Exit.End.Output, which come out of the loop and are used later. For example, the following program to reverse a Lisp list[2] is analyzed as the temporal composition of an instance of Car+cdr+null, which generates the stream of inputs to CONS at the point underlined below, with an instance of List-accumulation, which accumulates the stream in reverse order as the list in M.

```
(DEFINE REVERSE
   (LAMBDA (L)
      (PROG (M)
         LP (COND ((NULL L)(RETURN M)))
            (SETQ M (CONS (CAR L) M))
            (SETQ L (CDR L))
            (GO LP))))
```

Similary, the special case of iterative accumulation in which the Add roles are filled by instances of Set-add, and the Init is an empty set, can be viewed as the operation of making the stream of Input's to Add in the temporal available as a set outside the loop.[3] This overlay is shown in Table XXIV and Fig. 9-24.

Another special case accumulation plan which is abstracted in terms of sets is Iterative-aggregation, shown in Table XXV. In this plan the Add roles are filled by applications of an aggregative function (such as Plus, Times, or Union), and the Init of the accumulation is the identity element of that function. Iterative-aggregation can be viewed as a temporal implementation of Aggregate, as defined by the overlay Temporal-aggregate in Table XXV and as shown in Fig. 9-25.

Table XXIV. Set Accumulation.

TemporalPlan **iterative-set-accumulation** *specialization* iterative-accumulation
roles .exit(cond) .init(set) .add(set-add) .tail(iterative-set-accumulation)
constraints empty(.init)

TemporalOverlay **iterative-temporal-set-accumulation:** iterative-set-accumulation → set
properties ∀ *A* iterative-temporal-set-accumulation(*A*) = set(*A*.exit.end.output, *A*.exit.end.out)
correspondences
list>set(accumulation-stream(iterative-set-accumulation)) = set

---

1. Reverse is a standard relation on sequences defined in the appendix.
2. In which Push is implemented as **CONS**.
3. Notice that at this level of abstraction, we don't say exactly how this set (and therefore Set-add and Empty) are implemented.

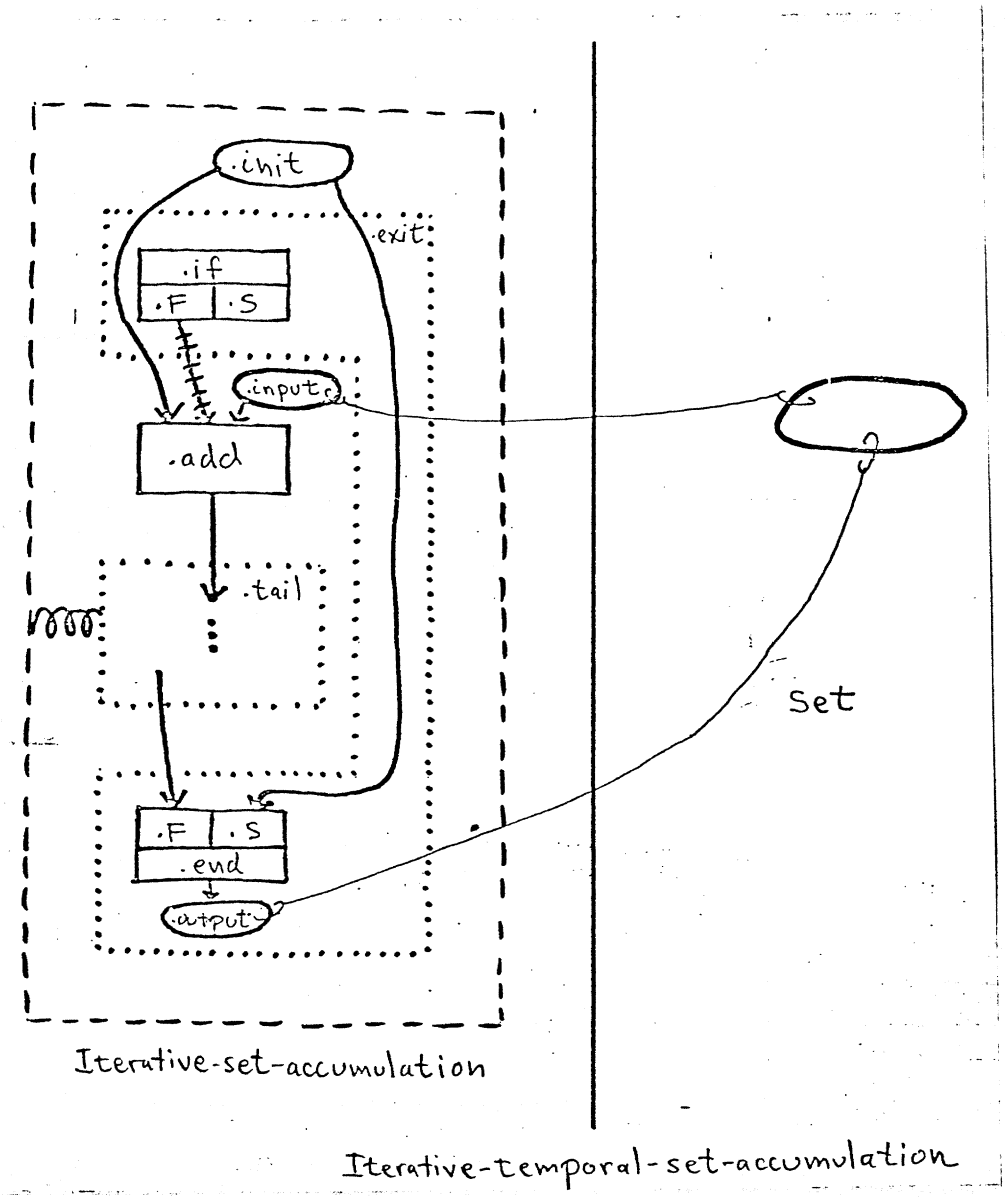


Figure 9-24. Accumulating a Temporal Set.

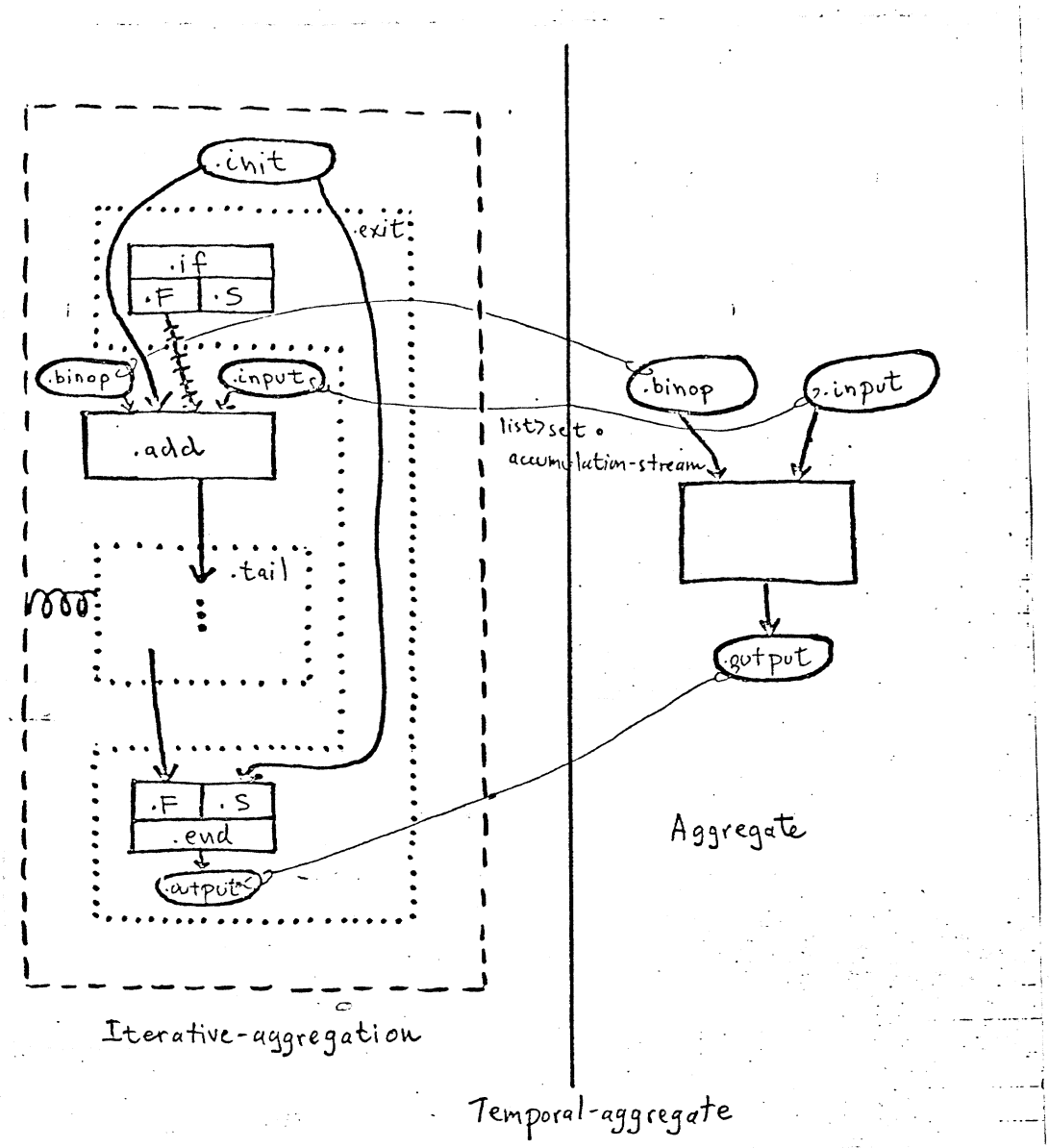


Figure 9-25. Temporally Aggregating a Set.

Table XXV. Temporal Aggregate.

TemporalPlan **iterative-aggregation** *specialization* iterative-accumulation
roles .exit(cond) .init(object) .add(@aggregative) .tail(iterative-aggregation)
constraints identity(.add.binop,.init)

IOspec **@aggregative** / .binop(aggregative-binfunction) .old(object) .input(object)
⇒ .new(object)
extension old+input+new
postconditions binapply(.binop,.old,.input)=.new

TemporalOverlay **temporal-aggregate:** iterative-aggregation → aggregate
correspondences
list>set(accumulation-stream(iterative-aggregation))=aggregate.input
∧ iterative-aggregation.add.binop=aggregate.binop
∧ iterative-aggregation.exit.end.output=aggregate.output
∧ iterative-aggregation.exit.end.out=aggregate.out

The stream of Input's to Add, viewed as a set, corresponds to the input to Aggregate. The aggregative function applied by Add is the Binop of Aggregate. The output of the end join of the iterative plan corresponds to the output of Aggregate.

A further overlay, not shown here, can be defined to analyze accumulation loops in which the function applied is aggregative, but the Init is not the identity element; as for example a summation loop which starts with an initial sum of 5. Such loops can be abstracted as an Aggregate operation in which the input set is obtained by adding the Init to the Accumulation-stream, viewed as a set.

Non-Iterative Temporal Abstraction

This section discusses singly recursive programs in which there is computation "on the way up", i.e. in which the recursive invocation is not the last step in the program. The kind of computation most commonly performed on the way up is accumulation, such as the following program with Lisp list accumulation on the way up.

```
(DEFINE COPYLIST
   (LAMBDA (L)
      (COND ((NULL L) NIL)
            (T (CONS (CAR L)(COPYLIST (CDR L)))))))
```

One way of thinking about this programming technique is to compare the program above with the REVERSE program of the last section, which has the same generation part, but in which the list

accumulation is done iteratively ("on the way down"). This comparison is made easier by re-coding REVERSE tail recursively as shown below.[1]

```
(DEFINE REVERSE
   (LAMBDA (L)
      (REVERSE1 L NIL))))

(DEFINE REVERSE1
   (LAMBDA (L M)
      (COND ((NULL L) M)
            (T (REVERSE1 (CDR L)(CONS (CAR L) M))))))
```

In effect, the non-iterative program above is using the stack provided by the Lisp language implementation to reverse the order of objects flowing from the list generation to the list accumulation, in order to cancel out the order reversal introduced by the accumulation. The rest of this section will show how to formalize this way of understanding accumulation on the way up in terms of the corresponding iterative accumulation with an intervening order reversal.[2] Similar plans and overlays for other basic recursive computations on the way up (generation, application, etc.) can be constructed, but are much less common in typical programming use.[3]

In terms of the plan calculus, the difference between iterative and non-iterative singly recursive (linear) temporal plans is whether there is anything but instances of Join (or Join-outputs) after the recursive invocation (Tail). Instances of Join and Join-outputs on the way up are required in iterative plans to specify via control flow that the entire computation ends when any of its tails end, and to return any final values. In the plan for the COPYLIST program above, which is non-iterative, an instance of @Function (the Add role of the accumulation) comes after the tail. The plans for iterative and non-iterative linear accumulation can be compared diagrammatically on the right and left sides, respectively, of Fig. 9-26.

Table XXVI defines these two plans as specializations of a more general plan called Linear-accumulation.[4] The constraints on this plan require only that the accumulation function applied (Add.Binop) be the same each time, and that the Add step occur once at each level in the recursion except when the exit test succeeds. Also, in both the iterative and non-iterative versions, the Init object is returned when the recursion terminates on the very first exit test.

---

1. The two different Lisp codings have the same plan.
2. The cancellation between the reversal of the order of inputs on the way up and the reversal introduced by the iterative accumulation is a particular property of List-accumulation. The reversal on the way up is a general property of non-iterative temporal abstraction.
3. In fact, even the other common accumulations, other than list accumulation, are seldom done on the way up, since the order reversal is immaterial when the streams are viewed as sets.
4. This table contains an equivalent restatement of the Iterative-accumulation plan introduced in Chapter Three, where only loops were being considered.

