MIT/LCS/TR-353

DATA BACKUP AND RECOVERY IN A COMPUTER ARCHITECTURE FOR FUNCTIONAL PROGRAMMING

Suresh Jagannathan

October 1985

This blank page was inserted to preserve pagination.

Data Backup and Recovery in a Computer Architecture for Functional Programming

by

Suresh Jagannathan

October, 1985

Funding for this work was provided in part by the National Science Foundation #NSF DCR-7915255 and the Department of Energy, #DOE DE-AC02-79ER 10473.

This empty page was substituted for a blank page in the original document.

DATA BACKUP AND RECOVERY
IN A
COMPUTER ARCHITECTURE FOR FUNCTIONAL PROGRAMMING

by

Suresh Jagannathan

Submitted to the Department of Electrical Engineering and Computer Science
on October 28, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

The VIM computer system, an experimental project under development in the Computation Structures Group at MIT, is intended to examine the efficient implementation of functional languages using the principles of data flow computation. In this thesis, we examine how to incorporate backup and recovery mechanisms into this system to guarantee that no online information is lost because of hardware malfunction. Our solution, which takes advantage of VIM's powerful applicative base language and its uniform treatment of data and files, integrates the operation of the backup and recovery system within the interpreter itself, resulting in a system that can ensure a high degree of data security without excessive performance degradation. Unlike schemes found in other systems to guarantee data security, operation of the backup facility requires no user intervention.

To present our algorithms rigorously, we first develop a formal operational model of system behaviour. This set-theoretic model views VIM as a state transition system with the interpreter serving as a state transition function. The specification language is a superset of the applicative language, VIMVAL. We enhance this model to include a concept of system failure and augment to the basic components in the system a backup state with the base language instructions now operating on both the VIM as well as the backup state. A formal proof demonstrating the correctness of our algorithms is also given. Issues concerning the implementation of these algorithms are also addressed in the thesis.

Thesis Supervisor : Jack B. Dennis

Title: Professor of Electrical Engineering and Computer Science

Keywords: Dataflow, Dynamic Dataflow Architecture, Functional Languages, Fault Tolerance, Data Backup and Recovery, Reliability, Formal Operational Models, Dataflow Graphs, Resiliency, Language Based Architecture.

This empty page was substituted for a blank page in the original document.

Acknowledgments

The result of knowledge should be the turning away from unreal things, while attachment to these is the result of ignorance. This is observed in the case of one who knows a mirage and things of that sort, and one who does not. Otherwise, what other tangible result do the knowers of Brahman obtain?

— Vivekachudamani of Sri Sankaracharya

I am grateful to my thesis advisor, Prof. Jack Dennis, for the many invaluable suggestions and insights he has provided during the development of this thesis. His continued interest in this work has been instrumental in the formulation of many of the ideas presented here.

I also thank Bhaskar Guharoy for the many enlightening conversations we have had together and for his help especially in the early stages of this work. It is a pleasure to acknowledge his assistance in the development of the formal model presented in Chapter Two.

The friendly and intellectually stimulating atmosphere of the Computation Structures Group provided an ideal environment in which to undertake this research. I thank the other members of the VIM group: Earl Waldin, Bradley Kuszmaul and Prof. Nikhil for many interesting discussions. Tom Wanuga, Kevin Theobald, Willie Lim and Tam-Anh Chu were responsible for many an enjoyable lunch.

Above all, I am indebted to my parents and brother, Aravind, without whose constant support and encouragement this document could never have been written.

This empty page was substituted for a blank page in the original document.

To my parents

This empty page was substituted for a blank page in the original document.

Table of Contents

List of Figures

This empty page was substituted for a blank page in the original document.

1.2 Motivation

found in our system provides a framework on which a highly secure system can be designed. Unlike its more conventional counterparts, the backup and recovery utilities we present are simple and efficient and do not require any user guidance to perform their task. We model the presence of failures and the actions undertaken by the backup and recovery mechanisms by defining a formal operational semantics of system behaviour. This formal model is used to give a precise description of the behaviour of the backup and recovery system. We demonstrate, using this formal model, that the backup algorithms developed properly records the relevant portions of the system state and that the recovery system does correctly restore the proper system state as well.

1.2 Motivation

The problem of guaranteeing data security is certainly not a new one. Virtually every major computer system developed includes some type of protection mechanism to safeguard the information entrusted to it. It is, therefore, natural for the reader to question why the study of data security for the VIM system is a problem worthy of investigation. There are two main reasons why we have not chosen to simply use backup and recovery algorithms developed for other systems for VIM. First, and foremost, backup and recovery systems in conventional systems are not able to provide full data security without becoming excessively complex and inefficient. In general, the implementations of these utilities are not able to provide full data security without incurring significant reduction in system performance. In addition to this major drawback, these schemes are based on a computational model which is much different from that found in VIM. The implementation strategy that we present in this thesis guarantees full data security and exploits the novel features of VIM in doing so. It is significantly different from other data security schemes proposed for both sequential and parallel systems. Some of the more interesting aspects of VIM pertinent to the issue of data security are cited below:

- VIM is based on a dynamic data flow architecture.

In a data flow computer system, all instructions in the program are viewed as potentially executable, with the only constraint being that they must have received all necessary operands and control signals. This execution model allows for a great deal of concurrency to be realized. Thus, our data security mechanism must be designed to be used in a highly concurrent environment.

- VIM interprets an applicative base language, namely, the language of dynamic data flow graphs.

A *dynamic data flow graph* is a directed acyclic graph where nodes represent instructions and arcs define data dependencies between these instructions. A graph is dynamic if the execution of a function activation is explicitly initiated by some *apply* operator. In our system, each function activation has its own graph created by the *apply* instruction. The base language instruction set is very powerful, containing instructions for function application and return, structure creation and manipulation as well as instructions to handle storage management. We can take advantage of this feature in the design of our data security mechanism by enhancing the semantics of these instructions to support the backup and recovery procedures.

- There is no distinction between *files* and *data* in VIM.

Unlike conventional systems, the units of storage allocation in VIM are the information units on which the primitive operations of VIM operate *i.e.* instructions, scalar values, arrays, and records. This feature will allow us to design backup algorithms that can be fully cognizant of how information is being created and manipulated in the system.

- The unit of transfer between disk and main memory is a small fixed-size unit of information called a *chunk*.

The small unit of information transfer will allow extensive concurrency of operation to be achieved by exploiting the data driven execution model to support a high level of information traffic between disk and main memory. The use of chunks as the unit of transfer poses interesting problems for the backup system when information needs to be copied from memory to a more stable storage medium. We discuss this issue later in the thesis.

1.3 Background

While hardware is generally reliable for the most part, catastrophes do indeed occur and it is necessary that the computer system designer be sensitive to this reality. Ideally, we would like to provide a guarantee to the users of our system that all accessible data will survive the effects of any hardware malfunction. The approach that is adopted, however, will naturally be strongly influenced by efficiency constraints. In most conventional computer systems that do not provide extra hardware support to achieve this aim, the cost of realizing full data security usually involves too much overhead and reduced performance to be a realistic goal. In these systems, users are

forewarned that some of the information they have entrusted to the system may not survive a hardware failure. Consequently, these users must take explicit action to preserve data they deem important by periodically "backing up" their information onto a more reliable storage medium. Information which is not transferred onto backup store is lost after a crash. It is usually not possible in most systems to recover this lost data by reexecuting the computations which initially produced it since there is no mechanism to monitor the creation and modification of information occurring in the system.

We shall say that the information held by a computer system is *fully secure* from loss or corruption if the system contains mechanisms which ensure the preservation of *all* data despite the presence of unreliable underlying hardware components. These mechanisms may be incorporated as backup and recovery software utilities or may take the form of replicated storage elements or may be implemented as some combination of both. The degree of data security provided by a computer system is one measure of its *reliability*. If a system is reliable, users of the system can confidently store and access data *whenever* desired. Note that data security does not necessarily imply reliability since a secure computer system need not mask failures to the external world. A system which does provide full data security, however, guarantees its users that no failure in the system will cause any accessible information to be lost even though users may be prevented from accessing it for a time if a failure takes place.

The extent to which users can access data when they wish is a measure of the *availability* of the system. In applications such as rocket guidance [4], for example, it is crucial that no single hardware failure makes the system unavailable for even a short period of time. The basic approach to masking failures of hardware from the external system is best done at the hardware level itself, by incorporating sufficient redundancy into the system [26]. In this thesis, we shall *not* be concerned with availability. Rather, we shall be focusing our attention on the problem of enhancing VIM to provide full data security for all online information.

For systems which need not mask failures, the standard approach used to guarantee data security has been to provide *backup* and *recovery* software utilities for the system. The backup utility serves to safeguard information on some storage device that is immune to the effects of hardware failure. The recovery utility is invoked when a hardware failure occurs and uses the information preserved by the backup facility to restore the system to a state which existed before the failure. The degree of data security provided by the system is dictated by the cost of the

backup and recovery process. Most current systems have had to sacrifice the goal of providing full data security because of the high overhead that would be involved in supporting the backup and recovery system. In these systems, the state restored by the recovery utility may not be the one which existed immediately prior to the failure. Moreover, it is usually the case that there is no means of recovering this desired state from the one produced by the recovery procedure. Thus, information loss is a possibility which users of the system must accept.

1.4 Previous Works

In this section, we briefly describe some previous efforts in the design of highly secure systems undertaken for both centralized and distributed systems as well as solutions proposed for providing fault-tolerance for a class of data flow architectures. We point out some major deficiencies and assumptions made in these proposals that make them not suitable for implementation in our system.

1.4.1 Data Security in Centralized Systems

In conventional single-processor machines, the simplest and perhaps most direct means of recording state information for recovery purposes is to establish a *checkpoint* describing all aspects of the system state. The checkpoint state is constructed by recording the state of the system at some point onto a reliable storage medium such as tape. The most obvious drawback of this scheme is the potentially large amount of data which must be examined -- an oftentimes unnecessary and expensive strategy since most of the objects in the system would probably not have been modified between successive checkpoints. To ameliorate this problem somewhat, many timesharing systems perform an activity referred to as *incremental dumping* to keep the backup system abreast of any modifications to the file hierarchy between successive checkpoints. In the event of a major mishap that necessitates the reconstruction of the file hierarchy, the system can be reloaded from the last checkpoint and appropriately modified by using the current incremental dumps to restore the system to a more recent state. All incremental dumps are copied onto magnetic tape. In order that the recovery phase of state restoration may proceed faster, incremental dumps are periodically consolidated to remove outdated copies of files. This consolidation process is known as *secondary dumping*.

Using incremental and secondary dumps to record information about the system state is the basic approach used by the Multics operating system [27] to provide secure service to its

users. A modification to the backup and recovery protocol employed by Multics was suggested by Benjamin [6] for the Data Network Computer. Instead of using magnetic tape as the medium to store backup copies, it was advocated that the computer system be integrated within a network of autonomous systems with each system being allowed access to the storage devices of the other processing elements. The backup facility maintains a consistent image of the file storage at a remote site within the network. The motivation for having such a backup system is the greater ease in managing the backup system that results when we do not have to deal with sequential access tape storage. The problem in using such a system, however, comes from the decrease in availability that may occur due to the extra dependency on communication lines etc.

Perhaps the most serious objection to the solution adopted by Multics is the cost involved in periodically scanning the file hierarchy to find those files which have been created or updated since the last incremental dump. The cost of performing a checkpoint in Multics increases linearly as the size of the system increases. To achieve the same degree of data security in a heavily used system where files are constantly created, destroyed and updated, as in a lightly used one, it is necessary that the backup system be invoked more frequently. This, in turn, implies degraded service to the user community. The basic reason why Multics (and other conventional centralized systems) are not able to provide full data security efficiently appears to be the inability on the part of the backup system to immediately discern when data has been created or altered and to reflect this fact onto the backup image of the system state. Because the mechanism by which data is created and updated is far removed from the file system with which the backup system interacts, the incremental dumping process is costly and inefficient. The end result is a computer system which cannot guarantee full data security without incurring excessive overhead.

1.4.2 Distributed Systems

Unlike single-processor machines or multi-processor machines under centralized control, it is difficult to perform global checkpoints in multi-processor systems under distributed control because of the lack of any system wide synchronization capability. Hence, although distributed systems may have the potential of providing a more secure computing environment as a result of the redundancy present in their architecture [29], exploiting such redundancy to achieve this end becomes much more difficult. A typical model characterizing a distributed system would be one where both the data as well as the code of a process is spread over several physical nodes in the

system. Communication between processes in such a model is usually through some form of remote procedure call [25] or message passing mechanism [20, 18, 31]. Failure in these systems can cause processes to *deadlock* [2] after the recovery phase completes. To avoid deadlock problems arising from the error recovery of a process that is a member of a collection of communicating processes, local distributed systems or network computers usually have the ability to set localized checkpoints for each process. Such checkpoints are referred to as *recovery points*. If a failure of a process is detected, it is necessary to not only roll back the failed process to its most recent recovery point, but to also reset all other processes that had exchanged information with the failed process since the time of the last recovery point in order to avoid deadlocks from occurring when operation is continued. If these recovery points are not set properly, it is quite likely to have a domino effect [2] whereby all processes initiate recovery actions that lead them to their earliest recovery point. Determining when to set recovery points and finding a consistent set of such points is a non-trivial process [31] and often substantially reduces the overall concurrency of the system.

Borg *et al* [7] and Barigazzi [5] present schemes for ensuring the security of data in a distributed message-based system. The basic idea in both proposals involves maintaining an inactive backup process on a different processor for each active process in the system . If a failure of a primary process occurs, the backup process takes over execution. In Borg's scheme, messages exchanged between two processes are also automatically recorded on backup processes as well. In addition, both the backup and recovery processes are periodically synchronized. When a failure occurs on a primary process, its backup begins execution in the state that the failed primary had at the last synchronziation point. The backup can catch up to the state the primary had just before the failure by recomputing based on the messages which were sent to it.

Barigazzi uses a similar approach in his system. However, instead of having messages sent from the primary to the backup process, an explicit recovery point is periodically established for every primary. Setting a recovery point for a process also requires establishing recovery points for all processes that communicated with this process since the last recovery point was set. Recovery in this scheme involves resetting all processes to their last recovery point. While both these schemes provide full data security, they do so at the cost of high overhead in maintaining up-to-date multiple copies of the primary process on disjoint processors. In addition, both proposals assume the transmission of messages to be a relatively inexpensive operation, an assumption which is certainly arguable.

In addition to general distributed systems, there has been much interest of late in distributed transaction systems for which a high degree of data security is essential. A sophisticated approach to handling data security for transaction based applications can be found in the *Argus* system. The *Argus* integrated programming language and distributed system being developed at MIT [23] is a comprehensive attempt to provide linguistic support for ensuring data security and consistency within a distributed computing environment. The language provides constructs to encapsulate vital data objects which are then guaranteed to survive crashes of the host processor with high probability. In addition, a mechanism is provided to express atomicity of process activity. When a hardware failure occurs, each outstanding atomic action is forced to abort. A two phase commit protocol is used to ensure that all transactions preserve the consistency of the active data. To guarantee data security, objects are distinguished as being either *stable* or *volatile*. All stable objects are written onto stable storage devices whenever a transaction completes. The integrity of stable objects is, therefore, preserved even though crashes of the host node may take place. Volatile data is presumed to be lost upon failure.

Perhaps the greatest point of contrast between the Argus implementation of data security and that which we want to provide in our system is the requirement in Argus that the user prespecify those objects which are to survive crashes. Our goal is to ensure that measures taken to provide data security are transparent to the user; no linguistic primitives are provided to specify the objects he wishes to survive failure and no explicit constructs are provided to control backup operations. While the gap between the programming model and the backup facility is not so severe in Argus as in Multics, the fact that the underlying system is based on an updatable memory execution model makes the task of ensuring a consistent state a complicated one involving expensive protocols such as two-phase commit and sophisticated algorithms to maintain a correct and up-to-date backup image. Moreover, the application domain for which Argus is best suited is a relatively restrictive one and the complexity of the system is probably not warranted for most non-transaction based computations.

1.4.3 Fault Tolerant Systems

There have been several proposals put forth for the design of fault-tolerant data flow computers which attempt to achieve data security (as well as availability) by incorporating low level fault masking capability into the system. We outline several of these proposals here to present an alternative approach to solving the problem of providing data security for a computer system.

An error recovery system in the context of a fault-tolerant data flow machine based on the Dennis-Misunas architecture [9] was described by Misunas in [24]. The model advocated was based on providing triple modular redundancy (TMR) for memory and the functional units. Each instruction cell acts as a voter and receives three results for each operand generating error packets if discrepancies are found. In addition, processor failures are handled by allowing the network to be reconfigured. The scheme involves extensive overhead in terms of increased packet traffic and extra hardware and is dependent on the role of acknowledgment signals to allow reconfiguration to take place. A fault-tolerant design for a static data flow architecture was also proposed by Leung [22]. In his proposal, a dynamic redundancy technique was employed whereby redundant units are used to detect and diagnose faults, with afflicted computations reexecuted when necessary.

Hughes [19] suggests that a checkpointing strategy suitably modified for data flow computation may be an appropriate mechanism for incorporating error recovery in a data flow machine. Basically, each cell (or instruction) which is checkpointed is marked. Whenever the backup system initiates a checkpoint, all active cells not previously checkpointed are saved. On recovery, all active cells that were checkpointed are restored. While simple to implement, recovery in this scheme requires the reinitialization of the entire state, which is an often unnecessary step. In addition, the memory of the system must be examined at each checkpoint to determine which items have not yet been copied; this is also quite wasteful in most instances. Finally, this proposal assumes that all information is resident in memory whenever the checkpoint process is invoked; by contrast, in the system which we envisage, data will be spread across the memory hierarchy between disk and main store during program execution.

All three of the proposals cited here are based on an architectural model much different from the dynamic data flow model designed for VIM. Moreover, they require special architectural enchancements to mask hardware faults. VIM is not intended for applications which have high availability requirements and, thus, there is no need for fault-masking strategies to be incorporated into the system. Consequently, the approaches to achieving data security taken by these schemes will not be directly applicable in satisfying the requirements we have set forth for our system.

1.5 Thesis Outline

In the next chapter, we present a detailed model of the VIM Computer System. We describe in detail the applicative base language being used and the dynamic data flow execution model employed. In addition, a formal semantics of system behaviour is also developed to rigorously define its operation. The manner in which users communicate with the system and the means by which long-lived objects are defined are also presented. To simplify the presentation, we assume a system supporting only a single user. Because of the applicative programming model used in VIM, this simplification does not invalidate its applicability to a multi-user system. We do not consider non-determinate computation in our model. This restriction also simplifies the algorithms. The complications involved in incorporating non-determinacy into the model is a topic which we discuss in Chapter Six.

The third chapter in the thesis presents the general strategy for the backup and recovery system. We define the criteria which the backup system will use in determining what information should be recorded on the backup storage medium. We also present the architectural enhancements necessary to support the backup and recovery utilities.

In the fourth chapter, we give a more detailed description of the backup and recovery algorithms. The alterations necessary to the interpreter are discussed. We also formalize the role of the backup and recovery system in the context of the abstract model given in Chapter Two and present the changes necessary to the base language instructions to support our algorithms.

The fifth chapter describes low level details in the transfer of information between the backup storage medium and the VIM system. We discuss how to guarantee the atomicity of information transfer so that a consistent image of the system state is maintained on the backup storage medium. We also mention how storage management is handled on the backup store as well as describing how to minimize the copying of information from system memory onto backup memory.

The final chapter presents a summary of the thesis and discusses some topics for further research. We pay particular attention to the changes which may need to be made to the backup and recovery algorithms if our model of system behaviour is augmented to allow non-determinate computation.

Chapter Two

The VIM System

The VIM Computer System is an experimental project in the Computation Structures Group intended for the investigation of a novel data flow architecture for the efficient support of functional languages. In this chapter, we discuss the design of the system and present an operational semantics for its applicative base language. The main programming language supported by VIM is VIMVAL, a major extension of the applicative language VAL [1]. VIMVAL programs are translated into the base language data flow graph representation which is then executed by the VIM *Interpreter*. Users communicate with VIM through a *Shell* program. The Shell provides a powerful interface to the system that allows users to build and manipulate VIM *environments* - the main facility for constructing long-lived structures. We discuss each of these components, VIMVAL, the Interpreter, and the Shell in the following sections.

2.1 The Applicative Language, VIMVAL

In this section, we present an informal overview of the VIM high-level applications language, VIMVAL. VIMVAL differs from VAL, its predecessor, in that functions are treated as first-class objects, free use of recursion and mutual recursion is allowed, and polymorphism is supported through a Milner-style type inference mechanism. Users can program in a structured and hierarchical manner through the use of a *module* construct described below.

In addition to various scalar types such as **integers, reals, characters** and **booleans**, VIMVAL also contains *structure* types such as **arrays** and **records**. Users can express history sensitive computation through the use of *streams* [30]. A stream is a potentially infinite sequence of homogeneous values which may be of any type (including *stream*) that allows users to write history sensitive code (such as the modeling of a conventional memory cell) within a functional framework. There are three operations on streams: **first** which returns the first element of a stream, **rest** which given a stream returns the stream without its first element, and **affix** which affixes an element to the head of a stream. We shall discuss the implementation of streams in greater detail in the next section.

Because functions are treated as first-class objects, they may be passed as arguments to other functions, returned as the result of a function activation and may be built into data

structures *e.g.*, **array[Function]**. The body of a function is an expression whose type is the result type of the function. The form of an expression may contain conditional expressions, function invocations, **tagcase** expressions which allow one of a series of expressions to be chosen based on the value of a tag, and **let** expressions which are viewed as sugaring for lambda abstractions. Conventional iteration in VIMVAL is expressed using tail recursion. Variables in VIMVAL are considered as only identifiers for a value. Once an identifier is defined, it cannot be changed — VIMVAL is a single-assignment language. The treatment of variables as identifiers for values as opposed to placeholders for memory cells which may be arbitrarily mutated as found in more conventional constructive languages is a distinguishing characteristic of VIMVAL.

A *module* in VIMVAL is a function which may be invoked from other modules or by a user command to the system. A module provides a mechanism for grouping related functions together. These functions may be invoked from within the module or, if they are passed as a result of the module, by functions external to the module. The body of a module may use names that are not bound by definitions in the module. These free names must be bound before the module can be run. Modules may be separately compiled with type consistency of identifiers used across modules being checked by a linker. Type specifications are optional in VIMVAL. Omitting type definitions allows free names in modules to be bound in possibly several contexts, each binding causing a different resolution of the types of the free variables. Users can, thus, express useful polymorphic functions using the type inference mechanism. For a more detailed exposition on the VIMVAL language, the reader should see [13] and [28].

2.2 The VIM Interpreter

The base language for the VIM system is the language of dynamic data flow graphs. Programs written in VIMVAL are translated into their data flow graph representation. Base language programs are evaluated by the VIM interpreter. A dynamic data flow graph is a directed acyclic graph in which nodes represent base language instructions and arcs are used to indicate data dependencies among instructions. There are two types of arcs in the graph: *value* arcs and *signal* arcs. An arc (s, t) is a *value* arc if it carries the value produced by the execution of node s to node t. A *signal* arc is used for performing a control function such as selecting which arm of a conditional expression is to be evaluated. A node is said to be *enabled* if and only if it has received all necessary values and signals. Enabled instructions can be executed in any order and the interpreter is free to choose the execution of any enabled instruction from any current function activation.

To illustrate the structure of a VIM data flow graph, we show in Fig. 2 the base language representation of the VIMVAL function shown in Fig. 1. Instructions are drawn as boxes, with the opcode of the instruction labeled inside the box. *Value* arcs are drawn from the bottom of the instruction which produces the value to the appropriate operand slot in the target instruction. Thus, operand number one for an instruction will be sent along the leftmost value arc entering that instruction. *Signal* arcs are drawn entering into the side of instructions. Each instruction has an index in the activation used for addressing purposes. The behaviour of the base language instructions are described in greater detail later in this chapter.

```
Function f(x :boolean; h, g : Function returns int)
  if x = true
      then h(1)   % h returns an integer.
      else g(2)   % g returns an integer.
  endif
endfun
```

Figure 1: A simple VIMVAL program

Any non-scalar value produced during the evaluation of an activation is placed on the VIM *heap*. The objects which may be found on the heap include function templates and data structures such as arrays and records. Associated with every object is a *unique identifier, uid*, which distinguishes this object from every other object on the heap. Thus, unlike simple scalar values, data structures and function templates are not transmitted along value arcs in an activation. Rather, the result of producing a complex structure is to place this structure on the heap and to send its uid to all target destinations instead. VIM employs a reference count mechanism to manage the heap. When there exists no reference to a structure on the heap from within any current activation, that structure can be removed from the heap and its space reused. The VIM heap may be considered to be a multi-rooted, directed acyclic graph where an arc (s, t), connecting nodes s and t, indicates that t is a component of s. Elements on the heap may be safely shared among objects. This feature is a result of the applicative nature of the base language. An object on the heap consists of a unique identifier and a structure value.

A distinctive feature of VIM base language programs is that no arc in the graph is ever reused. This is a consequence of the graph being acyclic with tail recursion used to model iteration. This model of data flow graphs requires that every new function application create a

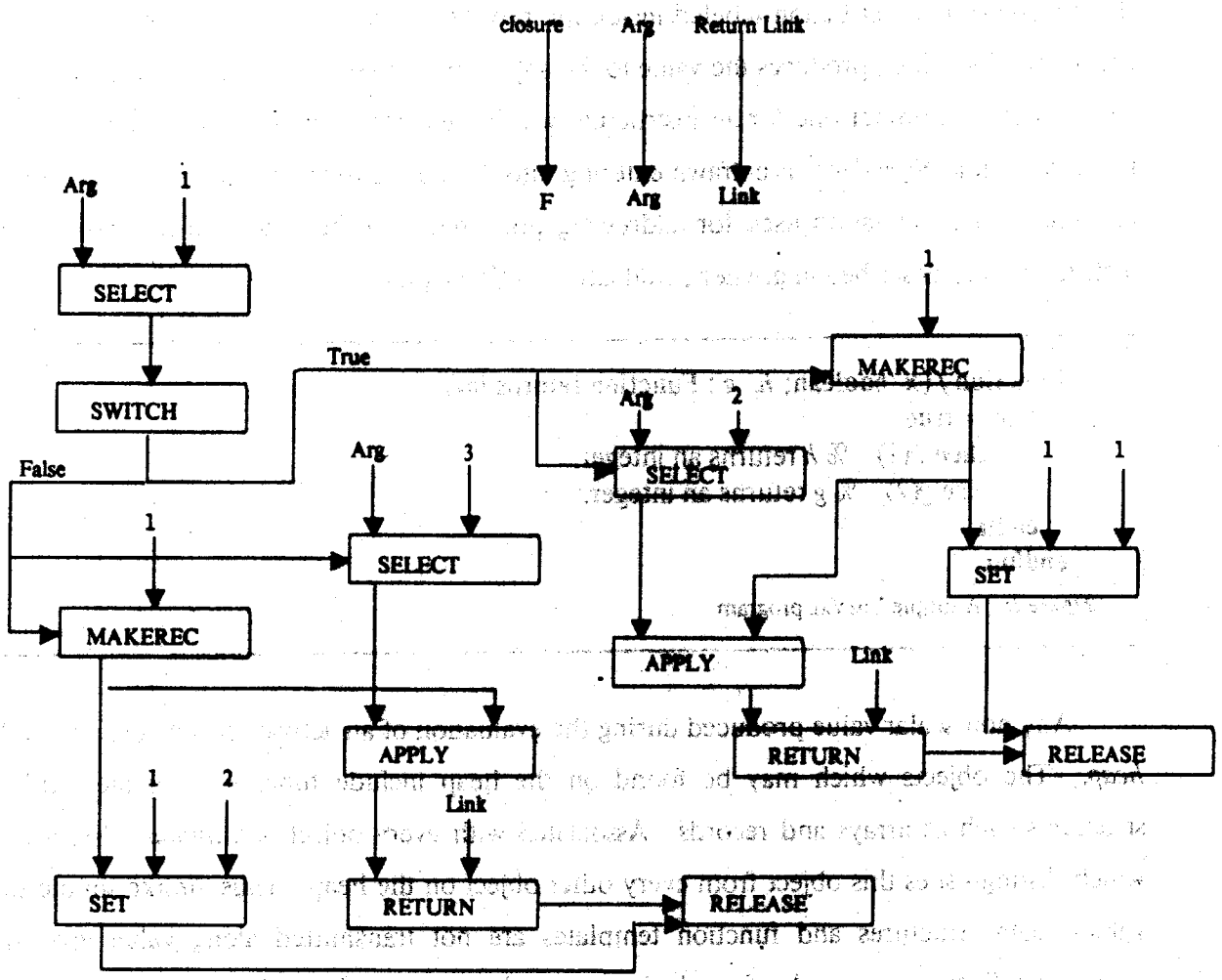
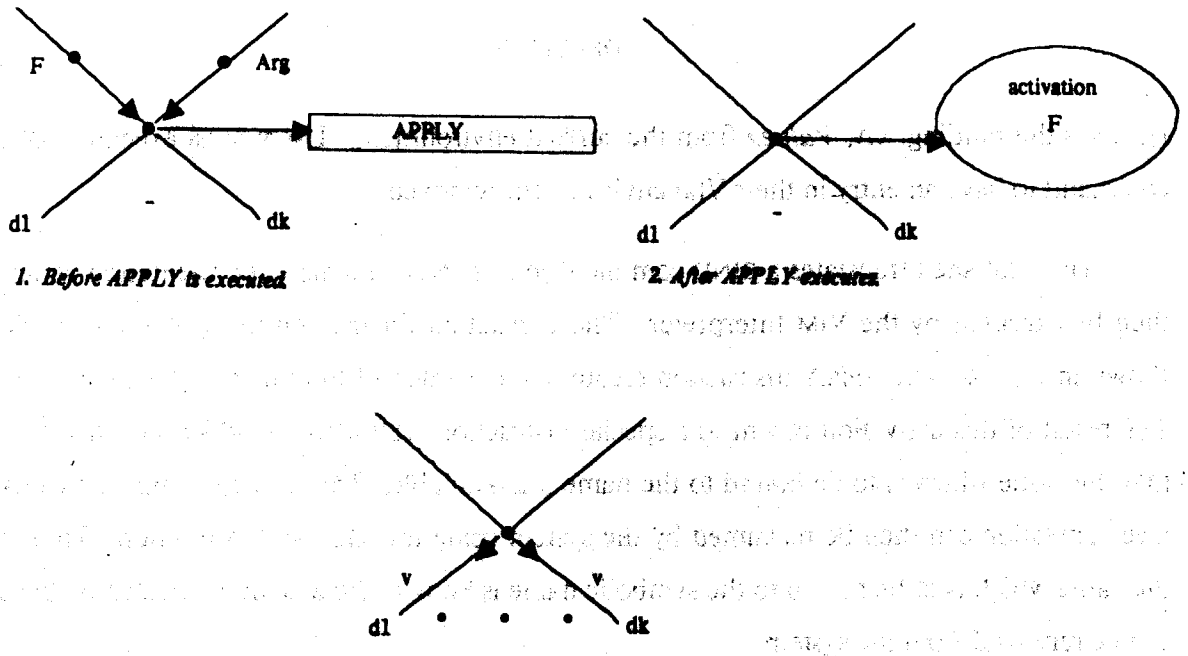


Figure 2: A VIM data flow graph

new copy of the function template and in this sense maybe thought of as a simplification of the tagged token dynamic data flow model used in the *U*-interpreter [3]. When a function is instantiated, the copy of the function is grafted at the point of application; when the activation returns a result, the activation graph contracts with the result being sent to all necessary target instructions. We illustrate this in Fig. 3.



