# EXCEPTION HANDLING AND RECOVERY IN APPLICATIVE SYSTEMS

by

Wilfred Edmund Plouffe, Jr.

Technical Report 158

Department of Information and Computer Science
University of California
Irvine, CA 92717

## ACKNOWLEDGMENTS

I am deeply indebted to Professors Kim P. Gostelow and Arvind, my advisors, for their guidance and support. Without their help, I could never have finished this dissertation. Professor George S. Lueker provided many comments which improved the presentation of my ideas.

I would also like to thank Lubomir Bic, Dennis Kibler, Joseph Minne, James Neighbors, and Robert Thomas for many stimulating and entertaining discussions.

The National Science Foundation and the Department of Information and Computer Science provided financial support for my graduate education. I am thankful for this support.

Sections 2.2.1 through 2.2.7 are summarized from the article "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" by John Backus. This paper was originally published in the Communications of the ACM, Volume 21, Number 8 (August 1978), Copyright 1978, Association for Computing Machinery, Inc., reprinted by permission. This permission is gratefully acknowledged.

# Chapter 1

# INTRODUCTION

## 1.1 Purpose

Applicative models, and dataflow in particular, are interesting alternatives to the von Neumann model for designing new computer systems. The applicative models have several advantages over the von Neumann approach: they have a clean and simple semantic base [Bac73, Kos73, Den75, Kos76, ArGo77a, Bac78]; they have a high potential for distributed processing and parallelism [Cha71, GIMT74, Rum75, PCDGS76, Tre77, Bac78, CDGPS78, HOS78]; and they are well suited to LSI implementation [Den75, ArGo77b, Mag79a, 79b]. However, applicative models also have several problems that must be solved before they are really viable alternatives to the von Neumann model. One such problem is exception handling and recovery.

This dissertation presents an exception handling and recovery model that is consistent with an applicative semantic base. The model is based upon the following principles:

1. the preservation of the semantic base of the applicative system, even in the presence of exception conditions;

2. the definition of an "error" value*;

3. the avoidance of non-terminating computations caused by the presence of exception conditions; and

4. the definition of distinguishing syntax for exception handling so that exception handling is distinct from the "normal" program flow.

It is shown through the use of examples that this model for applicative systems is comparable to current exception handling and recovery models for a von Neumann system.

## 1.2 Background

The class of exceptions, or exception conditions, of interest are those events caused by a program in execution. Events caused by external influence, such as a power fluctuation or a programmer trying to abort his program, are not considered. Some of the uses for exceptions include:

1. dealing with an operation's actual or impending failure (i.e., a domain or range failure);

2. indicating the significance of a valid result or the circumstances under which the result was obtained (e.g., a status variable or return code); or

3. monitoring an operation or measuring computational progress.

--------------------

*Although the type would be better described as "exception", the choice of "error" was made to agree with common usage. In fact, an error value may be used to represent an event that is not at all unusual (see the example in Section 1.2.3).

Note that "errors" (e.g., overflow or subscript out of bounds) are simply one type of exception. Even the label "exception" is misleading since these events may occur as part of "normal" programming, and thus may not represent an unusual situation at all, e.g., an end_of_file.

Current exception handling and recovery models assume a von Neumann semantic base. The von Neumann model has a single locus of control (usually represented by a program counter) and defines operations that execute on memory cells. This single locus of control restricts the potential for parallel execution, and reliance upon memory operations limits the possibilities for distributed execution. Thus, current exception models are inappropriate for an applicative system. The sections that follow present a brief overview of current models.

### 1.2.1 Terminology

This section introduces terminology common to exception handling and recovery models. A process (program in execution) is modelled as a directed graph with each operation execution modelled as a transition from one internal state of the process to another internal state (see Figure 1.1). Note that the von Neumann semantic model restricts the process to be represented by a single state (only one site of activity, the program counter, and a

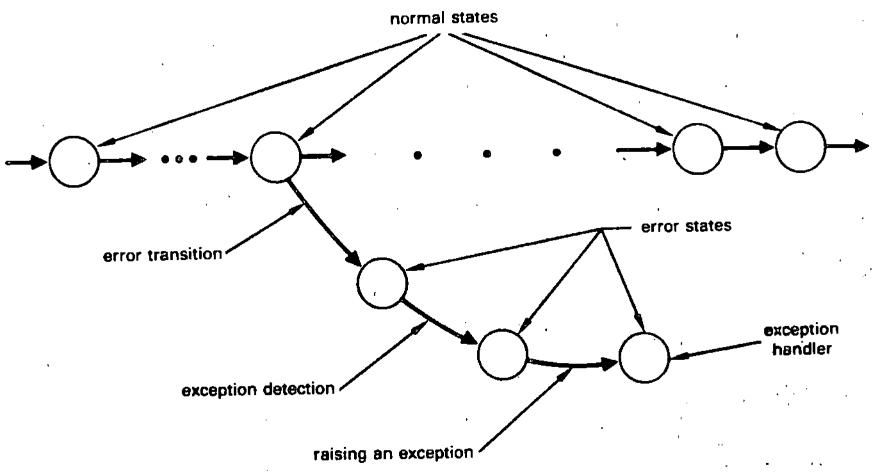

Figure 1.1a State Transition Model

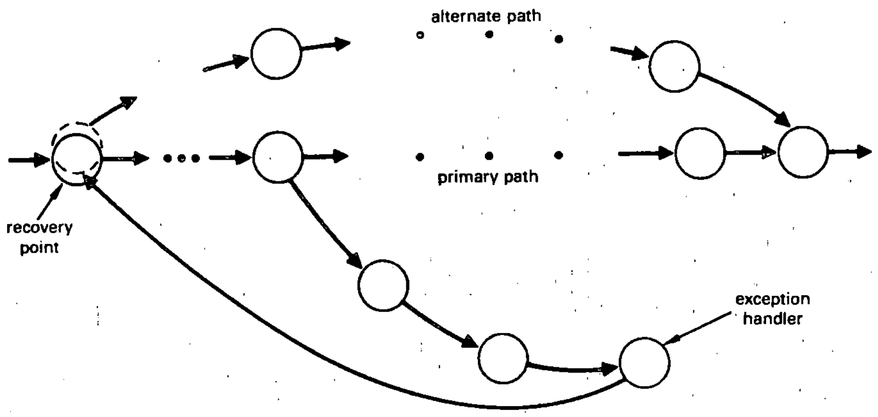

Figure 1.1b Backward Error Recovery

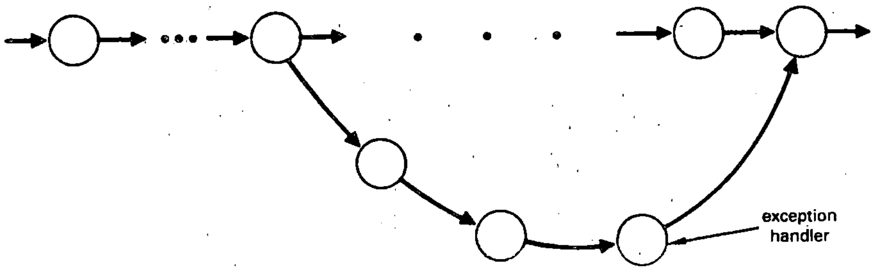

Figure 1.1c Forward Error Recovery

common shared memory). However, applicative semantic models do require such a restriction, and thus a process is represented by the set of all active states. The discussion that follows is concerned with only a single site of activity.

Each of the states represents either a normal state, during which the program is said to be operating in a normal and correct manner, or an erroneous state, during which some data items or control information is said to be in error (such classification of states may be subjective). If the transition from a normal state to an erroneous state (or from one erroneous state to another) is detected, either by the system or the user program, then an exception has been detected. Once detected, bringing the exception to the attention of the process is known as raising the exception. This is equivalent to making a transition to another (erroneous) state and starting execution of a special routine (the exception handler). Once raised, the process may then respond (or be required to respond) to the exception; this response is known as handling the exception and the routine that does this is known as an exception handler. The usual goal of an exception handler is to terminate the process gracefully, or to return the process to a normal state. The latter action is known as recovery.

The two general methods of exception recovery are backward error recovery and forward error recovery; they differ in the means used to return to a normal state. Backward error recovery attempts to return the process to a normal state by "backing up" the process to a known normal state (recovery point). Processing is then resumed from that point using an alternate computation path through the process graph, thus hopefully avoiding transition to an erroneous state. Forward error recovery attempts to return to a normal state by correcting the information that is in error; this process is very program dependent.

The sections that follow present examples of backward and forward error recovery mechanisms. The mechanisms were designed for a von Neumann system and assume that exception detection and raising mechanisms exist; they concentrate on exception handling with the goal of recovery.

### 1.2.2 Backward Error Recovery

Backward error recovery is the software analogue to the replacement of a defective hardware component by a standby spare. The technique involves replacement of the defective program module that caused the fault and then restart of the computation. One realization of this idea is the recovery block of Randell [Ran75, AnKe76, MeRa77, RLT78]. A recovery block is defined by the following syntax:

```
ensure <acceptance test>
by <primary alternate>
else by <other alternate>
    .
    .
    .
else error
```

The <primary alternate> corresponds to the block of an equivalent conventional program and is entered to perform the desired computation. The <acceptance test>, a Boolean expression without side effects, is then used to evaluate whether the alternate has performed acceptably (the process is in a normal state). An <other alternate>, if one exists, is entered to perform the computation if the preceding alternate fails to complete (e.g., an exception condition was raised or a time limit elapsed) or fails the <acceptance test>. However, before an alternate is entered, the state of the process is restored to that state current just before entry to the <primary alternate> (presumably this was a normal state). If the <acceptance test> is passed, any further alternates are ignored and the statement following the entire recovery block is the next to be executed.

Backward error recovery is conceptually easy to understand and use. This simplicity stems from the separation of the questions of exception detection (i.e., the <acceptance test>) and repair (i.e., the state restoration) from the question of how to provide continued service (i.e., the <other alternate>s). In addition, exception detection is simplified since it is not necessary to determine the actual information in error or the cause of the exception. However, the simplicity of backward error recovery masks two very serious problems.

The most serious of these problems arises during interprocess communication. Consider Figure 1.2. Let the dotted lines represent information flow between processes and the brackets represent recovery points. Should it become necessary to back up process C to recovery point 2, it is also necessary to back up process B to recovery point 2 because of the information flow between processes B and C. This also necessitaties backing up process A to recovery point 1, then process B to recovery point 1, and then all three processes to the start of the computation. This "domino effect" [Ran75] is intolerable for larger systems with long periods of interaction. Various means have been proposed to limit the effect: recovery lines (similar to the "sphere of control" of [Dav72, BjDa72]), structuring of "conversations" [Ran75], and a variant of audit trails [Rus77]. However, none of these is a totally satisfactory answer since each requires "throwing away" part of the computation; with luck, only the invalidated part is removed, but this cannot be guaranteed.

The second problem of backward error recovery concerns

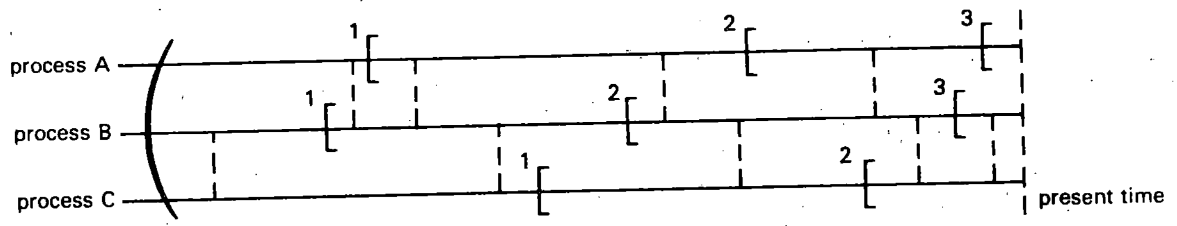

Figure 1.2 Domino Effect of Backward Error Recovery for Communicating Processes

the efficiency of the recovery process. Even though the process may be helped by providing a hardware "recursive cache" [Ran75] for the recovery, backward error recovery is inherently inefficient since it takes the "all-or-none" approach; there is no provision for establishing a partial recovery or using partial results. For instance, in some programs overflow exceptions are not significant; the result could have been represented by any suitably large number. Similarly, an underflow exception could be represented by zero (as is done on some machines). Such substitutions could also be appropriate for larger units of computation. However, any system which uses only backward error recovery is incapable of easily expressing value substitutions. The system is simply unable to distinguish between insignificant exceptions and major exceptions; in either case, the entire computation is thrown away during state restoration and an alternate method is tried.

### 1.2.3 Forward Error Recovery

Forward error recovery is analogous to the use of error correcting techniques and codes in order to cope with faulty hardware information storage and transmission components. Once an exception is detected, forward error recovery techniques try to return the process to a normal state by compensating for the exception condition and any damage that it may have caused before the exception was detected. Application of a forward error recovery technique to a particular program will involve damage assessment -- that is, it is necessary to determine, usually a priori, exactly what effects a particular type of exception can cause and to provide a program segment to correct those effects. It is because of this extra complexity that forward error recovery is attractive mostly for simple exception conditions where the damage is easily localized.

Most error recovery proposals are of the forward error recovery variety. This is probably because any backward error recovery technique requires either a language that is free from side effects for all procedure invocations or extra hardware support not normally available. Forward error recovery also has the benefit of allowing for the partial recovery that backward error recovery does not permit. Some examples of forward error recovery include PL/I ON conditions and recent proposals by Goodenough [Goo75], Wasserman [Was77], and Levin [Lev77]. CLU [LiSn77, LMSSS78] is one language which has implemented a forward error recovery scheme.

Consider the problem of writing a CLU procedure, sum_stream, which reads in a sequence of signed decimal integers from a character stream and returns the sum of the integers. The procedure must return one of the following four results:

1. the sum of the integers;
2. an OVERFLOW exception if the sum becomes too large to represent;
3. an INT_TOO_LARGE(<string>) exception if an individual decimal integer is outside the range of implemented integers; or
4. a BAD_FORMAT(<string>) exception if a stream element is not a signed decimal integer.

Procedure sum_stream uses procedure stream$getc(<stream>) which returns the next character or raises an END_OF_FILE exception if there are no more characters in the <stream>, and procedure string_to_integer(<string>) which returns the decimal integer corresponding to the <string> or raises any of the exceptions:

1. INVALID_CHARACTER(<char>) if one of the characters, <char>, is other than a digit or minus sign;
2. BAD_FORMAT if a minus sign follows a digit, there is more than one minus sign, or no digits; or
3. UNREPRESENTABLE_INT if an overflow occurs during conversion.

The example is described in [LMSSS78] as follows:

There are [logically] two loops within an infinite loop: one to skip spaces, and one to accumulate digits for conversion to a number. Notice the placement of the inner END_OF_FILE handler. [Note: handlers are denoted by the except when construct.] If END_OF_FILE is raised in the second inner loop, then the sum is computed correctly, and the first invocation of stream$getc will again raise END_OF_FILE. This time, however, the infinite loop is terminated and execution transfers to the other END_OF_FILE handler, which then returns the accumulated sum.

```
sum_stream =
  proc (s: stream) returns (int)
        signals (OVERFLOW,
                 INT_TOO_LARGE(string),
                 BAD_FORMAT(string))
  sum: int := Ø
  num: string := ""
  while true do
    c: char := stream$getc(s)
    except when END_OF_FILE:
                if (num = "")
                   then return (sum)
                end % if %
                c := ' '
    end % except %
    if c ≠ ' ' then num := string$append(num,c)
    elseif num ≠ "" then
             sum := sum + string_to_integer(num)
             num := ""
    end % if %
  end % while %
  except
    when UNREPRESENTABLE_INT:
           signal INT_TOO_LARGE(num)
    when BAD_FORMAT, INVALID_CHARACTER(*):
           signal BAD_FORMAT(num)
    when OVERFLOW: signal OVERFLOW
  end % except %
  end sum_stream
```

This example illustrates many of the traits of a forward error recovery system:

1. Exception handlers are associated with the syntactic unit that raises the exception. It is also quite common, as in CLU, to limit exception handling either to statements or to blocks.

2. An exception handler may return an alternate result and also alter the flow of control, e.g., the END_OF_FILE handler with the return clause.

3. Exceptions are handled at each level of the calling hierarchy. That is, if a procedure nested three levels deep is to signal a calling procedure at the top level, the signalled exception must pass through each of the intervening calls.

4. Finally, as with most useful forward error recovery systems, the exception handling capabilities can be used in "normal" programming. Consider the END_OF_FILE handler. The if statement does not represent an exception, rather it represents the expected normal termination of the while loop.