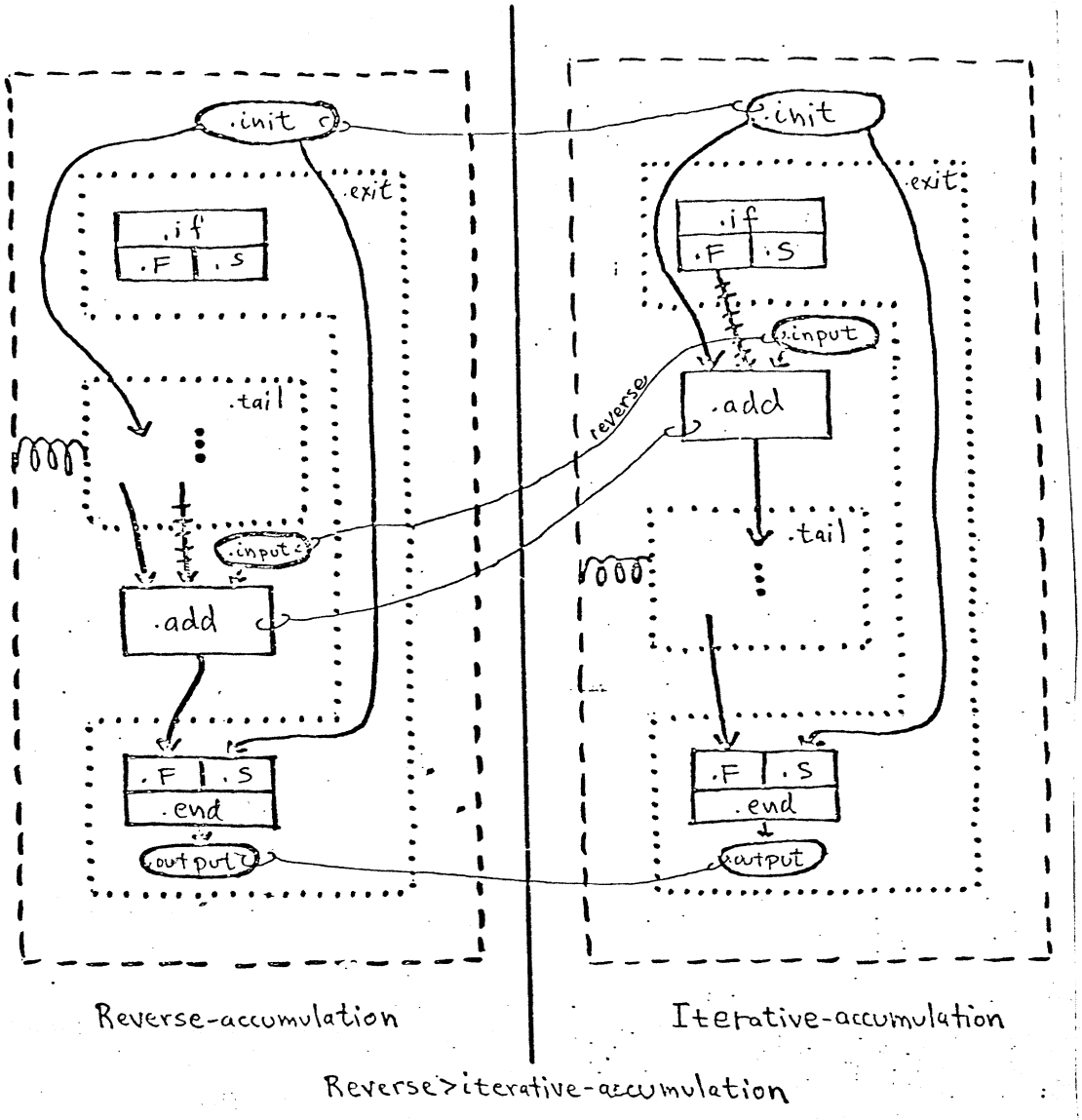


Figure 9-26. Accumulation on the Way Down vs. Up.

Table XXVI. Iterative and Non-iterative Accumulation.

TemporalPlan **linear-accumulation** *extension* iterative-termination
roles .exit(cond) .init(object) .add(old+input+new) .tail(linear-accumulation)
constraints .init=.exit.end.succeed-input
∧ (.exit.if.fail≠⊥ ⇋ .add.in≠⊥) ∧ (.add.in≠⊥ ⇋ .tail.exit.if.in≠⊥)

TemporalPlan **iterative-accumulation**
specialization linear-accumulation iterative-termination-output
roles .exit(cond) .init(object) .add(old+input+new) .tail(iterative-accumulation)
constraints .add.old=.init ∧ .add.new=.tail.add.old ∧ precedes(.add.out,.tail.add.in)

TemporalPlan **reverse-accumulation** *specialization* linear-accumulation
roles .exit(cond) .init(object) .add(old+input+new) .tail(reverse-accumulation)
constraints instance(join-outputs,.exit.end)
∧ .tail.exit.end.output=.add.old
∧ .add.new=.exit.end.fail-input
∧ .init=.init.tail ∧ precedes(.tail.add.out,.add.in)

Iterative-accumulation is obtained as a specialization of Linear-accumulation by adding the constraint that the Add step precedes the Tail, so that accumulation is done on the way down. There is then data flow from the New output of Add to Old input of the Add of the Tail. Also, in this form of accumulation, the Init at each level is the same as the output of the preceding Add. This can be seen in the REVERSE program above, in which the value returned is M, which is set to (CONS (CAR L) M) by the preceding repetition.

Reverse-accumulation, the plan for the non-iterative case, is obtained as a specialization of Linear-accumulation by constraining the Add step to follow the Tail, so that accumulation is done of the way up, and adding data flow to the Old input of Add from the New output of the Add of the Tail, via the join of the Tail. Also, in this form of accumulation, the Init is the same at each level, as can be seen in the COPYLIST program above, in which NIL is returned from whichever recursive invocation finally causes the COND to succeed.

Given this framework, the Accumulation-stream overlay is be generalized to apply to either instances of Iterative-accumulation or Reverse-accumulation.

Finally, as shown in Table XXVII and Fig. 9-26, the implicit order reversal of accumulation on the way up (as compared to on the way down) can now be modelled as an overlay, Reverse> iterative-accumulation, which establishes a correspondence between these two versions in which the type of Add operations, the Init's, and final outputs correspond, but the accumulation input streams are reversed. As mentioned earlier, similar overlays could be constructed between the iterative and non-iterative versions of other recursive plans, such as generation, application, etc.

Table XXVII. Temporal Reverse.

TemporalOverlay **reverse>iterative-accumulation:** reverse-accumulation → iterative-accumulation
correspondences
list>sequence(reverse(accumulation-stream(reverse-accumulation)))
= list>sequence(accumulation-stream(iterative-accumulation))
∧ reverse-accumulation.init = iterative-accumulation.init
∧ reverse-accumulation.add = iterative-accumulation.add
∧ reverse-accumulation.exit.end.output = iterative-accumulation.exit.end.output
∧ reverse-accumulation.exit.end.out = iterative-accumulation.exit.end.out

---

9.4 Recursive Structures

This section sketches how the epistemology of singly recursive data structures (lists, etc.) and temporal plans (loops, etc.) of the last two sections can be generalized to double and multiple recursion. Only a small amount of formal definition will be presented in this section, however, since the plans and overlays for doubly and multiply recursive structures tend to be longer and more detailed than those for linear structures, without introducing any fundamentally new ideas.

Table XXVIII shows the basic idea of double recursion. The data plan Double-recursion has two roles, Left and Right, which are either themselves instances of Double-recursion, or of type Atom, which is a primitive type used to terminate multiple recursions. Finite double recursion is defined analogously to finite single recursion. Recursion with a varying number of recursive instances at each level can be defined in terms of a single role which is constrained to be a set, each of which is either a recursive instance or an atom.

The doubly recursive data structure analogous to List is Binlist (binary list), a data structure with one head and two "tails", Left and Right.

Table XXVIII. Double Recursion.

DataPlan **double-recursion**
roles .left(double-recursion+atom) .right(double-recursion+atom)

Type **atom**

Type **double-recursion+atom** *uniontype* double-recursion atom

DataPlan **binlist** *extension* double-recursion
roles .head(object) .left(binlist+atom) .right(binlist+atom)

Type **binlist+atom** *uniontype* binlist atom

The binary data structure corresponding to Thread in the linear case is Bintree, and in the general case, Tree. In Lisp programming, binary trees are a more common data structure than binary lists, since a binary tree may be easily constructed out of dotted pairs, as described by Car-cdr-generator. A double recursion may be viewed as a binary tree in which the left and right recursive instances correspond to subtrees whose roots are successors of the root of the binary tree, as specified by the following overlay.

DataOverlay **double-recursion>bintree:** double-recursion+atom → bintree
definition T = double-recursion>bintree(R) ≡
[[instance(atom,R) ⇋ terminal(T,root(T))]
∧ [instance(double-recursion,R) ⊃
successor(T,root(T),root(double-recursion>bintree(R.left)))
successor(T,root(T),root(double-recursion>bintree(R.right)))]]

The basic plans for unbounded iterative computation introduced earlier in this chapter, generation, application, termination, filtering, and accumulation, can also be generalized to double and multiple recursion. For example, Table XXIX and Fig. 9-27 show the plan for doubly recursive generation, such as in the following code.

```
(DEFINE GENERATE
   (LAMBDA (S)
      ...(GENERATE (CAR S))...
      ...(GENERATE (CDR S))...))
```

Table XXIX. Binary Generation.

TemporalPlan **binary-generation** *extension* double-recursion
roles .current(object) .action-left(@function) .action-right(@function)
.left(binary-generation) .right(binary-generation)
constraints .current = .action-left.input ∧ .current = .action-right.input
∧ .left.action-left.op = .action-left.op ∧ .right.action-right.op = action-right.op
∧ .action-left.output = .left.current
∧ .action-right.output = .right.current

TemporalOverlay **temporal-binary-generator:** binary-generation → binary-generator
correspondences binary-generation.current = binary-generator.seed
∧ binary-generation.action-left.op = binary-generator.left
∧ binary-generation.action-right.op = binary-generator.right

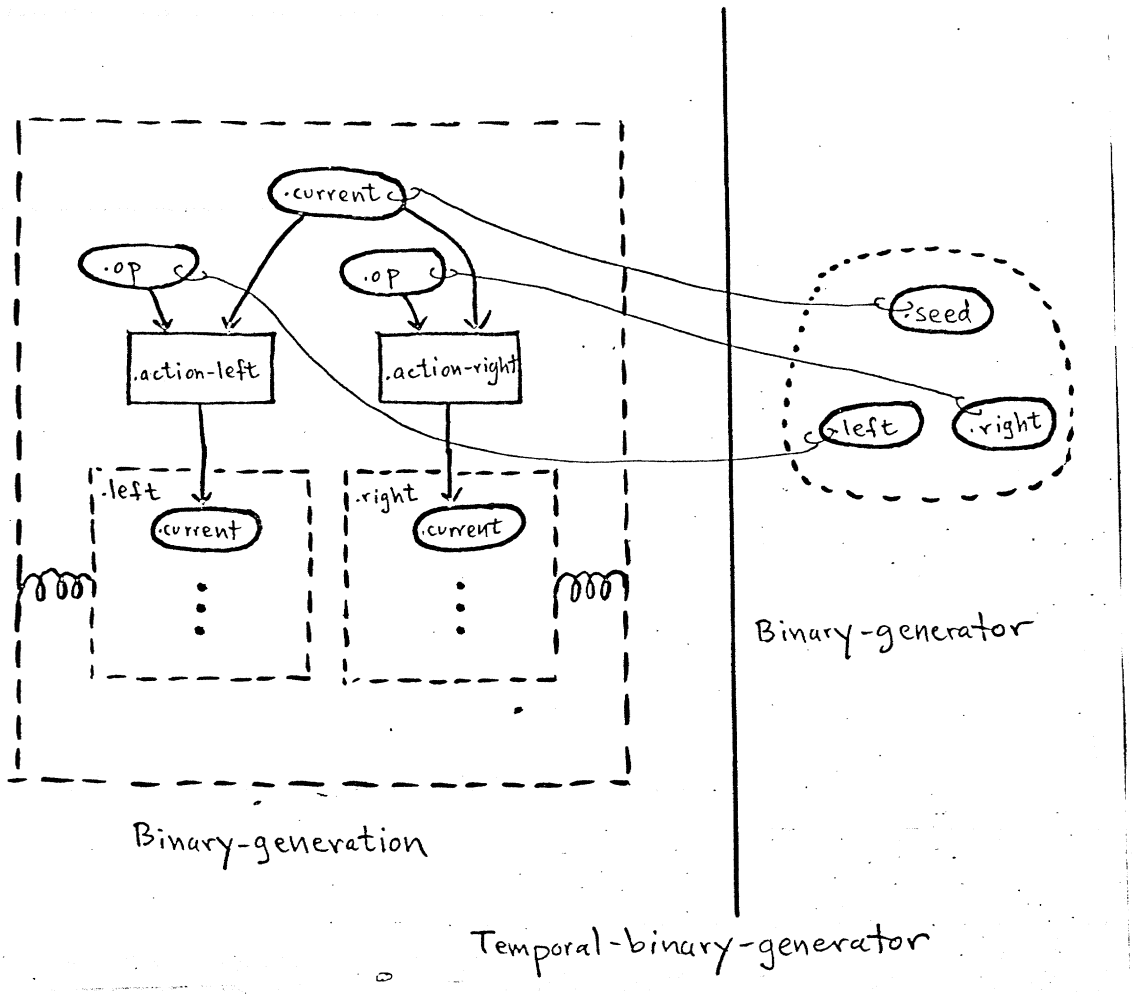


Figure 9-27. Binary Generation.

The overlay Temporal-binary-generator is analogous to Temporal-iterator for loops. It specifies how Binary-generation can be viewed as the temporal implementatation of the generator for a binary tree. For example, this is the overlay which relates the code above to the Lisp binary tree generator Car-cdr-generator.

Notice that there are no constraints in the Binary-generation plan between the order of execution of the left and right recursive invocations. Standard traversal orders for binary trees (such as pre- and post-order) are represented as specializations of Binary-generation. In the temporal view, these traversal orders can be viewed as overlays which "flatten" a tree into a linear structure in different ways.

Temporal abstraction of multiply recursive plans gives rise to tree structured streams and reverse streams. Operations on these temporal abstractions are the generalizations of the corresponding operations on temporal sequences, such as Iterate, Map, Truncate, and so on. A particularly important doubly recursive temporal plan is binary tree accumulation, as in the following code.

```
(DEFINE COPYTREE
   (LAMBDA (S)
      (COND ((ATOM S) S)
            (T (CONS (COPYTREE (CAR S))
                     (COPYTREE (CDR S)))))))
```

The plan for this program is the temporal composition of binary generation with binary truncation (on ATOM), and binary accumulation in which the accumulation function is Construct.

construct: double-recursion+atom $\times$ double-recursion+atom $\rightarrow$ double-recursion
definition $D = \text{construct}(L,R) \equiv [D.\text{left} = L \wedge D.\text{right} = R]$

Construct is the binary function which relates two instances of Double-recursion+atom with an instance of Double-recursion which has these as left and right parts. For binary trees in Lisp, application of Construct is implemented by CONS.