1. Before APPLY is executed *2. After APPLY executes*

3. After activation completes with return result v.

Figure 3: Function application in VIM

2.3 The VIM Shell

Users communicate with the VIM system through a system *Shell*. The VIM shell is responsible for accepting user commands, and translating them into the appropriate base language representation and then invoking the interpreter to execute this base language program whenever necessary. A user session typically consists of the user communicating with the *Shell* in an interactive mode, inputing *Shell* commands whose results are subsequently output to the user. Every user executes in a unique *environment*. A VIM environment relates symbolic names to values and acts as a repository for all long-lived objects. Objects referenced in an environment must be explicitly deleted by the user. The VIM environment plays the role of a directory structure in conventional systems. Users specify that a particular *⟨name, value⟩* pair is to be placed in his environment through the BIND command. The user command:

BIND *name* := *⟨expression⟩*

binds the value of the expression to the name specified. This binding is then placed in the user's environment. In addition to the BIND command, there is a DELETE command which, when given a name, removes the *⟨name, Value⟩* binding from the user's environment. The command:

DELETE *N*

removes the binding, *⟨N, Value⟩* from the current environment. Users must execute a DELETE command to have an entry in their VIM environment removed.

The VIM shell translates a BIND command into its base language representation which can then be executed by the VIM Interpreter. The translation for the command BIND $x := f(z)$ is shown in Fig. 4. The APPLY instruction creates an activation of the function f with argument z. The result of this activation is sent to a special instruction, TERMINATE, which informs the shell that the value which is to be bound to the name x is available. The storage occupied by this top level activation can then be reclaimed by the system using the RELEASE instruction. Thus, once the value which is to be bound to the symbolic name is known, the activation created by the shell can be removed from the system.

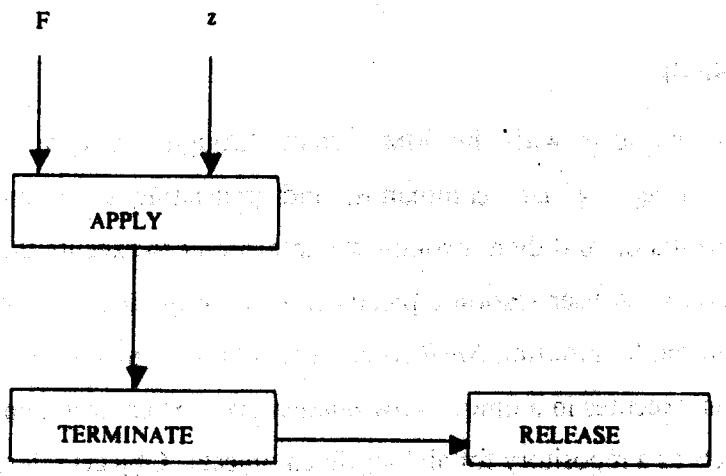


Figure 4: The translation of the BIND command

There is no translation into a base language program necessary for the DELETE command. This command is processed by the shell entirely — no assistance is required from the Interpreter to execute it.

The TERMINATE instruction described above is used to synchronize the operation of the interpreter program with the shell. If this synchronization mechanism were removed and BIND commands were allowed to arbitrarily overlap with one another, it is possible for incorrect environments to be constructed. To see why, consider two BIND commands input in succession: BIND(x, F) and BIND(x, G). It is clear that if the commands are processed correctly, x should finally get bound to the result of evaluating G. To ensure that this serialization takes place, however, requires that the second BIND command does not occur until the first one completes. As we shall see later, it is possible to still exploit a great amount of concurrency by allowing the computation of F to be still proceeding even if BIND (x, F) executes. This feature, known as *early completion*, is described in section 2.4.2.

In the next section, we will present a formal operational model for the VIM interpreter, base language and shell. This model, called **M1**, will be necessarily very abstract. We will not be considering, for example, internal representation of data structures in memory, paging of structures to and from memory or scheduling algorithms. We make two simplifications of the actual system in our model. First, we assume that all computation in the system is *determinate*. A computation is determinate if its output is totally specified by the value of its inputs; it's output does not depend on such factors as the relative arrival time of its inputs. Secondly, we assume that only a single environment exists in the system and, thus, there is no need for providing an explicit environment name to those instructions which manipulate environments. In the following chapters, we shall refine this model to describe the backup and recovery algorithms.

2.4 A Formal Operational Model - M1

The VIM system contains three major components: an *Interpreter*, a system *State* and a *Shell*. The *State* embodies all current information in the system *i.e.*, heap, activations, enabled instructions and environments. The interpreter executes a base language representation of a *Shell* command and returns the value of that command. The value returned by the interpreter is bound to a symbolic name in the user environment. The name being bound is determined by the current BIND command being processed. A shell command is translated into its base language representation and is then executed by the interpreter. This translation is performed by the Shell. The shell as described above is a function mapping from a *State* and a *session* to a new *State*. A session denotes the history of *Shell* commands input to the system. The shell translates

shell commands, invokes the interpreter to execute these commands in the current state, and binds the result of these commands to names in the user environment. It returns the state which is produced after evaluating all shell commands in the session. The abstract architecture of the VIM system is shown in Fig. 5.

In the following discussion, sets are denoted by bold font, elements of sets are denoted by italicized letters and names are indicated by a script strings of letters. Thus, **Set** is a set, $Elt \in$ **Set** and ***tag*** is a tag. We present our semantic definitions using VIMVAL-like syntax augmented with operations for performing set abstraction, set membership, etc. on the domains defined below. Function domains are specified using arrow ($\rightarrow$) notation. Thus, the domain equation, $\mathbf{A} = \mathbf{B} \rightarrow \mathbf{C}$ defines **A** to be the set of all functions with domain **B** and range **C**. Tuples are enclosed using angle brackets.

Formally, we define the VIM System to be a three-tuple:

VIM = ⟨*Shell*, *Interp*, **State**⟩ where

$\mathbf{State} = \langle \mathbf{Act} \times \mathbf{H} \times \mathbf{EIS} \times \mathbf{Env} \rangle$

$\mathbf{Act} = \mathbf{U_A} \rightarrow \mathbf{Activity}$

$\mathbf{H} = \mathbf{U_H} \rightarrow \mathbf{ST}$

$\mathbf{U_A}$ = the set of unique identifiers used for activations.

$\mathbf{U_H}$ = the set of unique identifiers used for structures.

EIS = the set of enabled instructions, described later.

$\mathbf{Env} = \mathbf{Name} \rightarrow (\mathbf{U} \cup \mathbf{Scalar})$

Activity = $\mathbf{N} \rightarrow$ **Instruction**, **N** being the set of natural numbers.

An element *Act* in the domain of VIM activations, **Act**, is a mapping from unique identifiers to *activities*. An activity is a function mapping from natural numbers to instructions and represents the code of an activation. An activity can be thought of an array of instructions, the i^{th} element in the array specifying the i^{th} instruction. The VIM heap, *H*, is modeled as a function from unique identifiers to structure types, **ST** defined below. The structure of the heap is determined from the mapping defined by the heap function. Scalar values are not represented on the heap.

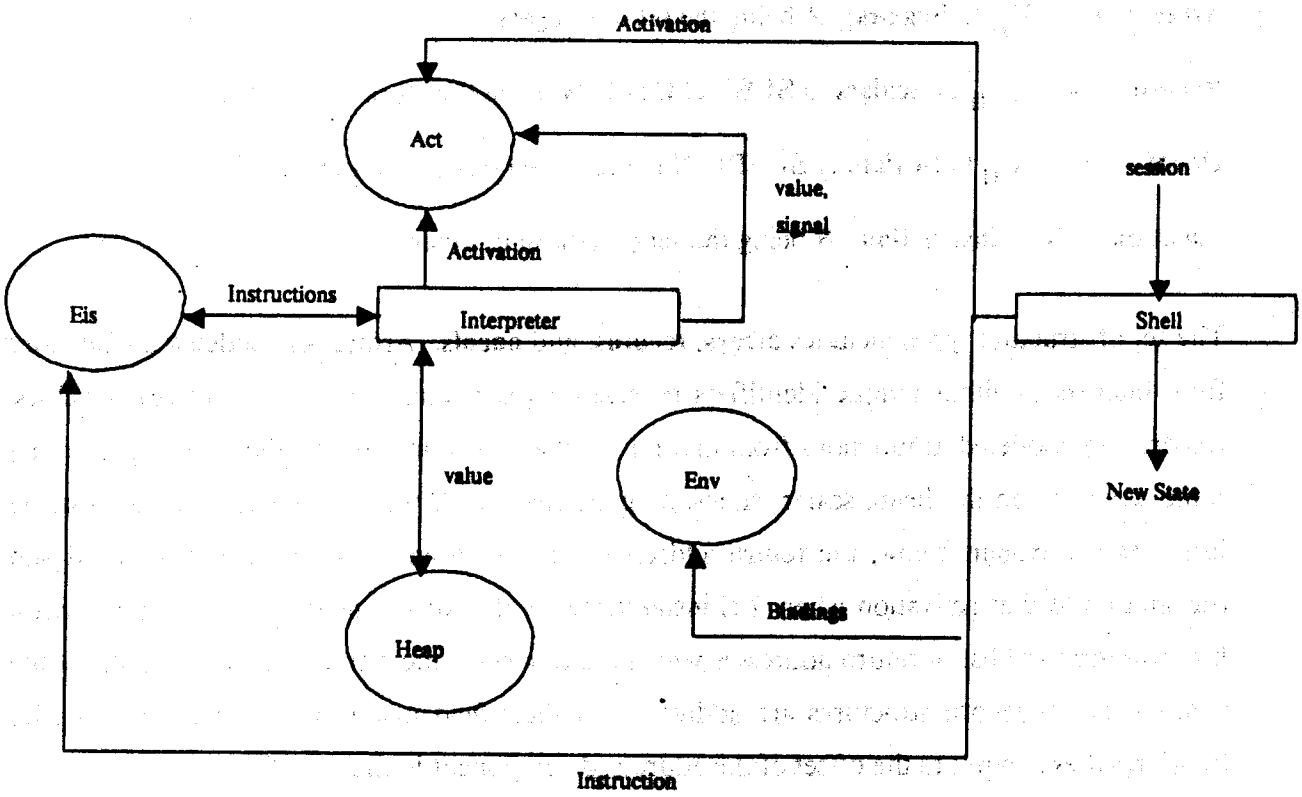


Figure 5: The Abstract VIM System

In the actual system, such values are transmitted directly along the value arcs in the graph. The *Environment* component in the *State* is also a function, mapping from a name, which is any sequence of characters, to either a unique identifier referencing a structure on the heap or a scalar value.

The data types supported by the system are given below:

Scalars = **Integers** ∪ **Reals** ∪ **Booleans** ∪ **Character** ∪ **Null**

Name = **Character***, the set of all character sequences.

Integers = the set of all integers ∪ {*undef*}
Reals = the set of all reals ∪ {*undef*}
Booleans = {*true*, *false*, *undef*}
Character = the set of characters in the machine ∪ {*undef*}

Null = {*nil*, *undef*}

$ST = \{Array \cup \{undef\}\} \cup \{Record \cup \{undef\}\} \cup \{Oneof \cup \{undef\}\} \cup \{Function\} \cup \{ECQ\} \cup \{Clsr\} \cup \{Dests\}$

$Array = Z \rightarrow (U_H \cup Scalars)$, Z being the set of integers.

$Record = N \rightarrow (U_H \cup Scalars \cup SUSP \cup Dest)$, N being the set of natural numbers.

$Oneof = N \rightarrow (U_H \cup Scalars \cup SUSP)$, N being the set of natural numbers.

$Function = N \rightarrow Instruction$, N being the set of natural numbers.

The set of structure types includes **arrays**, **records**, and **oneofs**. Arrays are modeled as functions from integers to either unique identifiers representing structures on the heap or scalar values. Records are modeled as functions from natural numbers to either unique identifers representing some structure on the heap, scalar values, suspensions, which we describe later, or destination lists. As we explain below, the return address of an activation is packaged into a record and transmitted to that activation when it is instantiated by the APPLY instruction. The destination list represents the list of return addresses which are to receive the result of the activation. While components of record structures are addressed by their field name in VIMVAL, the compiler translates these names to the offset of the addressed component in the record.

Note also that the set of functions is also included among the elements of the structure types in the system. This is consonant with our treatment of functions as first class citizens. An element of type Function is a mapping from natural numbers to instructions just as elements of the set of activations are. As we shall see below, the only difference between a function definition and its corresponding activation is that the latter is sensitive to the effect of instruction execution since operands and signals are received by the instructions within an activation whereas a function is a pristine object like any other VIM structure.

A function *closure* is a special record of two components: the first component is the uid of the function template of the closure and the second component is the list of free variable definitions found in the function. The closure of a function completely defines the bindings of the free variables in the function and, thus, must be defined before the function can be applied. Free variables are accessed by its index in the free variable list. The definition of a closure is given formally as:

$Clsr = \langle U_{II} \times (N \rightarrow (U_{II} \cup Scalar)) \rangle$

2.4.1 The Base Language Instructions

A base language instruction is an seven-tuple consisting of an *opcode*, three operand fields (not all need be used), an *operand* count used to indicate how many operands must arrive, a *signal* count used to indicate how many signals must be received, and the destination record containing the list of destinations for this instruction. The set of opcodes we will be considering in our operational model will include record structure operations, function application instructions, and operations on early completion elements. These classes of instructions will be the ones of greatest interest when we present our backup and recovery algorithms.

$$\textbf{Instruction} = \textbf{OPS} \times (U_H \cup \textbf{Scalars})^3 \times N \times N \times \textbf{Dests}$$

A destination has a type which is either **unconditional** if the result of the instruction is to be sent automatically to the destination, **true** or **false** used by a SWITCH instruction to control conditional evaluation. If the type of the destination is **true**, then the result is a signal which is sent to the destination instruction if and only if the SWITCH operator evaluated to true. A similar description applies for a **false** type destination. All destinations of an instruction must be within the same activation. The second component of a destination is the instruction number to which the signal or result value should be sent. If the destination is to receive a signal, then this must be specified. Otherwise, the operand field to which the result is to be sent must be provided. For convenience we shall refer to the elements of an instruction using dot, "." notation. For example, the **opcode** of instruction *I* shall be denoted as I.**opcode** etc.

$$\textbf{Dests} = \mathcal{P}(D)$$
$$D = \{\textbf{uncond}, \textbf{true}, \textbf{false}\} \times N \times \{\textbf{op1}, \textbf{op2}, \textbf{op3}, \textbf{signal}\}$$

An *enabled instruction* is a two-tuple (u, i) representing an instruction in some activation which has received all necessary operands and signals. Any enabled instruction can be executed by the interpreter. The applicative nature of the system guarantees that the behaviour of the program will be determinate regardless of the order in which enabled instructions in the program are executed.

$$EI = \langle U_A \times N \rangle.$$

$EIS = \mathcal{P}(EI)$ is the set of enabled instructions.

2.4.2 Early Completion Structures

According to the model of operation presented above, an instruction is allowed to execute only when it receives all necessary operands and signals. In the case when an instruction is to operate on a data structure such as an array or record, this means that the entire structure must be fully constructed before this instruction can execute. If the instruction only needs to examine a certain portion of the structure, then the execution model unnecessarily constrains parallelism in the program. To alleviate this problem, there is a facility in VIM known as *early completion.* Early completion structures allow greater concurrency of operation by allowing a data structure to be constructed and used before all of its components are available. The compiler designer can use this facility, for example, to generate code which will cause the results of an activation to be an early completion structure to allow the calling activation to use some of the results of the callee before all of them are known.

An element of the set *ECQ* is an *early completion element* [11]. An early completion element is a two tuple, (u, i), where $u \in U_A$ and $i \in N$. The early completion structure is essentially a queue containing target addresses of those instructions which require the value of this element in the structure. When the value is finally produced, it will replace the *ec*-structure and will be sent to all targets. This process is illustrated in Fig. 6.

$ECE = \langle U_A \times N \rangle$.

$ECQ = \mathcal{P}(ECE)$

An element, $(u, i) \in ECE$ denotes an instruction which has requested the value which is to replace this early completion structure. The uid u denotes a function activation, and i is the index in this activation of the target instruction.

2.4.3 Delayed Evaluation Using Streams

The astute reader would have noticed that the stream type presented in an earlier section is not defined in our formal model. We represent streams using the **record** and **oneof** types:

```
Stream[T] = oneof
        [empty : null
         non-empty : record
                     [first : T
                      rest  : Stream[T]]]
```

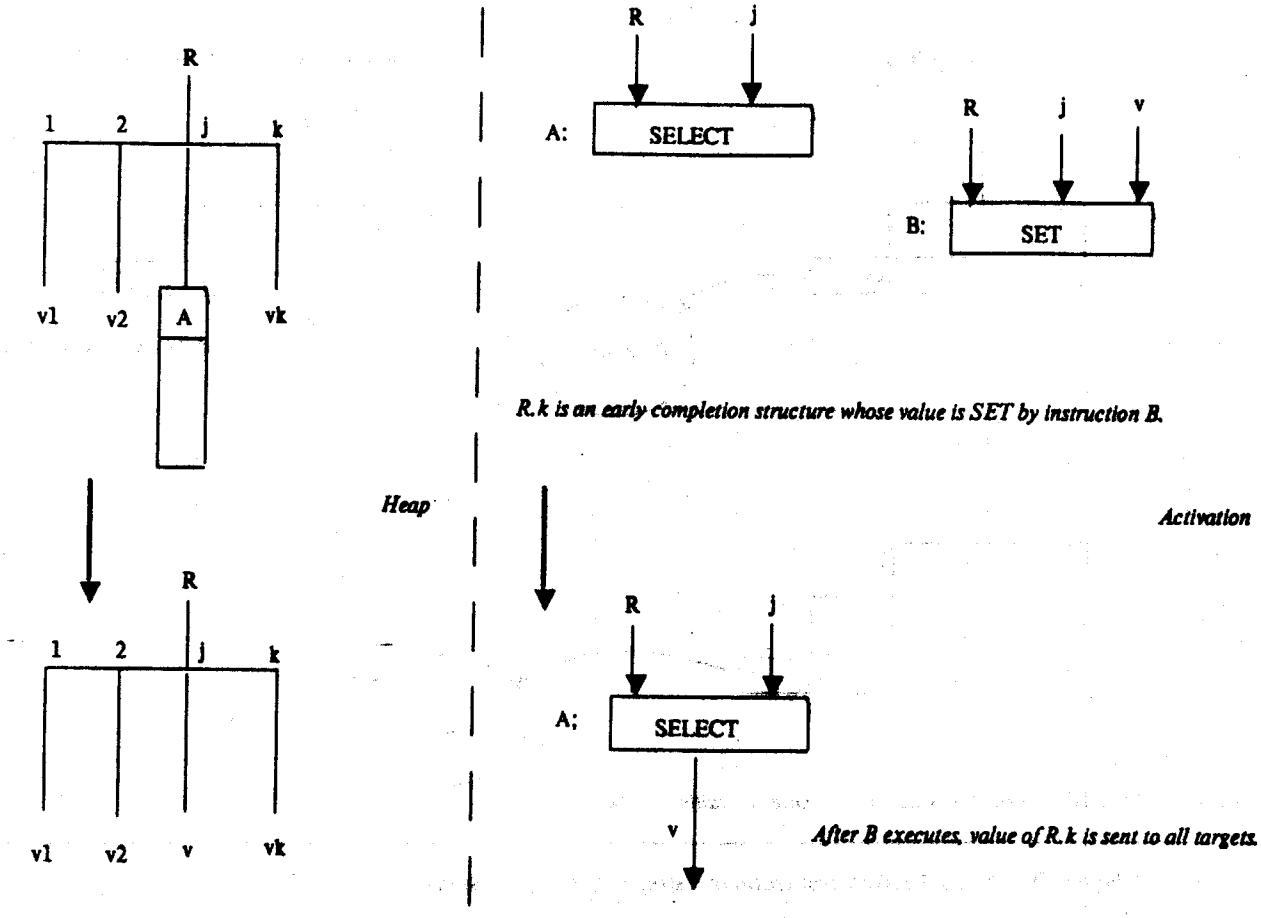


Figure 6: The Use of Early Completion Structures

We have previously mentioned that streams are potentially infinite structures. In a purely data driven execution model, the production of a stream may far outstrip its consumption. To avoid wasteful computation, streams are produced in a *demand-driven* fashion [16]. In demand driven evaluation, an element of a stream is produced only when its consumer requires it. To implement this feature, a special record element called a *suspension* is introduced. A suspension contains the address of the instruction in the stream producer responsible for instantiating production of the next stream element. When a consumer accesses a suspension, the suspension becomes replaced by an early completion queue and a signal is sent to the address which the suspension holds. A new record is created for the next element and the early completion queue will get replaced with the uid of this record. The head of this new record will contain the new

stream element and the tail of this record will again hold a suspension. We illustrate this process graphically in Fig. 7.

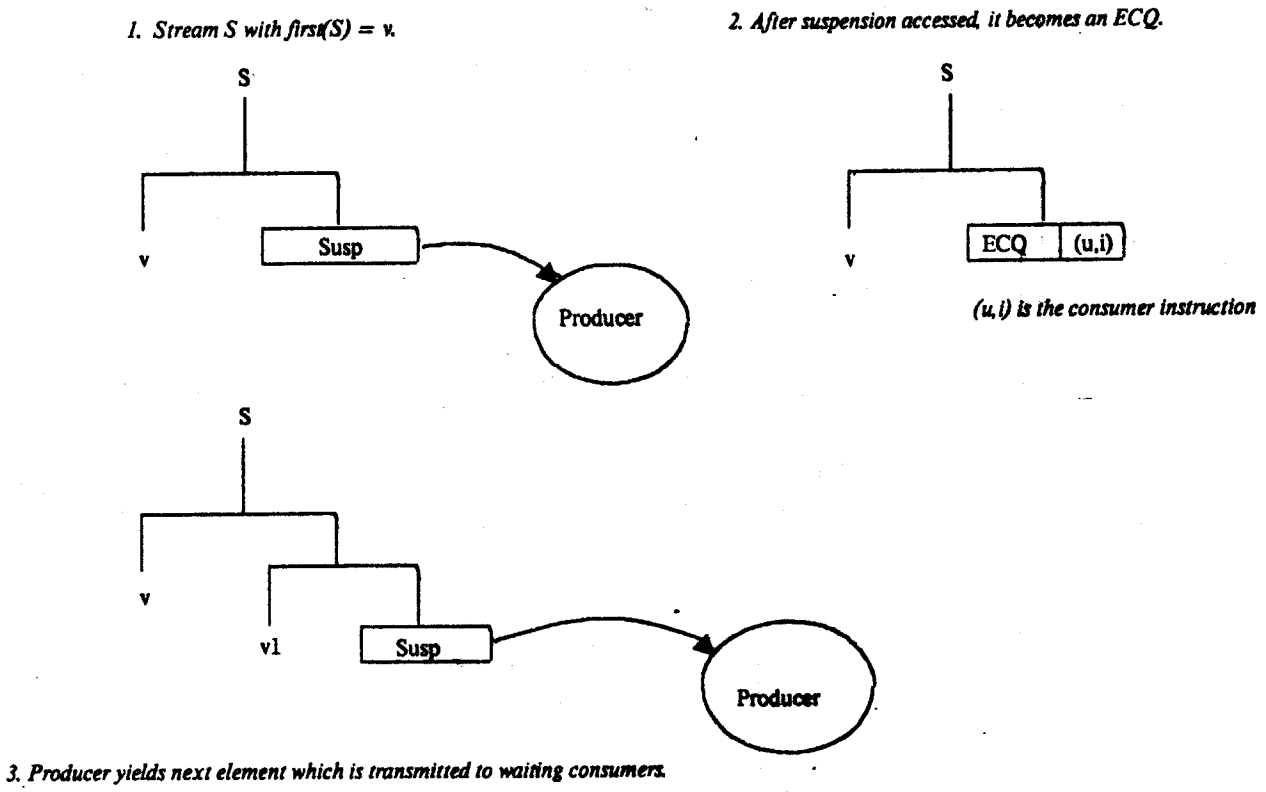


Figure 7: Demand driven evaluation of a stream using suspensions

$$SUSP = \langle U \times N \rangle$$

The suspension structure is a pair $\langle u, i \rangle$ which represents the address in the stream producer that is to be signalled when the suspension is accessed.

2.5 Semantic Functions for M1

In this section, we present an operational model for the VIM Shell Interpreter and base language instructions which extends the operational model defined in the previous section. The instructions which we examine here are chosen because of their relevance to the backup and recovery algorithms developed in the following chapters. We partition our presentation into four categories: formal description of the *Shell* and *Interpreter*, instructions which operate on structure types, instructions which are used for manipulating early completion structures and stream elements, and instructions which are concerned with function application and return.

In the next section, we define some auxiliary functions that will be useful for our presentation. These functions are used in the definitions of the above mentioned base language instructions.

2.5.1 Auxiliary Functions

There are several primitive auxiliary functions which we need to define before presenting our operational model. These functions operate on the heap, activation, and environment components of the VIM state. The functions *NewHeap*, *RemoveHeap* adds and remove an element to and from the heap respectively. The *NewHeap* function takes in three arguments: a current heap H, a uid u, and a structure, v. It returns a new heap, H', identical to H except that this new heap is defined for u such that $H'(u) = v$. This definition of *NewHeap* allows us to rebind existing uid's to different values, a feature which will be useful when implementing early completion structures and suspensions as we shall see later. The *RemoveHeap* function takes as arguments a heap H and uid u, and returns a new heap, H' identical to H except that $H'(u)$ is *undefined*.

NewHeap: $\mathrm{H} \times \mathrm{U_H} \times \mathrm{ST} \rightarrow \mathrm{H}$

Function *NewHeap* (H, u, v)
  $\forall\, u_1 \in \mathrm{U_H}$ **let** $H'(u_1) = H(u_1)$ **if** $u_1 \neq u$
  $= v$ **otherwise**
 in
  H'
 endlet
endfun

RemoveHeap: $\mathrm{H} \times \mathrm{U_H} \rightarrow \mathrm{H}$

Function *RemoveHeap* (H, u)

  $\forall\, u_1 \in \mathrm{U_H}$ **let** $H'(u_1) = H(u_1)$ **if** $u_1 \neq u$
  $=$ undefined **otherwise**
 in
  H'
 endlet
endfun

There are similar functions, *AddAct* and *RemoveAct* defined for the activation component of the state and *AddEnv* and *RemoveEnv* defined for the environment component as well.

Since the transmission of results and signals is an activity common to every instruction, we define an auxiliary function, *SendToDest* which is responsible for sending a value or a signal as the case may be to the specified target instruction and constructing a new activation function and new set of enabled instructions to reflect the effect of this transmission. *SendToDest* constructs a new activation component in which the target instruction found in the destination argument has been updated to reflect the transmission of the value or signal. If, as a result of this transmission, the target instruction becomes enabled *i.e.*, have its **opcnt** and **sigcnt** fields become zero, that instruction is appended to the enabled instruction set.

ResultType $= (U_H \cup \textbf{Scalar}) \cup \textbf{signal}$

SendToDest: **Act** $\times$ **EIS** $\times U_A \times$ **D** $\times$ **ResultType** $\rightarrow$ **Act** $\times$ **EIS**

Function *SendToDest*(*Act*, *EIS*, u_{FA}, ⟨*dc*, *i*, *opnum*⟩, *result*)

let $FA = Act(u_{FA})$
$I = FA(i)$
resultval = **if** *result* = **signal**
then *undef*
else *result*
endif

newopnum1 = **if** *opnum* ≠ **op1** **then** *I*.**op1** **else** *resultval*
newopnum2 = **if** *opnum* ≠ **op2** **then** *I*.**op2** **else** *resultval*
newopnum3 = **if** *opnum* ≠ **op3** **then** *I*.**op3** **else** *resultval*

newopcnt = **if** *result* = **signal**
then *I*.**opcnt**
else *I*.**opcnt**-1
endif
newsigcnt = **if** *result* = **signal**
then *I*.**sigcnt**-1
else *I*.**sigcnt**
endif

$I' =$ *I*.**opcode** $\times$ *newop1* $\times$ *newop2* $\times$ *newop3* $\times$ *newopcnt* $\times$ *newsigcnt* $\times$ *I*.**dests**

$\forall j \in \mathbb{N}\ FA'(j) = FA(j), \quad j \neq i.$
$= I', \quad j = i.$

NewAct = *AddAct*(*Act*, u_{FA}, FA') % new set of activations
NewEis = **if** (I'.**opcnt** = 0) $\wedge$ (I'.**sigcnt** = 0)
then $EIS \cup \{\langle u_{FA}, i\rangle\}$

```
            else EIS
          endif
  in
    NewAct, NewEis
  endfun.
```

Two functions which call *SendToDest* are *SendValue* and *SendSignal*. The function *SendValue* calls *SendToDest* for every target in the destination list of an instruction whose *opnum* is *not* **signal**. That is, all target instructions that are to receive the result of the instruction are sent the result value by *SendValue*. This operation is accomplished through the use of *SendToDest*. *SendSignal* operates in a similar fashion to *SendValue* except that it calls *SendToDest* for all targets in the destination list of an instruction whose *opnum* is **signal**.