As can be seen from the above example, forward error recovery may be quite useful for handling some exceptions, mainly those easy to characterize by both their cause and effect. However, forward error recovery is inappropriate when dealing with unexpected exceptions, for example, residual design faults in programs; in such a case, backward error recovery should be used when fault tolerance is a system requirement. Therefore, a mixed recovery system that combines the capabilities of both the forward and backward error recovery systems would seem to be most useful; see [MeRa77] and [RLT78].

## 1.3 Overview of the Dissertation

This chapter has reviewed two general models for exception handling and recovery for von Neumann systems; the chapters that follow introduce a model for applicative systems. Chapter 2 describes two different applicative systems: FP [Bac78] and Id [AGP78]. Chapter 3 presents the principles underlying the proposed exception handling and recovery model and then relates these principles to systems for FP and Id. Chapter 4 extends the basic model of Chapter 3 to include dataflow streams (at the present, there is no FP counterpart). Finally, the conclusions and an evaluation of the model are presented in the last chapter.

Chapter 2

# APPLICATIVE SYSTEMS

## 2.1 Introduction

The principles of the von Neumann machine were established more than 3Ø years ago to solve the programming problems of that time. These principles may be summarized as:

1. Sequential "flow of control" from one instruction to the next;
2. Each instruction operating on a single or small fixed number of data elements; and
3. All data and instructions residing in a single central store.

Much recent work has attempted to improve language design, software design, or hardware design by modifying one or more of these principles. Such attempts include structured programming which in its simplest form restricts the permissible control flows, distributed and parallel processing which attempt to distribute either the control flow or data store or both, and vector processing which allows a single instruction to specify processing on more than a single data element. However, such attempts achieve only incremental improvement over the basic von Neumann machine and may introduce other complications.

Researchers working with applicative systems have taken a more radical approach and have replaced the von Neumann principles with new principles that are more appropriate to modern programming practice. The most important principle is the removal of the memory concept. This eliminates side-effects from operation execution, and thus also eliminates possible race conditions during parallel processing. The net effect is a greatly simplified semantic base for these models and the proposed exception handling and recovery model.

The remainder of this chapter presents an overview of two applicative systems: FP, developed by John Backus at IBM [Bac78], in Section 2.2, and Id, developed by Arvind, Kim Gostelow, and Wil Plouffe at the University of California at Irvine [AGP78], in Section 2.3. These two applicative systems are used in Chapters 3 and 4 to relate the ideas of this dissertation to proposed systems.

## 2.2 Applicative Languages, FP

Applicative languages are one class of applicative systems. They are based upon reduction semantics -- a program "executes" by transformation of the program and its data from one valid state to another, where states themselves are programs. There are several proposals for applicative languages including Church's lambda calculus [Chu41], Curry's system of combinators [CuFe58], pure Lisp [McC6Ø], and functional programming systems [Bac78]. This section will concentrate on the functional programming system (or FP system) of John Backus. (The information contained in this section is summarized from [Bac78].)

An FP system comprises five elements:

1. A set O of objects;
2. A set F of functions mapping objects into objects;
3. A single operation, application;
4. A set FF of functional forms which are used to combine existing objects or functions to form new functions; and
5. A set D of definitions which define some elements of F and assign a name to each.

The subsections that follow discuss each of these elements and the semantics of FP program execution in detail.

### 2.2.1 Objects: O

An object x is either an atom, a sequence $<x_1,...,x_n>$ where each element $x_i$ is an object, or the distinguished object "bottom" ("undefined"). The selection of a set A, the set of all atoms, defines the set O. The set A contains nonempty strings of letters, digits, and special symbols which define the names of the atoms. We assume that certain atoms are elements of A and have certain meanings:

1. strings of digits correspond to "numbers";
2. phi denotes the empty sequence and may also be written as <>; and
3. T and F denote the Boolean values "true" and "false", respectively.

The set O contains all possible sequences that may be constructed using the atoms from set A with one constraint: if x is a sequence with bottom as an element, then x=bottom. This property is referred to as "bottom-preserving". Thus, we may define the elements of O by the following:

1. if x is an element of A, then x is an element of O;
2. if $x_1,\ldots,x_n$ $(n\geq 1)$ are elements of O and $x_i \neq$bottom for $1\leq i\leq n$, then $<x_1,\ldots,x_n>$ is an element of O; and
3. bottom is an element of O.

## 2.2.2 Application

There is a single operation in an FP system: application. If f is a function and x is an object, then f:x is an application and denotes the object resulting from applying f to x. Thus, +:<5,6> = 11 and 2:<A,B,C> = B.

## 2.2.3 Functions: F

Any function f in F maps objects into objects, f:O -> O. Also, all functions are bottom-preserving, that is, f:bottom = bottom. Each function in F is either primitive, i.e., defined by the FP system, defined (see Section 2.2.5), or a functional form, an expression representing a function.

There are several primitive functions that we assume are part of any FP system discussed in later sections. These functions are defined here using the notation of Backus [Bac78]:

$$p_1\text{->}e_1;\ \ldots;\ p_n\text{->}e_n;\ e_{n+1}.$$

(This notation is equivalent to the McCarthy conditional expression [McC60] $(p_1\text{->}e_1,\ldots,p_n\text{->}e_n,T\text{->}e_{n+1})$.) The following definitions are to hold for all objects x, $x_i$, y, $y_i$, z, and $z_i$ not equal to bottom:

Identity

Id:x ≡ x.

Selector Functions

$1{:}x \equiv x=\langle x_1,\ldots,x_n\rangle \text{ -> } x_1;$ bottom.

For any positive integer i,

$i{:}x \equiv x=\langle x_1,\ldots,x_n\rangle \ \&\ i\leq n \text{ -> } x_i;$ bottom.

Tail

```
tl:x ≡ x=<x1> -> <>;
       x=<x1,...,xn> & n≥2 -> <x2,...,xn>;
       bottom.
```

Atom

```
atom:x ≡ x is an element of A -> T; F.
```

Equals

```
eq:x ≡ x=<y,z> & y=z -> T;
       x=<y,z> & y≠z -> F;
       bottom.
```

Arithmetic Functions

```
+:x ≡ x=<y,z> & y,z are numbers -> y+z; bottom.

-:x ≡ x=<y,z> & y,z are numbers -> y-z; bottom.

*:x ≡ x=<y,z> & y,z are numbers -> y*z; bottom.

/:x ≡ x=<y,z> & y,z are numbers -> y/z; bottom.
```

Logical Functions

```
and:x ≡ x=<T,T> -> T;
        x=<T,F> or x=<F,T> or x=<F,F> -> F;
        bottom.

or:x ≡ x=<T,T> or x=<T,F> or x=<F,T> -> T;
       x=<F,F> -> F;
       bottom.

not:x ≡ x=T -> F; x=F -> T; bottom.
```

Append Left and Right

```
apndl:x ≡ x=<y,<>> -> <y>;
          x=<y,<z1,...,zn>> -> <y,z1,...,zn>;
          bottom.

apndr:x ≡ x=<<>,z> -> <z>;
          x=<<y1,...,yn>,z> -> <y1,...,yn,z>;
          bottom.
```

## 2.2.4 Functional Forms: FF

A functional form is an expression that denotes a function and consists of the functional operator and the functions or objects that are parameters of the operator. For example, f•g is the functional form representating composition of the functions f and g. The functional operator is represented by •, and the parameters are the functions f and g. When this is applied to an object x,

$$(f \cdot g):x = f:(g:x).$$

Given below are the definitions for functional forms used in later sections. (It is interesting to note that these definitions, while part of an FP system, are not required for defining an FFP system [Bac78]. In such a system, meta-composition may be used for defining functional operators.)

Constant

$$\bar{x}:y \equiv y=\text{bottom} \rightarrow \text{bottom};\ x.$$

Composition

$$(f \cdot g):x \equiv f:(g:x).$$

Construction

$$[f_1,\ldots,f_n]:x \equiv \langle f_1:x,\ldots,f_n:x\rangle.$$

Note that construction is bottom preserving:

$$\begin{aligned}[f_1,\ldots,f_n]:\text{bottom} &= \langle f_1:\text{bottom},\ldots,f_n:\text{bottom}\rangle\\ &= \langle \text{bottom},\ldots,\text{bottom}\rangle\\ &= \text{bottom}.\end{aligned}$$

Condition

```
(p->f;g):x ≡ (p:x)=T -> f:x;
             (p:x)=F -> g:x;
             bottom.
```

The condition functional form corresponds to the if statement of conventional languages. When there is no ambiguity, we write $(p_1\text{->}f_1;\ldots;p_n\text{->}f_n;g)$ for the form $(p_1\text{->}f_1;(\ldots(p_n\text{->}f_n;g)\ldots))$.

Insert

$$
\begin{aligned}
/f{:}x \equiv\ & x=\langle x_1\rangle \text{ -> } x_1;\\
& x=\langle x_1,\ldots,x_n\rangle \ \&\ n\geq 2 \text{ -> } f{:}\langle x_1,/f{:}\langle x_2,\ldots,x_n\rangle\rangle;\\
& \text{bottom}.
\end{aligned}
$$

Apply to All

$$
\begin{aligned}
\alpha f{:}x \equiv\ & x=\langle\rangle \text{ -> } \langle\rangle;\\
& x=\langle x_1,\ldots,x_n\rangle \text{ -> } \langle f{:}x_1,\ldots,f{:}x_n\rangle;\\
& \text{bottom}.
\end{aligned}
$$

Binary to Unary

```
(bu f x):y ≡ f:<x,y>.
```

While

```
(while p f):x ≡ (p:x)=T -> (while p f):(f:x);
                (p:x)=F -> x;
                bottom.
```

## 2.2.5 Definitions: D

A definition is an expression of the form

```
Def l ≡ r
```

where l is an unused function symbol and r is a functional form. For example,

```
Def last ≡ null•tl -> 1; last•tl
```

defines the function last to return the nth atom of a sequence comprising n atoms. The set D of definitions is well formed if there exists at most one definition for each function symbol.

## 2.2.6 Semantics

The FP system is determined by the following sets:

1. The set A of atoms (which then determines the set O of objects);
2. The set P of primitive functions;
3. The set FF of functional forms; and
4. The well formed set D of definitions.

The semantics of an FP system are defined by the reduction of an application to an object. There are four possibilities for the function f in the application f:x :

1. f is a primitive function, i.e., f is an element of P;
2. f is a functional form, i.e., f is an element of FF;
3. there is a definition in D that defined f in terms of a functional form, i.e., (Def f ≡ r) is an element of D; or
4. none of the above is true.

If f is a primitive function, then the result of the application is the result of applying the primitive definition for f to the parameter x. If f is a functional form, then the definition of the form defines how to compute f:x. If f is defined, (Def f ≡ r) is an element of D, then f:x is computed by r:x. If none of the above holds, then f:x is defined to be bottom*.

For example, using the definition of last given in Section 2.2.5, we may compute last:<1,2>.

```
    last:<1,2>           => (null•tl->1;last•tl):<1,2>
using condition form     => (last•tl):<1,2>
                            since (null•tl):<1,2>
                                  = null:(tl:<1,2>)
                                  = null:<2> = F.
using composition form   => last:(tl:<1,2>)
from def. of tl          => last:<2>
using def. of last       => (null•tl->1;last•tl):<2>
using condition form     => 1:<2>
                            since (null•tl):<2>
                                  = null:(tl:<2>)
                                  = null:<> = T.
from def. of selector 1 => 2
```

## 2.2.7 Expressive Power of FP Systems

Given the proper definitions, an FP system is capable of expressing any computable function on the set O. However, since FP systems are not history sensitive, the

--------------------

*If the computation f:x does not terminate, Backus defines the result to be bottom. This results in a dual meaning for bottom: it may represent an exception condition, i.e., +:<a,2>=bottom, or it may represent a nonterminating computation. We will distinguish these two meanings in Section 3 by defining "error" to represent an exception condition and using bottom to represent a nonterminating computation only.

definition of an FP system fixes the functions available since no program can alter the set of definitions D. Further, since functions map objects to objects and the definition of a function is a function expression, not an object, it is impossible for an FP system to have a function apply defined by

$$\text{apply}:\langle f,x\rangle \equiv f:x.$$

These limitations are not present in Formal Functional Programming (FFP) Systems or Applicative State Transition (AST) Systems [Bac78]; however, the additional semantics required for these systems are not needed for the purposes of this thesis and will not be introduced.

## 2.3 Dataflow Languages, Id

Dataflow languages are a second class of applicative systems. Much of the research on dataflow languages stems from the pioneering work of Jack Dennis at M.I.T. Major research projects are now in progress on various aspects of dataflow at the University of California at Irvine, the University of Utah, Texas Instruments Incorporated, the University of Newcastle Upon Tyne, the University of Manchester, and the French National Laboratory at Toulouse. This and the following sections will focus on a single dataflow language, Id [AGP78], for any examples; however, the principles of exception handling that are discussed are applicable to any dataflow language.

A dataflow program comprises a set of partially ordered operations where the partial order is determined only by the explicit need for intermediate results. This partial order is usually expressed using digraph notation. For example, Figure 2.1 is a representation for the expression (-b+sqrt(b^2-4*a*c))/(2*a). Operationally:

1. a dataflow operation executes when and only when all of the required operands become available, and
2. a dataflow operation is purely functional and produces no side-effects as a result of its execution.

Dataflow languages are usually presented by defining the base language operators that are available. This is analogous to defining a machine architecture by the machine code that the architecture is designed to execute. The approach taken here is to define the high-level dataflow language Id and then briefly to introduce operators necessary for particular examples.

Id is a block-structured expression-oriented single-assignment language. An Id program is simply a list of expressions. This section defines values and the basic expressions present in Id - blocks, conditionals, loops, and procedure applications.

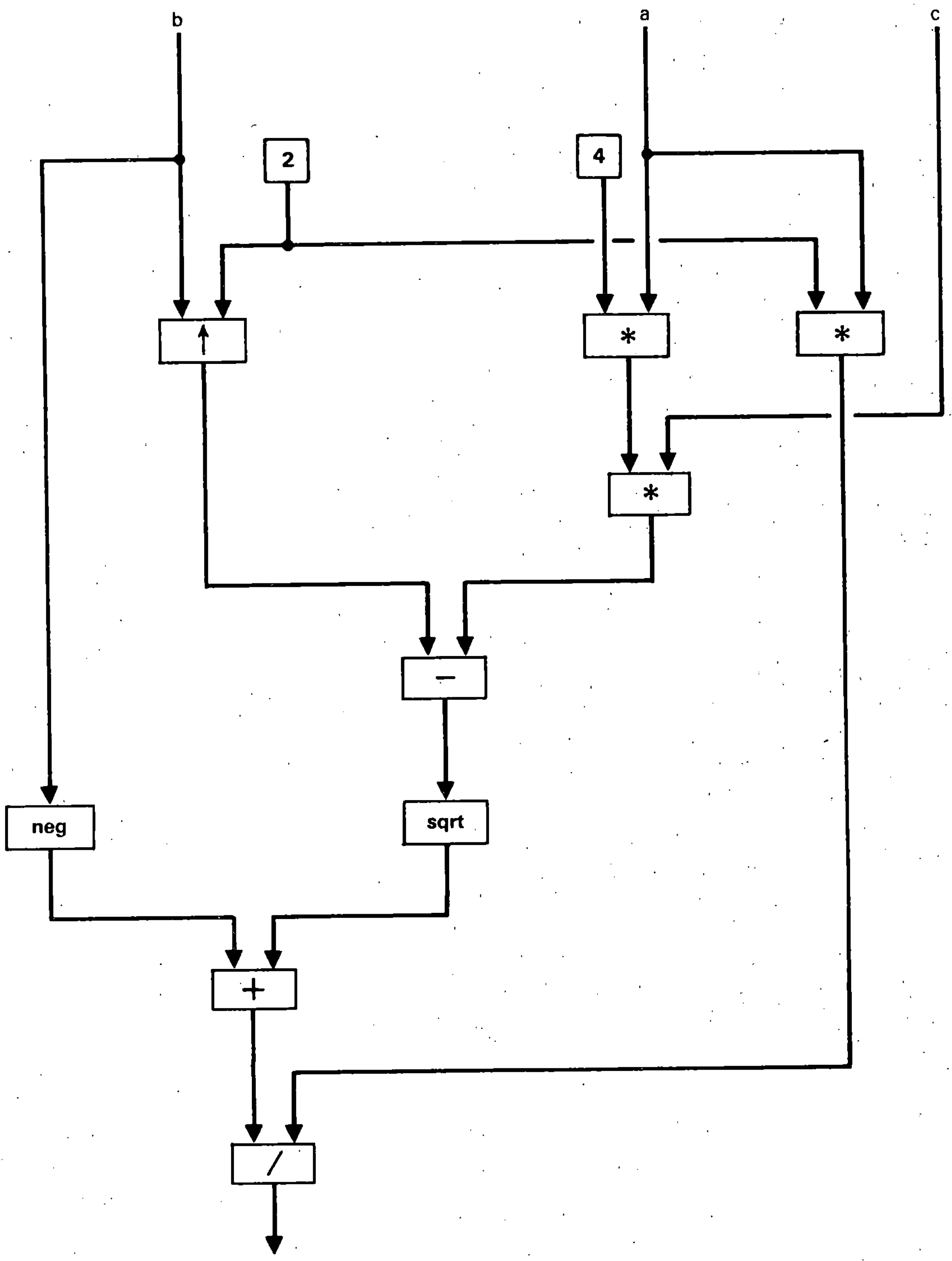

Figure 2.1  Translation of
(-b+sqrt(b^2-4*a*c))/(2*a)

### 2.3.1 Values

Id is not a strongly typed language. The type of an Id value is associated with the value itself and not with the Id variable. There are eleven primitive types of Id values: integer, real, Boolean, string, structure, procedure definition, manager definition, manager object, pdt (for programmer-defined data type), key (utilized for implementing security and protection), and error. The first four types correspond to the common conception of these types and will not be discussed here. Structure values are similar to arrays of other languages, but with some important distinctions. They are discussed below. Error values are introduced in the following sections, and the remaining values are discussed in [Bic78] and [AGP78].