CHAPTER TEN

PLANS INVOLVING SIDE EFFECTS

This chapter makes two basic points. First of all, since reasoning about plans involving side effects can in general be quite difficult,[1] the inspection method approach is to formalize many common forms of side effect usage as plans and overlays in the plan library and use them in the analysis, synthesis and verification of programs to bypass general reasoning.

The second basic point is that, whenever possible, plans in the library are written at a level of abstraction which does not make any commitment to whether or not side effects are used. This principle is exemplified by the input-output specifications shown in Table I. In each case, the "impure" specification is viewed as a specialization of the pure specification. #Set-add is the

---

Table I. Impure Input-Output Specifications.

IOspec **old+new** / .old(object) ⇒ .new(object)

IOspec **#old+new** / .old(object) ⇒ .new(object) *specialization* old+new
postconditions .old = .new

IOspec **#set-add** / .old(set) .input(object) ⇒ .new(set)
specialization set-add #old+new

IOspec **#set-remove** / .old(set) .input(object) ⇒ .new(set)
specialization set-remove #old+new

IOspec **#restrict** / .old(set) .criterion(predicate) ⇒ .new(set)
specialization restrict #old+new

IOspec **#newarg** / .old(function) .arg(object) .input(object) ⇒ .new(function)
specialization newarg old+new

IOspec **#newvalue** / .old(function) .value(object) .input(object) ⇒ .new(function)
specialization newvalue #old+new

---

1. See Shrobe [56] for an approach to the explicit control of reasoning about plans involving side effects.

specification for adding a member to a set by side effect.[2] If a dotted pair is modelled as a pair, then these last two input-output specifications model RPLACA and RPLACD.

Table II and Fig. 10-1 show an example of a very general form of side effect which can be capture using plans and overlay. The right hand side of the overlay in Table II is the plan for modifying a function (Update.Old) such that all domain elements which used to map to a given range element (Action.Old) now map to a new element (Action.New) computed by some action on the old range element. For example, this is the structure of the SYMBOL-TABLE-ADD in Chapter Two: a new bucket is computed from an old bucket of the table by Set-add; the table (modelled as function from indices to buckets) is then modified so that the new bucket is the new value of the index of the old bucket. The overlay #Old+input+new>action+update records the fact that computing the new range element by side effect obviates the step of modifying the function itself. This is the way the

---

Table II. Updating a Function by Side Effect.

TemporalOverlay **#old+input+new>action+update:** #old+input+new → #action+update
properties $\forall AP\,[P = \#\text{old+input+new>action+update}(A) \supset$
[(instance(#set-add,A) ⇋ instance(#set-add,P.action))
∧ (instance(#set-remove,A) ⇋ instance(#set-remove,P.action))
∧ (instance(#newright,A) ⇋ instance(#newright,P.action))
∧ (instance(#newleft,A) ⇋ instance(#newleft,P.action))
∧ (instance(#internal-thread-add,A) ⇋ instance(#internal-thread-add,P.action))
∧ (instance(#internal-thread-remove,A)
⇋ instance(#internal-thread-remove,P.action))]]
correspondences #old+input+new.old = #action+update.action.old
∧ #old+input+new.input = #action+update.action.input
∧ #old+input+new.in = #action+update.action.in
∧ #old+input+new.out = #action+update.update.out

TemporalPlan **#action+update**
roles .action(old+input+new) .update(#newvalue)
constraints .action.old = .update.value ∧ .action.output = .update.input
∧ cflow(.action.out,.update.in)

IOspec **old+input+new** / .old(object) .input(object) ⇒ .new(object) *extension* old+new

IOspec **#old+input+new** / .old(object) .input(object) ⇒ .new(object)
extension #old+new
specialization old+input+new

---

2. Note that the prefix character "#" is used to name impure input-output specifications as a mnemonic device.

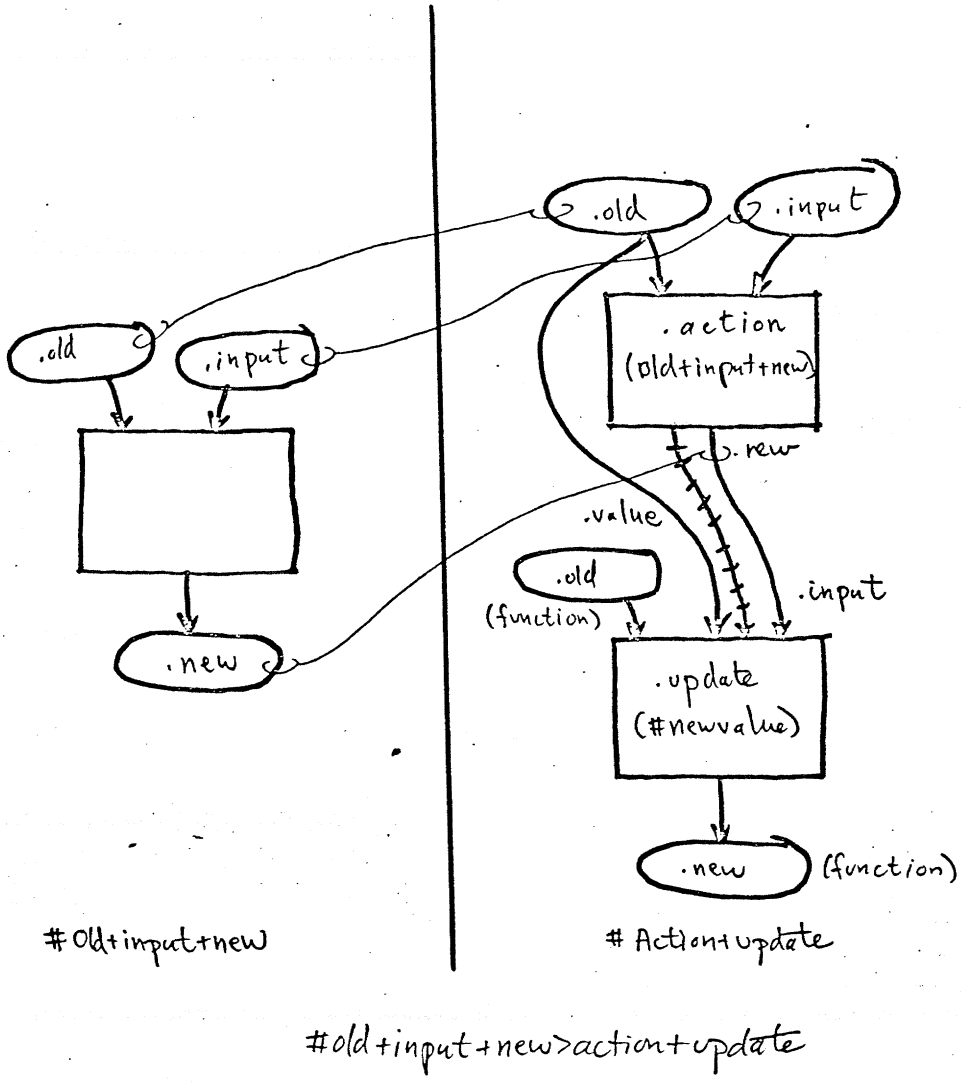


Figure 10-1. Updating a Function by Side Effect.

SYMBOL-TABLE-DELETE routine works (except for the special case handled separately before the loop): the new bucket is computed from the old bucket by side effect (splicing out), so that no subsequent ARRAYSTORE is required. Other specializations and extensions of #Old+input+new for which this implementation works are shown as properties of the overlay.

APPENDIX

PLAN LIBRARY REFERENCE MANUAL

1. SETS

Type **set**

Type **finite-set** *subtype* set
definition $\forall S$ instance(finite-set,S) $\leftrightharpoons$ (instance(set,S) $\wedge$ finite(size(S)))

Predicate **empty:** set $\rightarrow$ boolean
properties $\forall S$ (empty(S) $\leftrightharpoons$ size(S,0))
definition $\forall S$ empty(S) $\leftrightharpoons$ $\forall x \neg$member(S,x)

Predicate **universal:** set $\rightarrow$ boolean
properties $\forall S$ (universal(S) $\leftrightharpoons$ size(S,infinity))
definition $\forall S$ universal(S) $\leftrightharpoons$ $\forall x$ member(S,x)

Binrel **disjoint:** set $\times$ set $\rightarrow$ boolean
definition $\forall ST$ disjoint(S,T) $\leftrightharpoons$ $\forall x \neg$(member(S,x) $\wedge$ member(T,x))

IOspec **set-find** / .universe(set) .criterion(predicate) $\Rightarrow$ .output(object)
preconditions set-type(.universe) = argtype(.criterion)
$\wedge \exists x$ (member(.universe,x) $\wedge$ apply(.criterion,x) = true
postconditions member(.universe,.output) $\wedge$ apply(.criterion,.output) = true

IOspec **each** / .old(set) .op(function) $\Rightarrow$ .new(set)
preconditions set-type(.old) = domain-type(.op) $\wedge$ subset(.old,domain(.op))
postconditions set-type(.new) = range-type(.op)
$\wedge \forall y$ (member(.new,y) $\leftrightharpoons \exists x$ member(.old,x) $\wedge$ apply(.op,x) = y)

IOspec **restrict** / .old(set) .criterion(predicate) $\Rightarrow$ .new(set)
preconditions set-type(.old) = argtype(.criterion)
postconditions set-type(.new) = set-type(.old)
$\wedge \forall x$ (member(.new,x) $\leftrightharpoons$ (member(.old,x) $\wedge$ apply(.criterion,x) = true))

IOspec **set-add** / .old(set) .input(object) $\Rightarrow$ .new(set)
specialization old+input+new-set
preconditions instance(set-type(.old),.input)

postconditions set-type(.new) = set-type(.old) ∧ member(.new,.input)
∧ ∀ *x* (*x*≠.input ⊃ (member(.old,*x*) ⇋ member(.new,*x*)))

IOspec **set-remove** / .old(set) .input(object) ⇒ .new(set)
specialization old+input+new-set
preconditions instance(set-type(.old),.input)
postconditions set-type(.new) = set-type(.old) ∧ ¬member(.new,.input)
∧ ∀ *x* (*x*≠.input ⊃ (member(.old,*x*) ⇋ member(.new,*x*)))

Test **any** / .universe(finite-set) .criterion(predicate) ⇒ .output(object)$_{.succeed}$
condition ∃*x* member(.universe,*x*) ∧ apply(.criterion,*x*) = true
postconditions member(.universe,.output) ∧ apply(.criterion,.output)

1.1 Aggregating a Set.

IOspec **aggregate** / .input(finite-set) .binop(aggregative-binfunction) ⇒ .output(object)
preconditions subset(.input,argtype-one(.binop).instances) ∧ ¬empty(.input)
postconditions ∃*ST* (instance(finite-sequence,*S*)
∧ instance(finite-sequence,*T*)
∧ irredundant-sequence>set(*S*,.input)
∧ length(*T*) = length(*S*) ∧ first(*T*) = first(*S*)
∧ ∀ *i* (index(*T*,*i*) ∧ *i*≠1 ⊃
apply(*T*,*i*) = binapply(.binop,apply(*S*,oneminus(*i*)),apply(*T*,oneminus(*i*))))
∧ last(*T*,.output))

IOspec **sum** / .input(finite-set) .binop(aggregative-binfunction) ⇒ .output(object)
specialization aggregate
preconditions .binop = plus

IOspec **product** / .input(finite-set) .binop(aggregative-binfunction) ⇒ .output(object)
specialization aggregate
preconditions .binop = times

IOspec **aggregate-union** / .input(finite-set) .binop(aggregative-binfunction)
⇒ .output(object)
specialization aggregate
preconditions .binop = union

IOspec **aggregate-intersection** / .input(finite-set) .binop(aggregative-binfunction)
⇒ .output(object)
specialization aggregate
preconditions .binop = intersection

IOspec **max** / .input(finite-set) .binop(aggregative-binfunction) ⇒ .output(object)
specialization aggregate

preconditions .binop = greater

IOspec **min** / .input(finite-set) .binop(aggregative-binfunction) ⇒ .output(object)
specialization aggregate
preconditions .binop = lesser

1.2 Lists as Sets.

DataOverlay **list>set:** list+nil → set
properties ∀ *L,S* list>set(*L*,*S*) ⊃ (list>set(reverse(*L*),*S*)
∧ (instance(finite-list+nil,*L*) ⇋ instance(finite-set,*S*))
∧ (*L* = nil ⇋ empty(*S*)))
definition ∀ *L,S* list>set(*L*,*S*) ⇋
∀ *x* (member(*S*,*x*) ⇋ (*L*.head = *x* ∨ member(list>set(*L*.tail),*x*)))

DataOverlay **irredundant-list>set:** irredundant-list+nil → set
properties ∀ *L,S* irredundant-list>set(*L*,*S*) ⊃ length(*L*) = size(*S*)
definition ∀ *L,S* irredundant-list>set(*L*,*S*) ⇋ list>set(*L*,*S*)
∧ instance(irredundant-list+nil,*L*)

DataPlan **finite-list** *specialization* list finite-single-recursion

Type **finite-list+nil** *uniontype* finite-list nil

Type **irredundant-list+nil** *uniontype* irredundant-list nil

1.3 Sequences as Sets.

DataOverlay **sequence>set:** sequence → set
properties ∀ *S,T* sequence>set(*S*,*T*) ⊃ list>set(sequence>list(*S*),*T*)
definition ∀ *S,T* sequence>set(*S*,*T*) ⇋ ∀ *x* (member(*T*,*x*) ⇋ ∃ *i* *S*(*i*,*x*))

DataOverlay **irredundant-sequence>set:** irredundant-sequence → set
properties ∀ *S,T* irredundant-sequence>set(*S*,*T*) ⊃ length(*S*) = size(*T*)
definition ∀ *S,T* irredundant-sequence>set(*S*,*T*) ⇋ sequence>set(*S*,*T*)
∧ instance(irredundant-sequence,*S*)