SendValue: **Act** × **EIS** × $\mathbf{U_A}$ × **Dests** × **ResultType** → **Act** × **EIS**

Function *SendValue* (*Act, EIS,* u_{FA}, *dests, v*)

```
 let ValDest = {<dc, i, opnum> ∈ dests | opnum ∈ {op1, op2, op3}}
     <dc_1, i_1, opnum_1>, <dc_2, i_2,
opnum_2>, ..., <dc_n, i_n, opnum_n> =
                    n components of ValDest
 in
   SendToDest(
     SendToDest(...
       SendToDest(Act, EIS, u_FA, <dc_1, i_1, opnum_1>, v)
                 u_FA, <dc_2, i_2, opnum_2>, v)...)
 endlet
endfun
```

SendSignal: **Act** × **EIS** × $\mathbf{U_A}$ × **Dests** × → **Act** × **EIS**

Function *SendSignal* (*Act, EIS,* u_{FA}, *dests*)

```
 let SigDest = {<dc, i, opnum> ∈ dests | opnum = signal}
     <dc_1, i_1, opnum_1>.<dc_2, i_2, opnum_2>, ..., <dc_m, i_m, opnum_m> =
                    m components of SigDest
 in
   SendToDest(
     SendToDest(...
       SendToDest(Act, EIS, u_FA, <dc_1, i_1, opnum_1>, signal),
         u_FA, <dc_2, i_2, opnum_2>, signal)...)
```

```
    endlet
endfun
```

2.5.2 A Formal Model of the Shell

As we described above, the VIM shell serves as the interface between the interpreter and the VIM user. The formal definition of the shell is given below:

```
Command = C × Name × (Exp ∪ undef)
C = {BIND, DELETE}

Session: Stream[Command]

Shell: Session × State → State
```

```
Function Shell (Session, State)

  let <Act, H, EIS, Env> = State
      NewState =
          if ChooseToExecute (State, Session)
              then Shell(Session, Execute (State, Choose(EIS)))
              elseif empty(Session)
                  then State
                  else let c1 = first(Session)
                       in if c1.C = DELETE
                              then <Act, H, EIS, DelEnv(Env, Name)>
                              elseif c1.C = BIND
                                 then let % command is BIND

                                      FA = Translate(c1)
                                      uFA = new uid from UA
                                      Act' = AddAct(Act, uFA, FA)
                                      NewEIS = EIS ∪ {<uFA, 0> | FA(i).opcnt = 0
                                          ∧ FA(i).sigcnt = 0}

                                      State', v = Interp(State, Choice(NewEIS))
                                      <Act', H', EIS', Env> = State'
                                      Env' = AddtoEnv(Env, c1.name, v)
                                      in
                                      <Act', H', EIS', Env>
                                      endlet
                              endif
          endlet
endif
```

```
    in
      if empty (Session)
        then if EIS' ≠ {}
               then Shell (Session, Execute(Newstate, Choice(EIS')))
               else Newstate
             endif
        else Shell (rest(Session), Newstate)
      endif

    endlet
  endfun
```

The shell takes in as input a *stream* of shell commands. It calls the function *ChooseToExecute* which examines the current state and session and determines if the shell should process the next shell command or whether it should call the *Execute* function using the current enabled instruction set. This function allows the system to continue to process instructions even if the remainder of the session has not yet been input by the user. One possible implementation of this function would be a routine which examines the current input buffer — if the buffer is empty and there are instructions still to be executed, it returns true. If, on the other hand, there are shell commands available to be processed, it returns false. In the case when there are both commands and enabled instructions available, it can arbitrarily return either true or false. If the function returns false and the first command in the command stream is DELETE, then the shell removes the ⟨*name, value*⟩ binding from the current environment and processes the rest of the command stream. If it is a BIND command, however, it calls an auxiliary function, *Translate* with this command stream element. *Translate* returns an activity, *PA*, which embodies this command. For example, if the command input to *Translate* was BIND (*x* = *f* (*z*)), the activity returned would be of the form shown in Fig. 4. The result of evaluating this activity represents the value of the command. A new activation is constructed from this activity and this activation augments the current activation state. The enabled instruction component is also appropriately augmented to include all instructions in this new activation that have their operand and signal count already zero. The shell calls the interpreter with this new state and some enabled instruction from the set of enabled instructions. The choice of which enabled instruction to execute is made by the *Choice* function. The new state returned by the interpreter is used by the shell in processing the next shell command if there are any more to be processed. The value, *v*, returned by the interpreter is bound in the user environment to the symbolic name

which is an argument to the BIND command. If there are no more stream commands to be processed, the shell calls the *Execute* function described below to execute the remaining instructions found in enabled instruction set. If there are no more enabled instructions, the function returns with the final state.

2.5.3 A Formal Model of the Interpreter

The interpreter is a state transition function from states and enabled instructions to a state and either a unique id or a scalar value:

Interp (⟨*Act, H, EIS, Env*⟩, *Choice*(*EIS*)) ⊢ ⟨*Act'*, *H'*, *EIS'*, *Env'*⟩ × (U_H ∪ *Scalar*)

The *Choice* function, as we explained above, is used to determine which enabled instruction should be chosen for execution from the enabled instruction set. The definition of the *Interpreter* is given below:

Interp: **State** × **EI** → **State** × ($\mathbf{U_H}$ ∪ **Scalar**)

Function *Interp* (*State*, ⟨*u, i*⟩) %⟨*u, i*⟩ is an enabled instruction

```
  let
      ⟨Act, H, EIS, Env⟩ = State
      FA = Act(u)
      Newstate = Execute(State, ⟨u, i⟩)
      ⟨Act', H', EIS', Env⟩ = Newstate
  in
     if FA(i).opcode = TERMINATE
       then Newstate, FA(i).opnum1
       else Interp(Newstate, Choice(EIS'))
     endif
  endlet
endfun
```

The instruction which is chosen for execution must be part of some activation defined in the set of current activations, *Act*. The interpreter calls on an auxiliary function, *Execute*, defined below which contains the definitions of all the base language primitives. Note that the interpreter only returns its result when the TERMINATE instruction has executed. This restriction guarantees that environments will be updated correctly according to the order in which BIND commands were input. When the interpreter returns, a new command can be processed by the

shell. Note that because of the presence of early completion structures, there may still be many activities in progress at the time the TERMINATE instruction executes and the next shell command is processed. Thus, our model allows instructions found in activities created from the evaluation of different bind commands to execute in parallel. We do not come into problems in augmenting environments though, because, as we discussed earlier, bindings are always constructed in the proper serial order.

The *Execute* function examines the instruction being processed and performs the necessary function. The result of this function will be a new VIM state. The structure of this function can be given as follows:

Execute: State × EI → State

Function *Execute* (*State*, $\langle u_{FA}, k_{FA} \rangle$)

  let $\langle Act, H, EIS, Env \rangle = State$
    $FA = Act(u_{FA})$
    $I = FA(k_{FA})$
    $destinations = I.\mathbf{Dest}$,

    $NewEIS = EIS - \{\langle u_{FA}, k_{FA} \rangle\}$
  in
    if $I.\mathbf{opcode}$ = SET **then** ...
    elseif $I.\mathbf{opcode}$ = APPLY **then** ...
      .
      .
      .
    endif
  endlet
endfun

The destination list specifies those instructions in the current activation to which the result of executing this particular instruction should be sent. Recall that in addition to transmitting results, we may also need to send signals to destinations.

2.5.4 Formal Definition of Base Language Instructions

In this section, we present a formal definition of those base language instructions that will be useful to us in describing the backup and recovery algorithms later in the thesis. Keep in mind that these definitions are actually found within the *Execute* function given above.

2.5.4.1 The TERMINATE Instruction

The TERMINATE instruction is used to receive the result of evaluating a base language program. This result value is then bound by the shell in the user environment. The instruction takes in one argument, which is either a scalar value or a uid. It sends no results but will send a signal to a RELEASE instruction which is used to remove the activity from the set of current activities. We describe the operation of the RELEASE instruction later in the chapter. The interpreter picks up the result from the first operand slot in the instruction when it returns back to the shell.

---

if *I.opcode* = TERMINATE **then**
let
val = *H*(*I.op1*)

Act', *NewEis'* = *SendSignal*(*Act*, *EIS*, u_{FA}, *destinations*)

in
⟨*Act'*, *H*, *NewEis'*, *Env*⟩

endlet
endif

2.5.4.2 Structure Operations

The base language contains powerful instructions for the creation and manipulation of structure types. There are three structure operations of particular interest - CREATE, REPLACE and SELECT. The CREATE operator is used to create a structure of a particular dimension. In a functional language, a structure, once defined, cannot be subsequently altered. Replacement of an element α in a structure with an element β is done by creating a new version of the structure with β replacing α in the new version. The SELECT operation returns the value of a specified field in a structure.

The operations we describe below are for the **record** structure type but are very easily converted for the **array** or **oneof** type.

To create a record structure, we have a MAKEREC instruction. It takes in one argument, the size of the record structure to be created and constructs a record of such a dimension, setting all the fields in the record to be *undef*. In addition to the MAKEREC instruction, there is also a MAKERECEC instruction which constructs a record, all of whose elements are early completion structures.

```
if I.opcode = MAKEREC then
  let
    m = I.op1   % m is a natural number

    u_v = a new uid in U_H,
    Act', NewEis' = SendValue(Act, EIS, u_FA, destinations, u_v)
    Act'', NewEis'' = SendSignal(Act', NewEis', u_FA, destinations)
    H = NewHeap(H, u_v, MakeRecord(1, m, undef))
  in
    <Act'', H', NewEis'', Env>

  endlet
endif
```

The REPLACE instruction on records takes in three arguments, the uid of a record R, the field in the record which is to be replaced, f, and the value of the new element, x, which may be a scalar or a uid. It creates a new copy of the record, R', with field f in this copy having value x. We mention the REPLACE instruction here mostly for completeness as it will not be involved in the design of the backup and recovery algorithms we develop later in the thesis. For a detailed semantic description of this operation, the reader should see [17].

The SELECT instruction is given a record structure and the offset in the record of the field to be selected. If the item to be selected is an early completion structure, then the instruction queues itself onto the *ec*-queue. When the value of this field is finally known, the select instruction will be placed again on the enabled instruction queue so that it may execute. It is also possible that the item being selected may be a *suspension* if the record is part of a stream. Recall that the rest operation on streams is translated into a select operation on the second component of the head of the current stream. Because streams are produced in a demand driven manner, this field may be a suspension in which case the SELECT instruction will need to send a signal to the instruction referenced by the suspension. The field occupied by the suspension will then get changed to an early completion queue which will get SET in the activation responsible for producing the next stream element.

```
if I.opcode = SELECT then
  let
    R = H(I.op1)
    f = I.op2          % f must be a natural number

    t = R(f)
    Newstate = if t ∈ U_H ∧ H(t) ∈ ECQ
                 then <Act,
                       NewHeap(H, t, H(t) ∪ {<u_FA, k_FA>})
                       NewEis,
                       Env>
                 elseif t ∈ U_H ∧ H(t) ∈ SUSP
                   then let
                        <u', k> = H(t)
                        Act', NewEis' =
                          SendToDest(Act, NewEis, u', <uncond, k,signal>, signal))
                        in
                          <Act',
                           NewHeap(H, t, MakeECQ(<u_FA, k_FA>))
                           NewEIS',
                           Env>
                        endlet
                 else let
                       Act', NewEis' = SendValue(Act, NewEis, u_FA, destinations, t)
                       Act'', NewEis'' = SendSignal(Act', NewEis', u_FA, destinations)
                      in
                       <Act'', H, NewEis'', Env>
                      endlet
                 endif
  in
    Newstate
  endlet
endif
```

2.5.4.3 The Set Operation

The main operation on early completion elements is the SET instruction. The instruction takes in three arguments, a record *R*, an offset in the record which represents the field which is to be *set*, and a value, *x*. When the set instruction executes, it replaces the early completion structure found at the specified component with *x*. Moreover, all the elements in the *ec*-structure are appended onto the enabled instruction queue since the value of the field which these instructions initially requested is now available. The SET instruction, unlike the REPLACE

operation mentioned above, does not cause a new version of the structure to be created. Instead, the early completion structure is replaced with the value *in situ*. This does not violate any principle of referential transparency because no instruction is allowed to read a field which is an *ec*-structure. Since the SELECT instruction on structures prevents any of its targets from reading the value of the field until it is properly set, the applicative property of the base language is not compromised. Because all instructions which require the value of this field are on the early completion queue, SET does not send any results to any of its destinations, only signals.

---

if *I.opcode* = SET **then**
let
$u = I.op1$
$R = H(u)$
$f = I.op2$
$x = I.op3$
$u' = H(R(f))$

$\forall\, v \in N$
$R'(v) = R(v)$ **if** $v \neq f$
$= x$ **otherwise**

$Act', NewEis' = SendSignal(Act, NewEis, u_{FA}, destinations)$
$H' = NewHeap(H, u, R')$

in
$\langle Act',$
$H',$
$NewEis' \cup H(u')$
$Env\rangle$
endlet
endif

2.5.4.4 The Suspension Operator

The SETSUSP operator is responsible for setting a suspension in a record structure. It takes in three arguments, a record structure representing the head of the current stream, an offset into this structure where the suspension is to be placed, and an instruction address, *i*, representing the instruction which is to be signalled when the suspension is accessed. The offset must be an early completion element presumably constructed by a MAKERECEC instruction. SETSUSP sets in this record the value, $\langle u_{FA}, i\rangle$. u_{FA} being the uid of the activation in which the SETSUSP operator is executing. If the ec-structure is not empty, then SETSUSP signals the activation as well since such

a situation implies that some select operation has already attempted to read the next stream element. Like the SET instruction, SETSUSP does not send any results. The instruction is only used in the translation of the VIMVAL operator, **affix**, which is responsible for the construction of streams.

---

if *I.opcode* = SETSUSP **then**
let
u = *I.op1*
R = *H*(*u*)
f = *I.op2*
i = *I.op3*
Act', *NewEis'* = *SendSignal*(*Act*, *NewEis*, u_{FA}, *destinations*)

∀ *v* ∈ N,
R'(*v*) = *R*(*v*) **if** *v* ≠ *f*
= *MakeSusp*(⟨u_{FA}, *i*⟩) **otherwise**
in
if *R* (*f*) ∈ ECQ ∧ |*R* (*f*)| = 0
then ⟨*Act'*,
NewHeap (*H*, *u*, *R'*),
NewEis'
Env⟩
else let *Act''*, *NewEis'*=
SendToDest(*Act'*, *NewEis'*, u_{FA}, ⟨**uncond**, *i*, **signal**⟩, **signal**)
in
⟨*Act*, *H*, *NewEis''*, *Env*⟩
endlet
endif
endlet
endif

2.5.4.5 Function Application and Return

The instructions which will be of greatest interest to us in the coming chapters will be those concerned with the manipulation of functions. There are four instructions in VIM which deal with this: APPLY, TAILAPPLY, STREAMTAIL, and RETURN. The APPLY instruction is the standard function application instruction, taking a function closure and an argument record and constructing a new activation for this function. By convention, the first operand of the first instruction in the activation receives the closure of the function, thus allowing the activation to access the free variables of the function, the first operand of the second instruction receives the argument record, and the first operand of the third instruction in the activation receives the

destination list of the APPLY operator. APPLY uses an auxiliary function, *MakeDest* which packages the destination entries found in the destination list of the instruction into a record and places this record on the heap. *MakeDest* takes in three arguments, the current heap, the uid of the current activation and the destination component of the instruction. It returns a record, α, of two elements, the first element contains the uid argument, and the second contains the uid of the record containing the elements found in the destination list. The uid of α is passed as an argument to the called activation. Placing the destination components into a record allows them to be accessed by the RETURN instruction in the called activation.

```
if I.opcode = APPLY then
 let
  C   = I.op1
  arg = I.op2

  <u_f, free> = H(C)

  u' = a new uid from U_A,
  u'' = a new uid from U_H
  Act' = AddAct(Act, u', H(u_f))
  H'' = AddHeap(H, u'', MakeDest(H, u_FA, I.destlist))

  Act'', NewEis' =
   SendToDest
    (SendToDest
     (SendToDest
      (Act', NewEis, u', <uncond, 1, op1>, C),
       u', <uncond, 2, op1>, arg)
       u', <uncond, 3, op1>, u'')

 in
  <Act'', H', NewEis', Env>
 endlet
endif
```

There is no explicit iteration construct in either VIMVAL or the base language. Instead, iteration is modeled using *tail-recursive* functions wherein the result of the iteration is obtained in the final recursive call. Otherwise, while the recursive call is being processed, the calling activation would exist merely to route the result back to the caller. To avoid having the calling activation persist until the call is complete, there is a special base language instruction for

handling tail-recursion, TAILAPPLY. The TAILAPPLY operator differs from the APPLY instruction in that it requires a third operand, which is the return address to which the result should be sent. By providing its own return link to the callee, the caller need not wait for the recursion to complete. We illustrate this process in Fig. 8. The tailapply instruction sends only signals to its targets. Typically, the target of TAILAPPLY will be a RELEASE instruction, described below, which reclaims the space used by this activation.

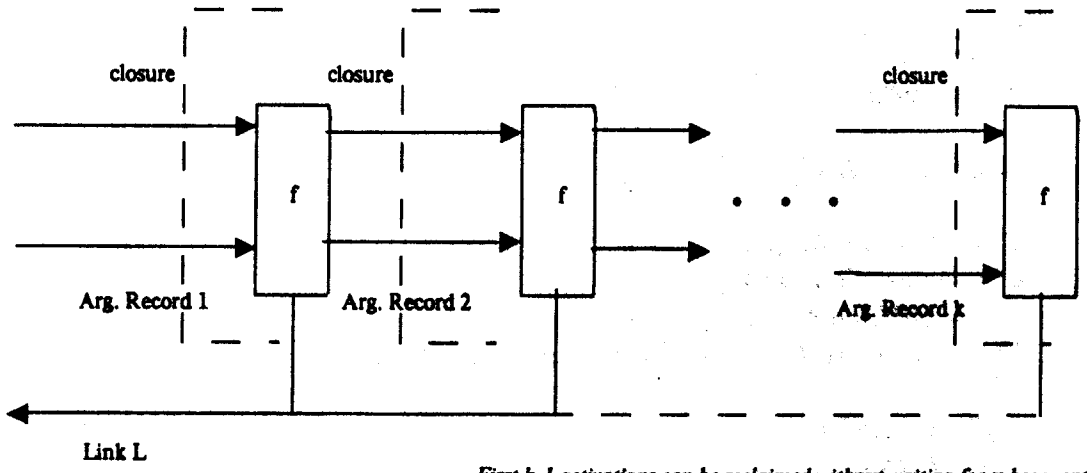


Figure 8: Tail application in VIM

```
if I.opcode = TAILAPPLY then
  let
    C = I.op1,
    arg = I.op2
    dest = I.op3

    ⟨u_f, free⟩ = H(C)

    u' = a new uid from U_A.
    Act' = AddAct(Act, u', H(u_f))

    Act'', NewEis' =
      SendToDest
        (SendToDest
          (SendToDest
```

```
        (Act', NewEis, u', <uncond, 1, op1>, C)
        u', <uncond, 2, op1>, arg)
        u', <uncond, 3, op1>, dest)

    Act''', NewEis'' = SendSignal(Act'', NewEis', u_FA, destinations)

  in
    <Act''', H, NewEis'', Env>
  endlet
endif
```

The STREAMTAIL instruction is similar to the TAILAPPLY operator in that both are used for implementing tail recursion. The STREAMTAIL instruction, however, is used in the implementation of a STREAM producer. Unlike the TAILAPPLY operator, the return link argument to STREAMTAIL is a record field in the last stream element created. When the next activation of the producer is instantiated, this return link will get SET to the uid of the new stream element. Thus, while the destination of the TAILAPPLY instruction is always an instruction, the return link of the STREAMTAIL operator must be a record field of the last stream element created. The basic structure of a stream producer using the STREAMTAIL instruction is shown in Fig. 9.

The RETURN instruction takes as input two arguments, a list of return addresses and a value. It sends to each of these return addresses the specified value and then sends signals to target instructions within its own activation. Unlike any other base language instruction, the execution of the RETURN operator affects instructions in activations besides its own. Thus, having the RETURN operator execute may lead to instructions in other activations becoming enabled. The value argument to the return instruction represents the value of the activation; no other effects of the activation will be visible outside of the value sent by the return operator to the receiving instructions in the calling activation. This property is a consequence of the applicative property of the base language. The RETURN instruction uses an auxiliary function, *GetDest*, which is the complement of the *MakeDest* function described earlier. *GetDest* when given the heap and the uid of the record, returns a set representing the destination list packaged in that record.

```
if opcode = RETURN then
  let
    DL = H(I.op1)  % the list of return addresses
    u_r = DL(1)  % uid of the calling activation
```

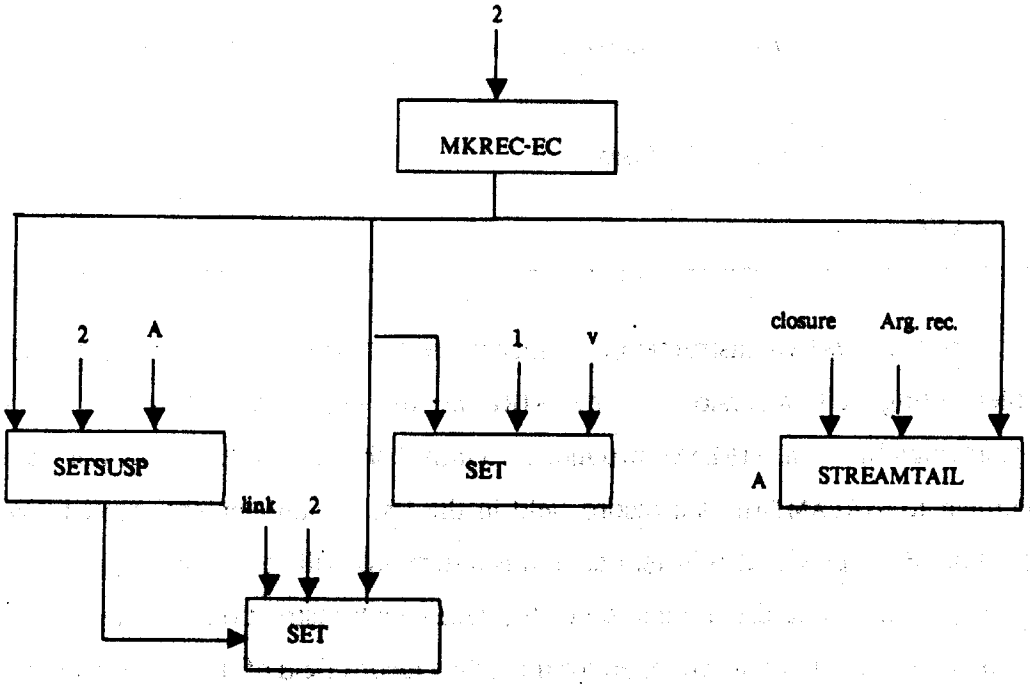


Figure 9: Skeleton of a Stream Producer

```
    targets = GetDest(H, DL(2))
    val = I.op2,          % the value to be returned

    Act', NewEis' = SendValue(Act, NewEis, u_F, targets, val)
    Act'', NewEis'', = SendSignal(Act', NewEis', u_FA, destinations)

  in
    <Act'', H, NewEis'', Env>
  endlet
endif
```

The last instruction we shall present is the RELEASE instruction. Unlike any of the other instructions presented thus far, RELEASE is not used in the implementation of any VIMVAL construct. Instead, it is used for memory management purposes in the machine. When all instructions within an activation have completed, the storage occupied by the activation may be reclaimed by the system. The RELEASE instruction performs this function. In a language without early completion structures, this operation would usually be a part of the RETURN

instruction, but because there may still be computations still in progress within the activation at the time the RETURN executes, it is necessary for a separate instruction to handle storage reclamation.

---

if *opcode* = RELEASE **then**
  ⟨*RemoveAct*(*Act*, u_{FA})
  H,
  NewEis,
  Env⟩
endif

2.6 Summary

In this chapter, we have presented a formal operational model for the VIM computer system. We introduced the application language, VIMVAL, and described the role of the Shell in the system. A rigorous definition describing the behaviour of some of the more interesting base language instructions was also presented. There are two key points raised in this chapter that should be noted. First, for the most part, an object created by an instruction is immutable. For those cases where it is not, as in early completion structures, the access and updating of these objects is carefully regulated to prevent incorrect information from being read. This feature of the system has major ramifications for the design of a backup system because it means that objects copied onto a backup storage device will, by and large, never need to be updated. The second characteristic of the system is the power of the individual base language instructions. Because of the expressive power of the base language, it should be possible to integrate the design of the backup and recovery system within the base language itself. The means by which this can be done is addressed in the following chapters.

We are now ready to develop the backup and recovery algorithms for the system. In the next chapter, we give a general overview of the approach we take in designing these procedures and the enhancements which need to be made to the system in order to support them. In subsequent chapters, we shall use the formal model given here to precisely describe the algorithms as well as to show their correctness. We will formalize our notion of "correctness" later in the thesis.

Chapter Three

The General Strategy

The goal of this thesis is to design efficient algorithms which guarantee complete security of all information in the VIM system against loss or corruption because of hardware malfunction. In this chapter, we discuss the general approach that we shall take in formulating these procedures. In the next section, we present a failure model of system operation. This model defines the appropriate context in which to reason about the design of the backup and recovery algorithms. In section 3.2, we raise some fundamental issues that must be addressed by the backup and recovery system. These issues are concerned with when backup is performed, how backup procedures are invoked and how the transfer of information from the main memory to the backup store is handled by the backup system. We shall be considering the problem of data security in the context of a single user system in which non-determinate computation is not allowed. Section 3.3 gives a high level design of the backup and recovery algorithms. We classify the information found in the VIM state into two different categories and discuss how the information in each of these categories is viewed by the backup procedures. We also describe the basic operation of the recovery algorithm in this section. Section 3.4 discusses the architectural enhancements that need to be made to the basic VIM architecture to efficiently support the implementation of these procedures. These enhancements are primarily concerned with the physical organization of the backup store. The last section is a summary of the chapter.

3.1 Failure Model

Many of the decisions that are made in the design of the backup and recovery system follow from the failure model that is assumed. A failure model is a specification of hardware behaviour characterizing the type of faults expected and the interaction between failed and non-failed components in the machine. Some of the factors which will influence the design of the backup and recovery algorithms that are described by the failure model include the frequency of failures in the system and the level of hardware error detection capability that is provided.

VIM is not a fault-tolerant system and, therefore, there will be faults that are not masked which will cause the system to behave erroneously. It is unreasonable to expect, however, that there will be no fault coverage in the system at all; like many conventional systems, VIM is

expected to provide enough fault coverage to correct many common errors arising from minor transient faults. Correction of single bit errors in memory, for example, is a feature which is found in many commercially available memory units and, thus, the services of the recovery utility should not be required when such an error is detected. In this thesis, we shall assume that the recovery utility is invoked only when errors cause information found on main memory or secondary store to be lost or corrupted. Power outage, short circuits, a malfunctioning disk head etc. are some examples of the type of faults which lead to such errors.

We do not expect that such faults will occur frequently; hardware is assumed to be reliable most of the time. We do make the assumption, however, that invalid information created because of an error is detected when it is accessed. For example, if a faulty disk head causes data to be written incorrectly onto disk, then when the data is read at some future time, the error will be detected. This assumption is important because it means that any information which the backup system observes and copies will either be correct or detected as being erroneous — invalid data is never maintained by the backup utility.

If the recovery utility is invoked, it will need to reconstruct the system state based on the information preserved by the backup facility. During this period, another failure may occur; the recovery facility must be robust enough to correctly restore the system state even following such circumstances. We discuss how this may be achieved later in the thesis.

3.2 Fundamental Issues

The backup system will need to interface with the interpreter and shell to monitor the progress of computations in the system. We need to decide, however, when it should actually get invoked and by whom. Secondly, once it is invoked, what information should it actually copy to the backup store? Thirdly, how should this copy operation be performed *i.e.* could normal system operation be intermixed with the execution of the backup procedures or must normal processing cease while the transfer of data is taking place?

The first question was already partially answered in the previous chapter where it was mentioned that the semantics of the base language instructions could be suitably altered to support the backup procedures. Unlike most conventional systems, the backup utility is not explicitly invoked by any process or user; it is implicitly activated whenever the appropriate base language instruction is fired. In a sense, the "logic" of the backup algorithms is distributed

among the various base language instructions described earlier. The question of which process activates the backup facility is not germane under this design. No one process is responsible for invoking the entire utility; different portions of the backup procedures are activated as instructions in an activation are enabled. In this sense, the backup facility is more an extension of the interpreter and shell, rather than as a separate program which is periodically invoked according to some predetermined policy. VIM offers the opportunity to make the backup facility more efficient because it is embedded within the interpreter and shell. This organization will allow us to design a backup facility that can observe the progress of computation to a greater degree than would otherwise be possible.

The answer to the second question involves determining how much information should be preserved and how the remainder of the system state can be derived from this data. While all data generated by the system could conceivably be copied onto the backup storage medium, it would not be a very practical solution because of the overhead incurred. Because all instructions in the base language generate data and update the system state, implementation of this strategy would involve modifying every base language instruction to send the result of their execution to backup store. This would clearly result in severe performance degradation. The backup algorithms will, therefore, need to maintain information about the system state in a condensed form which the recovery system can subsequently use to recover that part of the system state not explicitly preserved. As we shall see, most of the design effort for the backup and recovery system will be devoted to devising efficient algorithms to maintain and interpret these records. We discuss this issue in greater detail in the next section.

Because VIM is an applicative system, no data found on the heap, once created can be subsequently altered by either the backup system or the interpreter. Thus, having the backup procedures operate concurrently with normal system operation cannot cause any invariants over the data to be violated. Moreover, the data copied by the backup system will never be in an inconsistent state when the copy operation is performed because no updating of information takes place. Our backup procedures can, therefore, be allowed to execute concurrently with normal system operation[1] without the need for any explicit consistency checks.

During the recovery process, no shell commands are accepted by the system. If shell

[1]There is a caveat to this claim which will be explained in the next chapter

commands could be processed concurrently with the recovery process, it may be possible to have computations reference data which has still not been restored by the recovery procedures. In addition, this restriction also simplifies the interface between the recovery procedures and the shell by avoiding the need for any synchronization protocols between the two processes in updating (or deleting) environment entries. When the recovery procedures are invoked, they make no assumption about the integrity of the data which may still be accessible. Thus, the only information used in the reconstruction of the state is that found on the backup store. Of course, it is inefficient to restore the entire state of the system if only a fraction of it were affected by an error. Significant complexity is added to the backup and recovery procedures, however, if we require the system to support partial recovery. It is not clear whether the benefits derived from implementing partial recovery outweighs this increased complexity. We shall address this topic again later in the thesis.

In the next section, we present the high level design of the backup and recovery algorithms. The rationale for our design decisions have been mainly based on the effectiveness of these algorithms in addressing the questions raised in this section.

3.3 A High Level Overview of the Backup and Recovery Facilities

The design of the backup and recovery facilities are based on one important observation: *every computation in the system is associated with the evaluation of some shell command input to the system.* Thus, one immediate solution which presents itself is to simply record all shell commands on the backup state. This is obviously a correct solution since the behaviour of the system is presumed to be determinate. Reexecuting, in the proper serial order, the shell commands that were input to the system before the failure occured is, therefore, guaranteed to yield a correct state. This state will be identical to the state immediately prior to the failure except for the uid's associated with structures and activations. The uid's chosen during the recovery process may not be the same as chosen originally[2] but because these uid's are not visible to the programmer, no difference in the two states will be externally discernable. Obviously, such a scheme would inflict little degradation to system performance since only the text of the shell command need be maintained. On the other hand, recovery would be intolerably slow because every shell command is reexecuted from scratch with no information about the results of

[2]Recall from Chapter two that no restrictions are made on how uid's may be selected

these commands being kept on backup store. The recovery system, starting from some initial recovery state, would need to reexecute every shell command input to the system from the start of system operation because no information about the result of any of these commands are recorded. Such a major drawback makes this strategy unattractive for all practical purposes.