A structure value is either the distinguished value <> (the empty structure) or a set of (selector, value) pairs where each selector is distinct from all others in the set of pairs. A selector is either an integer or string value; a value may be any Id value. There are two base language operators defined on structures: select and append. If st is a structure value and sel a selector value, then select(st,sel) will return the value v if (sel, v) is an element of st. The Id syntax corresponding to select(st,sel) is st[sel].

The append operator creates a new structure. If st is a structure value, sel a selector value, and v any value, append(st,sel,v) will create a new structure identical to st but with (sel, v) replacing any pair in st with sel as the selector. The append operator does <u>not</u> modify existing structures; rather it creates a copy of the original structure and operates on the copy. Thus, append is free from side-effects. The Id syntax for append(st,sel,v) is st+[sel]v.

A variant of append is also used to delete a (<u>selector</u>, <u>value</u>) pair from a structure. This variant, denoted by delete(st,sel), returns a copy of the original structure st with any pair with <u>selector</u> equal to sel removed.

These operations are defined as follows, where st is the structure value $\{(s_1,v_1),\ldots,(s_n,v_n)\}$:

```
append(st,sel,v) ≡
   {(s_i,v_i) | (s_i,v_i) in st & s_i≠sel}
   union {(sel,v)}.

select(st,sel) ≡ (s_i,v_i) in st -> v_i;
                 undefined.

delete(st,sel) ≡
   (s_i,v_i) in st & s_i=sel
      -> {(s_i,v_i) | (s_i,v_i) in st & s_i≠sel};
   undefined.
```

### 2.3.2 Block Expressions

A block expression comprises a series of statements terminated by a return clause. The result of the block expression is the result obtained by evaluating the expression in the return clause.

```
block-expression ::=
     "(" statement-list [";"] return-clause ")" .

statement-list ::= statement {";" statement} .

statement ::= variable "<-" expression |
              procedure-definition .

return-clause ::=
     "RETURN" expression {"," expression} .
```

Statements are analogous to definitions: they define a variable name to be equivalent to the result of an expression evaluation or a procedure definition (see Section 2.3.5). Since statements are analogous to definitions, they are unordered and there can be only one statement defining each distinct variable (the single-assignment rule). Further, two variable definitions may not be mutually dependent (i.e., if a is defined by an expression utilizing the definition for b, b may not be defined by an expression that requires the definition for a).

Consider the following expression to compute the roots of a quadratic equation:

```
((-b+sqrt(b^2-4*a*c))/(2*a),
    (-b-sqrt(b^2-4*a*c))/(2*a))
```

An equivalent (but computationally more efficient) block expression is:

```
(x <- sqrt(b^2-4*a*c);
 y <- 2*a
 return (-b+x)/y, (-b-x)/y)
```

Both expressions require the inputs a, b, and c, and produce two values for results. The only difference would be in the actual computation – the expressions defining x and y in the block expression are actually computed twice for the list of expressions. The translation for the block expression is given in Figure 2.2.

## 2.3.3 Conditional Expressions

The conditional expression construct of Id corresponds to the if expression (or statement) of conventional languages.

```
conditional-expression ::=
      "(" "IF" Boolean-expression
          "THEN" expression {"," expression}
          "ELSE" expression {"," expression} ")" .

Boolean-expression ::= expression .
```

Consider the conditional expression

```
(if p(x) then f(x) else g(x))
```

The base language translation for the expression is given in Figure 2.3. Whenever a value for x is available, p(x) is

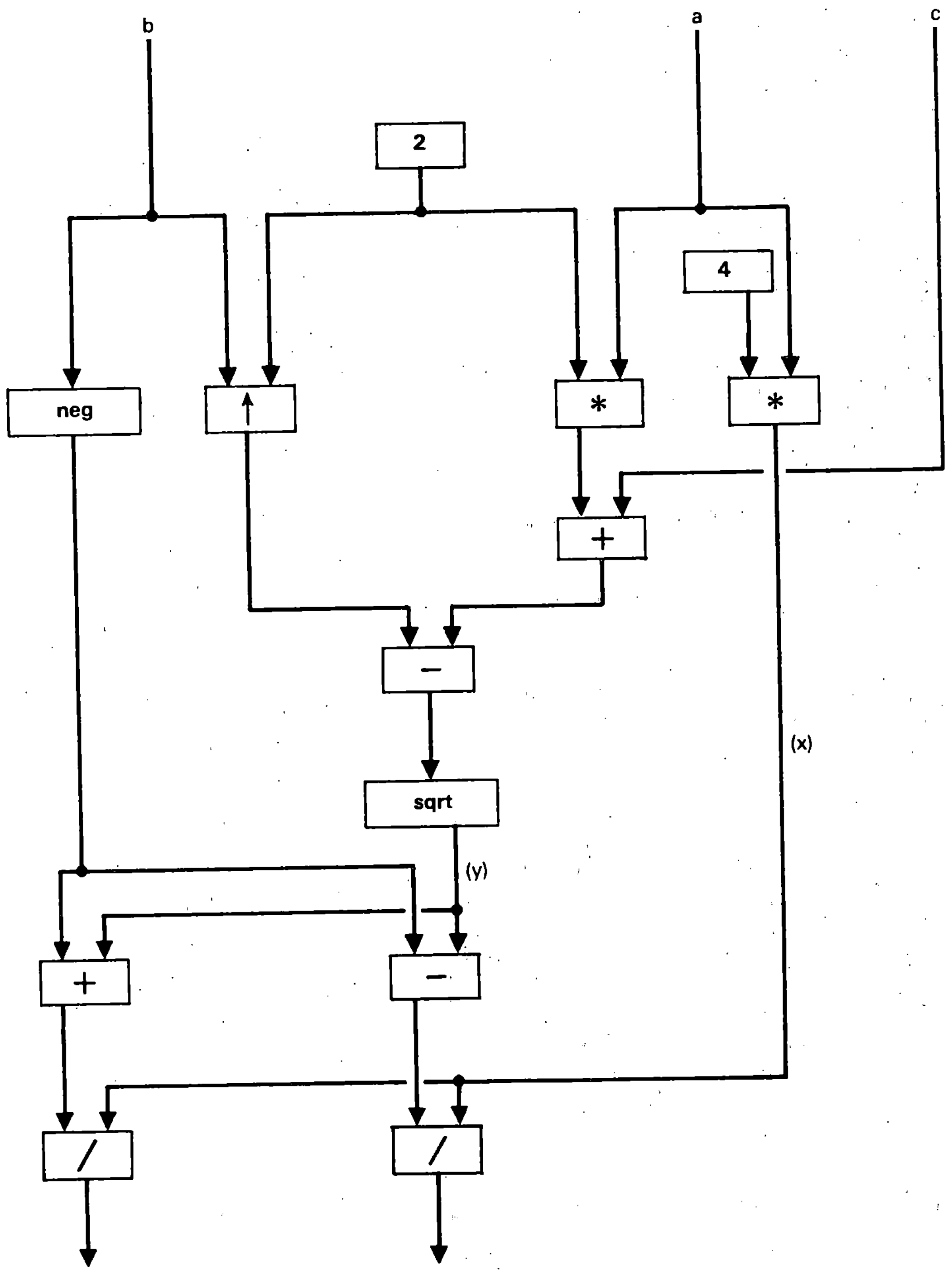

Figure 2.2 Translation of a Block Expression

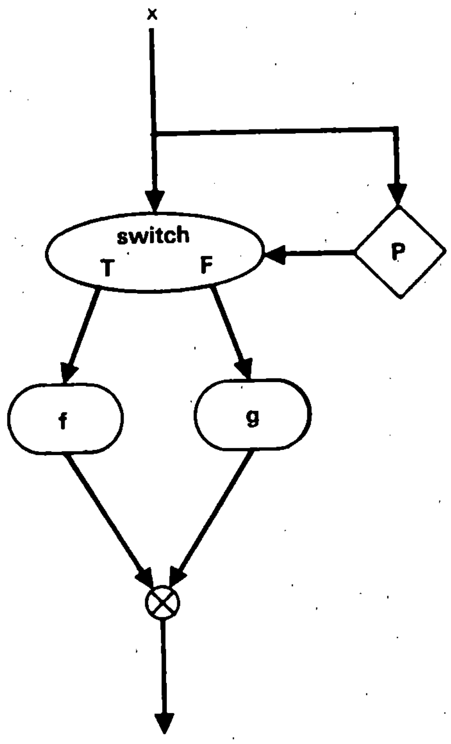

Figure 2.3 Translation of a Conditional Expression

evaluated for a Boolean result (the semantics are undefined for a non-Boolean result). If the result is true then an x value is sent to f, otherwise an x value is sent to g. This is accomplished using the switch operator. The $\otimes$ symbol is used to represent a legal merging of two output arcs. The result of the conditional expression is the result obtained from either f or g.

### 2.3.4 Loop Expressions

A loop expression comprises four distinct parts: an initial part which defines any initialization necessary for the loop body, a predicate part to control the loop iteration, a loop body comprising a series of statements, and a return clause for computing the result of the loop expression.

```
loop-expression ::= "(" initial-part [";"]
                        predicate-part
                        loop-body [";"]
                        return-clause ")" .

initial-part ::= "INITIAL" statement-list .

predicate-part ::= "WHILE" Boolean-expression "DO" .

loop-body ::= loop-statement {";" loop-statement} .

loop-statement ::= variable "<-" expression |
                   "NEW" variable "<-" expression .
```

An Id loop is essentially a set of first degree recurrence equations. Consider the following loop expression

```
(initial i <- 1;
         sum <- Ø;
 while i<n do
       nexti <- i + 1;
       new i <- nexti;
       new sum <- sum + nexti;
 return sum)
```

The initial part specifies the initial values of the recurrence relation.

$$i_{\emptyset} = 1 \text{ and } sum_{\emptyset} = \emptyset$$

The loop body specifies the recurrence relations. Any statement not beginning with the keyword NEW specifies the definition of a temporary variable. Thus the recurrence relations specified by the above loop are

$$i_{j+1} = i_j+1 \text{ and } sum_{j+1} = sum_j+i_{j+1}$$

The predicate part specifies the terminating conditions for loop interation. When the Boolean expression returns false, the current values of the variables are passed from the loop body to the return clause. The result of the loop expression is the result from the return clause.

The base language translation of the above loop expression is presented in Figure 2.4. The new operators D and D-1 are used to distinguish between values which belong to different iterations of the loop. The operators L and L-1 distinguish between values from different instantiations of the same loop (this occurs when one loop is nested within

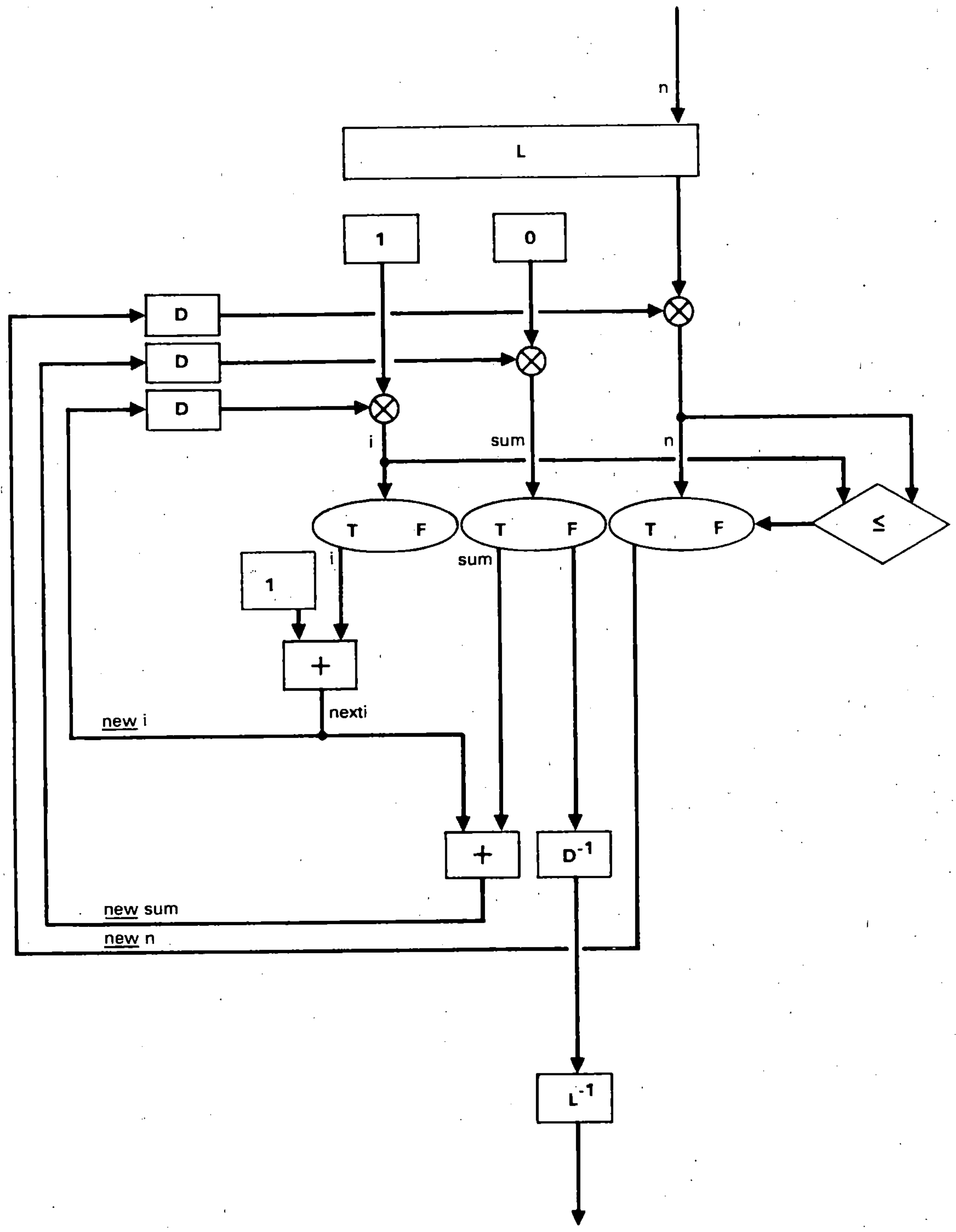

Figure 2.4 Translation of a Loop Expression

another). These operators are discussed in [AGP78].

### 2.3.5 Procedure Application

An Id procedure application is an expression -- a computation free from side-effects returning a result.

```
procedure-application ::=
     "APPLY" "(" variable [","  argument-list] ")" |
     variable "(" [argument-list] ")" .

argument-list ::= expression {"," expression} .

procedure-definition ::=
     "PROCEDURE" variable "(" [parameter-list] ")"
        expression .

parameter-list ::= variable {"," variable} .
```

Consider the following expression

```
(procedure sqrt (a)
     (initial x <- a/2
      while abs(x^2-a)>0.000001 do
            new x <- (x^2+a)/(2*a)
      return x);
 x <- sqrt(b+2*a);
 return x)
```

The above expression defines two variables, sqrt and x, naming a procedure definition value and a real value, respectively. The expression sqrt(b+2*a) invokes the procedure defined by the previous lines, passing b+2*a as the argument.