1.4 Internal Set Add.

TemporalPlan **internal-labelled-thread-add**
roles .old(labelled-thread) .add(internal-thread-add) .update(newarg)
.new(labelled-thread)
constraints .old.spine = .add.old ∧ .old.label = .update.old
∧ .add.input = .update.arg
∧ .add.new = .new.spine ∧ .update.new = .new.label
properties ∀ *PA* internal-thread>set-add(*P*,*A*) ⊃
(instance(#internal-thread-add,*P*.add) ⇋ instance(#set-add,*A*))
correspondences
labelled-thread>set(internal-labelled-thread-add.old) = set-add.old
∧ internal-labelled-thread-add.update.input = set-add.input
∧ labelled-thread>set(internal-labelled-thread-add.new) = set-add.new

DataOverlay **labelled-thread>set :** labelled-thread → set
properties ∀ *L* list>set(*L*) = labelled-thread>set(list>labelled-thread(*L*))
definition ∀ *TS* labelled-thread>set(*T*,*S*) ⇋
∀ *x* (member(*S*,*x*) ⇋ ∃ *y* (node(*T*.base,*y*) ∧ *T*.label(*y*,*x*)))

TemporalOverlay **push>set-add :** push → set-add
properties ∀ *PP* push>set-add(*P*,*A*) ⊃
(instance(irredundant-list,*P*.output) ⊃ instance(set-add-one,*A*))
correspondences list>set(push.old) = set-add.old
∧ push.input = set-add.input
∧ list>set(push.new) = set-add.new
∧ push.in = set-add.in
∧ push.out = set-add.out

1.5 Set Removal for Irredundant Lists.

TemporalOverlay **@tail+internal>restrict:** @tail+internal → restrict-one
correspondences
irredundant-list>set(@tail+internal.action.input) = restrict-one.old
∧ complement(@tail+internal.update.if.criterion) = restrict-one.criterion
∧ irredundant-list>set(@tail+internal.update.end.output) = restrict-one.new
∧ @tail+internal.action.in = restrict-one.in
∧ @tail+internal.update.end.out = restrict-one.out

IOspec **restrict-one** / .old(set) .criterion(predicate) ⇒ .new(set)
specialization restrict
preconditions $\exists x$ (member(.old,x) ∧ apply(.criterion,x) = false)
∧ $\forall xy$ (member(.old,x) ∧ member(.old,y)
∧ apply(.criterion,x) = false ∧ apply(.criterion,y) = false ⊃ $x = y$)

TemporalPlan **@tail+internal**
roles .action(@head) .update(cond)
.internal(internal-labelled-thread-find+remove)
constraints instance(@predicate,.update.if)
∧ instance(@tail,.update.then)
∧ instance(join-outputs,.update.end)
∧ instance(irredundant-list,.action.input)
∧ .action.output = .update.if.input ∧ cflow(.action.out,.update.if.in)
∧ .action.input = .update.then.input
∧ .update.then.output = .update.end.succeed-input
∧ update.else.in = .internal.find.in ∧ update.else.out = .internal.remove.out
∧ list>labelled-thread(.action.input,.internal.old)
∧ .update.if.criterion = .internal.composite.criterion
∧ list>labelled-thread(.internal.new,.update.end.fail-input

TemporalPlan **internal-labelled-thread-find+remove** *extension* internal-thread-find+remove
roles .old(labelled-thread) .new(labelled-thread) .composite(function+predicate)
.find(internal-thread-find) .remove(internal-thread-remove)
constraints .old.spine = .find.universe ∧ .new.spine = .remove.output
∧ .old.label = .new.label ∧ .old.label = .composite.op
∧ function+predicate>predicate(.composite,.find.criterion)

1.6 Discrimination

Type **discrimination** *subtype* function
  definition $\forall F$ instance(discrimination,F) $\leftrightharpoons$ range-type(F,set)
    $\wedge$ $\forall b$ (bucket(F,b) $\supset$ set-type(b) = domain-type(F))
    $\wedge$ domain(F,domain-type(F).instances)

DataOverlay **discrimination>set:** discrimination $\rightarrow$ set
  definition $\forall FS$ discrimination>set(F,S) $\leftrightharpoons$ (domain-type(F) = set-type(S)
    $\wedge$ $\forall x$ (member(S,x) $\leftrightharpoons$ member(F(x),x)))
  properties $\forall FS$ discrimination>set(F,S)
    $\supset$ (instance(finite-set,S) $\leftrightharpoons$ $\forall b$ (bucket(D,b) $\supset$ instance(finite-set,b)))

Binrel **bucket:** discrimination $\times$ set $\rightarrow$ boolean
  definition $\forall DS$ bucket(D,S) $\leftrightharpoons$ member(range(D),S)

1.7 Testing Membership in a Discrimination.

TemporalOverlay **discriminate+@member>@member:** discriminate+@member $\rightarrow$ @member
  correspondences
    discrimination>set(discriminate+@member.discriminate.op) = @member.one
    $\wedge$ discriminate+@member.discriminate.input = @member.two
    $\wedge$ discriminate+@member.in = @member.in
    $\wedge$ discriminate+@member.succeed = @member.succeed
    $\wedge$ discriminate+@member.fail = @member.fail

TemporalPlan **discriminate+@member**
  roles .discriminate(@function) .if(@member)
  constraints instance(discrimination,.discriminate.op)
    $\wedge$ .discriminate.output = .if.one $\wedge$ .discriminate.input = .if.two
    $\wedge$ cflow(.discriminate.out,.if.in)

1.8 Updating a Discrimination.

TemporalOverlay **discriminate+action+update>action:**
discriminate+action+update → old+input+new-set
properties ∀ *D A* discriminate+action+update>old+input+new-set(*D*,*A*) ⊃
((instance(set-add,*D*.action) ⇋ instance(set-add,*A*))
∧ (instance(set-remove,*D*.action) ⇋ instance(set-remove,*A*))
∧ (instance(#newvalue,*D*.update) ⇋ instance(#old+input+new,*A*)))
correspondences
discrimination>set(discriminate+action+update.discriminate.op)
= old+input+new-set.old
∧ discriminate+action+update.discriminate.input = old+input+new-set.input
∧ discriminate+action+update.action.input = old+input+new-set.input
∧ discrimination>set(discriminate+action+update.update.new)
= old+input+new-set.new
∧ discriminate+action+update.discriminate.in = old+input+new-set.in
∧ discriminate+action+update.update.out = old+input+new-set.out

TemporalPlan **discriminate+action+update** *extension* action+update
roles .discriminate(@function) .action(old+input+new-set) .update(newvalue)
constraints instance(discrimination,.discriminate.op)
∧ .discriminate.output = .action.old
∧ .discriminate.output = .update.value
∧ .action.new = .update.input
∧ .discriminate.op = .update.old
∧ cflow(.discriminate.out,.action.in) ∧ cflow(.action.out,.newvalue.in)

IOspec **old+input+new-set** / .old(set) .input(object) ⇒ .new(set)
specialization old+input+new

1.9 Associative Retrieval and Deletion.

Test **retrieve** / .universe(finite-set) .key(function) .input(object) ⇒ .output(object)
condition $\exists x$ member(.universe,x) ∧ apply(.key,x)=.input
postconditions member(.universe,.output) ∧ apply(.key,.output)=.input

IOspec **expunge** / .old(finite-set) .key(function) .input(object) ⇒ .new(finite-set)
extension old+input+new-set
postconditions $\forall x$ member(.new,x) ⇋ (member(.old,x) ∧ ¬.key(x,.input))

IOspec **expunge-one** / .old(finite-set) .key(function) .input(object) ⇒ .new(finite-set)
specialization expunge
preconditions $\exists x$ (member(.old,x) ∧ .key(x,.input))
∧ $\forall xy$ (member(.old,x) ∧ member(.old,y)
∧ .key(x,.input) ∧ .key(y,.input) ⊃ $x=y$)

IOspec **#expunge** / .old(finite-set) .key(function) .input(object) ⇒ .new(finite-set)
specialization expunge
postconditions .old=.new

1.10 Implementation of Associative Retrieval.

TemporalOverlay **any>retrieve**: any-composite → retrieve
correspondences any-composite.universe=retrieve.universe
∧ any-composite.composite.op=retrieve.key
∧ any-composite.composite.two=retrieve.input
∧ any-composite.if.output=retrieve.output
∧ any-composite.if.in=retrieve.in
∧ any-composite.if.succeed=retrieve.succeed
∧ any-composite.if.fail=retrieve.fail

TemporalPlan **any-composite**
roles .composite(function+two) .if(any)
constraints binrel+two>predicate(.composite,.if.criterion)

1.11 Implementation of Associative Deletion.

TemporalOverlay **restrict>expunge:** restrict-composite → expunge
properties ∀ *R E* restrict>expunge(*R*,*E*) ⊃
((instance(restrict-one,*R*.action) ⇋ instance(expunge-one,*E*))
∧ (instance(#restrict,*R*.action) ⇋ instance(#expunge,*E*)))
correspondences restrict-composite.old = expunge.old
∧ restrict-composite.composite.op = expunge.key
∧ restrict-composite.composite.two = expunge.input
∧ restrict-composite.action.new = expunge.new
∧ restrict-composite.action.in = expunge.in
∧ restrict-composite.action.out = expunge.out

TemporalPlan **restrict-composite**
roles .composite(function+two) .action(restrict)
constraints complement(binrel+two>predicate(.composite),.action.criterion)

DataPlan **hashing** *specialization* composed-functions
roles .one(function) .two(irredundant-sequence)

DataPlan **keyed-discrimination** *specialization* composed-functions
roles .one(function) .two(function)
constraints range-type(.two,finite-set)
∧ ∀ *S* (member(range(.two),*S*) ⊃ set-type(*S*) = domain-type(.one))
properties ∀ *D* instance(keyed-discrimination,*D*) ⊃
instance(discrimination,composed>function(*D*))

1.12 Retrieval from a Keyed Discrimination.

TemporalOverlay **discriminate+retrieve>retrieve:** keyed-discriminate+retrieve → retrieve
correspondences
discrimination>set(composed>function(keyed-discriminate+retrieve.composite))
=retrieve.universe
∧ keyed-discriminate+retrieve.if.key = retrieve.key
∧ keyed-discriminate+retrieve.if.input = retreive.input
∧ keyed-discriminate+retrieve.if.output = retrieve.output
∧ keyed-discriminate+retrieve.discriminate.in = retrieve.in
∧ keyed-discriminate+retrieve.if.succeed = retrieve.succeed
∧ keyed-discriminate+retrieve.if.fail = retrieve.fail

TemporalPlan **keyed-discriminate+retrieve**
roles .composite(keyed-discrimination) .discriminate(@function) .if(retrieve)
constraints .composite.one = .if.key ∧ .composite.two = .discriminate.op
∧ .discriminate.input = .if.input
∧ .discriminate.output = .if.universe
∧ cflow(.discriminate.out,.if.in)

1.13 Associative Deletion from a Keyed Discrimination.

TemporalOverlay **discriminate+expunge+update>expunge:**
keyed-discriminate+expunge+update → expunge
properties ∀ *DE* discriminate+expunge+update>expunge(*D*,*E*) ⊃
((instance(expunge-one,*D*.action) ⇋ instance(expunge-one,*E*))
∧ (instance(#newvalue,*D*.update) ⇋ instance(#expunge,*E*)))
correspondences
discrimination>set(composed>function(keyed-discriminate+expunge+update.old))
=expunge.old
∧ discrimination>set
(composed>function(keyed-discriminate+expunge+update.new))
=expunge.new
∧ keyed-discriminate+expunge+update.action.input = expunge.input
∧ keyed-discriminate+expunge+update.action.key = expunge.key
∧ keyed-discriminate+expunge+update.discriminate.in = expunge.in
∧ keyed-discriminate+expunge+update.update.out = expunge.out

TemporalPlan **keyed-discriminate+expunge+update**
extension discriminate+action+update
roles .discriminate(@function) .action(expunge) .update(newvalue)
.old(keyed-discrimination) .new(keyed-discrimination)
constraints .discriminate.op = .old.two ∧ .action.key = .old.one
∧ .new.two = .update.new ∧ .new.one = .old.one

2. FUNCTIONS

IOspec **@function** / .op(function) .input(object) ⇒ .output(object)
preconditions member(domain(.op),.input)
postconditions apply(.op,.input) = .output

IOspec **@binfunction** / .binop(binfunction) .one(object) .two(object) ⇒ .output(object)
preconditions $\exists z$ binapply(.binop,.one,.two.) = z
postconditions binapply(.binop,.one,.two) = .output

IOspec **newarg** / .old(function) .arg(object) .input(object) ⇒ .new(function)
preconditions instance(domain-type(.old),.arg) ∧ instance(range-type(.old).input)
postconditions .new(.arg,.input)
∧ $\forall xy$ (.old(x,y) ∧ $x \neq$ .arg ⊃ .new(x,y))
∧ domain-type(.new) = domain-type(.old) ∧ range-type(.new) = range-type(.old)

IOspec **newvalue** / .old(function) .value(object) .input(object) ⇒ .new(function)
preconditions instance(range-type(.old),.value) ∧ instance(range-type(.old),.input)
postconditions $\forall x$ (.old(x,.value) ⊃ .new(x,.input))
∧ $\forall xy$ (.old(x,y) ∧ $y \neq$ .value ⊃ .new(x,y))
∧ $\forall xy$ (.new(x,y) ⊃ .old(x,y) ∨ (.old(x,.value) ∧ y = .input))
∧ domain-type(.new) = domain-type(.old) ∧ range-type(.new) = range-type(.old)

Test **@predicate** / .criterion(predicate) .input(object)
condition apply(.criterion,.input) = true

Test **@binrel** / .criterion(binrel) .one(object) .two(object)
condition binapply(.criterion,.one,.two) = true

2.1 Algebraic Binary Functions.

Type **algebraic-binfunction** *subtype* binfunction
  definition $\forall F$ instance(algebraic-binfunction,F) $\leftrightharpoons$ (instance(binfunction,F)
    $\wedge$ instance(algebraic-relation,F))

Function **identity**: algebraic-binfunction $\rightarrow$ object
  definition $\forall Fe$ identity(F,e) $\leftrightharpoons$ (instance(on(F),e)
    $\wedge \forall x$ (instance(on(F),x) $\supset$ $F(x,e,x) \wedge F(e,x,x)$))

Predicate **commutative**: algebraic-binfunction $\rightarrow$ boolean
  definition $\forall F$ commutative(F) $\leftrightharpoons \forall xyfg$ ($F(x,y,f) \wedge F(y,x,g) \supset$ on(F).equal(f,g))