To see what optimizations can be made to alleviate this problem, let us examine how shell commands are used to alter the system state. The shell command of interest to us here is the BIND command. The BIND command binds a name to the result of some computation and places this binding in the user's environment. The value bound to the name represents a long-lived value — it survives the computation in which it was created. The data seen by the user of the system are precisely the values bound in his environment. Since these values have lifetimes greater than the computations in which they were defined, it would certainly be a major optimization if the backup facility maintained these values on the backup store. This would obviate the need for the recovery system to reexecute those commands whose evaluation produced these values. The backup facility must now, in addition to maintaining a log of the commands input to the system, also record all $\langle name, value \rangle$ bindings found in the user environments. These bindings cannot be arbitrarily changed because VIM is applicative; thus, once a binding is recorded on backup store, the backup facility need never again reexamine that entry in the user environment to check if it has been altered.

The data found in VIM may be classified into two categories; *quiescent*, which corresponds to the result values of computations associated with BIND commands that have been bound to a name in a user environment, and *transitional*, which corresponds to those values that are either part of some active computation or result values of a computation that have not yet been bound to a name in some environment. A computation consists of the collection of activations and data created during the evaluation of a shell command. Instructions in these activations produce transitional data since this data will survive for only so long as the computation in which it was created exists. The value finally produced by a computation will get bound in an environment and, as a result, will become quiescent. The backup facility, according to the scheme presented above, would only be aware of quiescent data. Transitional information corresponding to data produced during a computation would not be under the scrutiny of the backup procedures. When a failure takes place and the system state needs to be properly restored, the recovery facility first restores all quiescent data preserved by the backup facility onto the new recovery

state. It must then reexecute those shell commands found on the command log which had either not yet completed or whose result binding could not be recorded by the backup facility before the failure. These shell commands will be referred to as *volatile* commands.

Having the backup facility record only quiescent data is an optimization that reduces overall recovery time. It does so without excessive performance degradataion because the backup facility is only invoked when an ADDTOENV instruction executes to actually place the binding in the environment and, moreover, can perform the copy of the data in parallel with normal system operation. It would be an even greater optimization if the backup system could help reduce reexecution time of those volatile shell commands found on the log by recording information about those computations which were active at the time of failure. If there are many resource intensive, time-consuming activations in a computation, then having the backup facility ignore the presence of the transitional data produced in this computation means that the time to recover this state must be at least bounded from below by the time it takes to reexecute this entire computation. This is not very desirable since the computation could have already been executing for a very long period when the failure took place. Maintaining some record of the computation on the backup store would allow the recovery facility to avoid needlessly reexecuting those subparts of a computation which had already produced their result before the failure.

It is reasonable to expect that there will be many computations in progress at the time of failure. To record the progress of these computations, the backup system maintains information about these computations on a *computation record* on backup store. We discuss the structure of a computation record and how computation records are updated by the backup facility in the next chapter.

We can now present our intended model of system operation for VIM. As computations complete, causing values to be bound within some environment, the backup facility preserves these bindings on the backup storage medium. In addition, there will also be many active computations in progress. The backup facility maintains information about these computations as well. This information, embodied as a computation record, can be used by the recovery procedures to avoid needless reexecution of computations which had already produced their result before the failure. When the recovery system is invoked, it first restores all quiescent data found on the backup store. It then uses the computation records to restore the remaining part of

the state. The state after recovery is complete will be equivalent to the state which existed prior to the failure insofar as the structure and information content of both states will be the same. The states need not be identical, however, because the uid's associated with activations and structures may be different. The reason why the states would not be identical is because the order in which enabled instructions are chosen for execution may be different during the recovery process than before the failure. This does not compromise the correctness of the recovered state because of the applicative nature of VIM — no side-effects occur and, thus, no explicit ordering on instruction execution needs to be adhered to. We illustrate the system operation in Fig. 10.

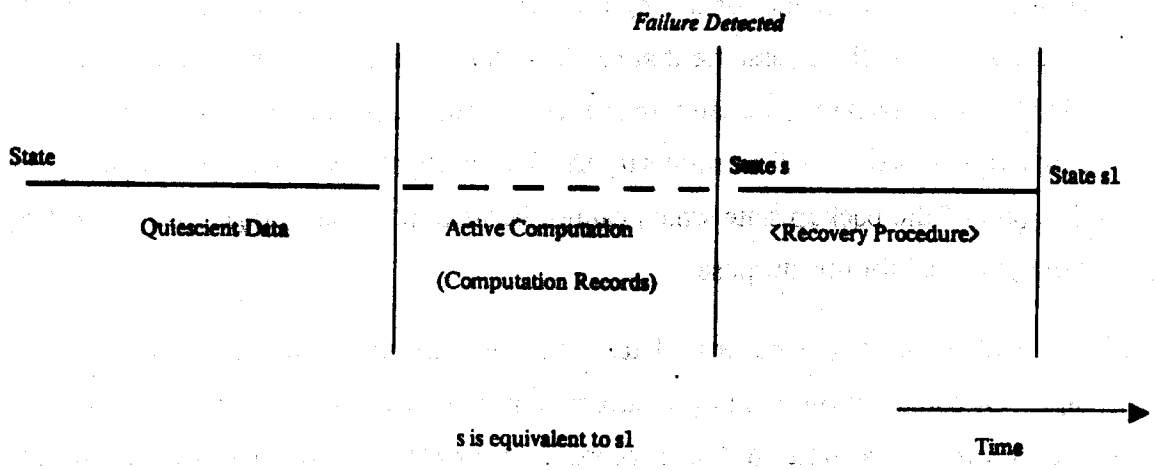


Figure 10: System Operation

3.4 Architectural Enhancements

Up to this point, we have only mentioned that the backup store on which backup data is kept has the property that information entrusted to it will survive failures of the machine with very high probability. The most common type of backup store used is magnetic tape. The sequential access nature of tape drives, however, makes it inconvienent to update the information found on the tape. Since the backup facility will be frequently updating computation records associated with active computations to reflect the progress of the computation, using tape as the only backup storage device would be impractical. Our design dictates the need for a fail-safe storage device from which information may be easily accessed.

updated, and deleted. We call a device which has these properties a *stable storage device.* It is not difficult to implement such a device on top of non-stable storage devices. Lampson [21] gives one implementation of a stable storage device in which disk storage is converted in to stable storage by maintaining multiple copies of the data on different disks and ensuring that all writes to disk are *atomic i.e.* the write either takes place on both disks or on none. Because both disks are guaranteed to have consistent information, data lost because of failure of any one disk can be recovered from the other. Advances in VLSI technology have also made it conceivable to consider a hardware implementation of stable storage using, for example, CMOS static RAMS and a backup battery supply. Because of the low power consumption of CMOS chips, information on RAM could be retained, despite power failure, using the backup battery supply. In this thesis, we shall not be considering implementations of stable storage but will assume that such a device is available for use by the backup and recovery utilities. Because of its relatively high cost, we shall also assume that stable storage is not very large (certainly much smaller than the size of the backup state) and, therefore, in order to guarantee that backup information is not susceptible to loss, it will be necessary to have another backup storage device capable of holding that part of the backup state which cannot be held in stable storage. We assume magnetic tape storage is used for this purpose.

Quiescent data is never updated by the backup procedures and, therefore, can be kept on tape. Of course, if the binding is subsequently deleted, the data will have to be removed from the backup state as well. A delete record indicating that a value has been removed from the user environment can be written onto tape in such situations. The computation records associated with active computations do need to be accessed and constructed relatively frequently. These records will, therefore, need to be held on stable storage. In addition, the command log, which contains all shell commands input to the system whose results have either not yet been produced or have not yet been recorded onto backup store, will also need to be held on stable storage. As we shall see, most computation records will be relatively short-lived and, thus, will not occupy stable storage for any significant amount of time. The final value of a computation record is quiescent and can be migrated onto tape, allowing the space used by the computation record to be reclaimed. It is expected that stable storage will always be able to support all computation records in the system because of their short lifetime. When the recovery system is invoked after a failure is detected, it will first read from the tape all the quiescent data and will restore as much of the environment image as possible from this data. Volatile shell commands are then executed

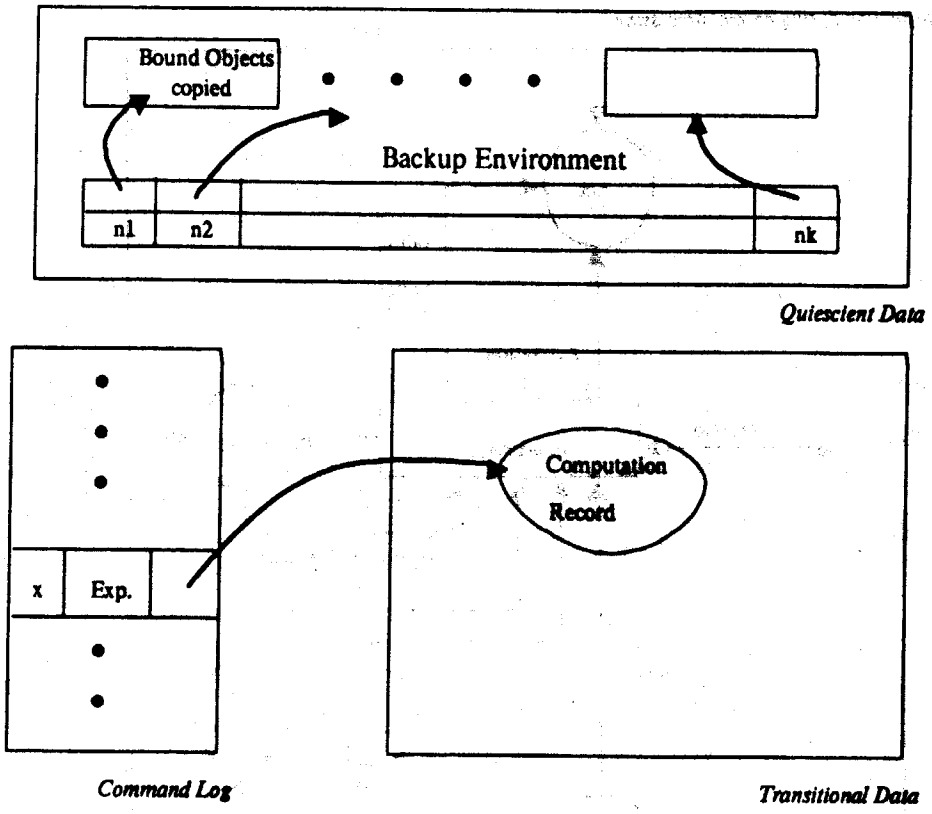


Figure 11: High Level Organization of the Backup Store

from the command log in the order in which they were originally input. The computation records found on stable store are used to reduce the overall reexecution time during this phase. When this phase is complete, the system can proceed with normal operation. The high level organization of the backup store is depicted in Fig. 11.

We illustrate the organization of the VIM system with the backup heap and environment in Fig. 12. The backup heap is used to hold all transitional data whereas quiescent data is held on the backup environment. The backup heap and environment constitute the VIM backup store. The Interpreter constructs computation records on the backup heap during normal processing and interprets them during recovery. In addition, the intermediate results of a computation are also stored on the backup heap by the interpreter. *<Name,Value>* bindings are placed on the backup environment by the *Shell* which also builds the command log found on the backup heap.

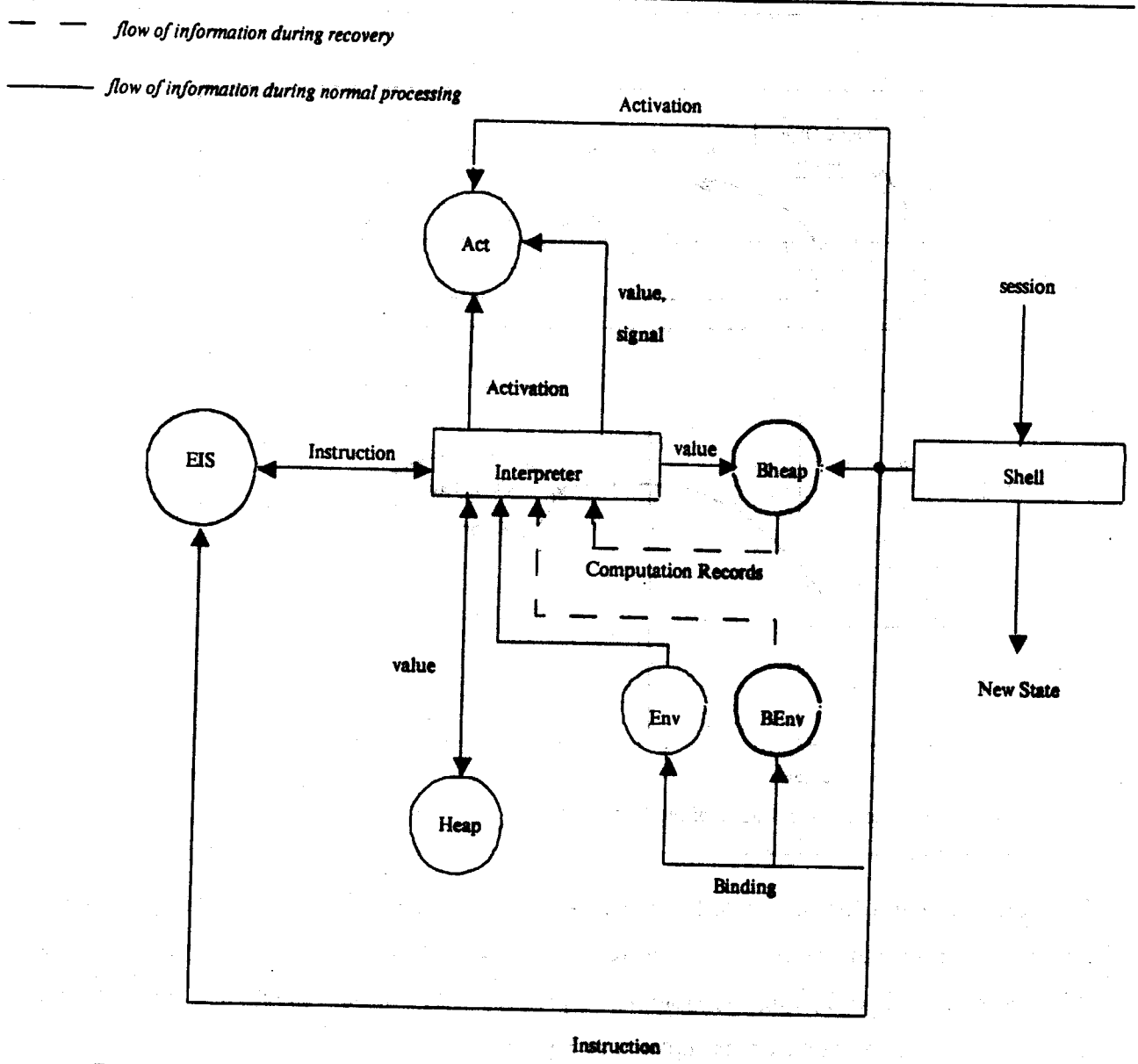


Figure 12: Abstract Architecture of the VIM System with Backup Store

3.5 Summary

In this chapter, we have presented a high level strategy for the backup and recovery algorithms for the VIM system. Our main observation about system behaviour was that all active computation in the system is associated with the evaluation of some shell command. The first solution proposed involved simply storing the log of all shell commands, reexecuting them from the beginning if a failure occured. While correct, because VIM is a determinate system, this

solution has the drawback of a very slow recovery time. A major optimization to this solution is to record all quiescent data *i.e.* data bound in some user environment. A further optimization, intended to reduce the overall recovery time in reexecuting volatile shell commands is to have the backup facility maintain some measure of information about all active computations. The recovery facility uses this information to avoid needless recomputation. Once this reexecution phase is complete, the state of the system is properly restored. During this reexecution phase, the order in which instructions are executed may be different from the original execution sequence. This may lead to different uid's being assigned to different structures but the overall structure of the heap and activations component remain identical. The reason why the order of instruction execution is not important during the recovery process is because VIM is an applicative system.

We also introduced the notion of stable storage in this chapter. Information about active computations will need to be frequently recorded by the backup system. A backup storage device on which data can be easily accessed and added is required to support these computation records. While quiescent data can be copied onto tape storage, computation records need to be maintained on stable store.

In the next chapter, we present the detailed organization of a computation record and discuss how the backup system monitors the progress of computation. As we noted earlier in this chapter, the logic of the backup procedures is actually distributed among certain base language instructions. We present a formal model of the VIM system supporting the backup and recovery procedures and argue that the information embodied in the computation records is consistent with the actual system state being represented.

Chapter Four

Constructing Computation Records

A major aspect of the backup and recovery algorithms for VIM concerns the construction of computation records. Recall from the last chapter that a computation record is used to record information about currently executing computations. In this chapter, we shall be primarily interested in how computation records may be constructed and maintained. In Section 4.1, we present the abstract representation of computation records. The main component in the record is known as an *activation descriptor entry* which embodies state information about an individual activation. In order to construct a computation record, changes to the operational behaviour of the base language instructions given in Chapter Two will be necessary. Section 4.2 discusses these changes as well as changes necessary to the shell and interpreter. In section 4.5, we present the altered operation of the base language instructions in terms of an abstract operational model, MR, which is an extension of model M1 presented in Chapter two. There are several major optimizations which can be made in managing computation records. These optimizations are also formalized in this section.

4.1 The Computation Record

In Chapter three, we argued that the backup system should record the progress of active computation in the system to help reduce recovery time. A computation record is an information structure constructed by the backup system for this purpose. Our focus in this section will be on determining how much information should be kept on the computation record to allow the recovery procedures to restore the system to its state prior to the failure. One simple scheme would be to periodically *checkpoint* all activations created by a computation. To checkpoint an activation means recording the state of all instructions which have not yet executed in that activation at the time of the checkpoint. The state of an instruction consists of its opcode, destination list, operand and signal count, as well as the value of its operands[3]. This approach would be very similar to that taken in many other parallel computer systems where a *recovery point* representing the state of one of possibly many concurrently executing processes is

[3] If an operand is a complex structure, this means recording all substructures referenced from the top-level structure as well.

periodically taken by the systems' backup facility. In our system, the recovery procedures would only need to find all enabled instructions in the computation records to begin the reexecution phase.

Periodically recording the state of all activations created by a computation is simple idea but has two major drawbacks which does not make it a feasible solution for our purposes. First, checkpointing all activations in a computation will probably be a costly task because the size of activations can be very big. Secondly, in order to guarantee that a consistent image of the computation is maintained on the backup utility, we would have to disallow any enabled instruction within any activation in that computation from executing while the checkpoint of that activation is being performed. To see why this is the case, consider two activations, α and β in the same computation where α has called β. If β is allowed to execute while a checkpoint of α is being made, then β may return its result to α and then execute a RELEASE operation. If the image of α on backup state does not reflect the return value sent by α, and β was not checkpointed before the RELEASE operation executed, the computation record would represent an incorrect state. Upon recovery, there would be no way to recover the return result value of β without reexecuting α. This is precisely a manifestation of the problems encountered in other concurrent systems that use recovery points to guarantee data security.

A more clever approach to recording state information about activations takes advantage of the applicative programming model VIM uses. A distinctive feature of an applicative language is that each function can be treated as a *constant applicative form caf*. A caf consists of constants combined by function composition and application. In conventional programming languages such as Pascal or Fortran, a function cannot be treated as a constant because its evaluation may cause side-effects to occur in the program. In VIM, having all functions be simply constants means that the behaviour of the function can be determined by just knowing its inputs. In the base language, an activation is the application of a function to some input. Because functions are caf's, we can embody an activation on the backup state by recording the function closure, its inputs and return link. Under this scheme, the recovery system would need to only APPLY the closure to the argument list to construct the corresponding activation being represented. An important advantage of this proposal over the checkpointing one is that no executing code is maintained on backup store. Because all data is immutable, the transfer operation of the data from main memory to backup store can proceed in parallel with the execution of any activations

which operate on this data. Moreover, the amount of information which needs to be copied is also greatly reduced since no data created by instructions within these instructions are preserved. Such data would be recovered when the activation is reexecuted.

A natural representation for a computation record in this scheme is as a directed tree in which nodes represent activations and edges indicate caller/callee relationships between pairs of activations. This tree is known as the *computation tree* for the computation. We illustrate this representation in Fig. 13.

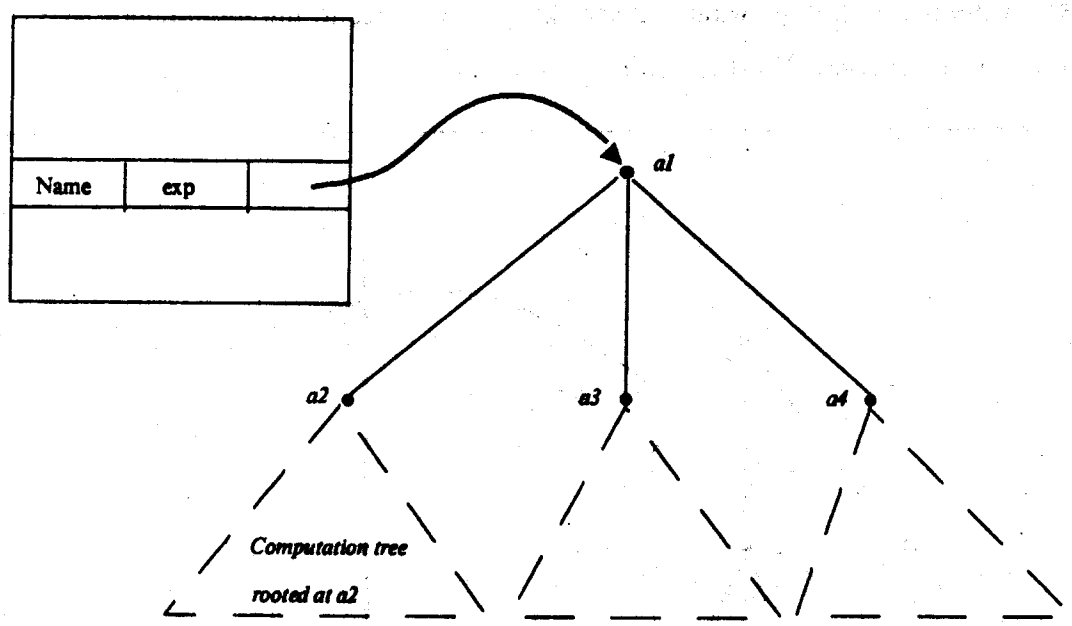


Figure 13: Representation of a Computation Record

The root of a computation record represents the initial activation constructed by the *Translate* function of the shell. Every node in the computation tree is labeled with the uid of its corresponding activation. If (s,t) is a member of E_c, the set of edges in a computation tree c, then activation t is instantiated from activation s. Each computation has a unique computation tree.

A node in a computation tree is called an *activation descriptor entry.* An activation descriptor contains the necessary information about an activation needed to restore the state of the activation. The representation of a computation record given above is simple but certainly does not help much to alleviate recovery time for transitional data. This is because computation

trees may become very large for long running computations. Reducing the size of the computation tree would speed up the recovery process. The information in an activation descriptor entry in this scheme contains the closure and argument list of that activation. During recovery, however, *every* activation represented by an activation descriptor entry in the backup state would be *reexecuted*. The time to reexecute all these activations would result in an unacceptably high recovery time. Clearly, what is needed is a mechanism to record results of activations as well as their instantiations. Thus, when a result of an activation is known, it replaces that activation in the computation tree. When this value is encountered by the recovery procedures, it is sent directly to the destination addresses, eliminating the need to reexecute any activation in the subtree rooted at the node containing the result. In this way, we can imagine the computation tree growing and shrinking in response to the instantiation and completion of function activations. This process is shown in Fig. 14.

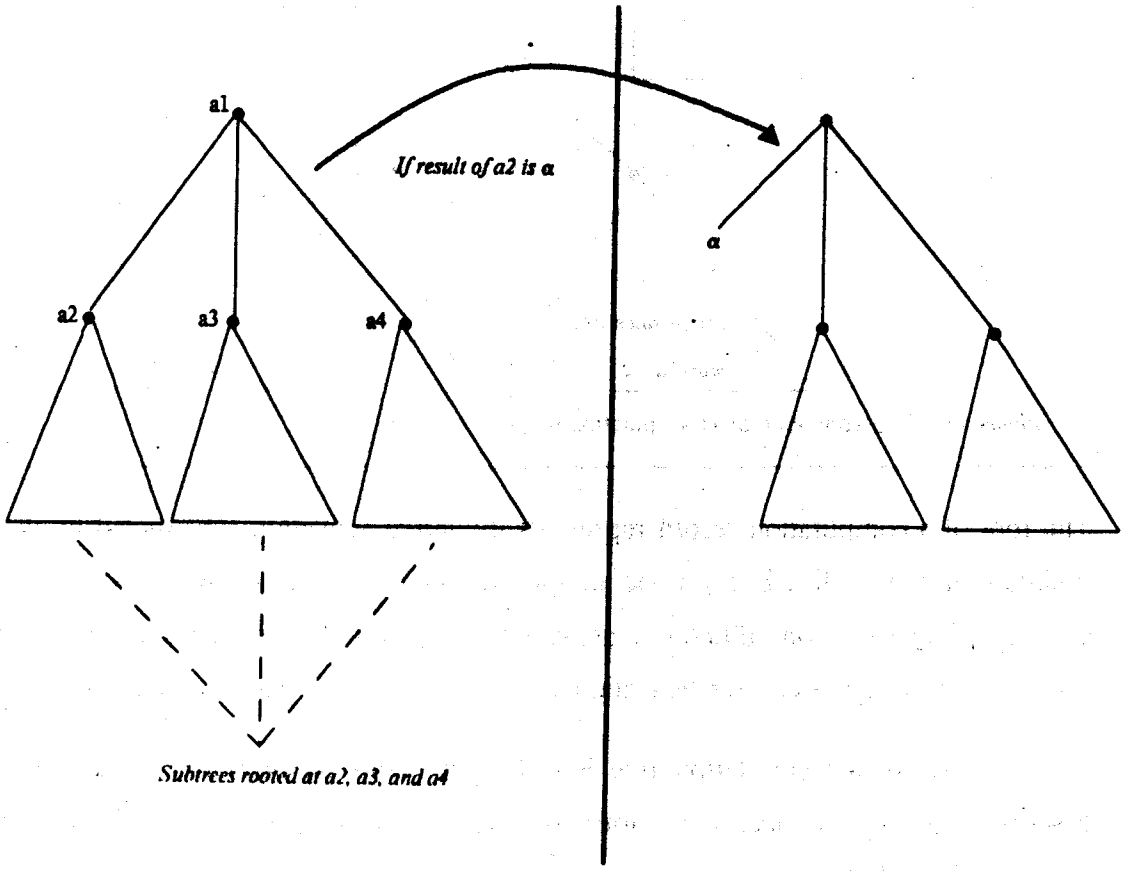


Figure 14: Dynamics of a Computation Tree

When an activation is instantiated, a new descriptor entry is placed in the computation tree and an edge is added from the caller to this new entry. When the result of an activation becomes known, it replaces this node in the computation tree. All subordinate activation descriptor entries are then removed. There are two features of our scheme that distinguish it from typical data backup strategies found in conventional systems. The first is that the updating of state information in our scheme is dependent totally on program behaviour. As we had mentioned in the last chapter, typical backup strategies use a predetermined policy to determine when the checkpointing is to be done. The second, and more important, difference is that because the construction of the computation tree takes advantage of the applicative programming model, the computation tree can be "pruned" whenever a result of an activation becomes known. In a language which permits side-effects and sharing of non-local variables, we would not be able to manage the backup image of the computation in this manner.

It now remains to show exactly how the interpreter, shell and base language semantics need to be modified to support the construction of these computation records. We examine the structure of an activation descriptor entry in greater detail in the next section.

4.2 The Activation Descriptor Entry

We had mentioned in the last section that an activation descriptor should contain enough information so that the recovery procedures could restore the state of the activation. We observed that if the function closure and argument list of the activation were preserved, then the recovery procedures need only apply the closure to the argument list, setting the return link, to restore the activation state.

The recovery process in this proposal is straightforward. All activation descriptor entries containing the function closure, argument record and return link can have the function application take place in parallel. Whenever an apply operation is to be executed during the recovery phase, a check is made to see if the function has already been instantiated. That is, all APPLY instructions during the recovery phase check to see if an activation descriptor already exists for the activation they are to initiate. If one exists, we can effectively ignore the instruction since the result of the application is already known. If no such descriptor exists, we perform the application. Result values found on an activation descriptor entry are used to prevent initiation of an activation. When a value is found in an *ADE*, it can be sent directly to the destination

address specified in its return link[4].

In a system in which errors requiring intervention of recovery procedures are assumed to be relatively infrequent, we may find the cost of even maintaining function closures and argument records too expensive. As we explain below, the main reason for needing the closure of the activation on the backup store is if we wish to have all activation descriptor entries evaluated in parallel. If we are willing to tolerate longer recovery time, we can signficantly reduce the amount of information which needs to be held on the activation entries by eliminating the need to hold even the function closure or argument record on backup store.

Instead of having all activations initiated in parallel, we can have the computation reexecuted from the initial activation descriptor in the computation tree. Whenever a new activation is about to be initiated, the recovery procedure first examines the corresponding activation descriptor entry for that activation (remember that there is a unique computation tree for every computation). If that entry contains a result, then that value is used directly and the new activation is not initiated. If, on the other hand, no result value is found in the descriptor, a new activation is constructed and processing proceeds as normal. No function closure or argument record needs to be maintained in this scheme because the computation is reevaluated from its initial activation — no parallel invocation of activations within the computation takes place. While the time to restore a computation is greater than if function closures were maintained in the backup state, it is bounded by the time the system would have taken to have processed this computation under normal circumstances. If function closures and argument list of activations were maintained, then the recovery system could exploit more parallelism than what was available during the original evaluation of the computation precisely because all activation descriptor entires in the computation record could be evaluated concurrently. It is important to keep in mind, however, that even if this extra informtion is not kept on the activation descriptors, the reexecution of the computation would still exhibit as much concurrency as it would have under normal conditions.

The information held by an activation descriptor entry must allow the recovery procedures to determine if an activation needs to be initiated or not. There are two forms of an activation descriptor. The first form is for those activations whose results were recorded by the backup

[4]We are assuming that uid's of activations are preserved in the backup state and are used during the recovery phase

facility. If an *ADE* for an activation α contains a value, then this value represents the result of α. The second form of an *ADE* is used when the result of an activation has not yet been recorded by the backup facility. In this case, the *ADE* contains the edges to all activations that were initiated from α before the failure occured. During recovery, α will be reexecuted. Recall that edges in the computation tree represent caller/callee relationships between activations.