The base language translation for the expression is given in Figure 2.5. The operator apply is defined in

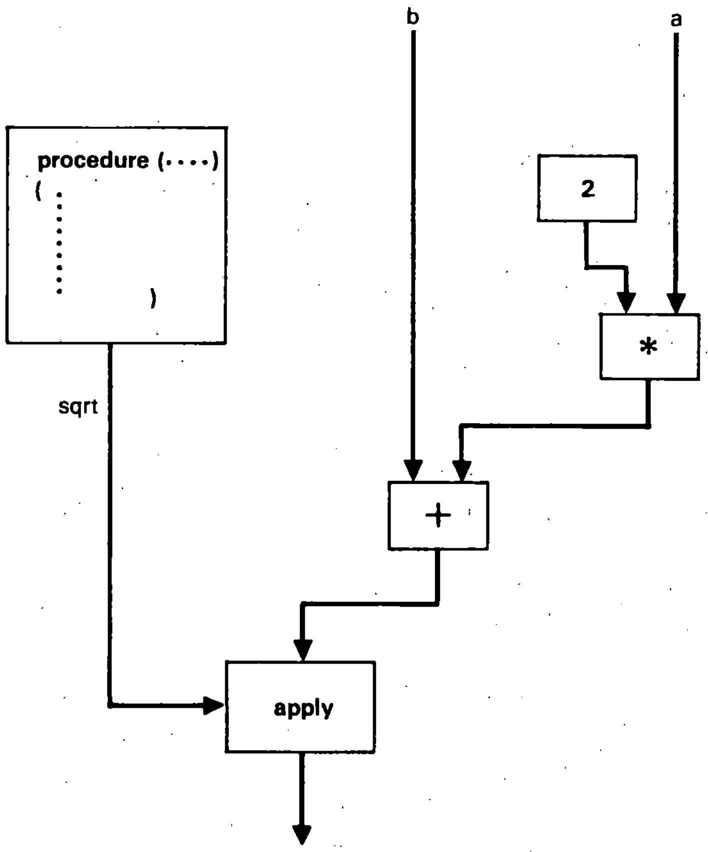

Figure 2.5 Translation of a Procedure Application

[AGP78]. Informally, it creates a new distinct context in which to execute the procedure sqrt, passes all of the arguments to that context as a single packet (structure), and receives the result packet (structure) from the return clause to distribute the result(s).

### 2.3.6 Expressive Power of Dataflow Systems

A dataflow language is capable of expressing any determinate function. In addition, there are several proposals for extending a dataflow language to handle nondeterminate computations such as input/output [GWG78, TFGJRS78, Ols80] or resource management [AGP77], and to define their semantics [Kos78].

## 2.4 Summary

This chapter has described two different applicatives systems: FP is an applicative language defined using reduction semantics and Id is a dataflow language defined using operational semantics. These two systems are used in the following chapters to relate the principles of the proposed exception handling and recovery model to actual implementations.

Chapter 3

# THE BASIC MODEL

## 3.1 Introduction

The basic model presented here for detecting, raising, and handling exception conditions in an applicative system is similar to a forward error recovery system. However, it differs substantially from forward error recovery for a von Neumann system in that it also has the expressive power of a backward error recovery system (this derives directly from the semantic base of an applicative system, i.e., the elimination of side-effects from the execution of an operation). The model for exceptions is based upon the following principles:

1. the preservation of the semantic base of the applicative system, even in the presence of exception conditions;
2. the definition of an "error" value;
3. the avoidance of non-terminating computations caused by the presence of exception conditions; and
4. the definition of distinguishing syntax for exception handling so that exception handling is distinct from the "normal" program flow.

The first principle implies that no major modification can be made to the semantic base of the applicative system. This rules out any of the conventional exception handling mechanisms based upon the flow of control, e.g., interrupts or special exit mechanisms. However, a value-based exception model does not require major modifications to the underlying semantic base.

In defining a value-based exception model, it is necessary to define the form of an exception condition. The second principle states that an exception will be represented as an error value; two possibilities exist for the definition of an error value:

1. error values may be defined as particular values within a larger type domain, and each type domain contains distinct error values; or

2. error values are defined as members of a distinct type domain.

The first alternative corresponds to strong typing of error values. Thus zero_divide would be a numeric value and undefined might be a Boolean value. This is the approach taken by VAL [AcDe79, Ack79]. However, the second alternative has been chosen for this work as this alternative adds the smallest amount of additional structure to the semantic model.

With the introduction of an error value, it is necessary to define the behavior of both individual operations, e.g., +, -, append, and select, and operations which implement the structured constructs such as conditional, looping, and procedure application. In particular, each operation and construct should force termination of the expression or construct whenever an exception condition is detected and raised. Thus an exception condition should not cause a nonterminating computation simply by its presence.

Finally, a syntax for exception handling is defined to encourage the use of the exception handling capabilities. This syntax separates the program code for the "normal" case from that for the exception condition, thus reducing the complexity of the program segments and increasing the understandability of the segments. At the same time, the "conceptual distance" between the expression which raises an exception and the handler which recovers from the exception should be small. This all implies that the handler should be distinct from, but "close" to, the associated expression.

The sections that follow discuss each of the major parts of the model in turn: the definition of an error value, the extension of the operator and syntactic construct definitions to total functions, and the association of exception handlers with expressions.

## 3.2 An Error Value

An error value is produced each time an operator, a language construct, or the programmer raises an exception. Each error value comprises three elements: a type which identifies the value as an error value, a class field which identifies the kind of exception detected, e.g., "type" or "end_of_file", and a parameter field which associates parameters or partial results with an exception.

For system-defined exception conditions, the parameter field of the error value should contain information about the context in which the exception was detected, e.g., for a conventional language, the name of the executing procedure and the statement number, or instruction address, within that procedure. For FP, this context information may be represented by a sequence containing the name of the executing defined function (from the set D) and the primitive function or functional form (an element of F or FF). For Id, this information is contained within the activity name for the unraveling interpreter [AGP78] which comprises the name of the executing procedure, the statement number, and an iteration count if the statement is part of a loop. If a statement number is returned as part of the context, the appropriate functions must be provided to associate statement numbers with the corresponding primitive function or functional form, e.g., statement 5 is the function "+". For this dissertation, the context information within a system-defined error value is represented by the name of the detecting operation, e.g., (error,overflow,/).

Four operations are defined on error values:

```
error: class x parameter -> (error,class,parameter)
errorclass: (error,class,parameter) -> class
errorparm: (error,class,parameter) -> parameter
errorp: value -> Boolean
```

The first operation, error, is the only means for creating an error value with the appropriate class and parameter fields. Since the only restriction is that the class argument not equal "composite", the programmer may define an error value that has the same form as a system-defined error value, such as (error,overflow,/). The next two operations, errorclass and errorparm, retrieve the class and parameter fields from an error value, respectively. The final operation, errorp, is a predicate that returns the error status of a value; true if the value is an error value, false otherwise.

Finally, there are also composite error values which represent the detection and raising of two or more separate exceptions. Such a value contains "composite" in the class field and a set of constituent error values in the parameter field, e.g.,

(error,composite,{(error,overflow,/),(error,domain,+)}).

The following two subsections relate these concepts to FP and Id.

### 3.2.1 FP Error Values

An FP error value is represented by a sequence consisting of three elements: the distinguished atom "err", the class field, and the parameters field. An alternative representation consists of the single distinguished atom err or a subset, E, of atoms which represent detected exceptions. Thus E may comprise the set of atoms err, type_err, domain_err, Boolean_err, overflow_err, etc. This second approach simplifies some of the operator definitions; however, parameters are not easily associated with a particular error value so the first approach has been used.

With the definition of an error value, the distinguished atom "bottom" no longer represents an error that has been detected, but represents only the semantics for a non-terminating computation. This distinction between an error and a non-terminating computation allows us to maintain the "bottom-preserving" property of FP functions and functionals while allowing for the handling and recovery of exceptions.

The following definitions of the four functions on error values hold for all objects x, y, and z not equal to bottom:

```
error:x ≡ x=<err,y,z> -> x;
          x=<y,z> & y is a string & y≠composite
            -> <err,y,z>;
          <err,domain,error>.

errorclass:x ≡ x=<err,y,z> -> y;
               <err,domain,errorclass>.

errorparm:x ≡ x=<err,y,z> -> z;
              <err,domain,errorparm>.

errorp:x ≡ x=<err,y,z> -> T; F.
```

Finally, an FP composite error value represents the set of constituent error values using a sequence. Since the order of elements in a set is not significant, the values

```
<err,composite,<<err,overflow,/>,<err,domain,+>>>
```

and

```
<err,composite,<<err,domain,+>,<err,overflow,/>>>
```

represent the same value.

## 3.2.2 Dataflow Error Values

An Id error value is a value of the type "error". Each error value includes the class and parameter subfields described above, and is represented by (error,class,parameter).

The four new operators for error values are defined as follows:

```
error(x,y) ≡ x is a string & x≠"composite"
                 -> (error,x,y);
               (error,"domain","error").

errorclass(x) ≡ x=(error,y,z) -> y;
                  (error,"domain","errorclass").

errorparm(x) ≡ x=(error,y,z) -> z;
                 (error,"domain","errorparm").

errorp(x) ≡ x=(error,y,z) -> true; false.
```

Finally, an Id composite error value represents the set of constituent error values using a structure of the form $<\#:n,1:v_1,\ldots,n:v_n>$ where $v_i$, $1 \leq i \leq n$, is one of the consituent error values. Again, order of the values within the structure representation is not significant.

## 3.3 Extension of Operator Definitions

Once an error value and operations on that value are defined, it is also necessary to define the actions of existing operations in response to error values received as input. In particular, all existing operations are defined as total functions. This entails strict domain checking for inputs and range checking for results. The operation must be defined to produce a result even if a domain or range exception is detected. (The result may be simply an error value or may be the result from an extended interaction with the user, such as may occur when using a dynamic debugger.) Moreover, the termination of individual operations leads to proofs of termination for programs and systems [Pat7Ø, ArGo77a].

When an operator receives more than one error value for input, several alternatives are possible:

1. return only one of the error values;
2. return a set (represented by either a sequence or a structure) containing all of the error values; or
3. return a single composite error value containing the information of each of the error values.

The final alternative was chosen for this model. This alternative retains the most information and requires no additional semantic error value infrastructure.

The function errorunion is used in the following sections to return a single error value given two input values, y and z. If only one of the values, y or z, is an error value, then that value is returned as the result. If both are error values but equal, then that value is returned. Finally, if both values are distinct error values, then errorunion returns a single composite error value as the result.

Errorunion has the following behavior:

```
errorunion(y,z) ≡
   not(errorp(y)) & not(errorp(z))
      -> (error,domain,errorunion);
   y=z -> y;
   errorp(y) & not(errorp(z)) -> y;
   not(errorp(y)) & errorp(z) -> z;
   errorclass(y)≠composite & errorclass(z)≠composite
      -> (error,composite,{y,z});
   errorclass(y)=composite & errorclass(z)≠composite
      -> (error,composite,errorparm(y) union {z});
   errorclass(y)≠composite & errorclass(z)=composite
      -> (error,composite,{y} union errorparm(z));
   errorclass(y)=composite & errorclass(z)=composite
     -> (error,composite,errorparm(y) union errorparm(z)).
```

Thus, the general behavior that is expected of an operation may be summarized as follows:

1. An operation must perform domain and range checking; if an exception is detected, the operation may produce an error value to represent raising the exception. An operation must not fail to terminate simply because a domain or range failure occurs -- it <u>must</u> produce some result.

2. An operation receiving an error value for input will propagate the error value received (assuming that the error value is outside the normal domain of the operation).

3. Finally, if an operation receives more than one error value for input, it will return a single composite error value as its result (again assuming that the error values are outside the normal domain of the operation).

### 3.3.1 Extensions for FP Systems

Some FP functions were defined in Section 2.2.3. Those definitions that must be changed are given below and are to hold for all objects x, $x_i$, y, $y_i$, z, and $z_i$ not equal to bottom:

#### Selector Functions

```
1:x ≡ x=<err,y,z> -> x;
      x=<> -> <err,missing,1>;
      x=<x1,...,xn> -> x1;
      <err,domain,1>.
```

For any positive integer i,

```
i:x ≡ x=<err,y,z> -> x;
      x=<> -> <err,missing,i>;
      x=<x1,...,xn> & i>n -> <err,missing,i>;
      x=<x1,...,xn> & i≤n -> xi;
      <err,domain,i>.
```

#### Tail

```
tl:x ≡ x=<err,y,z> -> x;
       x=<> -> <err,empty,tl>;
       x=<x1> -> <>;
       x=<x1,...,xn> & n≥2 -> <x2,...,xn>;
       <err,domain,tl>.
```

#### Atom

```
atom:x ≡ x=<err,y,z> -> x;
         x is an element of A -> T;
         F.
```

#### Equals

```
eq:x ≡ x=<err,y,z> -> x;
       x=<y,z>
          -> (y≠<err,y1,y2> & z≠<err,z1,z2>
                -> (y=z -> T; F);
             errorunion(y,z));
       <err,domain,eq>.
```

## Arithmetic Functions

```
+:x ≡ x=<err,y,z> -> x;
      x=<y,z>
         -> (y and z are numbers -> y+z;
               errorunion(numberp(y,+),numberp(z,+)));
      <err,domain,+>.

where numberp(x,f) ≡ x is a number -> <>;
                     x=<err,x1,x2> -> x;
                     <err,domain,f>.

-:x ≡ similar to (+:x).

*:x ≡ similar to (+:x).

/:x ≡ similar to (+:x).
```

## Logical Functions

```
and:x ≡ x=<err,y,z> -> x;
        x=<y,z>
           -> (x=<T,T> -> T;
                x=<T,F> or x=<F,T> or x=<F,F> -> F;
                errorunion(Booleanp(y,and),
                           Booleanp(z,and)));
        <err,domain,and>.

where Booleanp(x,f) ≡ x=T or x=F -> <>;
                      x=<err,x1,x2> -> x;
                      <err,domain,f>.

or:x ≡ similar to (and:x).

not:x ≡ x=<err,y,z> -> x;
        x=T -> F;
        x=F -> T;
        <err,domain,not>.
```

## Append Left and Right

```
apndl:x ≡ x=<err,y,z> -> x;
          x=<y,<>> -> <y>;
          x=<y,z> & z=<err,z1,z2> -> z;
          x=<y,<z1,...,zn>> -> <y,z1,...,zn>;
          <err,domain,apndl>.

apndr:x ≡ x=<err,y,z> -> x;
          x=<<>,z> -> <z>;
          x=<y,z> & y=<err,y1,y2> -> y;
          x=<<y1,...,yn>,z> -> <y1,...,yn,z>;
          <err,domain,apndr>.
```

Finally, some of the functional forms introduced in Section 2.2.4 must also be redefined to account for error values.

Constant

```
x̄:y ≡ y=bottom -> bottom;
      x=err -> <err,domain,constant>;
      x.
```

Condition

```
(p->f;g):x ≡ (p:x)=T -> f:x;
             (p:x)=F -> g:x;
             x=<err,y,z> -> x;
             <err,Boolean,condition>.
```

Insert

```
/f:x ≡ x=<err,y,z> -> x;
       x=<x1> -> x1;
       x=<x1,...,xn> & n>2 -> f:<x1,/f:<x2,...,xn>>;
       <err,domain,insert>.
```

Apply to All

```
αf:x ≡ x=<err,y,z> -> x;
       x=<> -> <>;
       x=<x1,...,xn> -> <f:x1,...,f:xn>;
       <err,domain,apply_to_all>.
```

While

```
(while p f):x ≡ (p:x)=T -> (while p f):(f:x);
                (p:x)=F -> x;
                x=<err,y,z> -> x;
                <err,Boolean,while>.
```

## 3.3.2 Extensions for Dataflow

The dataflow language operators must also be extended to handle error values. Most dataflow operations exhibit behavior similar to that of their FP counterparts. Consider the binary operator +:

```
+(x,y) ≡ x and y are numbers -> x+y;
         errorunion(numberp(x,"+"),numberp(y,"+")).

where numberp(x,f) ≡ x is a number -> lambda;
                     x=(error,$x_1$,$x_2$) -> x;
                     (error,"domain",f).
```

The dataflow structure operations also exhibit behavior similar to their FP counterparts, construction and selection.

```
append(st,sel,v) ≡
   st=<> & sel is a selector -> {(sel,v)};
   st={($s_i$,$v_i$) | $1 \le i \le n$} & sel is a selector
      -> {($s_i$,$v_i$) | ($s_i$,$v_i$) in st & $s_i \ne$sel}
         union {(sel,v)};
   errorunion(structurep(st,"append"),
              selectorp(sel,"append")).

select(st,sel) ≡
   st=<> & sel is a selector
      -> (error,"missing","select");
   st={($s_i$,$v_i$) | $1 \le i \le n$} & sel is a selector
      -> (($s_i$,$v_i$) in st & $s_i$=sel -> $v_i$;
          (error,"missing","select"));
   errorunion(structurep(st,"select"),
              selectorp(sel,"select")).

delete(st,sel) ≡
   st=<> & sel is a selector
      -> (error,"missing","delete");
   st={($s_i$,$v_i$) | $1 \le i \le n$} & sel is a selector
      -> (($s_i$,$v_i$) in st & $s_i$=sel
            -> {($s_i$,$v_i$) | ($s_i$,$v_i$) in st & $s_i \ne$sel};
          (error,"missing","delete"));
   errorunion(structurep(st,"delete",
              selectorp(sel,"delete")).
```

where structurep and selectorp have definitions similar to that of numberp except that they verify the types structure and selector (either a string or integer), respectively.