Predicate **associative**: algebraic-binfunction $\rightarrow$ boolean
  definition $\forall F$ associative(F) $\leftrightharpoons$
    $\forall xyzfg$ ($F(F(x,y),z,f) \wedge F(x,F(y,z),g) \supset$ on(F).equal(f,g))

2.2 Aggregative Binary Functions.

Type **aggregative-binfunction** *subtype* algebraic-binfunction
  definition $\forall F$ instance(aggregative-binfunction,F) $\leftrightharpoons$ (instance(algebraic-binfunction,F)
    $\wedge$ associative(F) $\wedge$ commutative(F) $\wedge \exists e$ identity(F,e))

Binfunction **plus**: integer $\times$ integer $\rightarrow$ integer
  properties instance(aggregative-binfunction,plus) $\wedge$ identity(plus,0)

Binfunction **times**: integer $\times$ integer $\rightarrow$ integer
  properties instance(aggregative-binfunction,times) $\wedge$ identity(times,1)

Binfunction **union**: set $\times$ set $\rightarrow$ set
  properties instance(aggregative-binfunction,union)
    $\wedge \forall S$ (empty(S) $\supset$ identity(union,S))
  definition $\forall STU$ union(S,T,U) $\leftrightharpoons$
    $\forall x$ (member(U,x) $\leftrightharpoons$ (member(S,x) $\vee$ member(T,x)))

Binfunction **intersection**: set $\times$ set $\rightarrow$ set
  properties instance(aggregative-binfunction,intersection)
    $\wedge \forall S$ (universal(S) $\supset$ identity(intersection,S))
  definition $\forall STU$ intersection(S,T,U) $\leftrightharpoons$
    $\forall x$ (member(U,x) $\leftrightharpoons$ (member(S,x) $\wedge$ member(T,x)))

Binfunction **greater**: integer $\times$ integer $\rightarrow$ integer
  properties instance(aggregative-binfunction,greater)
    $\wedge \forall i$ (i=minus-infinity $\supset$ identity(greater,i))
    $\wedge$ binrel>binchoice(le,greater)

definition $\forall ijk$ greater(i,j,k) $\leftrightharpoons$ ($j = k \leftrightharpoons$ le(i,j)) $\wedge$ ($i = k \leftrightharpoons$ le(j,i))

Binfunction **lesser:** integer $\times$ integer $\rightarrow$ integer
properties instance(aggregative-binfunction,lesser)
$\wedge \forall i$ (i = infinity $\supset$ identity(lesser,i))
$\wedge$ binrel>binchoice(ge,lesser)
definition $\forall ijk$ lesser(i,j,k) $\leftrightharpoons$ ($j = k \leftrightharpoons$ ge(i,j)) $\wedge$ ($i = k \leftrightharpoons$ ge(j,i))

2.3 Composed Functions.

DataOverlay **composed>function:** composed-functions $\rightarrow$ function
definition $\forall CF$ composed>function(C,F) $\leftrightharpoons$ (domain-type(F) = domain-type(C.one)
$\wedge$ range-type(F) = range-type(C.two)
$\wedge \forall xy$ ($F(x,y) \leftrightharpoons C$.two($C$.one($x$),$y$)))

DataPlan **composed-functions**
roles .one(function) .two(function)
constraints range-type(.one) = domain-type(.two) $\wedge$ subset(range(.one),domain(.two))

TemporalOverlay **composed>@function:** composed-applies $\rightarrow$ @function
correspondences composed-@functions.one.input = @function.input
$\wedge$ composed>function(composed-@functions.composite) = @function.op
$\wedge$ composed-@functions.two.output = @function.output
$\wedge$ composed-@functions.one.in = @function.in
$\wedge$ composed-@functions.two.out = @function.out

TemporalPlan **composed-@functions**
roles .composite(composed-functions) .one(@function) .two(@function)
constraints .composite.one = .one.op $\wedge$ .composite.two = .two.op
$\wedge$ .one.output = .two.input $\wedge$ cflow(.one.out,.two.in)

TemporalOverlay **newvalue-composite>newvalue:** newvalue-composite $\rightarrow$ newvalue
properties $\forall PN$ newvalue-composite>newvalue(P,N) $\supset$
(instance(#newvalue,P.action) $\leftrightharpoons$ instance(#newvalue,N))
correspondences newvalue-composite.action.value = newvalue.value
$\wedge$ newvalue-composite.action.input = newvalue.input
$\wedge$ composed>function(newvalue-composite.old) = newvalue.old
$\wedge$ composed>function(newvalue-composite.new) = newvalue.new
$\wedge$ newvalue-composite.action.in = newvalue.in
$\wedge$ newvalue-composite.action.out = newvalue.out

TemporalPlan **newvalue-composite**
roles .action(newvalue) .old(composed-functions) .new(composed-functions)
constraints .old.one = .new.one $\wedge$ .action.old = .old.two $\wedge$ action.new = .new.two

2.4 Updating a Bijection.

TemporalOverlay **newarg>newvalue:** newarg-bijection → @function+newvalue
properties $\forall NP$ newarg>newvalue(N,P) ⊃
(instance(#newarg,N) ⇋ instance(#newvalue,P.update))
correspondences newarg-bijection.old = @function+newvalue.update.old
∧ newarg-bijection.arg = @function+newvalue.action.input
∧ newarg-bijection.input = @function+newvalue.update.input
∧ newarg-bijection.new = @function+newvalue.update.new
∧ newarg-bijection.in = @function+newvalue.action.in
∧ newarg-bijection.out = @function+newvalue.update.out

IOspec **newarg-bijection** / .old(bijection) .arg(object) .input(object)
⇒ .output(bijection)
specialization newarg

TemporalPlan **@function+newvalue**
roles .action(@function) .update(newvalue)
constraints instance(bijection,.action.op) ∧ .action.op = .update.old
∧ .action.output = .update.value ∧ cflow(.action.out,.update.in)

2.5 Sequences.

Type **sequence** *subtype* function
definition $\forall F$ instance(sequence,F) ⇋ (domain-type(F,natural) ∧ $\exists L$ length(F,L))

Type **finite-sequence** *subtype* sequence
properties $\forall S$ (instance(finite-sequence,S) ⊃ $\exists x$ last(S,x))
definition $\forall S$ instance(finite-sequence,S) ⇋ (instance(sequence,S) ∧ finite(length(S))

Function **length:** sequence → cardinal
definition $\forall SL$ length(S,L) ⇋
$\forall i$ (($\exists x$ (S(i,x)) ⇋ (instance(natural,i) ∧ le(i,L)))

Binrel **index:** sequence × natural → boolean
definition $\forall Si$ index(S,i) ⇋ (instance(natural,i) ∧ le(i,length(S)))

Function **first:** sequence → object
definition $\forall Sx$ first(S,x) ⇋ S(1,x)

Function **last:** finite-sequence → object
definition $\forall Sx$ last(S,x) ⇋ S(length(S),x)

Function **butlast:** finite-sequence → finite-sequence
definition $\forall ST$ butlast(S,T) ⇋ oneplus(length(T),length(S))

$\wedge\ \forall ix\ (\text{index}(T,i) \supset (S(i,x) \leftrightharpoons T(i,x)))$

2.6 IOspecs with Sequences.

IOspec **term** / .op(sequence) .input(natural) ⇒ .output(object) *specialization* @function
preconditions index(.op,.input)

IOspec **newterm** / .old(sequence) .arg(natural) .input(object) ⇒ .new(sequence)
specialization newarg
preconditions index(.old,.arg)

IOspec **#newterm** / .old(sequence) .arg(natural) .input(object) ⇒ .new(sequence)
specialization newterm
postconditions .old = .new

IOspec **truncate** / .input(sequence) .criterion(predicate) ⇒ .output(finite-sequence)
preconditions $\exists i$ apply(.criterion,apply(.input,*i*)) = true
postconditions
$\forall i$ (index(.output,*i*) ↭ $\forall j$ (le(*j*,*i*) ⊃ apply(.criterion,apply(.input,*j*)) = false))
$\wedge\ \forall i$ (index(.output,*i*) ⊃ apply(.output,*i*) = apply(.input,*i*))
$\wedge$ apply(.criterion,apply(.input,oneplus(length(.output)))) = true

IOspec **truncate-inclusive** / .input(seqence) .criterion(predicate)
⇒ .output(finite-sequence)
preconditions $\exists i$ apply(.criterion,apply(.input,*i*)) = true
postconditions
$\forall i$ (index(.output,*i*) ↭ $\forall j$ (lt(*j*,*i*)
⊃ apply(.criterion,apply(.input,*j*)) = false))
$\wedge\ \forall i$ (index(.output,*i*) ⊃ apply(.output,*i*) = apply(.input,*i*))
$\wedge$ apply(.criterion,last(.output)) = true

IOspec **earliest** / .input(sequence) .criterion(predicate) ⇒ .output(object)
preconditions $\exists i$ apply(.criterion,apply(.input,*i*)) = true
postconditions apply(.criterion,.output) = true
$\wedge\ \exists i$ (apply(.input,*i*) = .output
$\wedge\ \forall j$ (lt(*j*,*i*) ⊃ apply(.criterion,apply(.input,*j*)) = false))

IOspec **iterate** / .input(iterator) ⇒ .output(sequence)
postconditions range-type(.output) = range-type(.input.op)
$\wedge$ first(.output,.input.seed)
$\wedge\ \forall ix$ (successor^{n}(*i*,generator>digraph(.input).input.seed,*x*)
↭ .output(oneplus(*i*),*x*))

IOspec **map** / .input(sequence) .op(function) ⇒ .output(sequence)
preconditions domain-type(.op) = range-type(.input)

postconditions range-type(.output) = range-type(.op)
∧ length(.input) = length(.output)
∧ ∀ *i* (index(.input,*i*) ⊃ apply(.output,*i*) = apply(.op,apply(.input,*i*)))

2.7 Segments.

DataOverlay **segment>sequence:** segment → sequence
properties ∀ *GS* segment>sequence(*G*,*S*) ⊃ length(*S*,difference(*G*.upper,*G*.lower))
definition ∀ *GS* segment>sequence(*G*,*S*) ⇋
∀ *ix* (*S*(*i*,*x*) ⇋ *G*.base(plus(*G*.lower,*i*),*x*) ∧ le(*i*,difference(*G*.upper,*G*.lower)))

DataPlan **segment**
roles .base(sequence) .lower(natural) .upper(natural)
constraints index(.base,.lower)
∧ index(.base,.upper) ∧ le(.lower,.upper)

DataPlan **upper-segment** *specialization* segment
roles .base(sequence) .lower(natural) .upper(natural)
constraints length(.base,.upper)

DataPlan **lower-segment** *specialization* segment
roles .base(sequence) .lower(natural) .upper(natural)
constraints .lower = 1

2.8 Binary Relations as Predicates.

DataOverlay **binrel+two>predicate:** binrel+two → predicate
definition ∀ *BP* binrel+two>predicate(*B*,*P*) ⇋ ∀ *x* (*P*(*x*) ⇋ apply(*B*.op,*B*.two) = *x*)

DataPlan **binrel+two**
roles .op(binrel) .two(object)
constraints instance(range-type(.op),.two)

DataOverlay **@binrel>predicate:** @binrel-composite → @predicate
correspondences
binrel+two>predicate(@binrel-composite.composite) = @predicate.criterion
∧ @binrel-composite.if.one = @predicate.input
∧ @binrel-composite.if.in = @predicate.in
∧ @binrel-composite.if.succeed = @predicate.succeed
∧ @binrel-composite.if.fail = @predicate.fail

TemporalPlan **@binrel-composite**
roles .composite(binrel+two) .if(@binrel)

constraints .composite.op = .if.criterion ∧ .composite.two = .if.two

DataOverlay **integer>predicate:** integer → predicate
definition ∀*Pi* integer>predicate(*P*,*i*) ⇋ ∀*j* (*P*(*j*) ⇋ *j* = *i*)
properties ∀*Pi* integer>predicate(*P*,*i*) ⊃
∃*B* (instance(binrel+two,*B*) ∧ *B*.op = eq ∧ *B*.two = *i*
∧ binrel+two>predicate(*B*,*P*))

2.9 Functions as Predicates.

DataPlan **function+two** *specialization* binrel+two
roles .op(function) .two(object)
constraints member(range(.op),.two)

TemporalOverlay **@function+equal>test:** @function+equal → @predicate
correspondences binrel+two>predicate(@function+equal.composite) = @predicate.criterion
∧ @function+equal.action.input = @predicate.input
∧ @function+equal.action.in = @predicate.in
∧ @function+equal.if.succeed = @predicate.succeed
∧ @function+equal.if.fail = @predicate.fail

TemporalPlan **@function+equal**
roles .composite(function+two) .action(@function) .if(@binrel)
constraints .composite.op = .action.op ∧ .composite.two = .if.two
∧ .action.output = .if.one ∧ cflow(.action.out,.if.in)
∧ .if.criterion = range-type(.action.op).equal

2.10 Function and Predicate Composites.

DataOverlay **function+predicate>predicate:** function+predicate → predicate
definition $\forall PB$ function+predicate>predicate(P,B) ⇋
$\forall x$ (P(x) ⇋ apply(B.criterion,apply(B.op,x)) = true

DataPlan **function+predicate**
roles .op(function) .criterion(predicate)
constraints range-type(.op) = argtype(.criterion)

TemporalOverlay **@function+predicate>predicate:** @function+predicate → @predicate
correspondences
function+predicate>predicate(@function+predicate.composite) = @predicate.criterion
∧ @function+predicate.action.input = @predicate.input
∧ @function+predicate.action.in = @predicate.in
∧ @function+predicate.if.succeed = @predicate.succeed
∧ @function+predicate.if.fail = @predicate+.fail

TemporalPlan **@function+predicate**
roles .composite(function+predicate) .action(@function) .if(@predicate)
constraints .composite.op = .action.op ∧ .composite.criterion = .if.criterion
∧ .action.output = .if.input ∧ cflow(.action.out,.if.in)

2.11 Complementary Predicates.

DataOverlay **complement:** predicate → predicate
properties symmetric(complement) ∧ instance(bijection,complement)
definition $\forall PQ$ complement(P,Q) ⇋ $\forall x$ ($P(x)$ ⇋ $\neg Q(x)$)