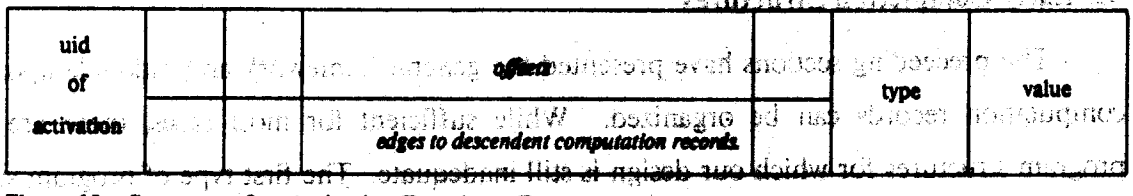


Figure 15: Structure of an Activation Descriptor Entry

We represent an edge in activation descriptor entry as a two-tuple *<offset, uid>* where *offset* represents the offset in the activation array of a function application instruction which instantiates the new activation and where *uid* represents the uid of the activation descriptor associated with the called activation. The *offset* component uniquely identifies the activation being initiated. When an application instruction is encountered during recovery, the activation descriptor corresponding to the activation to be instantiated is examined. If a value is present in this *ADE*, it can be sent directly to the destination instructions of the operator. The offset field is used to locate the *ADE* of the activation to be instantiated. The reason why we need the *offset* field at all is because the order in which instructions are executed during recovery may not be the same as the order in which they were originally executed. In sequential languages, every function application is ordered with respect to every other application. Reexecution of a given activation will not change this ordering. In a concurrent system such as VIM, there does not exist any *a priori* ordering on instruction execution. Thus, the instruction number, *offset*, of the function application operator i.e., APPLY, TAILAPPLY, or STREAMTAIL must be kept on backup heap to properly identify it during the recovery process.

The scheme we have given above requires little intervention by the backup facility. Function application is noted by adding a new activation descriptor to the appropriate computation tree. When the result of an activation is known, it replaces the descriptor entry for that activation on the backup state. We do not maintain argument records and closures of

activations, choosing instead to reexecute the computation from the beginning, only avoiding reexecution of activations whose results are already known. While recovery time in this proposal is greater than the one in which closures and argument records are maintained, there is a substantial reduction in the computational resources required by the backup facility. As we have mentioned previously, efficient implemention of this strategy will require alteration of some of the base language instructions. A formal definition of these operators is given in Section 4.5.

4.3 Early Completion Structures

The preceeding sections have presented the general framework and rationale upon which computation records can be organized. While sufficient for most cases, there are certain program structures for which our design is still inadequate. The first type of program structure not properly addressed in our presentation is the *early completion* structure. By itself, an early completion structure does not add useful information to the backup state since it only indicates that an activation has been initiated to produce the desired value. Copying an early completion structure onto the backup store would not allow the recovery system to restore the proper state unless the activation responsible for producing the value which is to replace that structure is also copied. This is obviously not a desirable situation to have to deal with.

For our purposes, a simpler (and more efficient) solution is to avoid copying early completion elements until all the early completion fields in the structure become set. When a structure which contains early completion fields is to be copied onto backup store, these *ec*-fields are labeled with a special flag indicating that the containing structure is to be eventually copied. When a SET instruction encounters such a field, it checks to see whether any other early completion elements exist in the structure; if so, no copy operation is performed; otherwise, the structure is copied onto the backup store. The *ADE* which is to reference this structure will initially have its value field reference a structure with a special value, **notcopied**. When all fields in the structure are known, this reference gets replaced with the reference to the fully defined structure. If the recovery routines encounter a structure with a value **notcopied**, it is treated as not being defined and is ignored during the reexecution process. The structure containing the early completion element being set may be a component of a larger structure which also needs to be copied onto the backup store. If this structure becomes fully defined as a result of executing this instruction, then it too will get copied onto the backup store. We illustrate the effect of the SET operator on the backup store in Fig. 16.

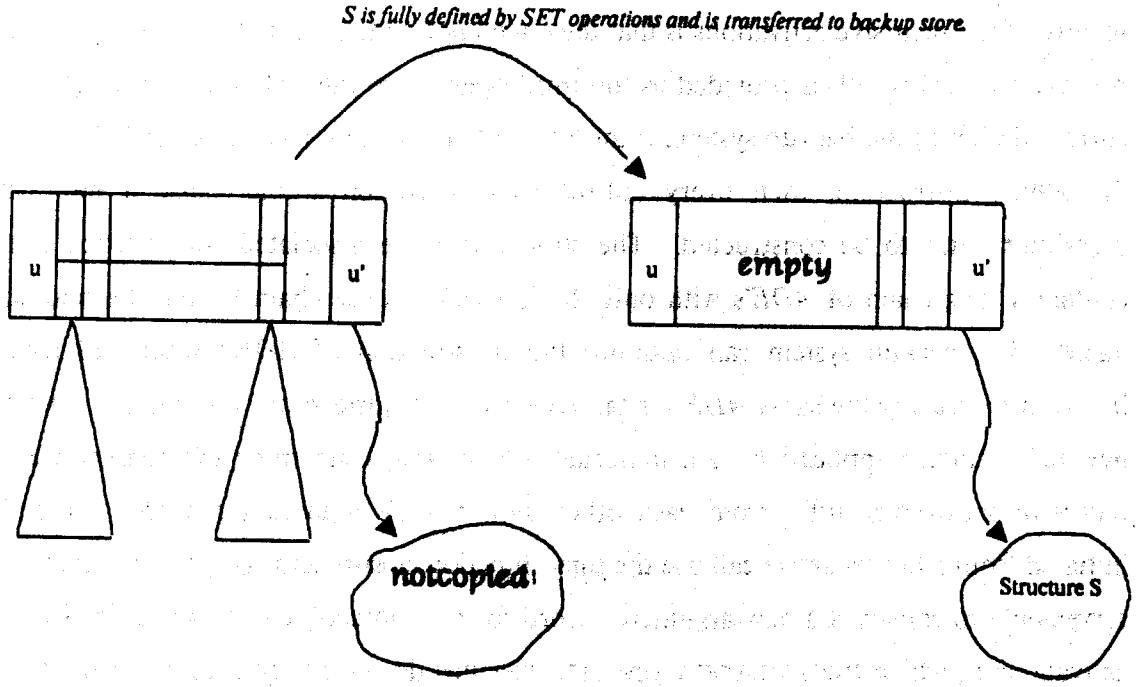


Figure 16: The Effect of the SET Operator on Backup Store

4.4 Further Enhancements

There are two other program structures whose behaviour cannot be efficiently captured by the backup facility by just modifying the semantics of the APPLY and RETURN operators. The first is the *tail recursive* program expressed using the TAILAPPLY operator. The second class of programs not handled by our system are those involving the production of *streams* implemented using the SETSUSP and STREAMTAIL instructions. Both these classes of programs use special function application and signalling operations which require more sophisticated algorithms than those presented above. In the next two subsections we discuss how the backup system should be augmented to handle tail recursive activations and demand driven evaluation of stream structures.

4.4.1 Tail Recursion

Recall that tail recursion is used to implement iteration in the base language. The key feature of tail recursive activations is that they need not persist until the recursive call completes because the return link is provided as the third operand to the TAILAPPLY instruction. In our current design of the backup system, if the TAILAPPLY operator was treated as being identical to the APPLY instruction, then every tail-recursive activation would cause a new activation descriptor entry to be constructed. The structure of the associated computation tree would contain a long chain of *ADE*'s with only the last *ADE* in the chain having the relevant result value. The backup system can optimize the construction of *ADE*'s when tail recursion is involved by *reusing* the same *ADE* for tail recursive calls instead of building new ones for each new tail recursive application. An important observation concerning tail recursion is that tail recursive activations differ from each other *only* in their argument records. All activations initiated from a tail recursive call use the same function closure and return link. Each activation serves only to construct a new argument record for the succeeding one to use. In fact, because activations in which the TAILAPPLY operator executes do *not* return a result value, there is no RETURN instruction which is executed. It should be clear that this behaviour is not well supported by our backup algorithms which very much depend on results of activations being recorded on backup store in order to help reduce recovery time of volatile commands. The reason for this incompatability is the fact that no tail-recursive activation except the last returns a result, making any intermediate tail recursive activation descriptor entries essentially useless.

We introduce a new type of activation descriptor for tail recursive activations which includes the argument record of the activation. When a function is instantiated by an APPLY instruction, an activation descriptor is constructed for it with type ***apply***. If this function was tail recursive, then during the evaluation of this function a TAILAPPLY instruction may execute. Execution of this instruction, while causing a new activation to be added to the set of activations in the system, does not necessitate a new activation descriptor to be constructed as well. Instead, we change the activation descriptor of the current activation to type ***tailapply***. The argument record passed as the second operand to the TAILAPPLY instruction is recorded in this activation descriptor. In addition, all edges emanating from this *ADE* are removed. The old state of the activation descriptor is thus replaced to reflect the new activation. Other function applications that take place in the activation are recorded in the ***tailapply*** *ADE* as was done in the ***apply*** *ADE*. Subsequent tail recursive calls in this activation will cause the same effect as took place

initially: the old argument record is replaced with the argument record of the new activation, and the edges emanating from the *ADE* are removed.

The inclusion of the argument record in the descriptor allows the recovery system to avoid reexecution of all the tail recursive calls leading up to the one represented on backup store. Since the closure and return link are the same, keeping the argument record in the *ADE* makes it unnecessary to reexecute any of the prior tail recursive activations originally executed from the initial APPLY. The representation of tail recursive activations we have chosen has two beneficial aspects. First, the depth of the computation tree is not increased for every tail recursive call since the new *ADE* can replace the *ADE* of the calling activation. This is because tail recursive activations send their result directly to the address specified in their third operand; the calling activation does not receive the result of the callee. Secondly, by storing the argument record on backup store, reexecution can begin by applying the function to this argument record and the return link provided by the APPLY instruction which initially instantiated this function.

In Fig. 17, we show some steps in the transformation of a computation tree which embodies the evaluation of the following function to illustrate the process described above:

```
Function Example (f: Function, n : Integer returns Integer)

  Function Tailexample(m,n : Integer, f:Function returns Integer)
    if m > n
      then m
      else Tailexample(f(m), n)
  endfun

  Tailexample(1,n,f)
endfun
```

4.4.2 Stream Structures

Our basic approach to recording the progress of computations is also not well suited for expressing the behaviour of computations involving the production of stream structures. Recall from Chapter Two that streams are produced by tail recursive functions in a demand driven fashion. The unique instruction in a stream producer which allows the lazy evaluation of a stream is the suspension operator. The backup and recovery algorithms as currently defined are not capable of modeling the kind of program behaviour exhibited by stream producers for reasons discussed below.

Steps in the computation of TailExample, with initial arguments: m = 1 n = 1 and f(x) = x + 1.

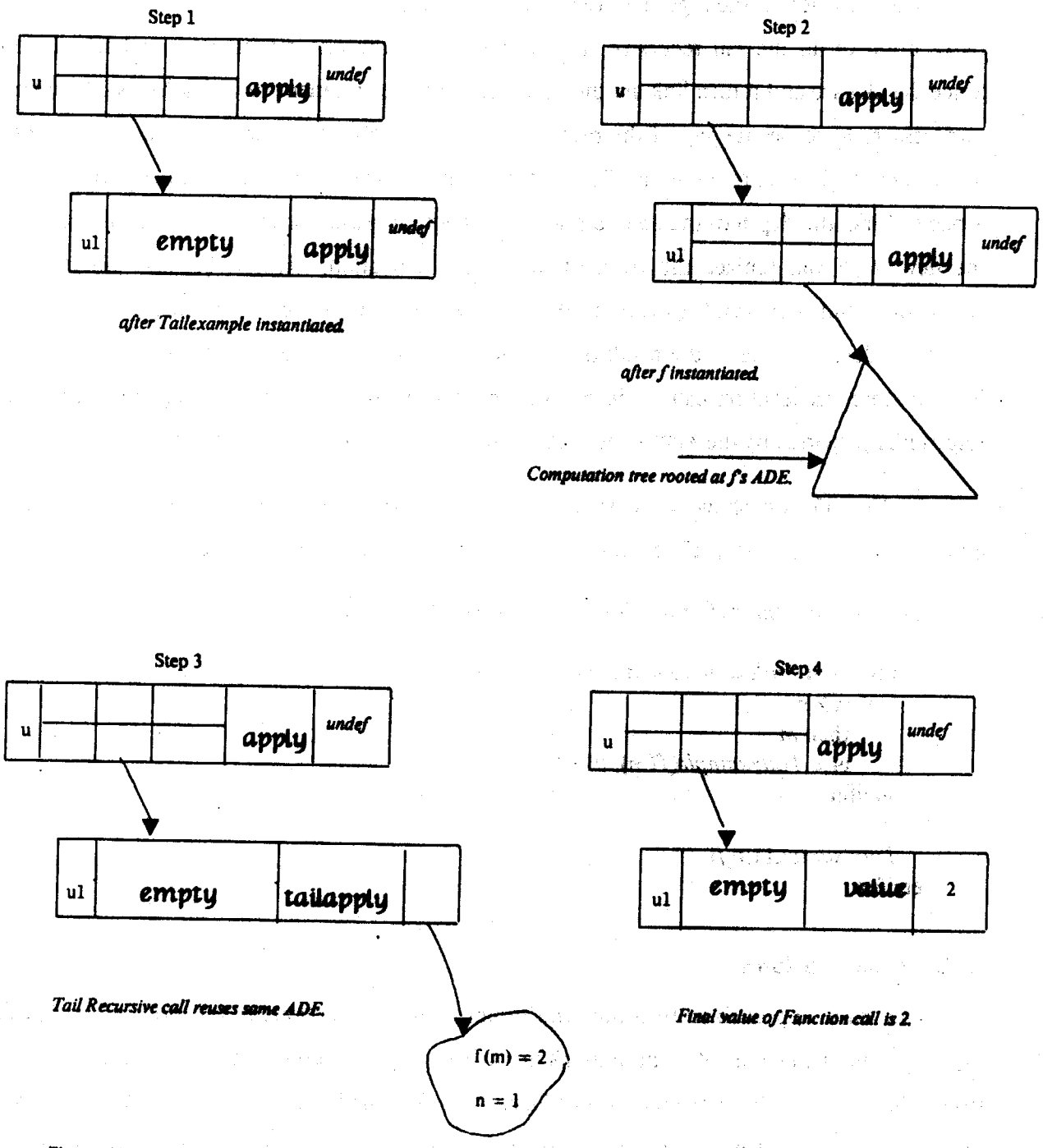


Figure 17: Handling Tail Recursion

4.4.2.1 Rationale

To see why our current method is insufficient, consider the structure of the computation tree produced by the backup algorithm (as currently defined) for a stream producing function. Since such a function is tail recursive, its associated activation would be represented by a *tailapply ADE*. The return link in a stream producer is used to connect together successive elements in a stream. When a new activation of a stream producer is initiated, the return link which is passed to this new activation is the uid of the last stream element. Thus, when the new stream element is produced, the field previously containing the suspension in the last stream element would now reference this new element. The new element, in turn, would either be a record whose second field is a suspension to the STREAMTAIL instruction in the current activation or the value **null** denoting the empty stream.

Now, consider the behaviour of the computation tree if the STREAMTAIL instruction were to be treated as being identical to the TAILAPPLY operator. Under this assumption, our backup algorithm would preserve the argument record for each activation of the stream producer function initiated, in accordance with the description of tail recursion given above. Notice that because a stream activation only executes a RETURN when no more tail recursive calls are necessary, the only result value that would be preserved on the backup store would be the last stream element produced. Intermediate elements which are constructed using the SET and SETSUSP operators would not be maintained on backup store. Moreover, recording only the argument record of the tail recursive activation for a stream producer would not be sufficient to restore the rest of the stream because the return link for each activation is different. Recall from Chapter two that the return link passed to an activation of a stream producer is actually the second field of the record representing the last stream element created by this producer. Thus, the return link of each call to the stream producer would be different.

This analysis indicates that the current design of the backup and recovery algorithms suffer from two drawbacks with respect to the handling of streams. First, because tail recursive activations associated with a stream producer differ from each other in more than just their argument records, we need to maintain more information about the activation on backup store. The extra information which needs to be recorded must obviously include the new stream element produced. The second drawback is the inability of the backup algorithms to incrementally construct a data structure on the backup store. When a stream element is created,

it does not define the entire stream but represents only one element in the stream. Because streams are created in a demand driven manner, whenever a new stream element is created, there is also an activation associated with it whose state is relevant to the backup system. The suspension signals an instruction in this activation to initiate production of the next stream element. Of most importance to the backup system is the argument record held by the STREAMTAIL instruction which instantiates the next stream activation. The backup system must record this information if it is to properly restore the state of the system. If the argument record is not copied, then there would be no way for the recovery system to generate any further stream elements beyond that which has been copied onto the backup store. We discuss the ramifications of this requirement below.

The instruction responsible for setting the suspension in the stream is the SETSUSP instruction. The argument to SETSUSP is the record representing the new stream element. We see that one means of noting the production of new stream elements, therefore, is to alter the behaviour of the SETSUSP instruction. The SETSUSP instruction, in addition to setting a suspension in the new stream element, also initiates the transfer of this stream element onto the backup store. Of course, the value of the stream may be an early completion structure in which case it will be the responsibility of the SET instruction to perform the actual transfer.

The instruction responsible for initiating a new activation of the stream producer is the STREAMTAIL instruction. The main operand to this instruction of interest to us is the argument record that is used to initiate the new activation. Recording the argument record serves a different purpose from its use in normal tail recursive activations. For streams, recording the argument record of the STREAMTAIL operator is essential to restoring the state of the stream producer activation to allow further generation of stream elements after the the recovery procedures complete.

The advantage in altering the behaviour of the SETSUSP instruction to initiate the copying of the stream element instead of the STREAMTAIL instruction is that the transfer of the stream element can take place *before* a demand is made for the next element. If we choose to record the creation of stream elements by making the STREAMTAIL instruction copy its return link structure, we would need to wait for the next demand to be made (since that it is when the STREAMTAIL instruction fires) before the copy operation of the current stream element can be started.

Because stream elements are produced in a demand driven fashion, if the evaluation of a bind expression yields a stream, the value field in ⟨*name, value*⟩ pair bound in the backup environment will contain a single element initially, namely the first element in the stream. As more elements of the stream are produced, they are added onto the backup image and are considered as part of the stream image in the backup environment.

4.4.2.2 Implementation

To monitor a stream producer, we introduce a new type of activation descriptor called a **stream** *ADE*. A **stream** *ADE* is similar to a **tailapply** descriptor in that both maintain information about a function activation other than just its return value. The **stream** *ADE*, however, in addition to containing the argument record for the next stream activation to be initiated, also contains the stream element result of its associated stream producer activation. These stream elements are linked together on the backup heap. During the recovery process, the backup stream image is first restored. The recovery system then constructs a skeleton of the activation of the stream producer. This skeleton is used to initiate production of the next stream element when the next demand is made. The only instruction that can be enabled in this activation is the STREAMTAIL operator whose argument record is taken from the backup store and whose return link is the address of the last stream element. The suspension field in this element is set to the address of the STREAMTAIL instruction. The reason for storing the argument record is to set up the skeleton activation to support the demand driven execution mechanism for streams. When the recovery procedure completes, a subset of the stream image recorded on backup store will be restored. Let $\langle x_1, x_2, \ldots, x_n \rangle$ be the stream elements recorded on backup store. Then the recovery system restores the first j elements, $j \leq n$, where x_j is the greatest element for which the argument record needed to create the next stream element has been preserved. A skeleton activation is constructed to yield the $j+1^{st}$ element when the demand for it is made. During the construction of the stream on backup store, argument records may be removed from the stream image when it can be determined that they will not be needed during the recovery process.

When the suspension field in the last element on the backup image is accessed during recovery, the STREAMTAIL instruction in the skeleton activation would fire, initiating the next activation of the producer to produce the $j+1^{th}$ element. We illustrate this process in Fig. 18.

To summarize, unlike all the other structures we have examined, there are *two* operators

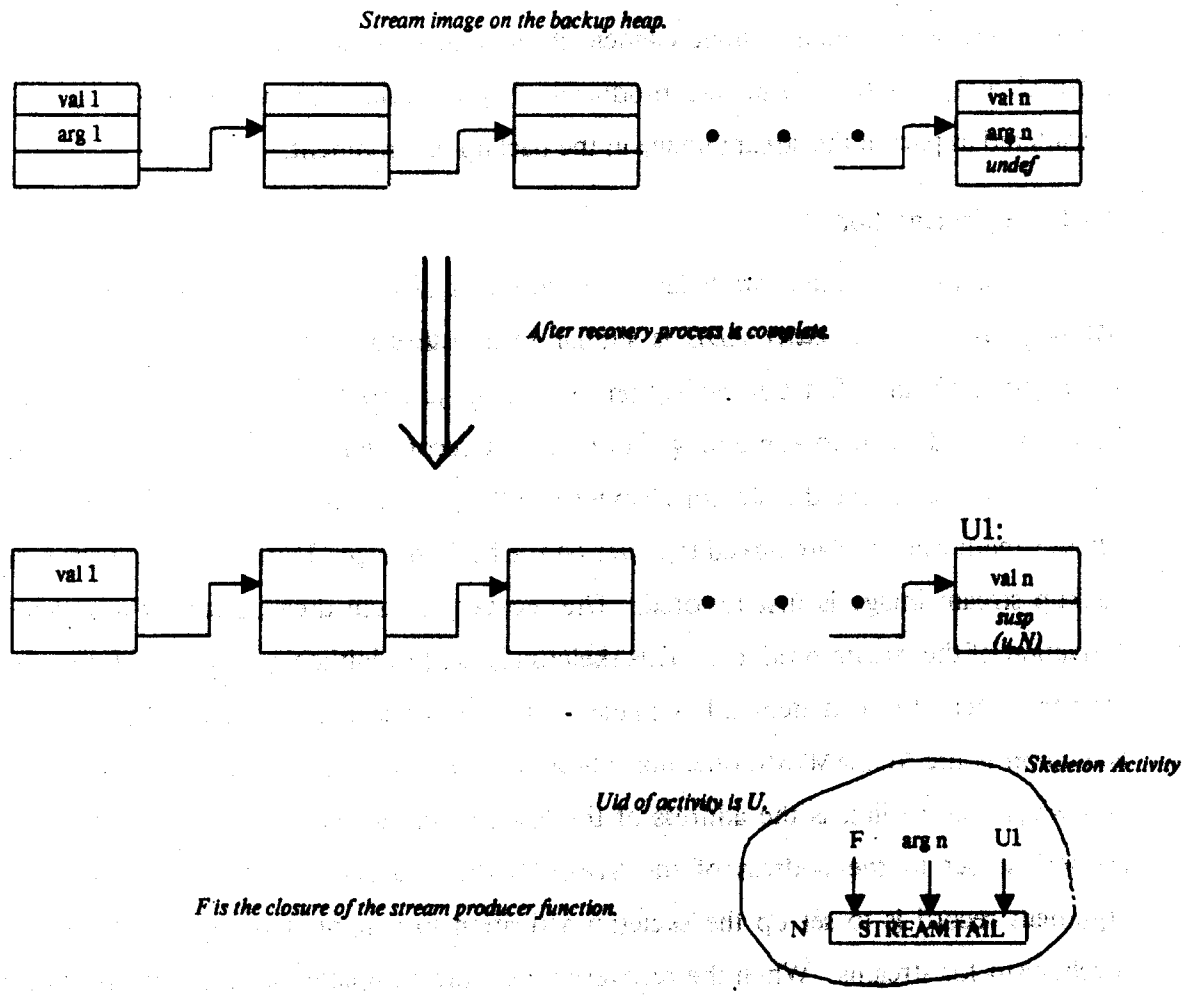


Figure 18: Reconstruction of a Stream Structure

which are responsible for maintaining a consistent image of a stream. The SETSUSP operator is responsible for initiating the transfer of the new stream element onto the backup store. The link field of the stream elements in the backup image is updated by the STREAMTAIL operator when it executes. The STREAMTAIL instruction is also responsible for copying the argument record to the backup store. The backup system should treat the argument record passed to the STREAMTAIL instruction in an activation and the stream element created within that activation collectively to ensure that a correct stream image is preserved. The implementation of this *⟨value, argument⟩* record is discussed in the next section.

In the next section, we formalize the backup algorithms outlined informally above. Our

formal model is an extension of the one given in Chapter two and mainly involves altering the definitions of those base language instructions responsible for the creation and updating of *ADE*'s on the backup store.

4.5 A Formal Model of Backup and Recovery

Beyond the need for modifiying the behaviour of some of the base language instructions, a formal model of VIM augmented with the backup and recovery algorithms must also have some concept of *failure*. A *failure* should be that special state which causes the system to invoke the recovery procedures. The modified operation of the base language instructions differ from their counterparts in Chapter two in that their execution results in a new backup state as well as in a new VIM state being constructed.

In the following presentation, we shall be using the same notation as in Chapter two.

4.5.1 The Backup State

Formally, the VIM system is now treated as a four-tuple:

VIM = ⟨*Shell*,*Interp*,**BackupState**,**VimState**⟩ where

VimState = ⟨Act × H × EIS × Env⟩ ∪ {*failed*}

BackupState = ⟨**Log** × **BHeap** × **BEnv**⟩

The Act,H,EIS,and Env components in the VimState are identical to their definitions in model M1. Notice that the *VimState* , in addition to having the same components as in M1 also contains the special value, *failed*. A failure in the system is modeled by having the *VimState* take on the value *failed*. All information found in the current *VimState* is "lost" if the *VimState* has the *failed* value. The domain **BackupState** is defined as a three-tuple where the first component in the tuple, **Log** represents the command Log of all volatile *Shell* commands[5], the second component, **BHeap**, denotes the set of structure values copied by the backup procedures, and the third component, **BEnv** contains all ⟨*name*,*value*⟩ bindings preserved by the backup utility. These values constitute the quiescent data in the system. The domain equation for **BEnv** is the same as that for the environment component in the VIM state, namely,

[5]Recall that *volatile* commands are those commands whose results have either not been produced or have not been copied onto the backup store.

$$\mathbf{BEnv} = \mathbf{Name} \rightarrow (U_H \cup \mathbf{Scalar})$$

The command log, as was described in Chapter three, is used to hold the record of all currently volatile *Shell* commands input to the system. By safely recording these commands, we are guaranteed of being able to restore the correct system state. The *Log* is a two-tuple, consisting of a function mapping from natural numbers to log entries and a size component indicating the size of the log. New log entries are appended to the end of the log.

$$\mathbf{Log} = \langle \mathbf{N} \rightarrow \mathbf{LogEntry} \rangle \times \mathbf{N}$$

$$\mathbf{LogEntry} = \langle \mathbf{Command} \times U_B \rangle$$

U_B = the set of uid's used for computation records.

As new *Shell* commands are input to the system, the text of the command is copied by the backup procedures onto the log. This text corresponds to the *Command* component of the *Logentry*. The second component in a log entry is a reference to the computation tree associated with this computation. This computation tree will reside on the backup heap.

Every element on the backup heap, *BHeap*, has an associated uid. There are three types of elements on the heap: structures which are normal VIM structures discussed previously, activation descriptor entries, and stream coordinator records that are used to package information about a stream activation. We discuss the role of the stream coordinator record in greater detail when presenting the operation of the suspension operator later in this chapter. Every *ADE* on the backup heap will either be referenced by some other *ADE* in the computation tree or will be referenced from a command log entry if it is the initial *ADE* in the computation tree.

$$\mathbf{BHeap} = (U_H \cup U_B) \rightarrow (\mathbf{ST} \cup \mathbf{Ade} \cup \mathbf{StreamRec})$$

$$\mathbf{StreamRec} = \langle \mathbf{Val} \times \mathbf{Arg} \times \mathbf{Link} \rangle$$

$$\mathbf{Ade} = \langle U_A \times \mathbf{AdeEntry} \times \mathbf{AdeType} \times \mathbf{Result} \rangle$$

$$\mathbf{Val, Arg} = U_H \cup \mathbf{Scalar} \cup \mathit{undef}$$

$$\mathbf{Link} = U_B \cup \mathit{undef}$$

$$\mathbf{AdeEntry} = (\mathbf{N} \rightarrow U_B \cup \mathit{empty}$$

AdeType = {*apply*,*tailapply*, *stream*, *value*}

Result = (U_H ∪ Scalar ∪ *undef*))

An activation descriptor entry is a structure of four components. The first component is the uid of the activation being represented. As we show below, this field is used to identify the activation descriptor so that the result of the activation can be properly forwarded to the *ADE* when it becomes known. The second component, the *AdeEntry*, is a function mapping from natural numbers to backup uids. If there are no entries in the *AdeEntry*, this component has value ***empty***. The domain of the *AdeEntry* function is the set of all instruction numbers in the corresponding activation which are either APPLY, TAILAPPLY, or STREAMTAIL operations. The range denotes the uid's of the *ADE*'s in the *BHeap* corresponding to these activations. Thus, if *j* was the instruction number in some activation α corresponding to an APPLY instruction, then *AdeEntry(j)* would be the uid of the *ADE* associated with the activation created by this APPLY instruction. Because the computation tree is pruned whenever a result of an activation is recorded, activation descriptor entries will have their *AdeEntry* field set to the value ***empty*** indicating that there are no subordinate *ADE*'s of this activation and that the result of this activation has already been recorded. The third component in the *ADE* contains the type of the descriptor. There are at least two types of *ADE*'s: ***apply*** *ADE*'s representing activations for which a result is not yet known; and ***value*** *ADE*'s which contain the result of the activation. In addition to these two types of descriptors, there are also special descriptors for tail recursive activations and stream producers which we described in the previous sections. The fourth field represents the *result* of the activation. It can either be a scalar or a uid which references the structure on the backup heap. All structures are associated with only one uid. Thus, the uid's used to reference structures on the backup heap are the same as those used to reference structures on the VIM heap. We shall use dot notation to refer to components of an activation descriptor.

When an *ADE* is initially constructed, there will be no value to place in its result field. Thus, this component is initially set to *undef*. When a result value is subsequently produced, it will replace the undefined element.

4.5.2 Early Completion

Early completion structures are represented in MR as follows:

$$ECQ = \mathcal{P}(ECE)$$

$$ECE = \langle U_H \times N \rangle \cup \{back\}$$

back is a special flag which indicates that this early completion structure is part of a structure which needs to be placed on the backup heap. This flag is placed in the queue by the *Copy* function. We present the definition of this function below. When the SET operator replaces an early completion structure containing this flag with a value, it will check if the structure containing this field can be copied by determining if there are any more early completion elements in the structure.

4.5.3 Auxiliary Functions

As was the case with the heap in M1, we have two auxiliary functions defined on the backup heap, *AddBHeap* and *RemoveBHeap*, which add and remove an element from the backup heap respectively. The definitions are omitted here but the reader may simply substitute *BHeap* for *H* in the definitions given for *AddHeap* and *RemoveHeap* in Chapter two.