The dataflow control operators are not as easily extended. These are the operators which implement the conditional, loop, and procedure application expressions. The result from a control operator should be equivalent to the corresponding result from an applicative language function. The control operators include switch and apply (others are defined in [AGP78]).

The switch operator is defined to have a single output value on either of two output lines (see Figure 2.3). The only exception condition detectable by the switch operator occurs if the predicate evaluates to a non-Boolean result. In this case, the entire conditional or loop expression should return an error value. Thus, the switch operator is defined by*:

```
switch(b,x) ≡ T-port : b=true -> x; ϕ.
              F-port : b=false -> x; ϕ.
              E-port : b=true or b=false -> ϕ; x.
```

The extended switch operator is used to translate an if expression

*Note that this definition can still be implemented using the original 2-port definition; but this requires that a single extended switch operator be replaced by one predicate and three 2-port switch operators.

```
(if p(x) then f(x) else g(x))
```

as shown in Figure 3.1. This translation yields results similar to those from the applicative condition functional form. If the predicate returns a non-Boolean value, the entire if expression will return an error value derived from the predicate result and any named inputs (variables) used within the if expression that are error values. Thus the semantics of the above if expression are

```
(if p f g):x ≡ p(x)=true -> f(x);
               p(x)=false -> g(x);
               errorunion(Booleanp(p(x),"if"),x).
```

Loop expressions are also translated slightly differently than in Section 2.3.4 when using the new switch definition. Consider the loop expression

```
(initial x <- f(xø);
 while p(x) do
       new x <- g(x);
 return h(x))
```

The translation for this loop is given in Figure 3.2. This translation yields results comparable to the applicative while functional form. In particular, it corresponds to the form:

```
(return-clause • (while predicate-part loop-body) •
     initial-part) : x.
```

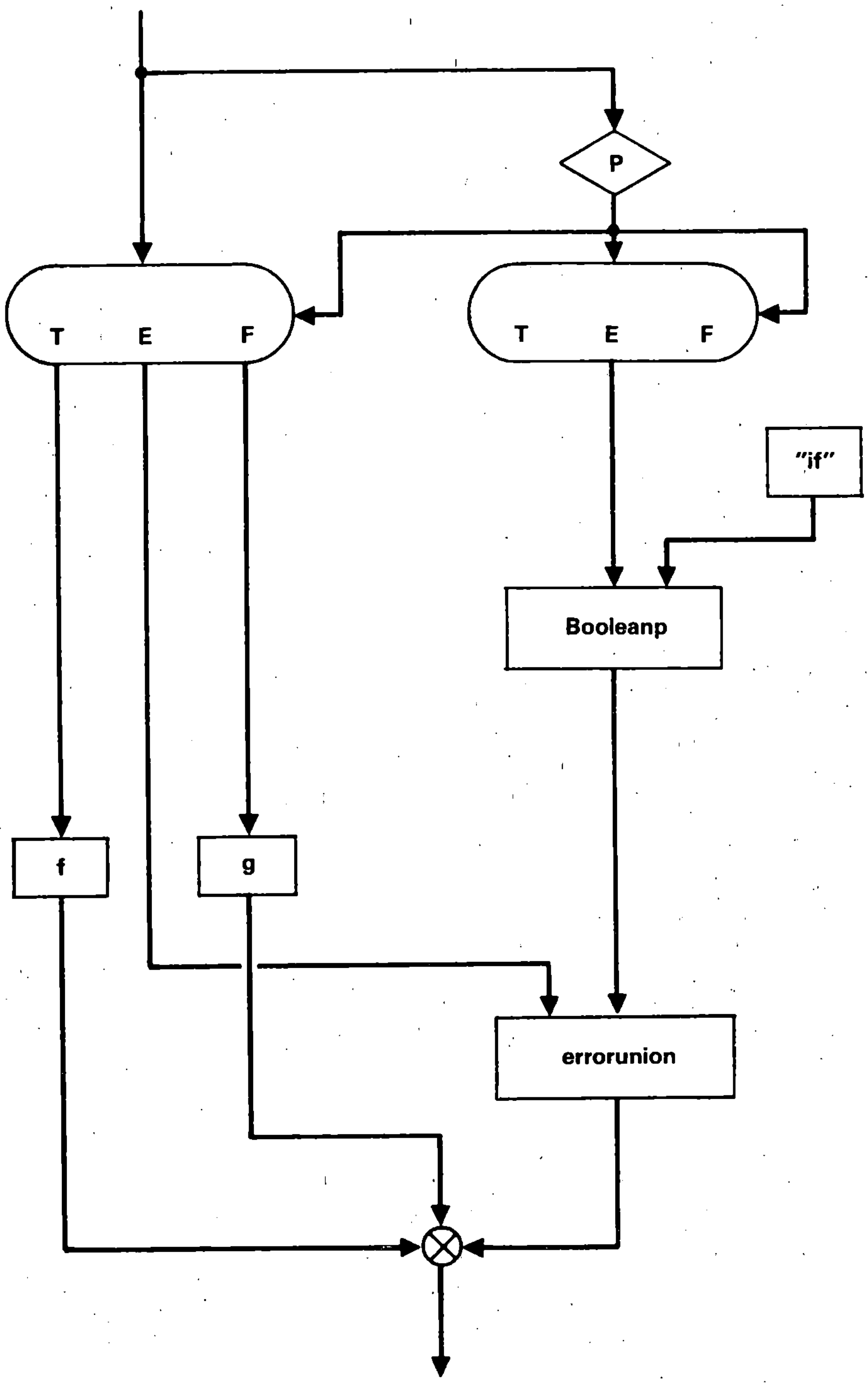

Figure 3.1 Translation of a Conditional Expression with Exception Conditions

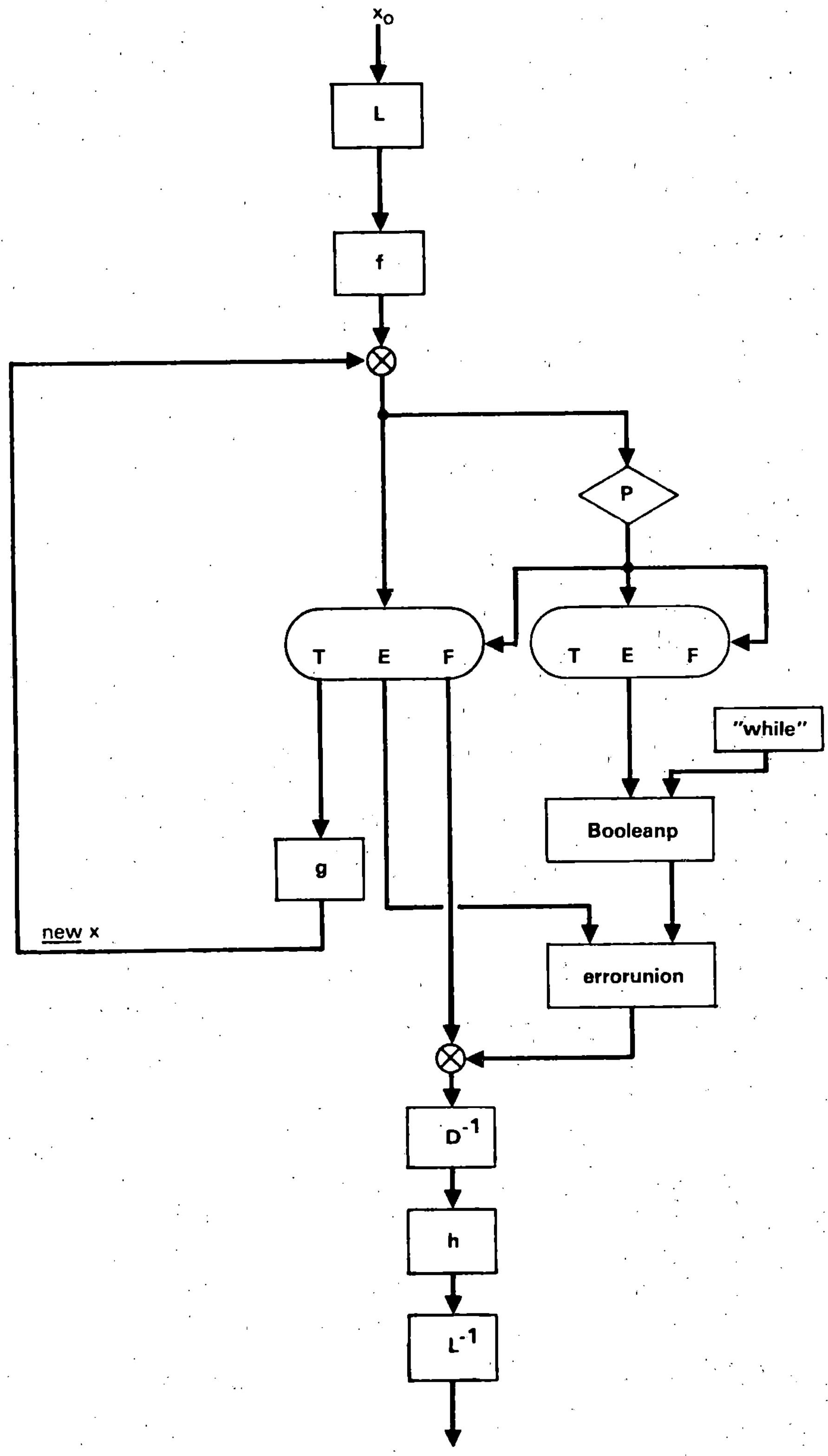

Figure 3.2 A Loop Expression with Exception Conditions

The dataflow apply operator is defined by

```
apply(p,x) ≡ p is a procedure definition -> p(x);
             errorunion(procedurep(p,"apply"),x).
```

This definition corresponds closely to that for the conditional expression -- the controlling value (here the procedure definition) only is checked for an exception. The result of this definition is that a procedure application is always attempted whenever a valid procedure definition is received, regardless of the state of the arguments.

## 3.4 Defining Exception Handlers

The previous sections introduced the concept of an error value and defined the behavior of operators with respect to error values. The purpose of this section is to introduce language syntax to make it easy for a programmer to use the underlying exception mechanisms. The new syntax corresponds to an exception handler.

An exception handler in this model combines two purposes: exception detection, represented by an exception condition, and exception recovery, represented by a recovery expression. Each exception handler is associated with an expression and handles exceptions only within the context of that expression.

A precondition handler is invoked before the associated expression is executed. The handler comprises a predicate and a simple-expression. If the predicate evaluates to true, then the precondition is satisfied and the associated expression is executed. If the predicate evaluates to anything but true, the precondition fails and the simple-expression within the handler is executed in place of the associated expression. Thus the precondition handler may perform domain checking for the associated expression and return a domain error value or a substitute result if any input condition is not satisfied.

The postcondition handler is invoked after the associated expression has executed. (It is not invoked if the precondition handler was executed in place of the associated expression.) The handler comprises a predicate and a simple-expression; it may not reference any values defined within the associated expression, but may reference the result of the associated expression using the symbol "@". (If the associated expression returns n values ($n \geq 1$), these may be referenced using "@[1]" thru "@[n]".) If the predicate returns true, then the postcondition is satisfied and the result from the associated expression is returned as the final result. If the predicate does not return true, then the postcondition fails and the result obtained by evaluating the simple-expression is returned as the final result. Thus, the postcondition handler is expected to perform range checking for the associated expression.

The syntax for an Id expression and handlers is:

```
expression ::= simple-expression
                  [precondition-handler]
                  [postcondition-handler] .

precondition-handler ::=
     "PRECONDITION" Boolean-expression
        "ELSE" simple-expression .

postcondition-handler ::=
     "POSTCONDITION" Boolean-expression
        "ELSE" simple-expression .

simple-expression ::= block-expression |
                      conditional-expression |
                      loop-expression |
                      arithmetic-expression |
                      "(" expression ")" .
```

This association of handlers with an expression is simply a syntactic extension to Id -- no new semantic operations are required. A similar extension is easily defined for an FP system. However, we first consider the semantics for the new Id syntax.

An Id expression which utilizes both a precondition handler and a postcondition handler may be represented as

$$\begin{array}{l} \text{expr}(x)\ \underline{\text{precondition}}\ p_1(x)\ \underline{\text{else}}\ e_1(x) \\ \qquad\quad \underline{\text{postcondition}}\ p_2(@,x)\ \underline{\text{else}}\ e_2(@,x) \end{array}$$

The expression is translated as follows:

```
(t1 <- p1(x);
 b1 <- (if type(t1)="Boolean" then t1 else false);
 return (if b1
          then (@ <- expr(x);
                t2 <- p2(@,x);
                b2 <- (if type(t2)="Boolean"
                          then t2
                          else false);
                return (if b2
                          then @
                          else e2(@,x)))
          else e1(x)))
```

The most common postcondition handler recovers from specific exceptions that may be raised within the associated expression. For instance, a post condition handler which recovers from either an overflow or a bad_format (illegal number representation) might be

```
expr postcondition
       (if errorp(@) then errorclass(@)≠"overflow"
                       and errorclass(@)≠"bad_format"
                     else true)
     else simple-expression
```

However, this type of handler is awkward to write and tends to obscure the actual "normal" computation. Thus, special syntax has been introduced to handle the common situation and is known as an error class handler.

```
postcondition-handler ::=
     "POSTCONDITION" Boolean-expression
        "ELSE" simple-expression |
     error-class-handler .

error-class-handler ::=
     "ERROR" {class-handler} [others-handler] .
```

```
class-handler ::=
     "CLASS" class-clause {"OR" class-clause}
        "->" simple-expression .

class-clause ::=
     string |
     string parameter {"AND" string parameter} .

parameter ::= "(" "@" "[" positive-integer "]" ")" .

others-handler ::=
     "OTHERS" "->" simple-expression .
```

An example of an error class handler follows:

```
expr(x) error class str1 -> e1(@,x)
              class str21 or str22 or ... or str2m
                 -> e2(@,x)
              ...
              class strn -> en(@,x)
              others -> en+1(@,x)
```

This expression is translated as follows:

```
(@ <- expr(x);
 return (if errorp(@)
             then (if errorclass(@)=str1 then e1(@,x)
                   elseif errorclass(@)=str21
                       or errorclass(@)=str22
                       or ...
                       or errorclass(@)=str2m
                          then e2(@,x)
                   ...
                   elseif errorclass(@)=strn
                          then en(@,x)
                   else en+1(@,x))
             else @))
```

Thus, the error class handler is useful when one or more particular exceptions may be raised by the associated expression. This form simplifies the predicate and expressions of the postcondition. The others-clause is executed when an exception is detected and is not handled by a class-clause.

If more than one value is returned from the associated expression, the class-clause must contain a parameter for each string. The parameter denotes the particular value which is to be compared with the string. Thus the class-clause

<u>class</u> $str_1$(@[1]) <u>and</u> $str_2$(@[2]) <u>and</u> $str_3$(@[4])

would translate to the following conditional

```
(if errorp(@[1]) and errorp(@[2]) and errorp(@[4])
   then errorclass(@[1])=str1 and errorclass(@[2])=str2
      and errorclass(@[4])=str3
   else false)
```

The exception handling syntax for an FP system is defined using two functional forms. The first form is equivalent to an Id general exception handler, and the second form contains only a postcondition handler. (Note that the problem of labelling the result from the simple expression is solved in a much cleaner fashion in FP. This results from using selectors and not labels to reference values.)