TemporalOverlay **@predicate>complement:** @predicate → @predicate
properties instance(bijection,@predicate>complement)
definition $\forall ST$ @predicate>complement(S,T) ⇋
S.criterion = complement(T.criterion)
∧ S.input = T.input ∧ S.in = T.in
∧ S.succeed = T.fail ∧ S.fail = T.succeed

2.12 Choice Functions.

Type **binchoice** *subtype* algebraic-binfunction
components .instances = binchoices .equal = trirel-equal
definition $\forall F$ instance(binchoice,F) $\leftrightharpoons$ instance(binfunction,F)
$\wedge \forall xy\,(F(x,y,x) \vee F(x,y,y))$

DataOverlay **binrel>binchoice:** binrel → binchoice
properties $\forall RF$ (binrel>binchoice(R,F) $\wedge$ instance(partial-order,R))
$\supset$ (instance(aggregative-binfunction,F) $\wedge \forall x$ (identity(F,x) $\leftrightharpoons$ bottom(R,x)))
definition $\forall RF$ binrel>binchoice(R,F) $\leftrightharpoons \forall xy\,(R(x,y) \supset F(x,y,y))$

TemporalOverlay **@binrel>choice:** @binrel+join → @choice
correspondences
binrel>binchoice(@binrel+join.if.criterion) = @choice.binop
$\wedge$ @binrel+join.end.output = @choice.output
$\wedge$ @binrel+join.if.in = @choice.in
$\wedge$ @binrel+join.end.out = @choice.out

IOspec **@choice** / .binop(binchoice) .one(object) .two(object) ⇒ .output(object)
specialization @binfunction

TemporalPlan **@binrel+join** *specialization* cond
roles .if(@binrel) .then(in+out) .else(in+out) .end(join-outputs)
constraints .if.two = .end.succeed-input
$\wedge$ .if.one = .end.fail-input

3. LISTS

Type **list+nil** *uniontype* list nil

IOspec **push** / .old(list+nil) .input(object) ⇒ .new(list)
specialization old+input+new
postconditions head(.new,.input) $\wedge$ tail(.new,.old)
$\wedge$ oneplus(length(.old),length(.new))

IOspec **pop** / .old(list) ⇒ .new(list+nil) .output(object)
postconditions head(.old,.output) $\wedge$ tail(.old,.new)

3.1 Upper Segment as List.

DataOverlay **upper-segment>list:** upper-segment → list+nil
definition ∀ *GL* upper-segment>list(*G*,*L*) ⇋
(instance(list,*L*) ⊃
∧ *G*.base(*G*.lower) = *L*.head
∧ ∃*H* (instance(upper-segment,*H*)
∧ *G*.base = *H*.base ∧ oneplus(*G*.lower,*H*.lower)
∧ upper-segment>list(*H*)) = *L*.tail)
∧ (length(*G*.base,*G*.lower) ⇋ *L* = nil)

TemporalOverlay **bump+update>push:** bump+update → push
correspondences upper-segment>list(bump+update.old) = push.old
∧ bump+update.update.input = push.input
∧ upper-segment>list(bump+update.new) = push.new

TemporalPlan **bump+update**
roles .bump(@oneminus) .update(newterm) .old(upper-segment) .new(upper-segment)
constraints cflow(.bump.after,.update.before)
∧ .old.lower = .bump.input
∧ .bump.output = .update.arg
∧ update.old = .old.base ∧ .update.new = .new.base
∧ .new.lower = .bump.output

TemporalOverlay **fetch+bump>pop:** fetch+bump → pop
correspondences upper-segment>list(fetch+bump.old) = pop.old
∧ upper-segment>list(fetch+bump.new) = pop.new
∧ fetch+bump.fetch.output = pop.output

4. DIRECTED GRAPHS

DataPlan **digraph**
roles .nodes(set) .edge(binrel)

DataPlan **tree** *specialization* digraph
properties $\forall G$ instance(tree,G) $\supset \forall xy$ (root(G,x) $\wedge$ root(G,y) $\supset x=y$)
roles .nodes(set) .edge(binrel)
definition $\forall G$ instance(tree,G) $\leftrightharpoons$ (instance(digraph,G)
$\wedge \exists x$ root(G,x) $\wedge \forall x$ ($\neg$successor*(G,x,x)))

DataPlan **bintree** *specialization* tree
roles .nodes(set) .edge(binrel)
definition $\forall T$ instance(bintree,T) $\leftrightharpoons$ (instance(tree,T)
$\wedge \forall x$ (node(T,x) $\wedge \neg$terminal(T,x) $\supset$ size(successors(T,x),2)))

DataPlan **thread** *specialization* tree
properties $\forall T$ instance(thread,T) $\supset$ ($\forall xy$ (terminal(T,x) $\wedge$ terminal(T,y) $\supset x=y$)
$\wedge \forall xyz$ (successor(T,x,y) $\wedge$ successor(T,z,y) $\supset y=z$))
roles .nodes(set) .edge(function)
definition $\forall T$ instance(thread,T) $\leftrightharpoons$ (instance(tree,T)
$\wedge \forall xyz$ (successor(T,x,y) $\wedge$ successor(T,x,z) $\supset y=z$))

4.1 Relations on Directed Graphs.

Binrel **node**: digraph $\times$ object $\rightarrow$ boolean
definition $\forall Gx$ node(G,x) $\leftrightharpoons$ member(G.nodes,x)

Trifunction **successor**: digraph $\times$ object $\times$ object $\rightarrow$ boolean
definition $\forall Gxy$ successor(G,x,y) $\leftrightharpoons$ (node(G,x) $\wedge$ node(G,y) $\wedge$ G.edge(x,y))

Binfunction **successors**: digraph $\times$ object $\rightarrow$ set
definition $\forall GSx$ successors(G,x,S) $\leftrightharpoons \forall y$ (member(S,y) $\leftrightharpoons$ successor(G,x,y))

Trifunction **successor***: digraph $\times$ object $\times$ object $\rightarrow$ boolean
definition $\forall Gxy$ (successor*(G,x,y) $\leftrightharpoons \exists i$ (successor^n(i,G,x,y)

Trifunction **successor**^{n}: natural $\times$ digraph $\times$ object $\times$ object $\rightarrow$ boolean
definition $\forall Gxyi$ successor^n(i,G,x,y) $\leftrightharpoons$
(($i=1 \wedge$ successor(G,x,y))
$\vee \exists z$ (successor(x,z) $\wedge$ successor^n(oneminus(i),G,z,y)))

Binrel **root**: digraph $\times$ object $\rightarrow$ boolean
definition $\forall Gx$ root(G,x) $\leftrightharpoons \forall y$ ((node(G,y) $\wedge x \neq y$) $\supset$ successor*(G,x,y))

Binrel **terminal:** digraph × object → boolean
definition $\forall Gx$ terminal(G,x) ⇋ (node(G,x) ∧ ¬∃y successor(G,x,y))

Binrel **digraph-equal:** digraph × digraph → boolean
properties instance(equivalence,digraph-equal)
definition $\forall GH$ digraph-equal(G,H) ⇋ (G.nodes = H.nodes
∧ $\forall xy$ (successor(G,x,y) ⇋ successor(H,x,y))

Binrel **subgraph:** digraph × digraph → boolean
properties instance(partial-order,subgraph)
definition $\forall GH$ subgraph(G,H) ⇋
($\forall xy$ (successor(G,x,y) ⊃ successor(H,x,y))
∧ $\forall xy$ (node(G,x) ∧ node(G,y) ∧ successor(H,x,y)
⊃ successor(G,x,y)))

4.2 Generators.

DataPlan **generator**
roles .seed(object) .op(binrel)
constraints instance(domain-type(.op),.seed) ∧ domain-type(.op) = range-type(.op)

DataPlan **iterator** *specialization* generator
roles .seed(object) .op(function)

DataOverlay **generator>digraph:** generator → digraph
properties $\forall RG$ generator>digraph(R,G) ⊃ root(G,R.seed)
correspondences generator.op = digraph.edge
∧ transitive-closure(generator) = digraph.nodes

Function **transitive-closure:** generator → set
definition $\forall RS$ transitive-closure(R,S) ⇋ (member(S,R.seed)
∧ $\forall yz$ (member(S,y) ∧ apply(R.op,y) = z ⊃ member(S,z))

Datastructure **natural-iterator** *instance* iterator
components .seed = 1 .op = oneplus

Assert generator>digraph(natural-iterator,natural-thread)

Datastructure **natural-thread** *instance* thread
components .nodes = naturals .edge = oneplus

TemporalOverlay **temporal-iterator:** iterative-generation → iterator
correspondences iterative-generation.action.input = iterator.seed
∧ iterative-generation.action.op = iterator.op

4.3 Binary Generators.

DataPlan **binary-generator**
roles .seed(object) .left(function) .right(function)

DataPlan **car-cdr-generator** *specialization* binary-generator
roles .seed(dotted-pair) .left(function) .right(function)
constraints .left = car ∧ .right = cdr

DataOverlay **binary>generator:** binary-generator → generator
properties ∀ *BG* binary>generator(*B*,*G*) ⊃
(instance(tree,generator>digraph(*G*)) ⊃ instance(bintree,generator>digraph(*G*)))
correspondences binary-generator.seed = generator.seed
∧ binrel-union(binary-generator.left,binary-generator.right) = generator.op

Binfunction **binrel-union:** binrel × binrel → binrel
definition ∀ *RST* binrel-union(*R*,*S*,*T*) ⇋ ∀ *xy* (*T*(*x*,*y*) ⇋ (*R*(*x*,*y*) ∨ *S*(*x*,*y*)))

4.4 Truncated Directed Graphs.

DataPlan **truncated-digraph**
roles .base(digraph) .criterion(predicate)
constraints ∀ *x* (node(.base,*x*) ⊃ (apply(.criterion,*x*) = true
∨ ∃*y* (successor*(.base,*x*,*y*) ∧ apply(.criterion,*y*) = true)
∨ ∃*y* (successor*(.base,*y*,*x*) ∧ apply(.criterion,*y*) = true)))

DataPlan **truncated-tree** *specialization* truncated-digraph
roles .base(tree) .criterion(predicate)

DataPlan **truncated-thread** *specialization* truncated-tree
properties ∀ *T* instance(truncated-thread,*T*) ⊃ (apply(*T*.criterion,root(*T*.base) = true)
∨ ∃*x* (successor*(*T*.base,root(*T*.base),*x*) ∧ apply(*T*.criterion,*x*) = true))
roles .base(thread) .criterion(predicate)

4.5 Finite Subgraphs.

DataOverlay **truncated>digraph:** truncated-digraph → finite-digraph
properties $\forall TF$ truncated>digraph-inclusive$(T,F) \supset$
$(\text{instance}(\text{thread},T.\text{base}) \leftrightharpoons \text{instance}(\text{thread},F))$
$\wedge (\text{instance}(\text{tree},T.\text{base}) \leftrightharpoons \text{instance}(\text{tree},F))$
definition $\forall GT$ truncated>digraph$(T,F) \leftrightharpoons$ (subdigraph(F,T)
$\wedge \forall x$ (node$(F,x) \leftrightharpoons$ (node(T,x)
$\wedge \exists y$ (successor$^{*}(T,x,y) \wedge$ apply$(T.\text{criterion},y) = \text{true})$
$\wedge \neg\exists z$ (successor$^{*}(T,z,x) \wedge$ apply$(T.\text{criterion},z) = \text{true})))$

DataOverlay **truncated>digraph-inclusive:** truncated-digraph → finite-digraph
properties $\forall TF$ truncated>digraph-inclusive$(T,F) \supset$
$(\text{instance}(\text{thread},T.\text{base}) \leftrightharpoons \text{instance}(\text{thread},F))$
$\wedge (\text{instance}(\text{tree},T.\text{base}) \leftrightharpoons \text{instance}(\text{tree},F))$
definition $\forall GT$ truncated>digraph-inclusive$(T,F) \leftrightharpoons$ (subdigraph(F,T)
$\wedge \forall x$ (node$(F,x) \leftrightharpoons$ (node(truncated>digraph$(T),x)$
$\vee \exists y$ (node(truncated>digraph$(T),y)$
$\wedge$ successor$(T,y,x) \wedge$ apply$(T.\text{criterion},x) = \text{true})))$

DataPlan **finite-digraph** *specialization* digraph
roles .nodes(finite-set) .edge(binrel)

4.6 Internal Thread Remove.

IOspec **digraph-remove** / .old(digraph) .input(object) ⇒ .new(digraph)
postconditions ¬member(.new.nodes,.input)
$\wedge \forall xy$ ($x \neq$ .input $\wedge$ $y \neq$ .input $\supset$ (successor(.new,x,y) $\leftrightharpoons$ successor(.old,x,y)))
$\wedge \forall x$ (successor(.old,x,.input)
$\supset \forall y$ (successor(.new,x,y) $\leftrightharpoons$ successor(.old,.input,y)))

IOspec **internal-thread-remove** / .old(thread) .input(object) ⇒ .new(thread)
specialization digraph-remove old+input+new
preconditions ¬root(.old,.input)

IOspec **#internal-thread-remove** / .old(thread) .input(object) ⇒ .new(thread)
specialization internal-thread-remove #old+input+new

4.7 Internal Thread Add.

IOspec **digraph-add** / .old(digraph) .input(object) ⇒ .new(digraph)
postconditions member(.new.nodes,.input)
∧ ∀*xy* (*x*≠.input ∧ *y*≠.input ⊃ (successor(.new,*x*,*y*) ⇋ successor(.old,*x*,*y*)))

IOspec **internal-thread-add** / .old(thread) .input(object) ⇒ .new(thread)
specialization digraph-add old+input+new
postconditions ¬root(.new,.input)
∧ ∀*x* (successor(.new,*x*,.input)
⊃ ∀*y* (successor(.old,*x*,*y*) ⇋ successor(.new,.input,*y*))

IOspec **#internal-thread-add** / .old(thread) .input(object) ⇒ .new(thread)
specialization internal-thread-add #old+input+new

4.8 Internal Thread Find.

IOspec **digraph-find** / .universe(digraph) .criterion(predicate) ⇒ .output(object)
preconditions ∃*x* (node(.universe,*x*) ∧ apply(.criterion,*x*) = true)
postconditions node(.universe,.output) ∧ apply(.criterion,.output) = true

IOspec **internal-thread-find** / .universe(thread) .criterion(predicate)
⇒ .output(object) .previous(object)
extension digraph-find
preconditions apply(.criterion,root(.universe)) = false
postconditions successor(.universe,.previous,.output)