There are also two functions defined on the command log, *AddLog* and *RemoveLog*. *AddLog* adds a new log entry to the end of the log and *RemoveLog* entry removes a defined entry on the log. In addition, we also define two functions over *AdeEntries* to replace and remove elements from a given *AdeEntry*.

AddLog: **Log** × **LogEntry** → **Log**

Function *AddLog*(*Log*,*Logentry*)

let $\langle LogVal, size \rangle = Log$

$LogVal'(m) = Logval(m)$ **if** $m \neq size+1$

$= Logentry$ **if** $m = size + 1$,

$size+1$

in

$\langle LogVal', size+1 \rangle$

endlet

endfun

RemoveLog: **Log** × N → **Log**

Function *RemoveLog* (*Log*, *n*)

```
  let LogVal, size = Log
      LogVal'(m) = LogVal(m) if m ≠ n
                 = undef if m = n
  in
    <LogVal', size>
  endlet
endfun
```

NewAdeEntry: **AdeEntry** × U_A × N → **AdeEntry**

Function *NewAdeEntry*(*AdeEntry*,*u*, *n*)

```
  let AdeEntry'(m) = AdeEntry(m) if m ≠ n
                   = u if m = n
  in
    AdeEntry'
  endlet
endfun
```

RAdeEntry: **AdeEntry** × N → **AdeEntry**

Function *RAdeEntry*(*AdeEntry*,*n*)

```
  let AdeEntry'(m) = AdeVal(m) if m ≠ n
                   = undef if m = n
  in
    AdeEntry'
  endlet
endfun
```

4.5.3.1 The Copy Operation

Before describing the Copy operation, let us first review the abstract representation of a structure on the VIM heap. The heap is modeled as a directed graph in which every node is labeled with a unique identifier. Nodes on the heap correspond to structures in our system. This representation allows for structures to be components of other structures e.g. an ARRAY of RECORDS. In our discussion, whenever we refer to a VIM structure, we also include this to mean all of the component structures which this structure references unless we explicitly state

otherwise. Thus, when a VIM structure is to be copied to the backup heap, it is also necessary that all component structures be transferred as well.

In order to preserve information found on the VIM state, it is necessary to have a function which can transfer data from the heap and environment components of the *VimState* to their respective counterparts in the *BackupState*. This *Copy* function is given below:

$Copy : (U_H \cup \textbf{Scalar}) \times \textbf{H} \times \textbf{BHeap} \rightarrow \textbf{H} \times \textbf{BHeap}$

Function *Copy* (*Val,H,BHeap*)

let *NewH, NewBH* =
if *Val* $\in$ **Scalars**
then *H,BHeap*
else let
RefStruct = $\{u \mid \exists\, m \in \mathrm{N}, R \in \textbf{Record}$ **s.t.**
$H(Val) \in \textbf{Rec} \wedge H(Val)(m) = u\}$
$u_1, u_2, \ldots, u_k$ = elements of *RefStruct*
ECStruct = $\{u \in U_H \mid H(u) \in \textbf{ECQ} \wedge Step(Val) = u\}$

NewH', NewBH' =
if *ECStruct* = {}
then if *RefStruct* $\neq$ {}
then *Copy*(u_1,
(*Copy* (u_2,...
(*Copy* (u_k), *H,BHeap*)...)
else *H,BHeap*
endif

else *AddBack*(*H, ECStruct*), *BHeap*
endif
in
NewH',
if *ECStruct* = {}
then *AddBHeap*(*NewBH', Val, H*(*Val*))
else *AddBHeap*(*BHeap.val*, ({**notcopied**}))
endlet
in
NewH.
NewBH
endlet
endfun

The *Copy* function takes as input the uid of the structure to be copied and first determines if there are any early completion elements in the structure. If there are, the structure is not copied; instead, those fields which are early completion queues are augmented with the special flag **back**. The function, *AddBack*, takes as arguments the current heap and the set of uid's which reference early completion elements in the structure. It returns a new heap in which the flag, **back**, has been added to each of these early completion structures. Determining if there are any early completion elements in the structure requires that all component structures be examined to see if they reference any such elements. The function *Step* takes as input the uid of the top level structure and returns the set of all uid's referenced from any substructure referenced from it that is associated with an early completion structure on the heap.

If there are no early completion elements in the structure, then it is transferred to the backup heap. Since a structure may reference many substructures, the *Copy* function copies all substructures referenced from the structure by recursively calling itself. A structure if fully copied only when it and all substructures it references have been placed on the backup heap. While the *Copy* function is easily expressed in our abstract model, it is significantly more complex in the actual implementation. We address the implementation problems in the next chapter.

4.5.4 The Shell

The command log contains the text of the shell comand input and the name to which the result of evaluating this command should be bound. This information is used by the recovery system which reinterprets the command text. The computation record associated with the log entry is used to avoid unneccesarily reexecuting operations whose results have already been recorded. Maintaining this information requires modifying the operation of the shell. The shell must now, in addition to the stream of shell commands and current *VimState*, also take as input the current *BackupState*. It returns a new *VimState* resulting from the evaluation of these commands and a new *BackupState* which contains a new log entry for every command input.

Shell: **Session** × **VimState** × **BackupState** → **VimState** × **BackupState**

Function *Shell* (*Session*, *VimState*, *BackupState*)

let ⟨*Act*, *H*, *EIS*, *Env*⟩ = *VimState*

```
    ⟨Log, BHeap, BEnv⟩ = BackupState
    NewState =
        if ChooseToExecute (VimState, Session)
            then Shell(Session, Execute (VimState, Choose(EIS)), BackupState)
            elseif empty(Session)
                then VimState, BackupState
                else let c1 = first(Session)
                     in if c1.C = DELETE
                            then ⟨Act, H, EIS, DelEnv(Env, Name)⟩, BackupState
                            elseif c1.C = BIND
                              then let % command is BIND

                                   FA = Translate(c1)
                                   uFA = new uid from UA
                                   Act' = AddAct(Act, uFA, FA)
                                   NewEIS = EIS ∪ {⟨uFA, 0⟩ | FA(i).opcnt = 0
                                         ∧ FA(i).sigcnt = 0}

                                   u1 = a new uid in UB
                                   NewAde = ⟨uFA, empty, {apply}, undef⟩
                                   BHeap' = AddBHeap(BHeap, u1, NewAde)
                                   Logentry = ⟨c1, u1⟩
                                   NewLog = AddLog(Log, Logentry)

                                   State', v = Interp(State, Choice(NewEIS))
                                  ⟨Act', H', EIS', Env⟩ = VimState'
                                   Env' = AddtoEnv (Env, c1.name, v)
                                  in
                                   ⟨Act', H', EIS', Env⟩,
                                   ⟨NewLog, BHeap', BEnv⟩
                                  endlet
                            endif
                     endlet
            endif
    in
      if empty (Session)
        then if EIS' ≠ {}
                then Shell (Session, Execute(Newstate, Choice(EIS')))
                else Newstate, NewBackupState
             endif
        else Shell (rest(Session), Newstate, NewBackupState)
      endif

    endlet
endfun
```

The new log entry constructed by the shell contains the text of the command input and a reference to the root of the computation tree to be associated with this computation. The uid field in the root activation descriptor entry contains the uid of the activation being instantiated. Its *AdeEntry* is undefined since no functions have been applied from this activtation yet. The log can be thought of as an array of log entries. The shell updates the log by adding the new log entry to some currently undefined index. Note that the interpreter is only invoked *after* the log entry has been recorded. The shell acknowledges a shell input by demanding the next command in the input stream only when the current shell command has been noted on the backup store. This guarantees that no processing will be done on a computation which cannot be recovered from failure since the text of the command is used by the recovery procedures to reexecute the command. The two functions which are responsible for updating the environment, *DelEnv* and *AddEnv* must also be modified. The *DelEnv* is responsible for removing an environment entry from the current envrionment. Since environment bindings are also part of the backup state, *DelEnv* must remove the entry from the *BEnv* as well.

4.5.4.1 Removing Log Entries

The *AddEnv* function is responsible for adding a new ⟨name,value⟩ binding to the current environment. The backup system maintains information in the form of a computation record about the computation that produced this value. This computation record need be kept on the backup store only so long as the binding was not placed in the image of the environment kept on the backup state. Once the binding is placed in the backup state environment and the result value is fully defined, the computation tree associated with the evaluation of this command can be removed from the backup heap. It may be the case, however, that the result value contains early completion structures, preventing it from being copied onto the backup heap. The value component in the binding in this case is notcopied. In such instances, it is the responsibility of the SET operator to remove the computation record when the value becomes fully defined. If there are no early completion structures in the result value, the *AddEnv* function places the binding on the backup environment and removes the log entry for this computation as well.

4.5.5 The Interpreter

Because base language instructions will be now be operating on the backup state as well as the VIM state, it is necessary to alter the behaviour of the interpreter. The interpreter is now a state transition function from a *VimState*, a *BackupState*, and an enabled instruction to a new *VimState* a new *BackupState* and a result value. Its definition is given below:

Interp: **VimState** × **BackupState** × **EI** → **VimState** × **BackupState** × (U_H ∪ **Scalar**)

Function *Interp* (*VimState*, *BackupState*, ⟨*u*,*i*⟩) %⟨*u*,*i*⟩ is an enabled instruction

```
let
    ⟨Act,H,EIS,Env⟩ = VimState
    FA = Act(u)
    ⟨Log,BHeap,BEnv⟩ = BackupState
    NewVimstate, NewBackupState = Execute(VimState,BackupState,⟨u,i⟩)
    NewVimstate' = Failure(NewVimstate)
    ⟨Act',H',EIS', Env'⟩ = NewVimstate'
in
    if failed(NewVimstate')
      then Recovery(BackupState)
      elseif FA(i).opcode = TERMINATE
        then NewVimstate',NewBackupState,FA(i).opnum1
        else Interp(NewVimState,NewBackupState,Choice(EIS'))
    endif
endlet
endfun
```

The *Failure* function models the introduction of a **failed** state in the system. It returns either the state **failed** or the state passed to it as input. The function *failed* returns true if the new state is a failed state and false otherwise. When *failed* returns true, the interpreter invokes the recovery procedures to restore the system to a correct state. The model of failure given here is, of course, simplistic insofar as it assumes that a failure does not occur during the middle of instruction execution. In the actual implementation of the system, care must be taken to guarantee that copy operations on the backup store are performed atomically. The model given here, however, is convenient for expressing the salient aspects of the backup algorithms, abstracting low level details such as preserving atomicity of copy operations. These issues are addressed in the next chapter.

4.5.6 Function Application

In this section, we present the modified operational semantics of two of the base language instructions responsible for function application in our system, the APPLY and the TAILAPPLY instructions.

4.5.6.1 The Apply Instruction

The effect of executing the APPLY operator is to augment the number of activations in the *VimState*. The addition of a new activation is reflected on the backup heap by adding a new activation descriptor entry for this new activation and creating a new *AdeEntry* function in the *ADE* of the currently executing activation. This new function maps from *offset* to *uid* where *offset* is the instruction number of the APPLY operator and *uid* is the unique identifier of the *ADE* representing the new activation on the backup heap. The new *ADE* will contain in its uid field, the uid of the new activation; its *AdeEntry* field is set to **empty**, and its type field is initialized to **apply** indicating that this activation was instantiated by an APPLY instruction. Since the result of this activation is not yet known, its *Result* is set to *undefined*.

---

if $I.opcode$ = APPLY **then**
let
$C = I.op1$,
$arg = I.op2$,

$\langle u_f, free \rangle = H(C)$,

u' = a new uid from U_A,
u'' = a new uid from U_A,
$Act' = AddAct(Act, u', H(u_f))$,
$H' = AddHeap(H, u'', MakeDest(H, u_{FA}, I.destlist))$,

$Act'', NewEis' =$
SendToDest
(*SendToDest*
(*SendToDest*
($Act', NewEis, u', \langle$**uncond**, 1, **op1**$\rangle, C$)
$u', \langle$**uncond**, 2, **op1**$\rangle, arg$)
$u'', \langle$**uncond**, 3, **op1**$\rangle, u'$),

u_{bh} = a newuid chosen from U_B,
$newade = \langle u', empty, \{apply\}, undef \rangle$,
$BHeap_1 = AddBHeap(BHeap, u_{bh}, newade)$,

$BHeap_2$ = **if** $\exists\ u_b$ *s.t.* $Bheap(u_b) = \langle u_{FA}, AdeEntry, Type, Result\rangle$,
then let $AdeEntry' = NewAdeEntry(BHeap(u_b), AdeEntry, u_{new}, i)$,
$UpdateAde = \langle u_{FA}, AdeEntry', Type, Result\rangle$,
in
$AddBHeap(BHeap_1, u_b, UpdateAde)$
endlet
else *BHeap*
endif
in
$\langle Act'', H', NewEls', Env\rangle$,
$\langle Log, BHeap_2, BEnv\rangle$
endlet
endif

It is possible that there is no computation record on the backup heap that contains the activation descriptor for the activation in which the APPLY instruction is found. This may happen if the result of the computation has already been bound in the backup environment before the APPLY executes. In this case, there is no change to the backup state. If there is an activation descriptor corresponding to the current activation, its *AdeEntry* is updated to reference this new *ADE*.

4.5.6.2 The TailApply Instruction

The TAILAPPLY operator is also modified to monitor the progress of tail recursive functions. If a TAILAPPLY operation executes as part of a tail recursive activation, we copy the argument record passed as the second input to the instruction to the backup heap, replacing the old argument record found in the activation descriptor entry. As we mentioned earlier, doing this substantially reduces the time needed to reexecute a tail recursive function. Copying the argument record becomes a non-trivial issue, however, because of the presence of early completion structures. If the argument record to be copied contains early completion elements, the copy operation can only take place after all such elements have been set. The old argument record on backup heap and the *AdeEntry* component cannot be replaced until the new record is fully copied. To ensure that this restriction is observed, it is necessary to create a new *ADE* corresponding to this new activation. We now need to link the old *ADE* and the new *ADE* together. Since the idea of noting the argument record on the backup heap is to avoid

reexecution of prior tail recursive calls, we interpose the new *ADE between* the activation descriptor corresponding to the current activation and its parent *ADE*. The parent *ADE* represents the activation in which the initial call to the tail recursive function was made. In this sense, the computation tree built from tail recursive activations is constructed "bottom-up", with old tail recursive *ADE*'s being pushed down the tree and new *ADE*'s being fitted in between its caller and the original caller of the function. We illustrate this process in Fig. 19. When an argument record is copied, we set the *AdeEntry* field in the descriptor to empty, effectively discarding prior *ADE*'s associated with this function. Thus, once an argument record is fully copied, all previous *ADE*'s of this tail recursive function are removed from the backup state since the link connecting these activation descriptors with the rest of the computation tree is severed. The formal definition of the modified TAILAPPLY instruction is given below:

```
if I.opcode = TAILAPPLY then
  let
    C = I.op1,
    arg = I.op2
    dest = I.op3

    <u_f, free> = H(C)

    u' = a new uid from U_A,
    Act' = AddAct(Act,u', H(u_f))
    Act'', NewEis' =
      SendToDest
        (SendToDest
          (SendToDest
            (Act', NewEis, u',<uncond,1,op1>, C)
            u',<uncond,2,op1>, arg)
            u',<uncond,3,op1>, dest)

    Act''', NewEis'' = SendSignal(Act'',NewEis',u_FA,destinations)

    u_new = new uid from U_B

    BHeap_1 = if ∃ u_b,u_ade,k s.t. BHeap(u_b)= <u_α,AdeEntry,Type,Result>
                ∧ AdeEntry(k) = u_ade ∧ BHeap(u_ade).u = u_FA
              then AddBHeap(BHeap,u_b,<u_α,
                     NewAdeEntry(AdeEntry,u_new,k),Type,Result>)
              else BHeap
              endif
```

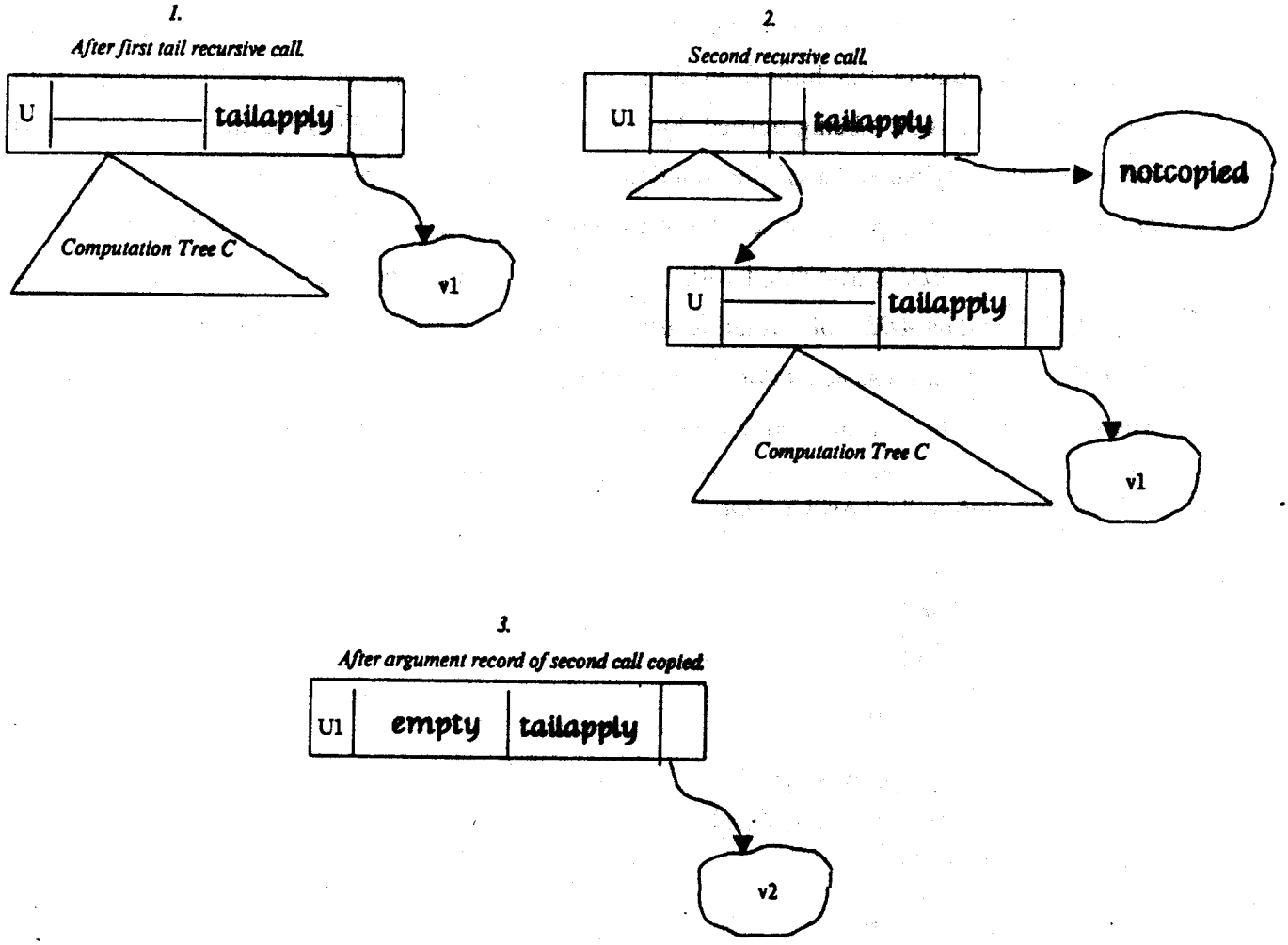


Figure 19: Recording Tail Recursive Functions

$$H', BHeap_2 = \textbf{if } BHeap_1 = BHeap \textbf{ then } H, BHeap \textbf{ else } Copy(arg, H, BHeap_1) \textbf{ endif}$$

$$BHeap_3 = \textbf{if } BHeap_2(arg) \neq \textbf{notcopied} \textbf{ then } AddBHeap(BHeap_2, u_{new}, \langle u', \textbf{empty}, \textbf{tailapply}, arg\rangle) \textbf{ elseif } \exists\, u_{ade} \text{ s.t. } BHeap(u_{ade}).u = u_{FA} \textbf{ then let } AdeEntry' = NewAdeEntry(\{\}, u_{ade}, i) \textbf{ in } AddBHeap(BHeap_2, u_{new}, \langle u', AdeEntry', \textbf{tailapply}, arg\rangle) \textbf{ endlet } \textbf{else } BHeap_2$$

```
            endif
        in
          ⟨Act'',
          H',
          NewEis''
          Env⟩,
          ⟨Log,BHeap3,BEnv⟩
        endlet
      endif
```

The backup heap is modified in three steps by the TAILAPPLY operator. In the first step, the *ADE* of the activation which initially instantiated this tail recursive function is modified to have its *AdeEntry* field reference the activation descriptor constructed for this new application. This new descriptor has its *AdeEntry* field initialized to reference the *ADE* of its caller. If no such *ADE* exists, then no change is made. In the next step, the argument record is placed on the backup heap. If the copy operation was successful, then, in the third step, the *AdeEntry* field of the newly created descriptor is set to **empty** since the recovery procedure can use the argument record directly during the restoration of the system state. If the structure was not copied, then a reference in the *AdeEntry* component of the new activation is set to its caller *Ade*. This allows us to still reference previous activations of the tail recursive function until the argument record is finally copied.

4.5.7 The Return Operator

One of the important features of our backup algorithm is that results of activations are recorded on the computation tree. This is the primary means of reducing reexecution time of volatile shell commands. The RETURN operator transmits the result of an activation to the backup heap. The return value, if it is a VIM structure will, in most cases, contain early completion elements. The SET operator is responsible, in these circumstances, for ensuring that the structure does finally get copied. If the return value is copied, then all *ADE*'s referenced from this descriptor are removed from the backup state. This has the effect of pruning the computation tree. Because *ADE*'s can be removed in this manner, it may be the case that when the RETURN instruction executes, there maybe no *ADE* associated with its activation since a return instruction in a parent activation may have placed its result on the backup heap. In this case, no action on the backup state is required by the instruction. The modified definition of the instruction is given below:

```
if opcode = RETURN then
  let
    DL = H(I.op1)  % the list of return addresses
    u_r = DL(1)  % uid of the calling activation

    targets = GetDest(H,DL(2))
    Val = I.op2,        % the value to be returned

    Act', NewEis' = SendValue(Act,NewEis,u_r, targets, val)
    Act'', NewEis'', = SendSignal(Act',NewEis',u_FA,destinations)

    Ade = <u_FA,AdeEntry,Type,Result>
    BHeap_1 = if ∃ u_b s.t. BHeap(u_b) = Ade
                then AddBHeap(BHeap,u_b,<u_FA,AdeEntry,value,Val>)
                else BHeap
              endif

    H',BHeap_2 = if Val ∈ U_H
                   then Copy(Val,H',BHeap')
                   else H,BHeap_1
                 endif

    H'',BHeap_3 = if ∃ u_b s.t. BHeap_2(u_b) = Ade
                    then if BHeap_2(Val) = notcopied
                           then H',BHeap_2
                           else let
                                  Ref = {n| AdeEntry(n) is defined}
                                  n_1,n_2,....,n_k = k elements of Ref
                                  AdeEntry' = RAdeEntry(...
                                    (RAdeEntry(AdeEntry,n_1),....n_k)))
                                  NewAde = <u_FA,AdeEntry',value,Val>
                                  in
                                    H',AddBHeap(BHeap_2,u_b,NewAde)
                                  endlet
                         endif
                  else H,BHeap
                endif
  in
   <Act'',
   H'',
   NewEis''
   Env>,
   <Log.BHeap_3,BEnv>
  endlet
endif
```

The backup heap is updated in three steps by the RETURN instruction. The first step determines if an activation descriptor for the current activation exists *i.e.* if this activation has an *ADE* which is part of some computation tree. If so, the type of this activation descriptor is changed to *value* indicating that a result has been created and the return value is written into the result field of that *ADE*. If the return value is a structure, then we initiate a *Copy* operation to place this structure on the backup heap. If the structure is fully copied because it contains no early completion elements or if the result was a scalar value, then all *ADE*'s referenced from this descriptor are removed from the backup heap, effectively pruning the computation tree as discussed earlier. If, on the other hand, the result was a structure contains early completion structures, it is the responsibility of the SET instruction to perform the pruning of the computation tree.

4.5.8 Stream Operations

There are two operations which manipulate stream activation descriptors on backup heap: the SETSUSP instruction and the STREAMTAIL instruction. These operators are responsible for recording the creation of new stream elements and noting argument records of activations to allow the recovery procedures to reconstruct the stream image and to permit demand driven evaluation of those elements not recorded.

4.5.8.1 The Suspension Operator

As we had described earlier, the SETSUSP operator needs to be altered to record the production of stream elements on the backup heap. Central to our algorithm is the use of a *stream coordinator record* which contains the element produced by a stream producer activation and the argument record to the STREAMTAIL instruction used to initiate the next activation of the producer. When a new stream activation is instantiated, a new *ADE* is created for it. Unlike the case with the APPLY operator, however, *ADE*'s created by the STREAMTAIL instruction are accessed through the stream coordinator record. The link field in the coordinator record is used to link together all stream activation descriptors in a chain, mirroring the construction of the stream on the *VimState*. The SETSUSP instruction is responsible for initially creating the coordinator record and for linking the elements in the backup stream image. When the STREAMTAIL instruction executes, it initiates the copy operation for the argument record of the

new activation being instantiated. In addition, it sets the link field in the coordinator record of its corresponding *ADE* to that of the new stream producer activation.

The SETSUSP operator takes as input the record representing the new stream element. It updates the *ADE* associated with this activation to type **stream** and constructs a new stream coordinator which is referenced from the *Result* of this descriptor. It then initiates the copy operation for this stream element. Each new invocation of the stream producer causes a new *ADE* to be constructed, with the link field in the coordinator of the previous activation set to this new *ADE*. In Fig. 20, we illustrate the effect of the SETSUSP instruction on the backup heap.

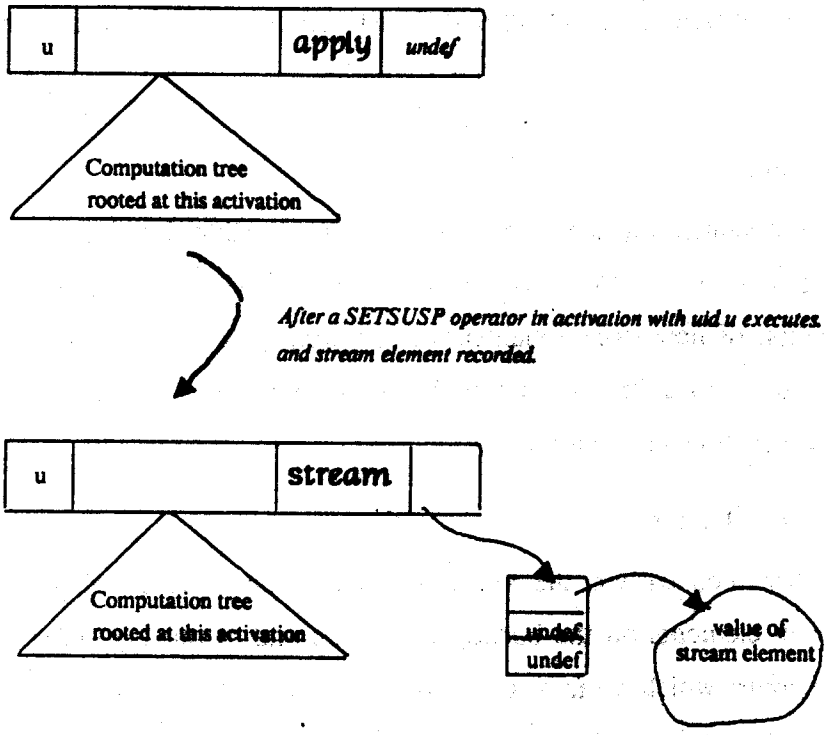


Figure 20: The Effect of the SETSUSP Instruction on the Backup Heap

The formal definition of the SETSUSP operator is given below:

```
if I.opcode = SETSUSP then
  let
    u = I.op1
    R = H(u)
    f = I.op2
    i = I.op3
```

```
Act', NewEis' = SendSignal(Act,NewEis,u_FA,destinations)

∀ V ∈ N
  R'(v) = R(v) if v ≠ f
        = MakeSusp(<u_FA, i>) otherwise
strecord = <R(1), undef,undef> %argument record and link not known yet.
H',BHeap_1 = if R(d) ∈ U_H
               then Copy(u,H,BHeap)
               else H, BHeap
             endif

u_new = a new uid from U_B
BHeap_2 = AddBHeap(BHeap_1,u_new,strecord)

H'',BHeap_3 = if ∃ u_b s.t. BHeap_2(u_b) = <u_FA,AdeEntry,Type,Result>
                then H',AddBHeap(BHeap_2,u_b,<u_FA,AdeEntry,stream,u_new>)
                else H,BHeap
              endif
in
 if R(f) ∈ ECQ ∧ |R(f)| = 0
  then <Act',
        NewHeap(H'',u,R'),
        NewEis',
        Env>,
        <Log, BHeap_3, BEnv>

  else let <Act'',NewEis''> = SendToDest(Act',NewEis',<uncond,i,signal>,signal)
       in
        <Act,
         H'',
         NewEis'',
         Env>,
         <Log, BHeap_3, BEnv>
       endlet
 endif
endlet
endif
```

There are three major tasks that the SETSUSP operator is responsible for in the update of the backup heap. The first is the construction of the stream coordinator record. It creates a record, whose first component is the value of the newly created stream element, and the second and third components are given value, *undef*. This second component is to hold the value of the argument record but may only be defined when the STREAMTAIL instruction in this activation

executes. The third component holds the link to the *ADE* of the next activation of the stream producer and can also only be set when the STREAMTAIL operator executes. The second task is the copying of the value component of the stream record. Recall that our definition of a non-empty stream element is a record of two fields, the first being the element itself and the second holding the suspension to the rest of the stream. The backup facility need only copy the first element of the record since the link field to the next stream element can be set by the recovery procedure when the stream is reconstructed. The third major task that the operator performs is the updating of the activation descriptor entry associated with the stream producer. It changes the type field of the *ADE* to **stream** and updates the *Result* to reference the new stream coordinator record. We next examine how the STREAMTAIL instruction needs to be modified to maintain a consistent and up-to-date image of stream production.