$\{(p_1,e_1);expr;(p_2,e_2)\}:x \;=\; (p_1:x)=T \rightarrow \{expr;(p_2,e_2)\}:x;\; e_1:x.$

$\{expr;(p_2,e_2)\}:x \;\equiv\; (p_2:<(expr:x),x>)=T \rightarrow expr:x;\; e_2:<(expr:x),x>.$

The first functional form is also equivalent to

$$(eq \cdot [p_1,\bar{T}] \to \{expr;(p_2,e_2)\};e_1):x$$

and the second functional form is equivalent to

$$((eq \cdot [p_2,T] \to 1;e_2) \cdot [expr,Id]):x$$

## 3.5 Example

This section presents an example which uses the exception handling capabilities of Id. Consider the problem of finding a root of a real function f. This may be also be expressed as "Find a value x such that f(x)=Ø." The usual approach to solving this problem is to use an algorithm such as Newton's Method which has quadratic convergence for monotonic functions. However, the basic assumptions for use of Newton's Method may not always be satisfied, and then another more stable, but slower algorithm will have to be used.

The basic assumptions behind Newton's Method are:

1. the function f, the first derivative f', and the second derivative f" are continuous and bounded on the interval of interest containing the zero of the function, and

2. the initial approximation xØ is sufficiently "close" to the root of the function that the algorithm will converge on that root.

1. It first tries Newton's Method. This function will return a condition code, cond1, specifying the state of the answer and the result, ans1. The condition code may be any of the following:

    1. "good" - the procedure calculated a close approximation to the root.

    2. "convergence" - the method was iterated for m times but failed to converge.

    3. "derivative" - problems were encountered when using the procedure fprime.

    4. "divergence" - the method started to diverge, that is, the error between successive iterations started to become larger.

    The value of ans1 is meaningful only for the first case.

2. If Newton's Method failed due to either derivative or convergence problems, root tries the secant method. The secant method is not tried when Newton's Method diverges since the secant method will also diverge (for most functions). The secant method will return a condition code, cond2, set to "good", "convergence", or "divergence". The value of ans2 is meaningful only for the first case.

3. Finally, if both Newton's Method and the secant method have failed (or Newton returned a "divergence" condition code), the method of false position is tried. This method is the most stable, i.e., the most likely to succeed, and may work when the others have failed. However, this method is tried last since it is also the slowest to converge to an answer. The result returned from this method, and thus also the procedure root, will either be a close approximation of the root (with a condition code set to "good") or a result with a condition code set to "convergence" or "divergence".

There are several possible exception conditions which may arise during the execution of the procedure. Two of these conditions involve passing incorrect arguments for either eps or max. Consider the case when a string argument is passed for the eps parameter. The first if expression

(<u>if</u> eps $\leq$ Ø.Ø <u>then</u> 1.ØE-6 <u>else</u> eps)

will return the error value (error,"domain","$\leq$"). This error value will propogate through Newton and be returned as the result for both ansl and condl. The value of condl, being an error value, will then be returned as the result of the return clause and thus from root also. However, this exception should not hamper execution of the procedure since the default value is an acceptable substitute. This substitution may be implemented using the exception handling mechanism:

e <- eps <u>precondition</u> eps>Ø.Ø <u>else</u> 1.ØE-6;

Other possible exception conditions involve passing incorrect arguments for f, fprime, or xØ, possible exception conditions arising from the use of either f or fprime, and possible exception conditions, e.g., "overflow" or "underflow", arising from the use of particular numeric values within Newton, secant, or false_pos. An incorrect argument for fprime should not cause termination of root, as either secant or false_pos could be substituted for Newton since these procedures do not require the use of fprime. (Note that this may be accomplished by returning a condition code of "derivative_error".) The remaining exception conditions should be handled within the individual procedures.

A better implementation of root would utilize the exception handling mechanism to recover from these possible exceptions. In addition, the "condition code" should be encoded within the result using error values. Such an implementation follows:

```
procedure root (f,fprime,xØ,eps,max)
   ( e <- eps precondition eps > Ø.Ø else 1.ØE-6;
     m <- max precondition max > Ø else 1ØØ;
     return Newton(f,fprime,xØ,e,m)
              error
                class "divergence" -> false_pos(f,xØ,e,m)
                others -> (secant(f,xØ,e,m)
                             error
                               others -> false_pos(f,xØ,e,m)))
```

## 3.6 Summary

This chapter has presented the basic model for exception handling and recovery in applicative systems. The principle elements of this model are:

1. the definition of an error value;
2. the definition of four new operators to create and manipulate error values;
3. the extension of all existing operators to total functions; and
4. the presentation of syntax for exception handling.

An example demonstrating the use of the model was also presented.

# Chapter 4

# DATAFLOW STREAMS

## 4.1 Introduction

The preceding chapter presented the basic model for applicative systems; this chapter extends the model to encompass dataflow streams. Dataflow streams [Lan65, Kah74, Bur75, Wen75, KaMa77, AGP78] provide a means of referencing an ordered collection of values, not all of which need to be present at any given instant of time. (Currently, there is no FP counterpart to a dataflow stream, though some work has been done in this area [Min77].) Thus streams differ from structures in that part of the stream may be used for further computation before the rest of the stream has been computed. Also, stream elements do not have to be computed in time order, and the potential for asynchrony is thus increased. Consider the following program:

```
X <- (initial x <- xø;
      while p(x) do
         new x <- g1(x);
      return all x);
y <- (initial y <- yø; z <- <>; i <- 1;
      for each x in X do
         temp <- g2(x);
         new z <- z + [i]temp;
            {equivalent to new z[i] <- temp}
         new i <- i + 1;
      return z);
```

This program also could have been written using structures:

```
x,n <- (initial x <- x_0; z <- <>; n <- 0;
        while p(x) do
           temp <- g_1(x);
           new x <- temp;
           new z <- z + [n+1]temp;
             {equivalent to new z[n+1] <- temp}
           new n <- n+1;
        return z,n);
y <- (initial y <- y_0; z <- <>; i <- 1;
      while i<n do
         temp <- g_2(x[i]);
         new z <- z + [i]temp;
             {equivalent to new z[i] <- temp}
         new i <- i + 1;
      return z);
```

If the while predicate p(x) eventually returns false for some value of x, both programs compute the same result. The difference lies in the time taken to compute the answer. A structure is returned from a loop only when every element has been inserted into the structure. However, each element of a stream is available from the loop as soon as it has been computed. For example, the first loop may be computing the third element of the stream X (equivalent to x[3] in the structure example) while the second loop is using the first element of the stream X (equivalent to x[1]) to compute y[1]. Thus the total elapsed time spent in the two loops may be reduced by as much as 50%. [AGP78] presents two examples where the asynchrony of streams may decrease the time complexity even more dramatically. However, this added asynchrony causes only minor problems for exception handling and recovery.

Exception handling with streams causes some difficulty since a stream value may not exist as a single entity at any one point in time. This implies that a stream cannot be represented by a single error token, and that an exception associated with a single stream element cannot, by itself, cause the entire stream to become an "error stream". Rather, a stream has a dual character: it is the union of its parts, but it is also an entity in its own right. While the elements of a stream may be error values, the stream itself has a distinct _status_, "normal" or "error".* This status is associated with the entire stream and not with any individual element of the stream. (For implementation, this status may be carried by the end-of-stream token. See [AGP78].)

The presence of an error status, rather than a normal status, usually indicates an unexpected end of the stream caused by either a program bug (e.g., taking the _rest_ of an empty stream) or an exception condition detected while forming the stream (e.g., receiving a non-Boolean result from a loop predicate while generating the stream). In the

--------------------

*This is slightly different from the semantics for structures. A structure may contain error values as data elements, but an "error structure" is a single (error) value.

following, a stream X with status s will be represented as $X_s$ where s is either "normal" or (error,y,z). The stream $X_s$ comprising stream elements $x_1$, ..., $x_n$ will be denoted $[x_1,\ldots,x_n]_s$. The empty stream $[]_s$ contains no elements (but it does have an end-of-stream token and a status).

## 4.2 Simple Operators

The stream functions and predicates, originally defined in [AGP78], must be extended for exception conditions on streams. In addition, several new functions and predicates are introduced to handle exception conditions on streams. Note that issues concerning domain exceptions with stream operators do not arise since these operators receive only streams as arguments and streams are not typed by the elements they contain.

The following definitions hold for all stream values $X = [x_1,\ldots,x_{nx}]_{sx}$ and $Y = [y_1,\ldots,y_{ny}]_{sy}$, and simple values x, y, and z:

```
errorstream(x) ≡ x=(error,y,z) -> []x;
                  [](error,"domain","errorstream").

errorp(X) ≡ sx=(error,y,z) -> true; false.

status(X) ≡ sx.

est(x) ≡ {generate an end-of-stream token
          with status equal to x}

empty(X) ≡ nx≥1 -> false; true.
```

```
first(X) ≡ $nx \geq 1$ -> $x_1$;
           sx≡(error,y,z) -> sx;
           (error,"empty","first").

rest(X) ≡ $nx \geq 1$ -> $[x_2,\ldots,x_n]_{sx}$;
          sx≡(error,y,z) -> $[\,]_{sx}$;
          $[\,]$(error,"empty","rest").

cons(y,X) ≡ $[y,x_1,\ldots,x_n]_{sx}$.

equalize(X,Y) ≡ nx<ny -> $[x_1,\ldots,x_{nx}]_{sx}$ &
                                $[y_1,\ldots,y_{nx}]_{sx}$;
                nx>ny -> $[x_1,\ldots,x_{ny}]_{sy}$ &
                                $[y_1,\ldots,y_{ny}]_{sy}$;
                sx="normal" or sy="normal" ->
                           $[x_1,\ldots,x_{nx}]$"normal" &
                                $[y_1,\ldots,y_{ny}]$"normal";
                $[x_1,\ldots,x_{nx}]$errorunion(sx,sy) &
                   $[y_1,\ldots,y_{ny}]$errorunion(sx,sy).

extend(X,x,Y,y) ≡ nx<ny -> $X'_s$ & $[y_1,\ldots,y_{ny}]_s$;
                  nx>ny -> $[x_1,\ldots,x_{nx}]_s$ & $Y'_s$;
                  $X'_s$ & $Y'_s$.

      where $X'_s$ = $[x'_1,\ldots,x'_{ny}]_s$,
                 $x'_i = x_i$ for $1 \leq i \leq nx$,
                 $x'_i = x$ for $nx+1 \leq i \leq ny$,
            $Y'_s$ = $[y'_1,\ldots,y'_{nx}]_s$,
                 $y'_i = y_i$ for $1 \leq i \leq ny$,
                 $y'_i = y$ for $ny+1 \leq i \leq nx$
      and s ≡ sx="normal" -> sy;
              sy="normal" -> sx;
              errorunion(sx,sy).
```

## 4.3 Control Operators

The stream control operators must also be redefined to account for exception conditions. These operators are used for translating the conditional, loop, and procedure application expressions. In all cases, these operators preserve the error status associated with input streams or produce an error status on output streams to represent exception conditions. However, all information contained within the stream is used and asynchrony is maximized.

The translation for conditional expressions containing streams is similar to that given in Section 3.3.2, Extensions for Dataflow. Consider

```
(if p(x) then f(Y) else g(Y))
```

Let Y represent a stream and f(Y) and g(Y) return streams. Then a stream must also be returned if p(x) returns a non-Boolean result. The result returned will be [](error,"Boolean","if"). The translation for the if expression is given in Figure 4.1.

The translation for a simple while loop containing streams is also similar to that given in Section 3.3.2. An empty error stream is returned if the loop predicate returns a non-Boolean result. Thus the loop expression

```
(initial X <- f(xø);
 while p(X) do
       new X <- g(X);
 return h(X))
```

has the translation given in Figure 4.2. The more complex stream looping constructs are discussed in the next section.

Finally, the operators for procedure application must also be extended to handle stream exceptions properly. The problem with streams in this context arises from the