TemporalPlan **internal-thread-find+remove**
roles .find(internal-thread-find) .remove(internal-thread-remove)
constraints .find.universe = .remove.old
∧ .find.output = .remove.input

4.9 Trailing Plans.

TemporalPlan **trailing** *extension* single-recursion
roles .current(object) .previous(object) .tail(trailing)
constraints .current = .tail.previous

TemporalPlan **trailing-search** *extension* trailing iterative-search
roles .current(object) .previous(object) .exit(cond) .tail(trailing-search)
constraints instance(join-two-outputs,.exit.end)
.current = .exit.if.input ∧ .previous = .exit.end.succeed-input-two
∧ .tail.exit.end.output-two = .exit.end.fail-input-two

4.10 Trailing Generation and Search.

TemporalPlan **trailing-generation+search** *extension* iterative-generation trailing-search
roles .current(object) .previous(object) .exit(cond) .action(@function)
.tail(trailing-generation+search)
constraints .current = .action.output ∧ .previous = .action.input

TemporalOverlay **trailing-generation+search>find :** trailing-iteration+search → internal-thread-find
correspondences
generator>digraph(temporal-iterator(trailing-generation+search)
= internal-thread-find.universe
trailing-generation+search.exit.if.criterion = internal-thread-find.criterion
trailing-generation+search.exit.end.output = internal-thread-find.output
trailing-generation+search.exit.end.two = internal-thread-find.previous
trailing-generation+search.action.in = internal-thread-find.in
trailing-generation+search.exit.out = internal-thread-find.out

4.11 Splicing Out.

TemporalPlan **spliceout**
roles .old(iterator) .new(iterator) .bump(@function) .splice(newarg)
constraints .old.op = .bump.op ∧ .new.op = .splice.out ∧ .old.seed = .new.seed
∧ .bump.output = .splice.input
∧ successor(generator>digraph(.old),.splice.arg,.bump.input)

TemporalPlan #**spliceout** *specialization* spliceout
roles .old(iterator) .new(iterator) .bump(@function) .splice(#newarg)

TemporalOverlay **spliceout>remove:** spliceout → internal-thread-remove
properties ∀*SR* spliceout>remove(*S*,*R*) ⊃
(instance(#spliceout,*S*) ⇋ instance(#internal-thread-remove,*R*))
correspondences
generator>digraph(spliceout.old) = internal-thread-remove.old
∧ spliceout.bump.input = internal-thread-remove.input
∧ generator>digraph(spliceout.new) = internal-thread-remove.new
∧ spliceout.bump.in = internal-thread-remove.in
∧ spliceout.splice.out = internal-thread.remove.out

4.12 Splicing In.

TemporalPlan **splicein**
roles .old(iterator) .new(iterator) .one(newarg) .two(newarg)
constraints .one.arg = .two.input ∧ one.new = .two.old
∧ successor(generator>digraph(.old),.two.arg,.one.input)

TemporalPlan #**splicein** *specialization* splicein
roles .old(iterator) .new(iterator) .one(#newarg) .two(#newarg)

TemporalOverlay **splicein>add:** splicein → internal-thread-add
properties ∀*SA* splicein>add(*S*,*A*) ⊃
(instance(#splicein,*S*) ⇋ instance(#internal-thread-add,*A*))
correspondences
generator>digraph(splicein.old) = internal-thread-add.old
∧ splicein.one.arg = internal-thread-add.input
∧ generator>digraph(splicein.new) = internal-thread-add.new
∧ find+splicein.one.in = internal-thread-add.in
∧ find+splicein.two.two.out = internal-thread-add.out

4.13 Labelled Directed Graphs.

DataPlan **labelled-digraph**
roles .spine(digraph) .label(function)
constraints subset(.spine.nodes,domain(.label))

DataPlan **labelled-thread** *specialization* labelled-digraph
roles .spine(thread) .label(function)

DataPlan **cdr-thread+car** *specialization* labelled-thread
properties $\forall P$ instance(cdr-thread+car,P)
$\supset$ list>labelled-thread(dotted-pair>list(P.spine.seed),P)
roles .spine(cdr-thread) .label(function)
constraints .label = car

DataOverlay **list>labelled-thread:** list $\rightarrow$ labelled-thread
definition $\forall TL$ list>labelled-thread(L,T) $\leftrightharpoons$
($\forall x$ (($x = L \vee \text{tail}^*(L,x)$) $\leftrightharpoons$ member(T.spine.nodes,x))
$\wedge$ tail = T.spine.edge $\wedge$ head = T.label

4.14 Trees as Partial Orders.

DataOverlay **tree>order:** tree $\rightarrow$ partial-order+bottom
properties $\forall TR$ tree>order(T,R) $\supset$ (root(T) = bottom(R)
$\wedge$ (instance(thread,T) $\leftrightharpoons$ instance(total-order,R)))
definition $\forall TR$ tree>order(T,R) $\leftrightharpoons$ $\forall xy$ (($x = y \vee \text{successor}^*(T,x,y)$) $\supset$ $R(x,y)$)

Type **partial-order+bottom** *subtype* partial-order
definition $\forall R$ instance(partial-order+bottom,R) $\leftrightharpoons$ (instance(partial-order,R)
$\wedge$ $\exists x$ bottom(R,x)

4.15 Intervals.

DataPlan **interval**
roles .base(total-order) .lower(object) .upper(object)
constraints .base(.lower,.upper)

DataOverlay **interval>truncated-thread:** interval → truncated-thread
properties $\forall IT$ interval>truncated-thread(I,T) ⊃
I.lower = bottom(tree>order(truncated>digraph(T)))
∧ I.upper = top(tree>order(truncated>digraph-inclusive(T)))
correspondences interval.lower = root(truncated-thread.base)
∧ interval.base = tree>order(truncated-thread.base)
∧ integer>predicate(interval.upper) = truncated-thread.criterion

5. LINEAR STRUCTURES

DataOverlay **list>sequence:** list+nil → sequence
properties instance(bijection,list>sequence)
∧ $\forall LS$ list>sequence(L,S) ⊃ (length(L) = length(S)
∧ (instance(irredundant-list,L) ⇋ instance(irredundant-sequence,S)))
definition $\forall LS$ list>sequence(L,S) ⇋
((L = nil ⇋ length(S,0))
∧ $\forall x$ (L.head = x ⇋ first(S,x))
∧ $\forall ix$ (($\exists M$ (tail^n(i,L,M) ∧ M.head = x)) ⇋ S(oneplus(i),x)))

DataOverlay **sequence>labelled-thread:** finite-sequence → labelled-truncated-natural-thread
properties instance(bijection,sequence>labelled-thread)
definition $\forall SL$ sequence>labelled-thread(S,L) ⇋ L.label = S
∧ $\exists T$ (truncated>digraph-inclusive(T,L.spine)
∧ T.criterion = integer>predicate(length(S)))

DataPlan **labelled-truncated-natural-thread** *specialization* labelled-thread
roles .spine(thread) .label(function)
constraints $\exists T$ (instance(truncated-thread,T) ∧ T.base = natural-thread
∧ .spine = truncated>digraph-inclusive(T))

Type **irredundant-sequence** *subtype* bijection
definition $\forall S$ instance(irredundant-sequence,S) ⇋ (instance(sequence,S)
∧ $\forall ijxy$ (S(i,x) ∧ S(j,y) ∧ $i \neq j$ ⊃ $x \neq y$))

DataPlan **irredundant-list** *specialization* list
definition $\forall L$ instance(irredundant-list,L) ⇋ (instance(list,L)
∧ $\forall Mx$ (tail*(L,M) ∧ head(M,x) ⊃ $x \neq L$.head)
∧ instance(irredundant-list,L.tail)

DataOverlay **sequence>thread:** irredundant-sequence → thread
properties instance(bijection,sequence>thread)
$\wedge\ \forall ST$ sequence>thread(S,T) $\supset$ length(S) = size(T.nodes
$\wedge\ \forall x$ (last(S,x) $\leftrightharpoons$ terminal(T,x))
definition $\forall ST$ sequence>thread(S,T) $\leftrightharpoons$ $\forall x$ (S(1,x) $\leftrightharpoons$ root(T,x))
$\wedge\ \forall xy$ ($\exists i$ (S(i,x) $\wedge$ S(oneplus(i),y)) $\leftrightharpoons$ successor(T,x,y))

DataOverlay **list>thread:** irredundant-list → thread
properties instance(bijection,list>thread)
definition $\forall LT$ list>thread(L,T) $\leftrightharpoons$ $\forall x$ (L.head = x $\leftrightharpoons$ root(T,x))
$\wedge\ \forall ix$ ($\exists M$ (tail^{n}(i,L,M) $\wedge$ M.head = x) $\leftrightharpoons$ successor^{n}(i,T,root(T),x))

6. MISCELLANY

6.1 Flags.

DataPlan **flag**
roles .arg(object) .criterion(predicate)
constraints instance(argtype(.criterion),.arg)

TemporalPlan **enflag+output**
roles .enflag(cond) .output(flag)
constraints instance(join-outputs,.enflag.end)
$\wedge$ .output.arg = .enflag.end.output
$\wedge$ apply(.output.criterion,.output.arg$_{\text{.enflag.end.succeed}}$) = true
$\wedge$ apply(.output.criterion,.output.arg$_{\text{.enflag.end.fail}}$) = false

TemporalPlan **enflag+deflag** *extension* enflag+output
roles .enflag(cond) .output(flag) .deflag(@predicate)
constraints .deflag.criterion = .enflag.output.criterion
$\wedge$ .enflag.end.output = .deflag.input
$\wedge$ cflow(enflag.end.out,.deflag.in)

TemporalOverlay **enflag+deflag>test:** enflag+deflag → test
correspondences enflag+deflag.enflag.if.in = test.in
$\wedge$ enflag+deflag.deflag.succeed = test.succeed
$\wedge$ enflag+deflag.deflag.fail = test.fail

BIBLIOGRAPHY

[1] W.S. Amey, "Computer Assisted Software Engineering (CASE) System", *4th Int. Conf. on Software Eng.*, Munich, Germany, Sept. 1979.

[2] P. Asirelli *et al.*, "A Flexible Environment for Program Development Based on a Symbolic Interpreter", *4th Int. Conf. on Software Eng.*, Munich, Germany, Sept. 1979.

[3] D.R. Barstow, "Automatic Construction of Algorithms and Data Structures Using A Knowledge Base of Programming Rules", Stanford Artificial Intelligence Laboratory Memo AIM-308, Nov. 1977.

[4] J.L. Bentley and M. Shaw, "An Alphard Specification of a Correct and Efficient Transformation on Data Structures", in *Proc. Specifications of Reliable Software*, 1979, pp. 222-233.

[5] R.M. Burstall, "Some Techniques for Proving Properties of Programs Which Alter Data Structures, *Machine Intelligence 7*, Edinburgh University Press.

[6] R.M. Burstall and J.A. Goguen, "Putting Theories Together to Make Specifications", in *Proc. of 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, pp.1045-1058.

[7] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 4, July 1979, pp. 402-417.

[8] T.E. Cheatham, J.A. Townley, G.H. Holloway, "A System for Program Refinement", *4th Int. Conf. on Software Eng.*, Munich, Germany, Sept. 1979, pp.53-62.

[9] O.J. Dahl and K. Nygaard "SIMULA - An ALGOL-Based Simulation Language", *Comm. of the ACM*, Vol. 9, No. 9, September 1966, pp. 671-678.

[10] O.J. Dahl, B. Myrhaug, and K. Nygaard, "SIMULA 67 -- Common Base Language", Norwegian Computing Center, Oslo, Norway, 1968.

[11] O.J. Dahl, E. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

[12] J.B. Dennis, "First Version of a Data Flow Procedure Language", *Proc. of Symposium on Programming*, Institut de Programmation, U. of Paris, April 1974, pp. 241-271.

[13] N. Dershowitz and Z. Manna, "The Evolution of Programs: Automatic Program Modification", *IEEE Trans. on Software Eng.*, Vol. SE-3, No. 6, November 1977, pp. 377-385.

[14] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J.J. Levy, "A structure Oriented Program Editor: A First Step Towards Computer Assisted Programming", Res. Rep. 114, IRIA, Pris, April 1975.

[15] D.M. Dungan, "Bibliography on Data Types", *Sigplan Notices*, Vol. 14, No. 11, November 1979, pp. 31-59.

[16] R.S. Eares, C.K. Hitchon, R.M. Thall, and J.W. Brackett, "An Environment for Producing Well-Engineered Microcomputer Software", *4th Int. Conf. on Software Eng.*, Munich, Germany, Sept. 1979.

[17] J. Earley, "Toward an Understanding of Data Structures", *Comm. of the ACM*, Vol. 14, No. 10, October 1971, pp. 617-627.

[18] J. Earley, "Relational Level Data Structures for Programming Languages", Computer Science Dept., U. of California, Berkeley, March 1973.

[19] L. Flon and J. Misra, "A Unified Approach to the Specificaton and Verification of Abstract Data Types", in *Proc. Specifications of Reliable Software*, 1979, pp. 162-169.

[20] R.W. Floyd, "Assigning Meaning to Programs", in *Mathematical Aspects of Computer Science*, J.T. Schwartz (ed.), Vol. 19, Am. Math. Soc., Providence Rhode Island, 1967, pp. 19-32.

[21] S.L. Gerhart, "Knowledge About Programs: A Model and Case Study", in *Proc. of Int. Conf. on Reliable Software*, June 1975, pp. 88-95.

[22] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology, Vol. IV*, (ed. Raymond Yeh), Prentice-Hall, 1978.

[23] C. Green and D. Barstow, "On Program Synthesis Knowledge", *Artificial Intelligence*, Vol. 10, No. 3, November 1978, pp. 241-281.

[24] D. Gries and N. Gehani, "Some Ideas on Data Types in High-Level Languages", *Comm. of the ACM*, Vol. 20, No. 6, June 1977, pp. 414-420.

[25] J. Guttag, "Abstract Data Types and the Development of Data Structures", *Comm. of the ACM*, Vol. 20, No. 6, June 1977, pp. 396-404.

[26] J. Guttag and J.J. Horning, "Formal Specification As a Design Tool", *7th Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, January, 1980, pp. 251-261

[27] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence", in *Proc. of 3rd Int. Joint Conf. on Artificial Intelligence*, Stanford University, August 1973, pp 235-245.