4.5.8.2 The StreamTail Operator

The STREAMTAIL instruction operates in conjunction with the SETSUSP operator in maintaining the stream coordinator records and activation descriptors. When this instruction executes, it initiates the transfer of the argument record passed as its second argument onto the backup heap. The stream coordinator corresponding to this activation is updated to reference this argument record. If both the value field and the argument record are fully copied, then all subordinate *ADE*'s can be removed from the backup heap. In this sense, the stream value and argument record passed to the STREAMTAIL instruction constitute the result value of this activation and just as the computation tree is pruned by the RETURN instruction when the return value is placed on the backup heap, we perform the same action here when both the stream value and argument record are safely recorded. Like the TAILAPPLY instruction, STREAMTAIL is also responsible for creating a new activation descriptor. The type of this *ADE* is set to **stream** and the link field in the current stream activation record is set to the uid of this descriptor. We give the formal definition of the STREAMTAIL instruction below:

---

if *I*.**opcode** = STREAMTAIL **then**
  let
    $C = I.\mathbf{op1}$,
    $arg = I.\mathbf{op2}$
    $dest = I.\mathbf{op3}$ % *dest* is a field in a stream record

    $\langle u_f, free \rangle = H(C)$

u' = a new uid from U_A,
$Act' = AddAct(Act,u', H(u_f)$
$Act'', NewEis' =$
SendToDest
(SendToDest
(SendToDest
($Act', NewEis, u', \langle uncond,1,op1\rangle, C$)
$u', \langle uncond,2,op1\rangle, arg$)
$u', \langle uncond,3,op1\rangle, dest$)

$Act''', NewEis'' = SendSignal(Act'', NewEis', u_{FA}, destinations)$

u_{newade} = new uid from U_B
$newade = \langle u', empty, stream, undef\rangle$
$BHeap_1 = AddBHeap(BHeap, u_{newade}, newade)$

$H', BHeap_2 = Copy(arg, H, BHeap_1)$
$H'', BHeap_3 =$ if $\exists\, u_b$ s.t. $BHeap_2(u_b) = \langle u_{FA}, AdeEntry, stream, u_s\rangle$
and if $BHeap_2(u_s) = \langle Val, Arg, Link\rangle$
then $H', AddBHeap(BHeap_2, u_s, \langle Val, arg, u_{newade}\rangle)$
else $H, BHeap$
endif

$H''', BHeap_4 =$ if $\exists\, u_a$ s.t. $BHeap_3(u_a) = \langle Val, Arg, u_{newade}\rangle$
then if ($Val \in Scalar \vee BHeap(Val) \neq notcopied$)
$\wedge\, BHeap(Arg) \neq notcopied$
$\wedge\, \exists\, u_b$ s.t. $BHeap(u_b) = \langle u_{FA}, AdeEntry, stream, u_s\rangle$
then let
$Ref = \{n \mid AdeEntry(n)$ is defined$\}$
$n_1, n_2, \ldots n_k$ = the k elements of Ref
$AdeEntry' = RAdeEntry(\ldots$
$(RAdeEntry(AdeEntry, n_1), \ldots n_k)))$
in
$H'', AddBHeap(BHeap_3, u_b, \langle u_{FA}, AdeEntry', stream, u_s\rangle)$
endlet
else $H'', BHeap_2$
else $H, BHeap$
endif
in
$\langle Act''',$
$H''',$
$NewEis''$
$Env\rangle,$
$\langle Log, BHeap_3, BEnv\rangle$
endlet
endif

There are four steps that the STREAMTAIL instruction follows in updating the backup heap. The first step is the construction of the activation descriptor for the new stream activation to be instantiated. In the next step, the instruction calls the *Copy* function to transfer the argument record to the backup heap. In the third step, the stream coordinator associated with this activation is updated to have its argument record reference this structure. Finally, the instruction checks to see if both the stream element and the argument record have been copied, and if so, removes the subordinate *ADE*'s from the backup heap. We have avoided describing the removal of argument records of old stream coordinatior records to simplify the presentation.

4.5.9 The Set Operator

The operation of the SET instruction is also modified to record information on the backup heap. A structure containing an early completion element may be part of an argument record to a TAILAPPLY instruction or may be part of a result record of an activation. In section 4.3, we argued that such a structure should augment the backup *State* only when it and all of its substructures are fully defined *i.e.* when they contain no early completion structures. If the early completion queue in the *ec* structure being set contains the special value, **back**, it means that this field is part of some structure which needs to be copied onto the backup heap. In fact, because elements on the heap can be shared, the structure containing this field may be a component of many structures, some of whom need to be copied to backup heap. The SET instruction examines each of these structures to see if there are any early completion structures in them which still need to get set. Those structures which are fully defined are copied.

The SET instruction also examines those activation descriptors or stream coordinators that have a reference to any of those structures that become fully defined because of its execution. If, for example, there is a **value** or **tailapply** *Ade* whose result field references a structure that becomes fully defined because of the SET instruction, then, as was done by the RETURN and TAILAPPLY instructions, all references to activation descriptors from the *AdeEntry* component of the associated *ADE* can be removed. Similarly, if the reference to a fully defined structure emanates from a stream coordinator, then the SET operator must check to see if the *AdeEntry* list of the associated stream activation descriptor can be reset using the criteria we discussed above. The instruction is also responsible for removing a log entry if the element being set is part of a

structure to be bound in an environment. We give the formal definition of the instruction below:

```
if I.opcode = SET then
  let
    u = I.op1
    R = H(u)
    f = I.op2
    x = I.op3
    u' = H(R(f))

    ∀ v ∈ N
      R'(v) = R(v) if v ≠ f
            = x otherwise

    Act', NewEis' = SendSignal(Act,NewEis,u_FA,destinations)
    H' = NewHeap(H, u, R')

    Parent = {u_p | u' ∈ Step(u)}
    u_1,u_2,...,u_k = k components of Parent

    H'',BHeap_1 = if back ∈ Q
                  then Copy(u_1,
                            (Copy(u_2,...,
                                  (Copy(u_k, H',BHeap)...)
                  else H',BHeap
                  endif

    BHeap_2 = if ∃ u_v s.t. BHeap_1(u_v) = <u_FA,AdeEntry,Type,u_v>
              then let Ref = {n | AdeEntry(n) is defined}
                   n_1,n_2,...,n_k = k components of Ref
                   AdeEntry' = R(AdeEntry:=(BAdeEntry(AdeEntry,n_1),...n_k)))
                   in if Type = value ∨ Type = tailapply
                      then if BHeap(u_v) ≠ notcopied
                           then H'
                                AddBHeap(BHeap_1,u_v,<u_FA,AdeEntry',Type,u_v>)
                           else BHeap_1
                           endif
                      elseif Type = stream
                      then let <val,arg,link> = BHeap(u_v)
                           in (if val ≠ notcopied ∨ BHeap(val) ≠ notcopied)
                                 ∧ BHeap(arg) ≠ notcopied
                               then H''.
```

```
                                    AddBHeap(BHeap1,ub,<uFA,AdeEntry',Type,uv>)
                                else H'',
                                    BHeap1
                            endlet
                        else H,BHeap
                    endif
                endlet
            else BHeap
        endif
    NewLog = CheckLog(Log,Parent)

    in
      <Act',
      H',
      NewEis' ∪ H(u')
      Env>,
      <NewLog, BHeap2, BEnv>
    endlet
endif
```

Because objects on the heap can be shared, the record whose field is being set may be a substructure of several objects. The set *Parent* denotes the uid's of these structures. Recall that the auxiliary function *Step* when given the uid of an early completion structure returns the set of all uids associated with structures which have a path on the heap to this ec-structure. If the flag, **back**, is found in the early completion queue, the instruction calls the *Copy* function to attempt to copy all those structures whose uid's are in *Parent*. This function will only copy those structures which do not contain any other early completion elements. The backup heap returned from the calls to *Copy* will contain all those structures referenced from *Parent* which were able to be copied because they no longer contained any ec-structures.

If the activation descriptor referencing a copied structure was of type **value** or **tailapply**, then the subtree rooted at the *ADE* is removed. If, on the other hand, this structure was referenced from a stream coordinator record, the instruction removes the subtree of the corresponding stream activation descriptor if *both* the value and argument field in the coordinator have been completely defined.

We also introduce one other auxiliary function, *CheckLog*. Some of the structures in the *Parent* set maybe values which are to be bound in the user environment. Among these

structures, there maybe some which are now fully defined as a result of executing the SET operation and are placed on the the environment image on the backup state. These structures are the result values of active computations. These computations are represented on the backup heap as computation records which are referenced from the entries in the command log. The function *CheckLog* removes these log entries and returns the new log.

4.6 Summary

In this chapter, we presented the backup and recovery algorithms for the VIM system. Our attention focused on the representation and manipulation of computation records which contain information about the progress of volatile commands in the system. A computation record is represented as a directed tree where nodes in the tree denote activations and edges signify caller/callee relationships. To update a computation record requires altering the semantics of the APPLY and RETURN instructions. The APPLY operator adds a new node to the tree and the RETURN operator replaces a node with the result value of its corresponding activation.

A node in the computation tree is referred to as an *activation descriptor entry* and embodies information about an activation in some active computation. There are two basic types of *ADE*'s: **apply** and **value**, the former used to denote an activation whose result value is not yet known and the latter designating an activation whose value has been recorded on the backup heap.

The main disadvantage with this simple scheme is that it is not sufficiently expressive to handle early completion or stream structures and is inefficient when tail recursive functions are involved. Early completion is handled by requiring that all *ec*-elements in a structure be SET before the structure is copied. Ensuring that such a structure eventually does get copied necessitated the modification of the SET instruction to update the backup heap when the structure becomes fully defined. Tail recursive functions are represented on the backup heap using a special **tailapply** *ADE* which holds the argument record of the tail recursive activation. We based the design of the **tailapply** *ADE* on the observation that the recovery procedures could avoid executing intermediate activations of a tail recursive function if the argument records to the activations of the functions were recorded. During recovery, the function could be instantiated directly with the argument recorded found on the backup heap. Finally, we introduced a special descriptor to handle stream structures. A **stream** *ADE* contains a reference

to a *stream coordinator record* which embodies information about a stream activation. In addition to storing the stream element produced in that activation, we also keep the argument record needed to produce the next stream element. The construction and maintenance of stream coordinator records required modifying the operation of the SETSUSP and STREAMTAIL instructions.

The algorithms presented in this chapter were developed for a very abstract machine in which such issues as structure organization, guaranteeing atomicity of information transfer from memory to backup heap, and memory management of the backup heap were not addressed. In the next chapter, we discuss these issues.

Chapter Five

Implementation Issues

In this chapter, we are concerned with concrete problems that arise when we attempt to implement the backup algorithms presented in Chapter Four for the VIM system. These algorithms were described in the context of an abstract model MR which was used as a vehicle to rigorously describe their behaviour. Issues which were conveniently hidden in our model will now have to be more thoroughly addressed. In particular, we will focus our attention on how the *Copy* function may be implemented. The complexity of the operation arises because of the representation of structures in our system. Guaranteeing that the copy operation will always be *atomic* is a non-trivial task given the storage representation for data structures that VIM uses. Section 5.3 concentrates on this issue. Section 5.4 describes a simple storage management policy for stable storage.

5.1 The Copy Operation

The copy operation, as described in Chapter Four, is responsible for copying a VIM structure object to the backup heap. An efficient implementation of this operation is necessary for any practical realization of our backup algorithms. In our abstract model, the copy operation was treated as a function which transmitted the success of the copy back to its caller. The callers of this function were base language instructions that required result values or argument records to be placed on the backup heap. There is no concurrency between the execution of the caller *i.e.* instructions and the *Copy* function in this model. An instruction that calls the *Copy* function must wait for the copy to be complete before it can continue execution. In an actual implementation, having the interpreter wait for the transfer of data from memory to stable storage before beginning execution of the next instruction would lead to intolerably slow performance. Examination of the formal description of these instructions, however, reveals no reason why the transfer of data to stable storage cannot proceed in parallel with the execution of other instructions. Our approach to implementing the backup algorithms is to consolidate the "logic" for modifying the backup state into a backup system kernel that is responsible for updating the computation records on backup store and initiating the copy operation to perform data transfer to stable storage. Base language instructions invoke the kernel, passing as arguments, the type of operation to be performed *i.e. set, result, tailapply* etc. and the necessary

operands *i.e.* data structure to be copied. The instructions can then proceed with normal execution without having to wait for any results from the kernel. The responsibility of updating the backup state would then be primarily in the purview of the backup system kernel. Overlapping execution of base language instructions with transfer of data to and from backup store makes the backup algorithms presented a much more attractive solution. If sufficient concurrency can be exploited in the base language programs, then we conjecture that updating stable storage should not cause major degradation in system performance.

An important requirement of the *Copy* function is that it be *atomic*. In database literature, atomicity is a property of a transaction whose overall effect is all-or-nothing: either all the changes made to the data by the transaction happen, or none of the changes happen. Thus, all transactions appear externally as indivisible operations. This requirement is essential to support recoverability of data after hardware failures occur. Atomicity in VIM is a property of an activation that refers to the point at which the effects of the activation are perceived by other executing activities in the system. It is necessary that the *Copy* function be atomic because the effect of its execution *i.e.* the augmenting of backup state information, should be made visible to the recovery procedures only after the entire structure has been completely copied to backup store. If a failure were to take place during the middle of a copy operation, and the function was *not* atomic, the recovery system would see data in the backup state that does not correspond with any data that was present in the Vim state at the time the failure took place. Guaranteeing atomicity is a non-trivial issue for two reasons. First, data structures in VIM may be arbitrarily complex; if the structure to be copied represented a node α on the VIM heap, then the entire graph rooted at α must also be copied onto the backup heap. The other complexity involved with the copy operation is due to the way data structures are represented in VIM. We discuss the storage organization of the VIM heap in the next section and then present our solution to guaranteeing atomicity for this operation.

5.2 Storage Organization in VIM

VIM structures have been thus far treated as monolithic entities that are created and manipulated as a single unit. This representation was useful in the presentation of our backup and recovery algorithms but is far removed from the actual representation of structures in our system. The representation chosen for structures in VIM is influenced by the organization of physical memory and the desire to have only information needed by the computation be resident

in main memory. VIM is based on a hierachically organized physical memory consisting of main memory and disk in which information is brought into main memory only upon demand. To facilitate the transfer of information between memory and disk, the virtual address space is partitioned into a number of fixed size pages. The organization of the address space in VIM differs from conventional demand paged systems, however, in the page size chosen. To avoid the overhead of unnecessarily paging in unwanted information, the unit of storage allocation in VIM is a small page of 24-32 words, known as a *chunk*. Having a small page size is the primary means of exploiting parallelism in base language programs. In our data flow execution model, there is no dependency between any two enabled instructions and thus, I/O service required by one instruction does not prohibit the execution of the other in the interim period. By having a high level of concurrency of data transfer between the disk and main memory, the processing unit is seldom expected to be idle waiting for a pending I/O request to be serviced during program execution. It is expected that, in general, there will be enough enabled instructions during program execution to make disk access completely transparent.

Because of the small page size used, each chunk holds at most a single object. Complex structures such as arrays and records are held in a number of chunks. The representation chosen for these structures should be sensitive to the applicative nature of the programming model by allowing information to be shared between structures whenever possible. One implementation well suited to our goal of efficient sharing is to represent VIM structures as k-ary trees of chunks with the leaves of the tree containing the elements of the structure and the internal nodes of the tree containing pointers to other chunks [17]. Because structures can be complex, leaf chunks may hold uid's to other structures in addition to containing scalar values. The choice of a tree organization to represent chunks allows a high degree of sharing to be achieved. For example, to construct a new version of a structure differing from its predecessor in a single element, the system need only construct a new path from the root of the new structure to the leaf chunk which is to hold the new element; all other elements which are still common to both structures are still shared between them by having both structures use the same paths to the common leaf chunks. The REPLACE instruction briefly described in Chapter Two, for example, which returns a new version of a record differing from the old version in a single element operates in this manner.

The address of an element in a structure is specified by a two-tuple, $\langle uid, accesspath \rangle$. *uid* denotes the uid of the structure in the system. The access path is the base k representation of the

offset of the desired element in the structure. The *length* of the access path for a given structure is the height of its VIM tree representation.

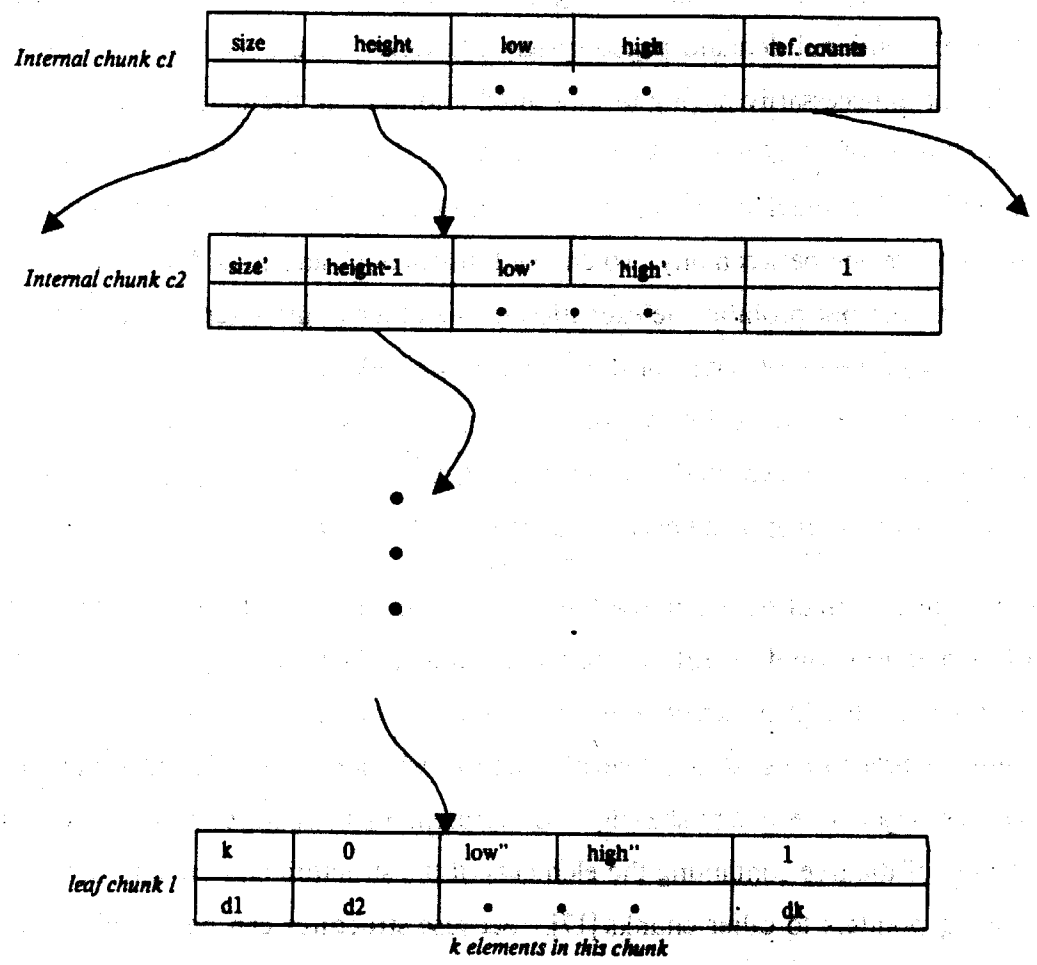


Figure 21: The Representation of a VIM structure

Abstractly, we view a chunk as a three tuple:

$$\textbf{Chunk} = \textbf{Cid} \times \textbf{Header} \times \textbf{Data}$$

$$\textbf{Header} = \mathbf{N} \rightarrow \mathbf{N}$$

$$\textbf{Data} = \textbf{Internal} \cup \textbf{Leaf}$$
$$\textbf{Internal} = \textbf{Cid}^m$$
$$\textbf{Leaf} = (\mathbf{U} \cup \textbf{Scalar})^m$$

Cid = the domain of chunk identifiers

The *Cid* of a chunk is the chunk's unique identifier and maybe thought of as the virtual address of the chunk in the system. It should be noted that the domain of chunk id's is not related to the domain of structure uid's which are used to uniquely identify objects on the VIM heap. The *Header* field in a chunk contains administrative information about the chunk. In particular, the number of references to this chunk in the system, the height of this chunk in the VIM tree, and the high and low indicies of the elements of the subtree rooted at this chunk are all information kept in the header field. If a chunk has no outstanding references to it, it is placed on a *freelist* and its space may be reused when needed. The last component is the *data* portion of the chunk. For an internal chunk, this consists of the *Cid's* of its immediate descendents in the tree. For a leaf chunk, this consists of either *uid's* if the elements of this structure are themselves structures or *scalar* values. The size of the data portion is some fixed *m*. The size of the chunk is *m* plus the size of the header portion.

For a complete description of the storage organization of VIM structure, the reader should see [17]. In the next section, we examine the problem of making the copy operation atomic. Structures which are to be copied from the VIM heap onto the backup heap must be copied atomically *i.e.*, a structure consisting of many chunks should be deemed as being copied only when all of the chunks which comprise it have been transferred onto backup store and not before. In our discussion, we shall refer to a chunk found in main store as a VIM chunk and a chunk found on backup store as a backup chunk. We shall use the word *structure* to refer to any VIM structure *e.g.* array or record.

5.3 Performing the Copy Operation

When a base language instruction which needs to augment the backup state executes, it invokes the backup system kernel which maintains all information about the current status of structures which are in the process of being copied. The kernel is responsible for updating the computation tree and command log on backup store based on the algorithms given in Chapter Four. When a structure is to be copied, the kernel calls the *Copy* function. This function takes as input the uid of the structure to be copied and produces as its output an acknowledgement, **copied** if the structure was fully copied onto backup store or **notcopied** if it could not be. The latter acknowledgement occurs when a structure (or any of its substructures) contains early completion elements. The backup system kernel uses this acknowledgement to determine when activation descriptors should be given a new type and when command log entries can be removed.

A major complexity in implementing the copy function for the VIM system is due to the representation of VIM structures as trees of chunks. For the sake of uniformity and simplicity, structures found on the backup heap will also have the same representation as their counterparts on the VIM heap *i.e.* trees of chunks. At the start of system operation, all chunks on the backup store will be on a freelist. Whenever a VIM chunk needs to be copied onto backup store, a backup chunk is removed from the backup freelist and the contents of the VIM chunk are copied onto it. This approach will remove the need for sophisticated interpretation of the data found on the backup store by the recovery system. Activation descriptor entries comprising a computation record are treated as records which occupy a single chunk.

To see how the Copy function can be implemented, let us first consider a simple scenario. Suppose that the function is to transfer a structure whose leaves contain n scalar values onto the backup heap, where $n > m$. This structure will be represented on the heap as a tree of chunks. Let the leaf chunks of the structure be labeled $l1$, $l2$,...,lj. Clearly, these leaf chunks can be copied onto the backup heap without any preprocessing by the copy routine since the leaves contain only data. Let $b1,b2,...,bj$ be chunks on the freelist on the backup store. Then, the data found on $l1$ can be copied onto $b1$, that of $l2$ can be copied to $b2$ etc. After the leaf chunks have been thus copied, the routine proceeds with copying the internal chunks as well. Copying an internal chunk is a little more complex because these chunks contain references to other VIM chunk id's. The copy routine translates these references to their appropriate backup store equivalents. Since the copy operation is being performed in a bottom-up fashion though, all chunks referenced by an internal chunk would already have been given chunk id's on the backup store. If an internal chunk C has references to chunks $c1,c2,...,cm$, then the corresponding backup chunk has references to $b1,b2,...,bm$ where bi is the backup chunk id corresponding to li. The copy routine updates the activation descriptor to reference this structure only when all chunks have been written to backup store. If a failure takes place before the ADE can be updated, it will appear to the recovery system that no information about this activation was recorded by the backup system. Note that all chunks at a given height in the tree can be copied in parallel.

The copy routine performs a bottom-up breadth-first traversal of the structure. When a chunk is copied onto backup store, its backup chunk id is recorded in the header portion of the chunk. If the copy routine is invoked later on, to copy a chunk which has already been copied

on the backup store, it can avoid recopying the chunk by first checking to see if that chunk already has a backup chunk id. By simply recording these id's, the same level of sharing found on the VIM heap is achieved on the backup heap as well.

In this simple scenario, guaranteeing atomicity is a fairly easy task because the structure is not recorded as being copied until the copy funtion transfers all Vim chunks to the backup heap. The situation becomes slightly more complex when we consider the actions which need to be undertaken for copying more complicated structures. In particular, the solution given above is not satisfactory for handling structures whose elements are themselves structures *e.g.* an ARRAY of RECORDS. Let us first consider the situation without early completion. Suppose that a leaf chunk, *li*, which is to be copied contains references to other structures on the heap. It is necessary that each of these subordinate structures must themselves be fully copied before this leaf chunk is allowed to be written onto backup store. Our description of the copy routine in Chapter Four had it recursively call itself to copy these substructures. When all structures referenced from this chunk have been copied, then the leaf chunk can itself be transferred onto backup store. Because the routine only returns to its caller when all substructures have been copied, the copy action is atomic since the backup state will be updated to acknowledge the existence of this structure *only* when the top-level copy routine completes and returns its result to the kernel. Because we are not considering early completion elements, the *Copy* function is guaranteed to return an acknowledgment, **copied**. No partially copied structure can ever be referenced by any activation descriptor since such references are *only* set by the kernel after the copy operation is complete.

5.3.1 Early Completion

The presence of early completion elements in the leaf chunks of structures further complicates the copy procedure. In Chapter Four, we noted that it is necessary for the backup system to be able to determine when there are no further early completion elements still to be SET before the backup state can be updated. In our implementation, we can determine this information through the use of reference counts. Every chunk has two reference counts associated with it: the **refcnt** is the total number of references to this chunk found in the system, and the **setcnt** is the number of SET instructions still pending which are to set early completion elements found in this chunk. The meaning of the **refcnt** is the same for an internal chunk as it is for a leaf chunk. The **setcnt** field in the header of an internal chunk is the number of early

completion elements which need to be set in the tree rooted at this chunk. Thus, the **refcnt** field in the root chunk of a structure holds the total number of references to this structure and the **setcnt** field indicates the total number of early completion elements found within the structure[6]. When an early completion structure gets SET, the **setcnt** field in the headers of all chunks on the path to this element get decremented.

Let us first consider a simple structure *i.e.* one which does not reference any other structure on the heap. To implement the copy operation in the presence of early completion structures, we do the following. When a leaf chunk with **refcnt** greater than zero is encountered by the copy procedure, we do not copy it onto the backup store. Instead, we allocate a backup chunk id for this chunk and continue examining the other chunks in this structure. To inform the SET operator which will eventually replace the early completion element found in this chunk that it should copy the chunk onto backup store, the copy routine also labels the chunk with the tag ***back***. At the time the early completion element becomes defined, the SET operator will note the fact that this chunk belongs to a structure that is to be copied to backup store. If the **setcnt** of this leaf chunk is zero, the operator invokes the backup system kernel to copy this chunk onto the backup chunk allocated for it. In addition, if there are no outstanding SET instructions that are to be executed for this structure, determined by examining the **setcnt** of the root chunk for this structure, the kernel also updates the activation descriptor to reference this structure according to the algorithms given in the last chapter. Notice that computation records are *only* updated when all elements in a structure are fully defined and copied onto the backup heap. Thus, it will never be the case that activation descriptors are aware of a structure for which not all elements are known. Atomicity is still guaranteed even with early completion structures because a structure becomes a part of a computation record only when no further ec-elements are referenced from it.

We next examine how the backup system should handle early completion structures belonging to structures that are components of a larger structure which is to be copied. When a leaf chunk which contains uid's to other structures is encountered by the copy function, the **setcnt** of the root chunk of this subordinate structure must be examined. If it is greater than zero, it means there are early completion elements which still need to be set in this structure. As

---

[6] This does not include the **refcnt** or **setcnt** of its substructures.

before, the copy routine is recursively invoked to copy the chunks found in this substructure. It is necessary that the backup state does not get updated with the parent structure unless all subordinate structures of this parent become fully defined. Our implementation follows closely with the algorithms given in the last chapter. Every root chunk of a structure contains a field in its header, *Parent*, containing the uid of all structures which reference this structure that are to be copied. This field is mangaged by the copy function as it traverses the heap. Associated with each uid in the list, we also store the chunk id of the leaf chunk in the parent structure from which the reference emanates. When all early completion elements are set in a structure that is to be copied onto backup store, all structures referenced in its *Parent* list are examined. The backup state is updated with those structures referenced in this list which have become fully copied. Determining whether a structure has been fully copied involves keeping track of the number of chunks that still need to be transferred and the status of its substructures. Counters are used to record this information. Every root chunk, in addition to the *Parent* list, also contains a *ToBeCopied* counter indicating the number of chunks still to be copied onto backup store. When this value becomes zero, the backup state can be updated. Every leaf chunk also contains a counter indicating the number of substructures which have not yet been fully copied. When this counter reaches zero, the leaf chunk can be copied onto backup store and the *ToBeCopied* counter can be decremented. Much of the detail involved in implementing the incrementing and decrementing of these counters is not very interesting and is omitted here. The important point to note with respect to early completion is that the backup system kernel is responsible, not only for updating the backup state with the structure containing the early completion element being set, but also for updating the backup state with all of its parent structures that need to be copied as well.

5.4 Storage Management

One other implementation detail that deserves brief mention is the manner in which storage management is handled on stable store. The organization we have chosen for stable store lends itself to a very simple and efficient storage management strategy. Recall that stable storage is used to hold transitional data represented in the form of computation records. Each computation record embodies the progress of some active computation in the system. When the result of a computation is recorded on stable store, the associated computation recorded can be deleted. Deleted computation records are added onto a freelist of backup chunks. We expect

that the set of quiescient data found on stable store will be periodically written onto tape. After the transfer, the space occupied by these data items on stable store can be reclaimed. It is not necessary, however, to wait for the transfer of information to tape before space can be reused on stable store. Once the result of a computation is bound in an environment, the storage occupied by the activation descriptor entries of the corresponding computation record can be reused. We use a variant of a mark and sweep garbage collection policy to reclaim structures referenced from activation descriptor entries in a computation record that can be reclaimed. Once the result of a computation is known and is copied onto the backup store, all chunks in that structure are marked. We can then reclaim all chunks belonging to structures referenced from *ADE*'s in that computation that are not marked.

The storage management policy is very simple for our system because we can determine precisely when data is no longer accessible by observing the dynamics of the command log — removal of an item in the log implies that the associated computation record can be reclaimed. Waiting for a computation to complete before reclaiming the storage it occupies on stable storage obviates the need for introducing a complicated garbage collection policy on stable store.

It may be the case that the value of a computation is recorded on the backup environment even before all the values of the *ADE*'s in the subtrees of the computation are. Since the result of the computation has been recorded in the environment the associated computation record can be placed on the backup store freelist. Removing the computation record, however, necessitates taking some action to inform the activations whose results would normally be recorded in these *ADE*'s that the computation record is no longer part of the backup heap. To do this, we maintain a uid entry table which contains referencing information for all activations in the system. Entries in this table include the uid of the activation, the address of the activation in memory and the address of its *ADE* on backup store and a reference to a boolean flag which indicates whether the computation record has been reclaimed or not. When the computation record is removed from the backup state, its associated flag is set to true and the result value is not copied. Only the root activation of a computation can change this flag. Clearly, this operation is outside of the purely functional programming paradigm thus far used; such resource managers are to be written using the *guardian* construct [10] which allows this sort of non-applicative behaviour to expressed in an applicative setting.