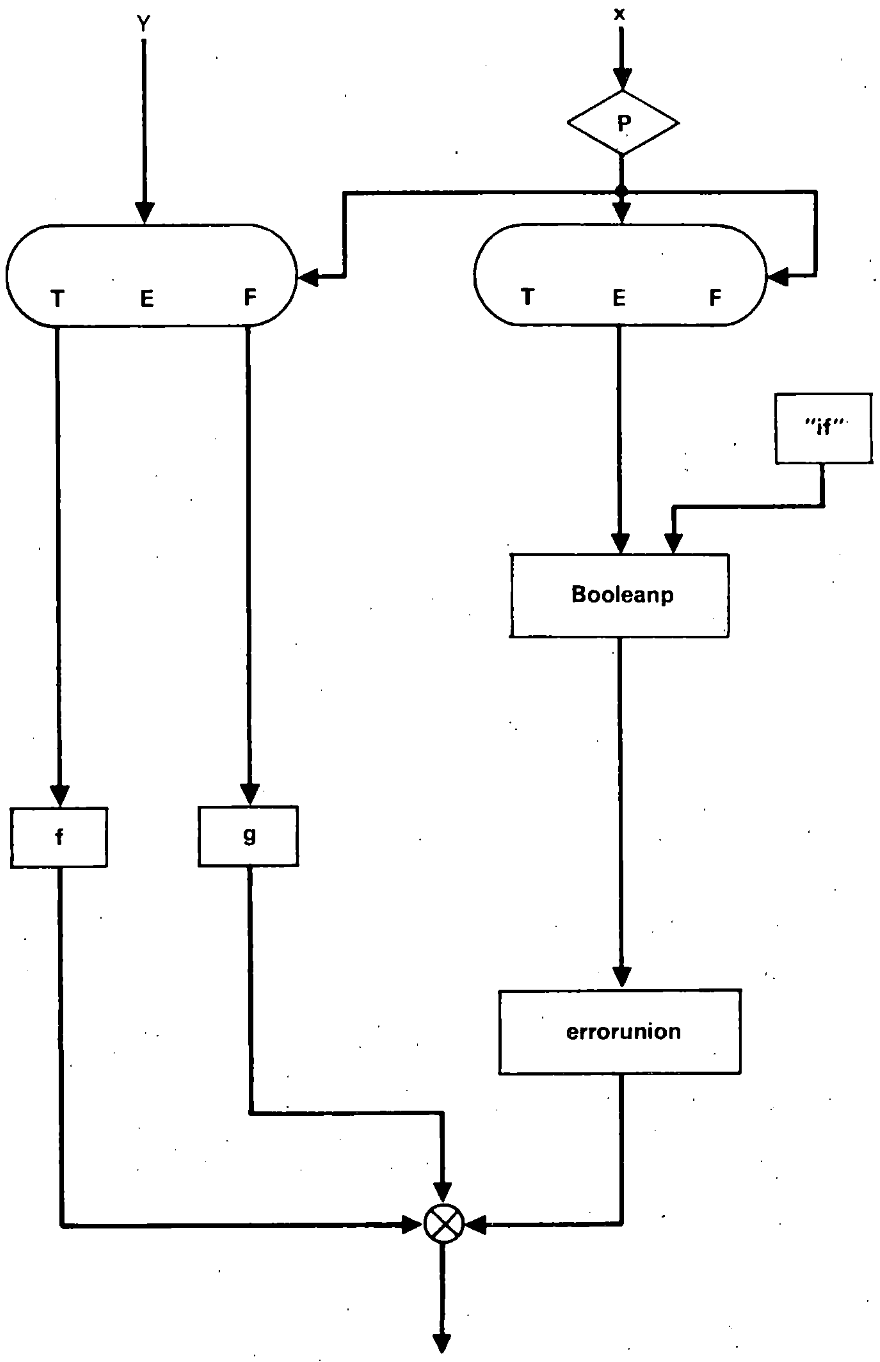

Figure 4.1 Translation of a
Streamed Conditional Expression

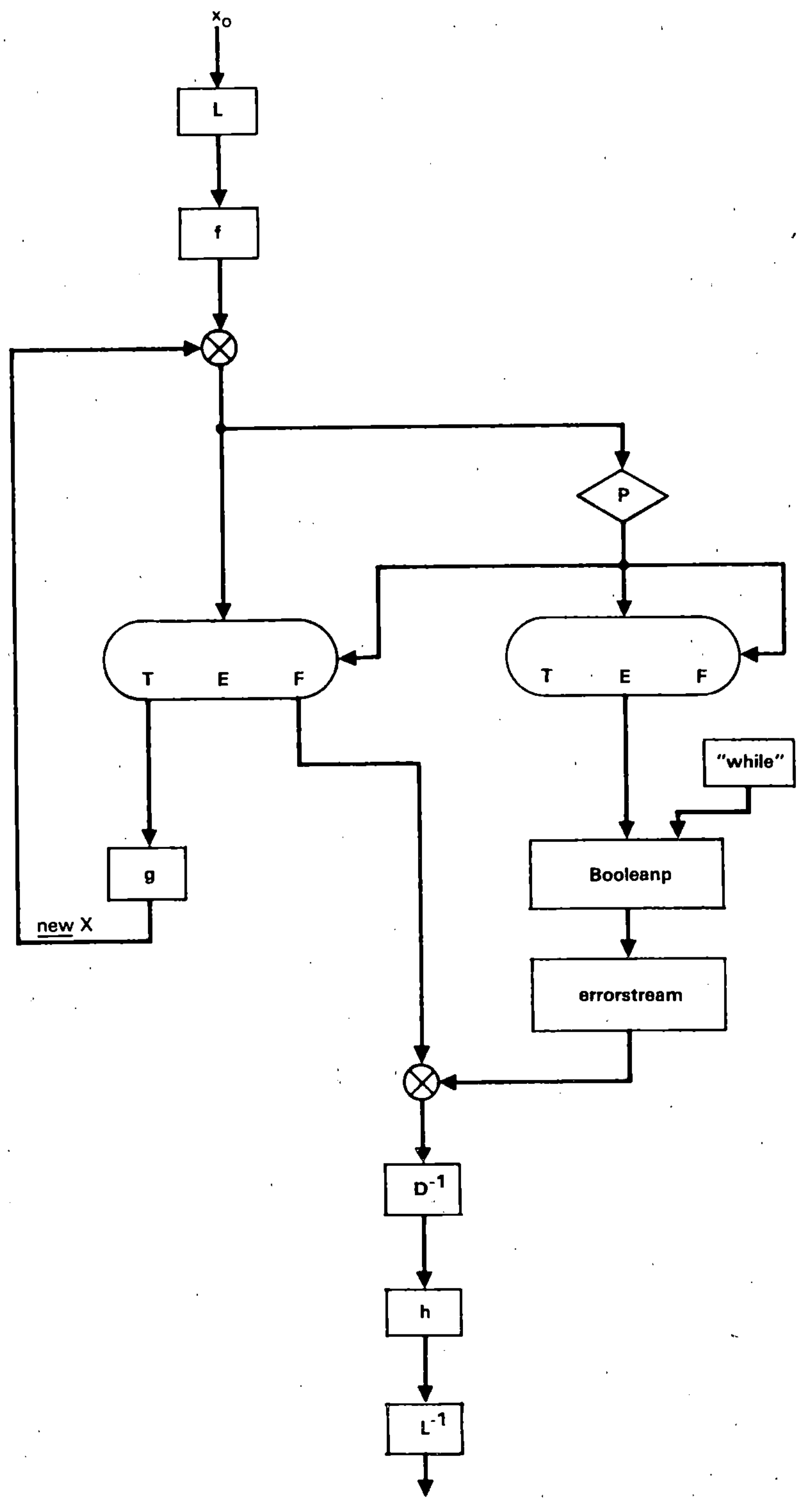

Figure 4.2 Translation of a Streamed Conditional Expression

restriction that a stream may not be an element of a structure. (Recall from Section 3.3.2 that a single argument is passed to the procedure being invoked. If a stream could be part of this argument, no further semantics would be needed; however, the current Id semantics do not allow this.) Thus each stream argument needs to be handled independently of all other arguments. However, the number of stream arguments may not equal the number of stream parameters. It is the task of the apply operator to resolve this conflict. Apply discards any extraneous stream arguments and generates error streams for any missing stream arguments. Similar actions are taken for the number of stream results actually produced versus the number expected.

## 4.4 Stream Loop Expressions

Several new loop constructs are defined for dataflow streams. These constructs were introduced to effectively utilize the potential asynchrony present in the definition of a stream. They include a for each construct, a return all construct, and a for each - while construct.

### 4.4.1 The For Each Construct

The for each construct divides the stream into its component elements. Each element is sent to a distinct iteration of the loop, the ith element to the ith iteration. The use of this construct produces potentially much greater asynchrony than may be obtained through the use of a while loop. For example, let stream X = [1,2,3,4,5] in the following loop:

```
(initial sum <- 0;
 for each x in X do
    new sum <- sum + x;
 return sum)
```

The result from the loop is the sum of the individual components of X, i.e., 15.

This construct is translated using the E operator. This operator receives a single stream value as input and generates many simple (non-stream) values as output; the ith component of the input stream produces the ith output value which is directed to the ith iteration of the loop. Let $X=[x_1,...,x_n]_s$, then

$$E(X) \equiv \{x_1,...,x_n,est(s)\}$$

The translation of the loop given above is presented in Figure 4.3. There are no exception conditions associated with this construct since it is based solely on the structure of the stream arguments.

## 4.4.2 The All Construct

The all construct collects a single value from each iteration of a loop to form a stream. The ith iteration will supply the ith component of the resulting stream. The

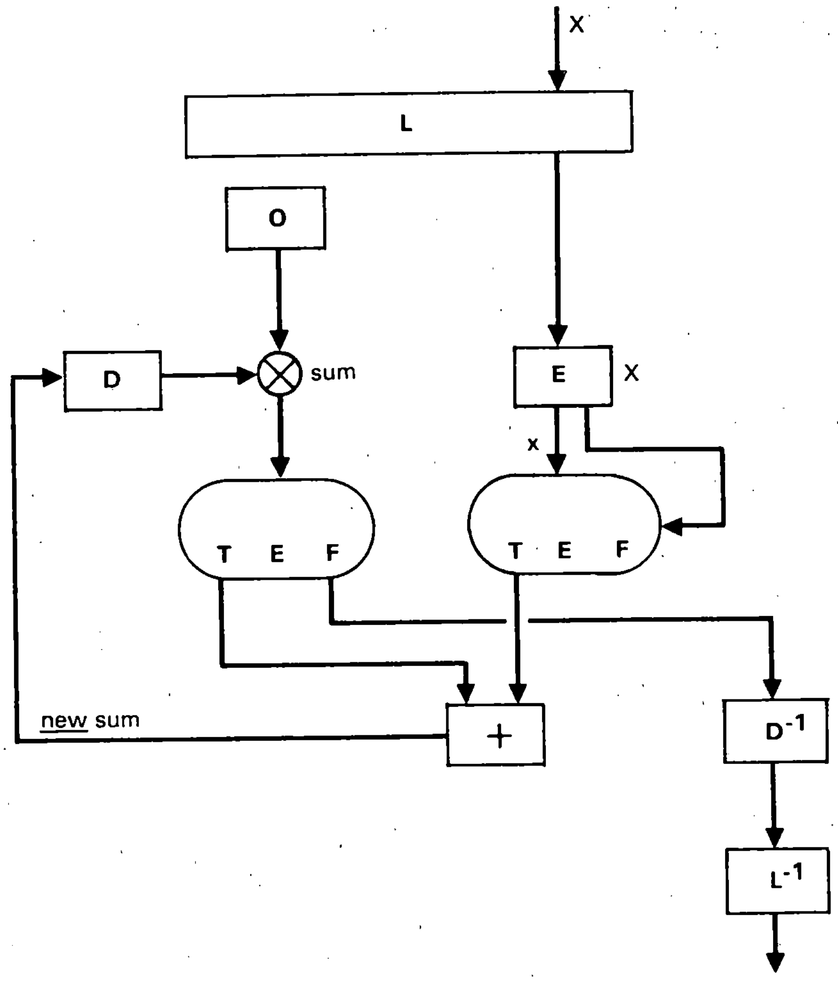

Figure 4.3 Translation of a For Each Loop Expression

use of this construct produces much greater asynchrony than may be obtained by constructing a stream using most other means. For example, consider the stream X equal to [1,2,3,4,5] and the loop

```
(initial sum <- 0;
 for each x in X do
    nextsum <- sum + x;
    new sum <- nextsum;
 return sum, all nextsum)
```

The results from the above loop are 15 and [1,3,6,10,15], respectively.

This construct is translated using the $E^{-1}$ operator:

$$E^{-1}(\{x_1,\ldots,x_n,s\}) \equiv [x_1,\ldots,x_n]s.$$

The translation of the loop given above is presented in Figure 4.4. There are no exception conditions directly associated with this construct; however, the output stream has an error status "inherited" from the end-of-stream token.

## 4.4.3 The For Each - While Construct

The for each - while construct combines the aspects of the for each construct with that of the while loop. This construct directs elements of the stream argument to distinct iterations of the loop until either the while predicate returns false or the stream terminates. The only

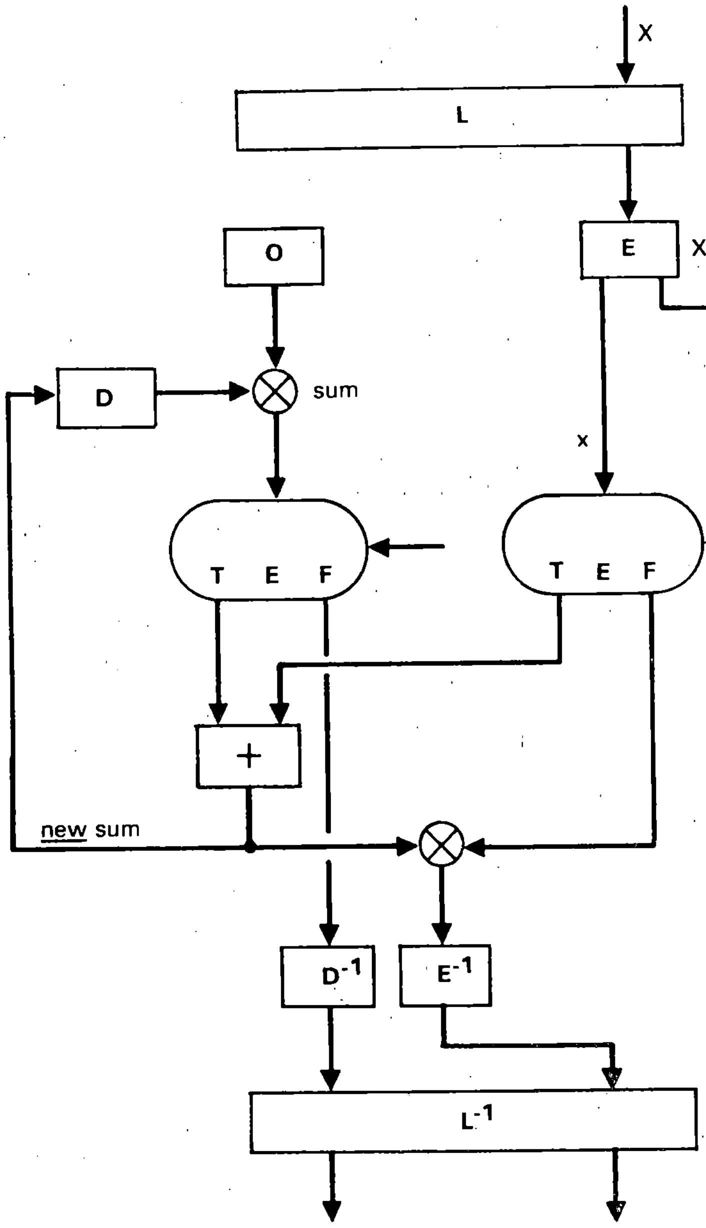

Figure 4.4 Translation of a Return All Clause

exception condition associated with this construct is a non-Boolean result from the while predicate. However, if the loop terminates due to encountering the end of the input stream, the error status of the input stream is passed to any output streams.

```
(initial x <- a;
 for each y in Y while p(x,y) do
     new x <- f(x,y);
 return x, all x)
```

The translation of this loop is presented in Figure 4.5. This translation differs slightly from that given in [AGP78]; the difference results from the requirement that the output stream "inherit" the status of the input stream when the loop terminates because of the end of the input stream. If the loop terminates because the while clause returns false, the output stream will have a "normal" status.

## 4.5 Example

The previous sections have defined the various operators and constructs for stream programming; this section presents an example which uses streams and examines possible exception conditions. The example is the procedure sieve which implements the Sieve of Eratosthenes algorithm to generate prime numbers. The input argument, LIST, is a stream of integers, 2 through n, inclusive. The procedure

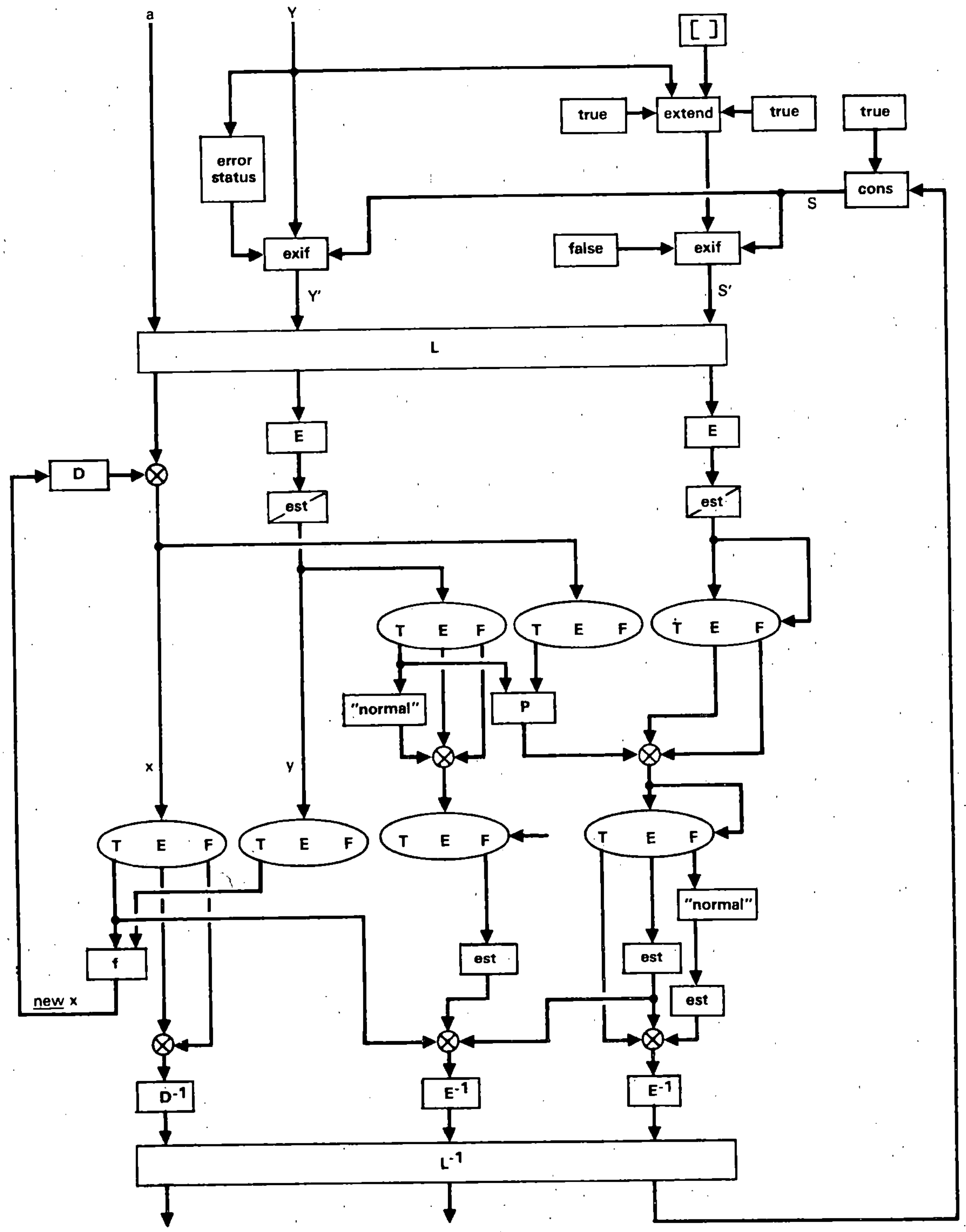

Figure 4.5  Translation of a For Each-While Loop Expression

will iteratively create sieves, each of which filters out multipliers of the first item of the LIST received as input with each iteration generating a single prime and a new LIST for input to the next iteration.

```
procedure sieve (LIST)
   (initial
      procedure partial (prime, LIST)
         (if empty(LIST) then LIST
          elseif mod(first(LIST),prime)=0
                then partial(prime,rest(LIST))
          else cons(first(list),
                   partial(prime,rest(LIST))))
    while not empty(LIST) do
       prime <- first(LIST);
       new LIST <- partial(prime,rest(LIST))
    return all prime)
```

There are two possible exception conditions which may arise in the above procedure: the stream LIST may have an error status, or one of the elements of LIST may not be an integer value. Since the procedure partial returns LIST whenever LIST is empty (i.e., the end-of-stream token), the new LIST will have the same error status as the original. However, this status is not inherited by the resulting stream of prime numbers (the loop predicate "empty" always returns either true or false). If the error status of LIST is to be passed on to the stream of primes, the loop predicate should be changed to the following:

```
(if empty(LIST)
   then (if errorp(LIST)
           then status(LIST)
           else false)
   else false)
```

If an element of LIST is not an integer, then the expression mod(first(LIST),prime)=Ø also returns an error. The result of this is to cause the conditional expression, and thus the procedure partial, to return a stream which contains neither the stream element nor any succeeding elements. In addition, the stream will have an error status resulting directly from use of the mod function.