[28] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Comm. of the ACM*, Vol. 12, No. 10, October 1969, pp. 576-580,583.

[29] C.A.R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica* 1,4, 1972, pp. 271-281.

[30] Y.I. Ianov, "The Logical Schemes of Algorithms," in *Problems of Cybernetics*, Vol. 1, Pergamon Press, New York (English Translation), pp. 82-140.

[31] E. Kant, "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach", Stanford Artificial Intelligence Laboratory Memo AIM-331, September 1979.

[32] D.E. Knuth, *The Art of Computer Programming*, Vol. 1,2,3, Addison-Wesley, 1968,1969,1973.

[33] B. Liskov *et. al.*, "Abstraction Mechanisms in CLU", *Comm. of the ACM*, Vol. 20, No. 8, August 1977, pp. 564-576.

[34] B.H. Liskov and S.N. Zilles, "an Introduction to Formal Specifications of Data Abstractions," *Current Trends in Programming Methodology, Vol. I*, (ed. Raymond Yeh), Prentice-Hall, 1977.

[35] R.L. London, M. Shaw, and W.A. Wulf, "Abstraction and Verification in Alphard: A Symbol Table Example", Carnegie-Mellon University and USC Info. Sciences Institute Technical Reports, 1976.

[36] J.R. Low, "Automatic Data Structure Selection: An Example and Overview", *Comm. of the ACM*, Vol. 21, No. 5, May 1978, pp. 376-384.

[37] Z. Manna, *Mathematic Theory of Computation*, McGraw-Hill, 1974.

[38] Z. Manna and R. Waldinger, "Knowledge and Reasoning in Program Synthesis", *Artificial Intelligence 6*, 1975, pp. 175-208.

[39] Z. Manna and R. Waldinger, "The Logic of Computer Programming", *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 3, May 1978, pp. 199-229.

[40] Z. Manna and R. Waldinger, "Synthesis: Dreams => Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 4, July 1979, pp. 294-327.

[41] R.E. Mayer, "A Psychology of Learning BASIC", *Comm. of the ACM*, Vol. 22, No. 11, November 1979, pp. 589-593.

[42] J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence 4*, American Elsevier, 1969.

[43] J. Misra, "An Approach to Formal Definitions and Proofs of Programming Principles", *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 5, September 1978, pp. 410-413.

[44] J. Misra, "Some Aspects of the Verification of Loop Computations", *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 6, November 1978, pp. 478-485.

[45] M.S. Moriconi, "A Designer/Verifier's Assistant", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 4, July 1979, pp. 387-401.

[46] J.C. Reynolds, "Reasoning About Arrays", *Comm. of the ACM*, Vol 22, No. 5, May 1979, pp. 290-298.

[47] C. Rich and H.E. Shrobe, "Initial Report On A LISP Programmer's Apprentice", MIT Artificial Intelligence Laboratory TR-354, December 1976.

[48] S.J. Rosenschein and S.M. Katz, "Selection of Representations for Data Structures", in *Proc. of Symp. on Artificial Intelligence and Programming Languages*, Rochester, N.Y., August 1977, pp. 147-154.

[49] P.D. Rovner, "Automatic Representation Selection for Associative Data Structures", Computer Science Dept., University of Rochester, TR10.

[50] E.D. Sacerdoti, "The Nonlinear Nature of Plans", in *Advance Papers of the 4th Int. Joint Conf. on Artificial Intelligence*, Tblisi, Georgia, USSR, September 1975, pp. 206-214.

[51] E. Schonberg, J.T. Schwartz and M. Sharir, "Automatic Data Structure Selection in SETL", *Proc. 6th POPL Conf.*, 1979, pp. 197-210.

[52] J.T. Schwartz, "On Programming", An Interim Report on the SETL Project, Courant Institute of Mathematical Sciences, New York University, June 1975.

[53] J.T. Schwartz, "Correct-Program Technology", in "Correct-Program Technology/Extensibility of Verifiers", Martin Davis and J.T. Schwartz, Computer Science Report #12, Courant Institute of Mathematical Sciences, New York University, September 1977.

[54] M. Shaw, "Abstraction and Verification in Alphard: Design and Verification of a Tree Handler", *Proc. 5th Texas Conf. on Computing Systems*, 1976, pp. 86-94.

[55] M. Shaw, W.A. Wulf, and R.L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators", *Comm. of the ACM*, Vol 20, No. 8, August 1977, pp. 553-563.

[56] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", MIT Artificial Intelligence Laboratory TR-503, April 1979.

[57] J.M. Spitzen, K.N. Leavitt, and L. Robinson, "An Examle of Hierarchical Design and Proof", *Comm. of the ACM*, Vol. 21, No. 12, December 1978, pp. 1064-1075.

[58] G.L. Steele and G.J. Sussman, "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two), MIT Artificial Intelligence Laboratory Memo No. 453, May 1978.

[59] W. Teitelman, "A Display Oriented Programmer's Assistant", *Proc. of 5th Int. Joint. Conf. on Artificial Intelligence*, Cambridge, Massachusetts, August 1977.

[60] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 237-247.

[61] R.R. Willis, E.P. Jensen, "Computer Aided Design of Software Systems", *4th Int. Conf. on Software Eng.*, Munich, Germany, Sept. 1979.

[62] N. Wirth, *Systematic Programming, An Introduction*, Prentice-Hall, 1973.

[63] W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Trans. on Software Eng.*, SE-2, No. 4, December 1976, pp. 253-265.

INDEX

#action+update 260
#expunge 270
#internal-thread-add 288
#internal-thread-remove 287
#newarg 259
#newterm 277
#newvalue 259
#old+input+new 260
#old+input+new>action+update 261,260
#old+new 259
#restrict 259
#set-add 259
#set-remove 259
#splicein 290
#spliceout 290
@binrel 50
@function 47
@oneminus 85
@predicate 50
@tail+internal>restrict 147
@aggregative 251
@binfunction 273
@binrel 273
@binrel+join 281
@binrel-composite 278
@binrel>choice 281
@binrel>predicate 278
@choice 281
@function 273
@function+equal 279
@function+equal>test 279
@function+newvalue 276
@function+predicate 280
@function+predicate>predicate 280
@predicate 273
@predicate>complement 280
@reverse 246
@tail+internal 267
@tail+internal>restrict 267
@transitive-closure-function 238
aggregate 50
aggregate-intersection 50
aggregate-union 50
aggregative-binfunction 50
any 51
any>retrieve 133
application-in-stream 219
application-out-stream 220
bijection 139
binfunction 45
binrel 50
bintree 55
bump+update 60
bump+update>push 86
car 79
car+cdr+null 112
cardinal 45
cascade-iterative-termination 214
cdr 79
composed-@functions 81,49
composed-functions 47,80
composed>@function 82
composed>function 80
cond 72
cotruncate 107
counting 63
digraph 55
digraph-add 58
digraph-remove 58
dim 80
discriminate+action+update 53
discriminate+action+update>action 138
discriminate+expunge+update>expunge 144
discriminate+retrieve>retrieve 119

discrimination 53
each 51
earliest 49
element 80
eq 80
expunge 53
expunge-one 142,53
fetch 96
fetch+update 60
fetch+update>pop 87
function 45
generation-stream 217
generator 57
hashing 49
in+out 71
integer 45
internal-labelled-thread-find+remove ... 148,149
internal-thread-add 58
internal-thread-find 58
internal-thread-remove 58
interval 58
irredundant-sequence 49
iterate 57
iterative-accumulation 212,64
iterative-aggregation 64
iterative-application 204,102,64
iterative-filtering 205,64
iterative-generation 202,63
iterative-list-accumulation 64
iterative-search 210,65
iterative-set-accumulation 64
iterative-termination 208,63
iterator 57
join 73
join-output 73
keyed-discriminate+expunge+update 54
keyed-discriminate+retrieve 54
keyed-discrimination 54
labelled-digraph 59
labelled-thread 59
length 49
list 60
map 50
max 50
member? 51
min 50
natural 45
newarg>newvalue 141
newterm 49
newvalue 47
newvalue-composed 49
newvalue-composite>newvalue 84
nil 60
null 80
object 45
old+new 47
partial-order 50
pop 60
predicate 50
product 50
push 60
restrict 51
restrict-one 145
restrict>expunge 146
retrieve 118,53
segment 60
segment. 76
segment>sequence 83
sequence>set 109
set 45
set-find 51
single-recursion 60
splicein 59
splicecut 153,59

spliceout>remove 154
sum 50
temporal-binary-generator 257
term 49
terminated-iterative-search 100
test 71
test>@choice 282
thread 55
trailing 65
trailing-generation+search 150,64
trailing-generation+search>find 151
trailing-search 159
tree 55
truncate 49
truncate-inclusive 49
truncated-digraph 57
truncated-list-generation 112
truncated-thread 57
upper-segment 60
upper-segment>list 85
accumulation-stream 246
aggregate 264
aggregate-intersection 264
aggregate-union 264
aggregative-binfunction 274
algebraic-binfunction 274
any 264
any-composite 270
any>retrieve 270
application-in-stream 218
application-out-stream 218
associative 274
atom 255
binary-generation 256
binary-generator 286
binary>generator 286
binchoice 281
binlist 256
binlist+atom 256
binrel+two 278
binrel+two>predicate 278
binrel-union 286
binrel>binchoice 281
bintree 284
bucket 268
bump+update 283
bump+update>push 283
butlast 276
car+cdr 225
car+cdr+null 237
car-cdr-generator 286
cascade-iterative-termination 213
cascade>iterative-termination 213
cdr-thread+car 291
co-earliest 233
commutative 274
complement 280
composed-@functions 275
composed-functions 275
composed>@function 275
composed>function 275
construct 258
cotermination-fail-stream 233
cotermination-in-stream 233
cotruncate 234
counting 203
digraph 284
digraph-add 288
digraph-equal 285
digraph-find 288
digraph-remove 287
discriminate+@member 268
discriminate+@member>@member 268
discriminate+action+update 269

discriminate+action+update>action 269
discriminate+expunge+update>expunge 272
discriminate+retrieve>retrieve 272
discrimination 268
discrimination>set 268
disjoint 263
double-recursion 255
double-recursion+atom 255
double-recursion>bintree 256
each 263
earliest 277
empty 263
enflag+deflag 293
enflag+deflag>test 293
enflag+output 293
expunge 270
expunge-one 270
fetch+bump>pop 283
filtering-in-stream 240
filtering-succeed-stream 240
finite-digraph 287
finite-list 265
finite-list+nil 265
finite-sequence 276
finite-set 263
finite-single-recursion 198
first 276
flag 293
function+predicate 280
function+predicate>predicate 280
function+two 279
generation-stream 216
generation>list 225
generator 285
generator>digraph 285
greater 274
hashing 271
identity 274
index 276
integer>predicate 279
internal-labelled-thread-add 266
internal-labelled-thread-find+remove 267
internal-thread-add 288
internal-thread-find 288
internal-thread-find+remove 288
internal-thread-remove 287
intersection 274
interval 292
interval>truncated-thread 292
irredundant-list 292
irredundant-list+nil 265
irredundant-list>set 265
irredundant-sequence 292
irredundant-sequence>set 265
iterate 277
iterate+truncate 230
iterate+truncate-inclusive 230
iterate+truncate-inclusive>truncated-thread 230
iterate+truncate>truncated-thread 230
iterative-accumulation 211,254
iterative-aggregation 251
iterative-application 201
iterative-cosearch 209
iterative-cosearch+nil 209
iterative-cotermination 206
iterative-cotermination+nil 206
iterative-filtering 201
iterative-generation 201
iterative-list-accumulation 246
iterative-list-accumulation>@reverse 246
iterative-search 209
iterative-search+nil 209
iterative-set-accumulation 248
iterative-steady-state 207

iterative-temporal-any 242
iterative-temporal-each 240
iterative-temporal-find 240
iterative-temporal-restrict 240
iterative-temporal-set-accumulation 248
iterative-temporal-transitive-closure 238
iterative-termination 206
iterative-termination+nil 206
iterative-termination-output 206
iterative-termination-output+nil 206
iterative-termination-predicate 206
iterative-termination-predicate+nil 206
iterative-termination>steady-state 207
iterator 285
keyed-discriminate+expunge+update 272
keyed-discriminate+retrieve 272
keyed-discrimination 271
labelled-digraph 291
labelled-thread 291
labelled-thread>set 266
labelled-truncated-natural-thread 292
last 276
length 276
lesser 275
linear-accumulation 254
list+nil 281
list-generation 225
list>labelled-thread 291
list>sequence 292
list>set 265
list>thread 293
lower-segment 278
map 277
max 264
min 265
natural-iterator 285
natural-thread 285
newarg 273
newarg-bijection 276
newarg>newvalue 276
newterm 277
newvalue 273
newvalue-composite 275
newvalue-composite>newvalue 275
node 284
old+input+new 260
old+input+new-set 269
old+new 259
partial-order+bottom 291
plus 274
pop 281
product 264
push 281
push>set-add 266
restrict 263
restrict-composite 271
restrict-one 267
restrict>expunge 271
retrieve 270
reverse-accumulation 254
reverse>iterative-accumulation 255
root 284
segment 278
segment>sequence 278
sequence 276
sequence>labelled-thread 292
sequence>set 265
sequence>thread 293
set 263
set-add 263
set-find 263
set-remove 264
single-recursion 198
single-recursion+nil 198

splicein 290
splicein>add 290
spliceout 290
spliceout>remove 290
steady-state-costream 233
steady-state-stream 227
subgraph 285
successor 284
successor* 284
successorⁿ 284
successors 284
sum 264
tail* 198
tailⁿ 198
temporal-aggregate 251
temporal-binary-generator 256
temporal-co-earliest 233
temporal-cotruncate 234
temporal-earliest 231
temporal-iterate 221
temporal-iterator 285
temporal-map 221
temporal-truncate 228
temporal-truncate-inclusive 228
term 277
terminal 285
terminated-iterative-search 242
termination-fail-stream 227
termination-in-stream 227
thread 284
times 274
trailing 289
trailing-generation+search 289
trailing-generation+search>find 289
trailing-search 289
transitive-closure 285
tree 284
tree>order 291
truncate 277
truncate-inclusive 277
truncated-digraph 286
truncated-generation>list 237
truncated-list-generation 237
truncated-thread 286
truncated-tree 286
truncated>digraph 287
truncated>digraph-inclusive 287
union 274
universal 263
upper-segment 278
upper-segment>list 283