Coding errors may also introduce exception conditions. For instance, the procedure partial might have been written as:

```
procedure partial (prime,LIST)
    (item <- first(LIST)
     return (if mod(item,prime)=Ø
                then partial(prime,rest(LIST))
                else cons(item,
                          partial(prime,rest(LIST)))))
```

The problem with the above code is that there is no check to verify that there is at least one element in LIST before the execution of first. The result from executing partial is similar to the case of an illegal value in the LIST stream -- a stream is returned with an error status.

Other coding errors do not necessarily terminate execution. Consider the following code:

```
procedure partial (prime,LIST)
    (item <- first(LIST);
     REMAINDER <- partial(prime,rest(LIST));
     return (if mod(item,prime)=Ø
                then REMAINDER
                else cons(item,REMAINDER)))
```

This procedure does not terminate since the recursion is not dependent on either the structure of the stream LIST nor the validity of any of the stream elements.

## 4.6 Summary

This chapter has extended the basic model for exception handling and recovery to include dataflow streams. This was accomplished by associating a status, either "normal" or an error value, with each stream, and defining the operators errorstream, errorp, status, and est to manipulate that status. In addition, the Id stream constructs were extended to define their behavior when an exception occurs and to allow an output stream to "inherit" the status of an input stream. Finally, an example was given to discuss the exception capabilities of the new definitions.

# Chapter 5

# EVALUATION

## 5.1 Introduction

The proposed model for detecting and handling exception conditions has two major differences from existing systems: the use of a value-based, applicative semantic model such as FP or dataflow, and the definition of an error value. This section will evaluate the usefulness of the proposed model by comparing it to existing exception systems. Several illustrative examples are considered.

## 5.2 Comparison to Backward Error Recovery

Backward error recovery was discussed in Section 1.2.2. If an exception is detected during execution of a recovery block or results from the block fail an acceptance test, an alternate computational method is used to determine the result of the recovery block. This same behavior is easily simulated by the proposed model.

Consider the following recovery block:

```
ensure p(x)
by f(x)
else by g(x)
else error
```

This block may be translated to the following:

```
(f(x)
 postcondition p(@)
          else (g(x)
                postcondition p(@)
                         else error("failed",x)))
```

## 5.3 Comparison to Forward Error Recovery

Forward error recovery was discussed in Section 1.2.3. When an exception is detected, the appropriate handler is invoked. Handlers are usually constrained in their actions to simple value substitution for erroneous data (although the calculation of the substitute values may be rather involved) or changing the flow of control. Such actions are easily simulated with the proposed model, although the structure of the resulting program may be quite different.

Consider the simple example of a multiplication operation raising an underflow exception. It is often desired to substitute zero for the value of the computation and then continue. This is accomplished by placing an "underflow" handler following each such possible multiplication operation. For example,

```
z <- ( a*b error class "underflow" -> Ø.Ø ) + c
```

Other examples may not translate as easily into this model, particularly if the forward error recovery system is also used to direct the program's flow of control. This difficulty stems in part from the applicative language basis of the model and the inability to express directly control flow concepts in the manner of von Neumann languages. Consider the following CLU procedure:

```
    sum_stream =
        proc (a:array) returns (int)
          signals (OVERFLOW,
                   UNREPRESENTABLE_INT(string),
                   BAD_FORMAT(string))
        sum: int := Ø
        i: int := 1
        while true do
          sum := sum + get_number(a,i)
          i := i+1
        end % while %
        except
          when MISSING_ELEMENT :
                return (sum)
          when UNREPRESENTABLE_INT(f: string) :
                signal UNREPRESENTABLE_INT(f)
          when BAD_FORMAT(f) :
                signal BAD_FORMAT(f)
          when OVERFLOW :
                signal OVERFLOW
        end % except %
        end sum_stream
```

The equivalent dataflow procedure is:

```
procedure sum_stream (a)
   (initial
      sum <- Ø;
      i <- 1;
      more <- true;
   while more do
      temp <- sum + get_number(a,i);
      new sum <-
         temp error class missing_element -> sum
                    class unrepresentable_int
                       or bad_format
                       or overflow -> @
                    others ->
                       error("failure",
                             <1:"unhandled exception",2:@>);
      new i <- i+1;
      new more <-
         true precondition not(errorp(temp)) else false;
   return sum)
```

The dataflow translation differs considerably from the original CLU procedure. This occurs since the CLU procedure used the exception handlers to abort the while loop when an exception is detected; this is particularly noticeable with the use of the missing_element exception to force a "normal" termination of the loop. However, dataflow exception handlers are not capable of directly expressing such flow of control concepts, nor does it seem desirable to do so. Therefore, the flow of control information must be made explicit by use of the variable "more". Note that the dataflow translation has also accounted for the CLU default exception handling by defining the "failure" exception.

## 5.4 Comparison to Mixed Strategies

Several proposals for a mixed strategy have been presented. The principle motivation behind such a strategy is the idea that a forward error recovery method is capable of handling foreseen exception conditions but is inadequate for handling residual design faults within a program block. However, these design faults may be easily handled by a recovery block using an alternate computational method.

Since the proposed model is capable of simulating both backward and forward error recovery, it is capable of simulating a mixed strategy system. An example of this mixed strategy occurs in the procedure root, Section 3.5. If an exception occurs within procedure Newton, an alternate computational method is employed, but choice of which method to use is influenced by information obtained during the execution of procedure Newton. Thus, root resembles a recovery block, but utilizes information that is present only in a forward error recovery system.

## 5.5 Conclusions

The proposed model solves the original problem: to design a mechanism for handling exception conditions within an applicative system. In addition, it exhibits certain desirable traits:

1. separation of exception handling -- the code for handling exceptions is distinct from the code for "normal" programming.

2. conceptual distance -- the code for handling an exception is conceptually close to the code which may detect the exception. This is accomplished by maintaining a close physical distance; the handler may immediately follow the expression which raises the exception.

3. efficiency -- detecting and raising an exception imposes very little overhead on a user program since these mechanisms may be easily incorporated into the machine architecture. Handling an exception may impose both sequencing constraints and processing overhead; however, such overhead is unavoidable in a distributed processing environment.

4. ease of use -- the best argument for any exception handling mechanism is that it is easy to use and has the capabilities desired by a programmer. The proposed model satisfies both objectives.

## 5.6 Future Research

Exception handling and recovery will continue to be of interest, particularly for fault-tolerant systems. The work presented here concerns but one small area of exception handling. Other areas yet to be fully explored include the use of error values in the von Neumann model of computation, handling interprocess communication (monitor) exceptions in dataflow and other systems, handling hardware related exceptions, and aborting or suspending a nonterminating computation.

# BIBLIOGRAPHY

[AcDe79] Ackerman, W.B.; and Dennis, J.B. VAL - A value-oriented algorithmic language: Preliminary Reference Manual. MIT/LCS/TR-218, Lab. for Computer Science, M.I.T., Cambridge, Ma., June 1979.

[Ack79] Ackerman, W.B. Data flow langauges. National Computer Conf., June 1979, pp. 1087-1095.

[AnKe76] Anderson, T.; and Kerr, R. Recovery blocks in action: A system supporting high reliability. Proc. 2nd International Conf. on Software Engineering, Oct. 1976, pp. 447-457.

[ArGo77a] Arvind; and Gostelow, K.P. Some relationships between asynchronous interpreters of a data flow language. In Formal Description of Programming Concepts (E.J. Neuhold, Ed.), North-Holland, Aug. 1977, pp. 95-119.

[ArGo77b] Arvind; and Gostelow, K.P. A computer capable of exchanging processing elements for time. In Information Processing 77 (B. Gilchrist, Ed.), North-Holland, Aug. 1977, pp. 849-853.

[AGP77] Arvind; Gostelow, K.P.; and Plouffe, W.E. Indeterminacy, monitors, and dataflow. Proc. Sixth ACM Symp. on Operating Systems Principles, Operating Systems Review (ACM) 11, 5 (Nov. 1977), 159-169.

[AGP78] Arvind; Gostelow, K.P.; and Plouffe, W.E. An asynchronous programming language and computing machine. Tech. Report #114a, Dept. of Information and Computer Science, Univ. of Calif., Irvine, Ca., May 1978 (revised Sept. 1978).

[Bac73] Backus, J. Programming language semantics and closed applicative languages. Proc. ACM Symp. on Principles of Programming Languages, Oct. 1973, pp. 71-86.

[Bac78] Backus, J. Can programming be liberated from the von Neumann style? Comm. ACM 21, 8 (Aug. 1978), 613-641.

[Bic78] Bic, L. Protection and security in a dataflow system. Ph.D. Thesis, Tech. Rep. #126, Dept. of Information and Computer Science, Univ. of Calif., Irvine, Ca., Oct. 1978.

[BjDa72] Bjork, L.A.; and Davis, C.T. The semantics of the preservation and recovery of integrity in a data system. Rep. TR Ø2.54Ø, IBM, San Jose, Ca., Dec. 1972.

[Bur75] Burge, W.H. Recursive Programming Techniques. Addison-Wesley, Reading, Ma., 1975.

[Cha71] Chamberlin, D.D. The "single-assignment" approach to parallel processing. Proc. AFIPS 71 FJCC, Nov. 1971, pp. 263-269.

[Chu41] Church, A. The Calculi of Lambda-Conversion. Princeton Univ. Press, Princeton, N.J., 1941.

[CDGPS78] Comte, D.; Durrieu, G.; Gelly, O.; Plas, A.; and Syre, J.C. Parallelism, control and synchronization expression in a single assignment language. SIGPLAN Notices (ACM) 13, 1 (Jan. 1978), 25-33.

[CuFe58] Curry, H.B.; and Fey, R. Combinatory Logic, Vol. 1. North-Holland, New York, 1958.

[Dav72] Davis, C.T. A recovery/integrity architecture for a data system. Rep. TR Ø2.528, IBM, San Jose, Ca., May 1972.

[Den75] Dennis, J.B. First version of a data flow procedure language. MAC Tech. Memo. 61, Lab. for Computer Science, M.I.T., Cambridge, Ma., May 1975.

[GIMT74] Glushkov, V.M.; Ignatyev, M.B.; Myasnikov, V.A.; and Torgashev, V.A. Recursive machines and computing technology. In Information Processing 74 (J.L. Rosenfeld, Ed.), North-Holland, Aug. 1974, pp. 65-7Ø.

[Goo75] Goodenough, J.B. Exception Handling: Issues and a proposed notation. Comm. ACM 18, 12 (Dec. 1975), 683-696.

[GWG78] Gurd, J.; Watson, I.; and Glauert, J. A multilayered data flow computer architecture. Dept. of Computer Science, Univ. of Manchester, Manchester, England, July 1978.

[HOS78] Hankin, C.L.; Osmon, P.E.; and Sharp, J.A. A data flow model of computation. Dept. of Computer Science, Westfield Coll., Univ. of London, Hampstead, London, Mar. 1978.

[Kah74] Kahn, G. The semantics of a simple language for parallel processing. In Information Processing 74 (J.L. Rosenfeld, Ed.), North-Holland, Aug. 1974, pp. 471-475.

[KaMa77] Kahn, G.; and MacQueen, D. Coroutines and networks of parallel processes. In Information Processing 77 (B. Gilchrist, Ed.), North-Holland, Aug. 1977, pp. 993-998.

[Kos73] Kosinski, P.R. A data flow language for operating systems programming. Proc. ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices (ACM) 8, 9 (Sept. 1973), 89-94.

[Kos76] Kosinski, P.R. Mathematical semantics and data flow programming. Proc. Third ACM Symp. on Principles of Programming Languages, Jan. 1976, pp. 175-184.

[Kos78] Kosinski, P.R. A straightforward denotational semantics for non-determinate data flow programs. Proc. Fifth ACM Symp. on Principles of Programming Languages, Jan. 1978, pp. 214-221.

[Lan65] Landin, P.J. A correspondence between Algol-6Ø and Church's lambda-notation: Part I. Comm. ACM 8, 2 (Feb. 1965), 89-1Ø1.

[Lev77] Levin, R. Program structures for exception condition handling. Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., June 1977.

[LiSn77] Liskov, B.; and Snyder, A. Structured exception handling. Computation Structures Group Memo 155, Lab. for Computer Science, M.I.T., Cambridge, Ma., Dec. 1977.

[LMSSS78] Liskov, B.; Moss, E.; Shaffert, C.; Scheiffer, B.; and Snyder, A. CLU Reference Manual (Draft) Computation Structures Group Memo 161, Lab. for Computer Science, M.I.T., Cambridge, Ma., July 1978.

[Mag79a] Mago, G.A. A network of microprocessors to execute reduction languages, Part I. Intl. J. of Computer and Info. Sci. 8, 5 (Oct. 79), 349-385.

[Mag79b] Mago, G.A. A network of microprocessors to execute reduction languages, Part II. Intl. J. of Computer and Info. Sci. 8, 6 (Dec. 79), 435-471.

[McC6Ø] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM 3, 4 (Apr. 196Ø), 184-195.

[MeRa77] Melliar-Smith, P.M.; and Randell, B. Software Reliability: The role of programmed exception handling. Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices (ACM) 12, 3 (Mar. 1977), 95-1ØØ

[Min77] Minne, J. Stream programming in RED Languages. Dataflow Note #24, UCI Dataflow Arch. Project, Dept. of Information and Computer Science, Univ. of Calif., Irvine, Ca., Dec. 1977.

[Ols8Ø] Olsen, E. Input/output for dataflow computer systems. Masters Thesis, Dept. of Information and Computer Science, Univ. of Calif., Irvine, Ca., (to appear).

[Pat7Ø] Patil, S.S. Closure properties of interconnections of determinate systems. Record of the Project MAC Conf. on Concurrent Sys. and Parallel Computation, ACM, June 197Ø, pp. 1Ø7-116.

[PCDGS76] Plas, A.; Compte, D.; Durrieu, G.; Gelly, O.; and Syre, J.C. LAU System Architecture: A parallel data-driven processor based on single assignment. Proc. 1976 Intl. Conf. on Parallel Processing, Aug. 1976, pp. 293-3Ø2.

[Ran75] Randell, B. System structure for software fault tolerance. IEEE Trans. Sfw. Eng. SE-1, 2 (June 1975), 22Ø-232.

[RLT78] Randell, B.; Lee, P.A.; and Treleaven, P.C. Reliability issues in computing systems design. Computing Surveys 1Ø, 2 (June 1978), 123-165.

[Rum75] Rumbaugh, J.E. Data flow languages. Proc. 1975 Sagamore Computer Conf. on Parallel Processing, Aug. 1975, pp. 217-219.

[Rus77] Russell, D.L. Process backup in producer-consumer systems. Proc. Sixth ACM Symp. on Operating Systems Principles, Operating Systems Review (ACM) 11, 5 (Nov. 1977), 151-157.

[Tre77] Treleaven, P.C. Principle components of data flow computers. No. 1Ø8, Computing Lab., Univ. of Newcastle, Newcastle upon Tyne, England, July 1977.

[TFGJRS78] Treleaven, P.C.; Farrell, E.P.; Ghani, N.; Jones, S.B.; Randell, B.; and Smith, P.J. The design of highly concurrent computing systems. No. 126, Computing Lab., Univ. of Newcastle, Newcastle upon Tyne, England, July 1978.

[Was77] Wasserman, A.I. Procedure-oriented exception-handling. Tech. Report #27, Medical Information Science, Univ. of Calif., San Francisco, Ca., Feb. 1977.

[Wen75] Weng, K.S. Stream-oriented computation in recursive data flow schemes. MAC Tech. Memo. 68, Lab. for Computer Science, M.I.T., Cambridge, Ma., Oct. 1975.