Chapter Six

Conclusion

This thesis has proposed a design for the VIM computer system which guarantees the security of all online information against loss or corruption as a result of hardware failure. We developed backup procedures that are intended to execute concurrently with normal computation. These procedures record the progress of all computation in a compact form on a backup storage medium. When a computation completes, its result is bound to some name in the VIM users' environment structure. Once this binding is added to the backup environment image, the computation record associated with this computation is removed from the backup state. We make no assumption as to the integrity of any data which survives a failure except for data found on the backup medium. The backup medium consists of two main devices: tape storage used to hold all data bound to identifiers in the user environment, and stable storage which contains information in the form of computation records about all active computations in the system. When the recovery procedure is invoked after a failure, it first restores the VIM environment structure using the backup environment image. It then begins reexecution of those commands which were executing at the time the failure occured. The time to reexecute these commands is greatly reduced because of the information kept on the corresponding computation records. Once all commands have been reexecuted, the recovery process is complete and the system can accept further commands from its users.

6.1 Contributions of the Thesis

The contributions of this thesis have been two-fold. First, we developed backup and recovery algorithms for the VIM system. These algorithms are very different from those found in more conventional systems. The unique features of VIM that required a novel approach to providing data security lie in its applicative programming model and in the use of a uniform representation for both data and programs. This homogeneity eliminated the need to maintain distinction between files and data, thus allowing the backup system to be more closely integrated with the VIM interpreter than would otherwise be possible. The use of an applicative base language made it possible to have a simple organization of backup store because data in such a model never changes its value. We exploited the expressive power of the base language instructions by distributing the logic of our algorithms among the salient base language

instructions. The APPLY operator, responsible for creating a new function activation, was augmented to append to the appropriate computation record a new activation descriptor corresponding to the activation to be instantiated. Similar enhancements were made to the RETURN operator and various structure operators as well. By embedding these algorithms within the interpreter itself, it was possible to achieve a measure of data security far greater than what is possible in conventional systems. In addition, such a design makes the operation of the backup facility completely transparent to users of the system. Similarly, once the recovery procedures restore the system state after a failure, subsequent computations will not be able to determine that a failure occurred by examining the restored state. The fact that the base language is applicative also freed us from having to introduce complicated backup storage policies. Data once written onto to backup store is never updated; it is either garbage collected or bound to a symbolic name in the backup environment.

In order to rigorously specify these algorithms, we developed a formal operational model of system behaviour for VIM. This model views VIM as a state transition system with the VIM interpreter being a state transition function. We presented the definitions of some of the more interesting base language instructions in this model using a variant of the functional language VIMVAL. This basic model was later enhanced to incorporate the backup system as well. The backup state was treated as a separate component of VIM. The interpreter was now treated as a state transition system on a two-tuple consisting of a VIM state and a backup state. Beyond being an important tool for expressing our algorithms, this model also allowed us to give a formal proof of correctness of our algorithms (presented in the Appendix).

6.2 Future Research

One important area of investigation that was not addressed in this thesis is the issue of correctly preserving the state of *non-determinate* computations. A non-determinate computation is one which may exhibit different behaviours for the same inputs. This type of computation contrasts with determinate computations for which repeatable behaviour is guaranteed. A major assumption made in this thesis was that all computation was determinate. This allowed us to design a recovery system which can reexecute computations whose results are not recorded on the backup store by preserving the arguments to the computation. Such a design is possible because any computation in this model is guaranteed to produce the same result when presented with the same inputs. The basic non-determinate operator in the VIM base language is the

MERGE instruction. The merge operator is enabled whenever an input arrives on either of its two input arcs. Thus, the behaviour of this operator is characterized by the arrival order of its inputs. Such behaviour is inherently non-determinate.

An important area of future research is augmenting the design proposed in this thesis to handle non-determinate computation. The changes made must take into consideration the fact that non-determinate computations cannot simply be reexecuted by the recovery system since there is a multiplicity of output behaviours possible. Reexecuting such a computation with the same inputs is, therefore, not guaranteed to produce the identical outputs.

Most transaction systems such as airline reservations, banking, etc. are based on non-determinate computation. Such systems also typically have very high data security requirements. Enhancing the VIM backup and recovery system to support non-determinate computation would be an important step in understanding how highly secure transaction systems can be written in an applicative computer system. Issues of atomicity and indivisibility of transactions could then be addressed in this context.

It would also be a challenging task to map the abstract specification of the algorithms given in Chapter 4 into an efficient implementation on the VIM system. Realization of these algorithms will require optimizations not addressed in the thesis. Such optimizations include minimizing the amount of copying done to stable storage and efficiently reclaiming storage on the backup medium. These problems which were hidden in our abstract model were briefly addressed in Chapter 5. A truly viable implementation will need to confront these issues in much greater detail.

Appendix A

Proof of Correctness

In the previous chapters, we have developed a formal model of the VIM system to describe our backup and recovery algorithms. Beyond being a convenient vehicle in which to precisely state our algorithms, the formal model can also be used in proving the correctness of our design. Intuitively, demonstrating the correctness of the backup and recovery procedures involves showing that the recovery procedures do not restore an incorrect state based on the information kept on backup store by the backup procedures. The computations which execute after the recovery procedures complete should not be able to discern the fact that a failure had occurred before. Thus, the state of the system observed by any computation instantiated after recovery from failure must be equivalent to the observable state which existed prior to the failure. In VIM, computations observe the effects of other computations through the VIM environment structure. It is, therefore, necessary that the environment image restored by the recovery system be equivalent to the environment which existed before the failure. We present a formal definition of environment equivalence later in this appendix. To establish that environment equivalence is preserved, we examine the state transitions produced by the system when interpreting the command log during recovery and compare it with the state transitions that result when interpreting the same log when no backup state information is used. To show that environment equivalence is preserved across the two transition sequences, we use the fact that no instruction is executed during the recovery process that would not also have been executed by the VIM interpreter evaluating the same command when no information on the backup state is utilized. Our proof is as follows: We first examine the state created by executing a particular instruction in the transition sequence of the recovery process. We then demonstrate that this environment is equivalent to the environment component of the state created by executing the equivalent instruction under normal interpretation. Using a simple induction argument, we then show that the environment component of the final recovery state is equivalent to the environment component that would have resulted if no failure had taken place.

In the following sections, we formally define our notions of transition sequence and equivalence. We then prove our main theorem: The system state after the recovery process completes is equivalent to the state that would have resulted had no failure taken place. This

implies that it is not possible for any computation to observe the effects of a failure once the recovery procedures complete.

A.1 Definitions and Terminology

State Transition Sequences

Because we are considering VIM to be a determinate system, we can model the execution of a program as a sequences of states. The *Execute* function which when given a current state and enabled instruction returns the state created by executing the instruction. The operation of some of the base language instructions are different depending upon whether the system is executing using the normal or recovery interpreter. In our proof, we shall be concerned with examining state transition sequences constructed by the respective interpreters on the command stream preserved on the backup log.

Definition 1: The *System State* is a two-tuple, $\langle VimState, BackupState\rangle$ where *VimState* and *BackupState* were defined in Chapter Four. The *recovery command stream* for a system state S is a sequence of commands: $\langle c_1,c_2,...,c_k\rangle$ where $BackupState = \langle Log,BHeap,BEnv\rangle$ and $Log(i).command = c_i$.

Definition 2: A *state transition relation*, $\vdash$, is a relation on states. Let $S = \langle Act,H,EIS,Env\rangle$ and $T = \langle Act',H',EIS',Env'\rangle$. Then, $S \vdash T$ if $\exists\ (u,i) \in EIS$ *s.t.* $Execute(S,(u,i)) = T$.

Definition 3: A *state transition sequence*, $\langle S_1,S_2,...,S_k\rangle$, is a sequence of states such that $S_i \vdash S_{i+1}$ for $1 \le i \le k$-1.

For notational convenience, we shall sometimes denote a state transition sequence $\langle S_1,S_2,...,S_j\rangle$ as $S_1 \vdash^{*} S_j$.

The recovery process is divided into two phases. In the first phase, the contents of the backup environment are restored onto the VIM environment structure. Once this is done, the command log is reexecuted; during this reexecution phase, backup information is used to avoid unnecessary recomputation. The state of the system after the first phase completes is called the *initial recovery state*. The heap in the initial recovery state contains all structures referenced from environment entries in the backup environment image.

Definition 4: Let S_k be a VIM state such that $Failure(S_k) = true$. Let B_k, the backup state corresponding to S_k, be $\langle Log,BHeap,BEnv\rangle$.

Then, the *initial recovery state* for S_k is the state: $\langle\{\}, H_0, \{\}, Env_0\rangle$ where

$H_0(u)$ = if $\exists$ *name* s.t. $BEnv(name) = u$ and $BHeap(u) = v$ then v.
= if $\exists$ R, u_1, m s.t. $(H_0(u_1) = R) \wedge$
$(R(m) = u) \wedge (BHeap(u) = v)$ then v.
= *undef* otherwise.

$Env_0 = BEnv$

Unlike instructions executed during normal operation, instructions executed during the recovery process use information found in the backup state. The APPLY instruction, for example, avoids instantiation of new activations if the value for an activation which previously executed has been recorded on backup store. The command log found on the backup state is a sequence of commands whose final results have not yet been not recorded in the backup environment structure. When a failure occurs, the recovery system reexecutes the commands found on the log, using backup state information it has accumulated about the computation. To demonstrate that the recovery process interprets this information correctly, we examine how the VIM Interpreter would execute these same commands if *no* backup state information is used. The transition sequences produced by the respective interpreters is known as a *transition sequence pair.* If the recovery system interprets the information found on the backup state correctly, it follows that the final states in the two transition sequences must have equivalent environment components.

Definition 5: Let R be the recovery command stream for state S and let $STS = \langle S_1, S_2, \ldots, S_f\rangle$ be the state transition sequence produced during the evaluation of the call: $Shell(R, S_1, \langle\{\},\{\},BEnv\rangle)$. Then, S_1 is the initial recovery state and S_f is the *ideal recovery state* for R. EIS_f is φ, the empty set. STS is called the *standard transition sequence* for system state S.

Let R be the recovery command stream for system state S and let $RTS = \langle R_1, R_2, \ldots, R_f\rangle$ be the state transition sequence produced during the evaluation of the call: $Recovery(\langle Log, BHeap, BEnv\rangle)$. Then, R_1 is the initial recovery state and R_f is the *final recovery state* for R. EIS_f is φ, the empty set. RTS is called the *recovery transition sequence* for system state S.

Definition 6: A *sequence pair* for a system state, S, is a two-tuple: $\langle RTS, STS\rangle$ where $RTS = \langle R_1, R_2, \ldots, R_j\rangle$ and $STS = \langle S_1, S_2, \ldots, S_k\rangle$, $j \leq k$ where R_j and S_k are the final recovery state and ideal recovery state for the command stream, resp. $R_1 = S_1$, the initial recovery state for system state S.

State Equivalence

In the following definition, we shall use the symbol, ≡, to denote the equivalence relation between corresponding state components in different states. Thus, if Env_i and Env_j are two environments in states S and S′, then $Env_i \equiv Env_j$ if these two environments are equivalent.

Definition 7: Instruction Equivalence: Let FA and FA' be two activities. Then, $FA(m) \equiv FA'(n)$ for $m,n \in$ N if:

- $FA(m)$.opcode = $FA'(n)$.opcode.
- ∀ *opnum* ∈ {opnum1,opnum2,opnum3}, $FA(m)$.*opnum* = $FA'(n)$.*opnum* if $FA(m)$.*opnum* and $FA(n)$.*opnum* ∈ Scalar.
- ∀ *opnum* ∈ {opnum1,opnum2,opnum3}, $FA(m)$.*opnum* ≡ $FA'(n)$.*opnum* if $FA(m)$.*opnum* and $FA'(n)$.*opnum* ∈ U.
- $FA(m)$.opcnt = $FA'(n)$.opcnt.
- $FA(m)$.sigcnt = $FA'(n)$.sigcnt.
- ∀ (*dc*,*k*,*opnum*) ∈ FA(i).dest, ∃ (*dc*,*l*,*opnum*) in $FA'(j)$.dest *s.t.* $FA(k) \equiv FA'(l)$.

Activity Equivalence: Two Activities FA and FA' are equivalent if for every $m \in N$, $FA(m) \equiv FA'(m)$.

Object Equivalence: Let H_i and H_j be two heaps defined such that $H_i(u) = v$ and $H_j(u') = v'$ for $u,u' \in$ U. Then, $v \equiv v'$ if

- v = *undef*, v' = *undef*.
- $v \in$ Scalar, then $v' \in$ Scalar and $v = v'$.
- $v \in$ Record, then $v' \in$ Record and $v(i) \equiv v'(i)$, for every $i \in$ N.
- $v \in$ ECQ, then $v' \in$ ECQ and ∀ $(u_k,i) \in v$, ∃ $(u_j,m) \in v'$ *s.t.* $(u_k,i) \equiv (u_j,m)$.
- $v \in$ SUSP, $v' \in$ SUSP *s.t.* if $v = (u_m,m)$ and $v' = (u_n,n)$, $Act(u_m)(m) = Act(u_n)(n)$ OR $H_i(v) \in$ Record and $v' \in$ SUSP and ∃ state S_k where $S_j \vdash S_k$ and $H_k(v') \equiv H_i(v)$.
- $v \in$ U, then $v' \in$ U and $H_i(v) \equiv H_j(v')$.

Environment Equivalence: Let Env_i and Env_j be two environments in states S_i and S_j. Then, Env_i and Env_j are equivalent if for every n_1 *s.t.* $Env_i(n_1) = v$ ∃ n_2 *s.t.* $Env_j(n_2) = v'$ where if

- v = *undef*, v' = *undef*.
- $v \in$ Scalar, then $v' \in$ Scalar and $v = v'$.
- $v \in$ U, then $v' \in$ U and $H_i(v) \equiv H_j(v')$. H_i and H_j are the heap components in states S_i and S_j resp.

Containment and Completeness

In order to show that environment equivalence is preserved between states in the recovery transition sequence and the standard transition sequence, we will need to examine the effect of

executing equivalent instructions in the two states. Thus, it is necessary that for every enabled instruction in the recovery state, there be an equivalent instruction in the corresponding VIM state. This property is called *containment*.

Definition 8: Let $\langle RTS,STS\rangle$ be a sequence pair for a system state, S, where R_i is an element of RTS and S_j is an element of STS. R_i is *contained* in S_j if $\forall (u,m) \in EIS_i \exists (u',n) \in EIS_j$ s.t. $FA(m) \equiv FA'(n)$ where $Act_i(u) = FA$ and $Act_j(u') = FA'$.

Definition 9: If $\langle RTS,STS\rangle$ is a sequence pair for a system state, S, and R_i is an element of RTS and S_j is an element of STS, then R_i is *complete* wrt S_j if $Env_i \equiv Env_j$ and R_i is contained in S_j.

Computation Sets

In Chapter 4, we introduced the computation tree as an abstraction to describe the instantiation of function activations in a computation. The following three definitions formalize this idea.

Definition 10: Let an activation a be instantiated in state S. Then, the *computation set* C of a is defined as follows:

- $a \in C$.
- Any activation instantiated from an activation in the computation set C is also in C.

That is, the computation set of an activation (u,a) represents the transitive closure over all activations in the computation tree rooted at (u,a).

Definition 11: A *computation sequence* CS for an activation instantiated in state S_1 with computation set C is a state transition sequence, $\langle S_1,S_2,S_3,\ldots,S_k\rangle$ where k is the smallest integer such that none of $EIS_{k+1},\ldots,EIS_j$ contain any instructions from activities found in C.

Definition 12: A value, $v \in$ Scalar $\cup$ ST, is *accessible* in a state $\langle Act,H,EIS,Env\rangle$ if either:

- There is some activity A s.t. $A(n) = I$ and $I.opnum = v$ for $opnum \in$ {op1.op2.op3}, and $Act(u) = A$.
- There is some activity A s.t. $A(n) = I$ and $I.opnum = u$ for $opnum \in$ {op1.op2.op3}, $Act(u') = A$ and $H(u) = v$.
- There is some $v' \in$ Record s.t. $v'(m) = v$ or $v'(m) = u$ where $H(u) = v$ and v' is accessible.

A.2 Proof of Correctness

Lemma 1: Let $R_i = \langle Act_i, H_i, EIS_i, Env_i \rangle$ be an element of *RTS* and $S_j = \langle Act_j, H_j, EIS_j, Env_j \rangle$ be an element of *STS*. Suppose $R_i \vdash R_{i+1}$ on an enabled instruction, (u,i), where $Act_i(u) = FA$ and $FA(i).opcode \neq$ APPLY. Then, if R_i is complete wrt S_j, $\exists$ a state S_k *s.t.* $S_j \vdash^* S_k$ and R_{i+1} is complete wrt S_k.

Proof: To prove the lemma we need to show that environment equivalence and containment holds between R_{i+1} and some state S_k for any instruction whose **opcode** is not APPLY. We prove the lemma by examining the different classes of instructions defined in our model and show that the lemma holds over each of these classes. If R_i is complete wrt S_j, then, by the property of containment, there exists an enabled instruction, $(u',j) \in EIS_j \equiv (u,i)$.

1. **Scalar Operations:** The effect of executing a scalar operation does not alter the environment and so, both environments in R_{i+1} and S_{j+1} remain equivalent. By definition of instruction equivalence, we know that every destination $(dc,i,opnum)$ in $FA(i).dest \equiv$ some destination, $(dc,j,opnum)$ in $FA'(j).dest$. Since the same scalar values are sent to equivalent instructions, these destinations remain equivalent. Thus, equivalent instructions become enabled in R_{i+1} and S_{j+1}. Hence, R_{i+1} is contained in S_{j+1} and, so, the lemma holds for scalar operations.
2. **Structure Operations:** A structure operation performed in state R_i might cause the new state R_{i+1} to have a different heap. Note that the environment component does not change when any structure operation executes. To see that the lemma still holds for these types of instructions, we examine the various structure operations in our system.

   a. CREATE: A create instruction executed in R_i will produce a new structure on the heap in R_{i+1} with uid u_1 and size n, with all elements of the structure having value *undef*, where n is the operand to the instruction. The equivalent instruction in EIS_j, (u',j) when executed in S_j will produce a new structure on the heap in S_{j+1} with uid u_2 and size n. By our definition of heap equivalence, H_{i+1} and H_{j+1} are still equivalent. Showing that R_{i+1} is still contained in S_{j+1} follows the same argument as given above.

   b. SELECT: A select operation can be performed on a structure element that is either a scalar or structure value, an early completion structure or a suspension. We examine each of these in turn:

      i. *value:* There are two cases to be considered if the item selected from H_i is a scalar or a structure. In the first case, the item selected from H_i would also be a scalar or structure value. By the property of containment, the operands to the SELECT instruction must be equivalent. Since the heaps and environment components do not change in this case, $H_{i+1} \equiv H_{j+1}$. R_{i+1} is still contained in S_{j+1} because all destinations of the SELECT in R_i are equivalent to destinations in S_j and, following the argument given in (1), these instructions would remain equivalent. Thus, R_{i+1} is complete wrt S_{j+1} in this case.

In the second case, the item selected in H_i is a record representing a stream, but the corresponding item in H_j is a suspension. This case arises when the APPLY operator is to instantiate a function represented by a **stream** *Ade*. The stream image on stable store is restored by the recovery procedure. In state S_j, the suspension triggers the instantiation of a new activation, A, to produce the new stream element. Let $\langle S_{j+1}, S_{j+2}, \ldots, S_n \rangle$ be a state transition sequence where EIS_n contains all enabled instructions in EIS_{j+1} except those belonging to A (or any of A's descendents) in the computation tree rooted at A. Then, H_n will contain the new stream element and, by the property of object equivalence, the two streams in H_{i+1} and H_n would be equivalent. R_{i+1} is obviously still contained in S_n. Thus, R_{i+1} is still complete wrt S_n.

ii. *early completion:* If the item selected from H_i is an early completion structure, then the select instruction, (u,l) is added onto the queue. Since operand values are equivalent (because of containment), the SELECT operation performed in S_j would also select an early completion structure from H_j. The SELECT instructions in both states would get added to the *ec*-queue. Since no new instructions become enabled, R_{i+1} is contained in S_{j+1}.

iii. *suspension:* If the item selected from H_i is a suspension, the instruction referenced in the suspension is signalled. By our property of containment, the item selected in S_j must also be a suspension and the instruction referenced in this suspension would also be signalled. The equivalent instruction would get signalled in both H_i and H_j and, thus, instruction equivalence is still preserved. The execution of the suspension operator changes the field to an early completion queue in both states containing the SELECT instruction. Since the two SELECT operators in R_i and S_j are equivalent, the early completion queues are also equivalent. Thus, R_{i+1} is complete wrt S_{j+1} in this case.

c. SET: By property of containment, the value SET using H_i must be equivalent to the one SET using H_j. In addition, every value in the *ec*-queue set with the value in H_i is equivalent to an element in the *ec*-queue set with the value in H_j. Thus, after these instructions become reenabled, instruction equivalence is still preserved and, thus, R_{i+1} is still contained in S_{j+1}.

3. **Terminate Operation:** Execution of the TERMINATE instruction causes control to return to the shell, with the *⟨name,value⟩* binding added to the user environment. Since R_i is contained in S_j, the operand to the terminate instruction must be equivalent in both R_i and S_j. Env_{i+1} is, therefore, equivalent to Env_{j+1}. Following the same argument given in (1) and we see that R_{i+1} is still contained in S_{j+1}.

4. **Return Operation:** The RETURN instruction takes two arguments, the return value and the target list. By the property of containment the return value and target list in the equivalent instructions in R_i and S_j must be equivalent. R_{i+1} is contained in S_{j+1} since the target addresses of the return operator are equivalent and the result values are equivalent in the two states.

5. **Tailapply Operation:** The TAILAPPLY instruction takes in three arguments, the function closure, argument list and return link. By the property of containment, all three operands must be equivalent in the executing instructions in corresponding states. The result of executing the instruction is to add a new activation and to signal instructions in its own activation. Because the closures and argument lists are equivalent, the instructions enabled in the new activation must also be equivalent in R_{i+1} and S_{j+1}. Following the argument given in (1), it is easily seen that R_{i+1} is contained in S_{j+1}. A similar argument can be applied in the analysis of the **streamtail** operator and is omitted here.

Since we have shown that the lemma holds for all instruction classes (excluding APPLY), the lemma is proved. □

The effect of executing a function is visible only in the result value returned by that function. Thus, given a computation sequence for some activation, the values that are still accessible after all activations in the computation sequence have completed are precisely those returned by that activation to its caller.

Lemma 2: Let $\langle S,S_1,\ldots,S_k\rangle$ be a computation sequence for an activation A. Then, the values accessible in S_{k+1}, but not in S are those values accessible from the instructions found in the destination list of the activation, A.

Proof: (*by contradiction*). Suppose there is some value, v, that is accessible in S_{k+1} but not in S and is also not accessible from the return link to the activation. This value must have been created by some instruction that is an element of some activation in the computation set of A. Let this activation be α. In order for this structure to be accessible after the termination of this activation, it must be returned as a result of that activation since any references to it from instructions within α are lost once the RELEASE instruction executes. Let its caller be β. In order for the value to be accessible in S_{k+1}, it must, by the same reasoning as above, be returned as a result of this activation as well. Continuing this argument, we see that the value can only be accessible in S_{k+1} if it is returned as the result of A. But this contradicts our original hypothesis and so the lemma is proved. □

Theorem 1: Let $\langle RTS,STS\rangle$ be a sequence pair for a system state S and let $R_i \in RTS$ and $S_j \in STS$. Then, if $R_i \vdash R_{i+1}$ and R_i is complete wrt S_j, then $\exists$ a state S_k s.t. $S_j \vdash^* S_k$ and R_{i+1} is complete wrt S_k.

Proof: Our proof is by induction. The induction step shows that the theorem holds over all instruction classes for our system.

Basis: Let $RTS = \langle R_1,R_f\rangle$ and $STS = \langle S_1,S_f\rangle$. Since $R_1 = S_1$, and by definition of the initial recovery state, $EIS_1 = \varphi$, $R_1 = R_f$ and $S_1 = S_f$. Thus, $R_f = S_f$ and the theorem is satisfied.

Hypothesis: Suppose that the theorem holds for all sequence pairs $\langle RTS,STS\rangle$ where RTS is of upto length i, $i > 2$.

Step: We show that the theorem holds for a sequence pair $\langle RTS, STS \rangle$ where RTS is of length $i+1$.

Non-Apply Instruction: Refer to Lemma 1.

Apply Instruction: There are four types of activation descriptors that are found on a computation record. These descriptors can either be of type: ***apply***, ***value***, ***tailapply*** or ***streamtail***. When an APPLY instruction executes in R_i, it examines the activation descriptor for the activation to be initiated.

1. ***apply*** or nonexistent *Ade*: If no *Ade* exists, then state R_{i+1} is complete wrt state S_{j+1} since the APPLY instruction executing in state R_i uses no backup information. Since R_i is complete wrt S_j and equivalent instructions execute in these two states, then by analysis similar to the one given for the TAILAPPLY operator in Lemma 1, R_{i+1} and S_{j+1} remain equivalent. This same analysis holds when the APPLY operator is to instantiate an activation which has an activation descriptor entry of type ***apply***. Since no result is found in the *Ade*, a new activation is created by the APPLY.

2. ***value***: If a ***value*** *Ade* exists for this activation, then R_{i+1} differs from R_i in that the value v is accessible from an activity in R_{i+1} but is not accessible from this same activity in R_i. The equivalent instruction executing in S_j would cause the instantiation of an activation, (u_j, A), because no backup information is used. By Lemma 2, there is a state transition sequence $\langle S_j, S_{j+1}, \ldots, S_{j+k} \rangle$ such that the values accessible in S_{j+k} but not accessible in S_j are those values accessible from instructions found in the destination list passed to the activity, A. These values represent the result of the activation and thus must be equivalent to the values found in the *Ade* of the activation used in R_i (since we have no non-deterministic computation). Since the APPLY instruction executing in R_i sends the value of the activation found on the *Ade* to all destinations, and by the property of containment, the two APPLY instructions executing in R_i and S_j have equivalent destination instructions, state R_{i+1} must be contained in state S_{j+k}. Since the environment image does not change in either state, R_{i+1} is complete wrt S_{j+k} and, so, the theorem holds for this case.

3. ***tailapply***: If the *Ade* for an activation has type TAILAPPLY, then R_{i+1} differs from R_i in that R_{i+1} will contain a new activity, A, whose argument record is retrieved from the activation descriptor. When the corresponding APPLY instruction executes in S_j, it will cause a new activity, A', to be created whose argument record is not necessarily the same as in A. Consider the state transition sequence, $\langle S_j, S_{j+1}, \ldots, S_{j+m} \rangle$ where $S_{j+m-1} \vdash S_{j+m}$ on a TAILAPPLY instruction which creates a new activity, B, such that $B \equiv A$. There must be such a transition sequence since all computation in the system is determinate. B is a descendent in the computation tree rooted at A'. Fix the transition sequence from S_j to S_{j+m} so that EIS_{j+m} contains no instruction from any activation on the path from A' to B (or descendents of any such activations) in the computation tree rooted at A'. This is clearly possible because of the non-determinacy of the *Choice* function. The only new values accessible in S_{j+m} not accessible in S_j are those referenced from B. Since $B \equiv A$, R_{i+1} is still contained

in S_{j+m}. Since the environment image is not updated in either transition sequence, environment equivalence is still preserved. Thus, R_{i+1} remains complete wrt S_{j+m}.

4. **Streamtail:** The case when the *Ade* is of type STREAMTAIL is very similar to the TAILAPPLY *Ade* given above. In this instance, R_{i+1} differs from R_i in two ways. First, the stream image on stable store is restored onto H_{i+1}. Secondly, a skeleton activation is created to instantiate production of the remainder of the stream. In state S_f the apply operator will cause a new activation, A, to be created to produce the first element of the stream. To satisfy the property of containment, we consider the transition sequence, $\langle S_f, S_{j+1}, ..., S_{j+m} \rangle$ where EIS_{j+m} contains no instruction from any from A (or any of its descendents) in the computation tree rooted at A. At this point, the first stream element is completely defined and there is a suspension in A which is not yet enabled to produce the next element. By our definition of object equivalence, the corresponding streams in S_{j+m} and R_{i+1} are equivalent. Thus, all destination instructions of the APPLY in the corresponding states which become enabled also remain equivalent. Hence containment still holds between S_{j+m} and R_{i+1}. Since the environment image does not change, R_{i+1} remains complete wrt S_{j+m}.

Since we have shown the theorem for each of the classes of *Ade*'s which may be found on a computation record, the theorem is proved. □

Corollary: If $\langle RTS, STS \rangle$ is a sequence pair, then R_f is complete wrt S_f where R_f and S_f are the final states for *RTS* and *STS* respectively.

Proof: (*by contradiction*)

Suppose that R_f was not complete wrt to S_f. Since $EIS_f = \varphi$, we know that R_f is contained in S_f. It must, therefore, be the case that environment equivalence is not preserved between the two states. By Theorem 1, we know that R_f must be complete wrt some state S_j where $S_j \vdash^* S_f$. As a consequence of the hypothesis, there must be some environment altering instruction executed in the transition sequence, $\langle S_j, S_{j+1}, ..., S_f \rangle$ that is not executed in *RTS* or vise versa. This is a contradiction since by definition 5, the same command stream is used in producing both the ideal recovery state and final recovery state. Thus, our hypothesis that R_f is not complete wrt S_f must be false and the corollary is proved. □

References

1. Ackerman, W. B. and Dennis, J. B. VAL --- A Value Oriented Algorithmic Language: Preliminary Reference Manual. 218, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1978.

2. Anderson, T., Lee, P. A., and Shrivastava, S. K. System Fault Tolerance. In *Computing Systems Reliability*, Anderson, T. and Randell, B., Eds., 1979.

3. Arvind and Gostelow, K. P. "The U-interpreter". *COMPUTER, 15,2* (Feburary 1982), 42-49.

4. Avizienis, et. al. " The STAR (Self-Testing And Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design". *IEEE-TC C-20*, 11 (November 1971), 1312-1321.

5. Barigazzi, G., and Strigini, L. Application Transparent Setting of Recovery Points. 13th Annual Symposium on Fault Tolerant Techniques, IEEE, 1983, pp. 48-55.

6. Benajamin, Arthur. Improving Information Storage Reliablility Using a Data Network. TM-78, Laboratory for Computer Science, MIT, Cambridge, Mass., 1976.

7. Borg, A., and Baumbach, J., and Glazer, S. A Message System Supporting Fault Tolerance. 6th Annual Symposium on Operating Systems, ACM-SIGOPS, 1983, pp. 90-99.

8. Dennis, J. B. First Version of a Data Flow Procedure Language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, Springer-Verlag, 1974, pp. 362-376.

9. Dennis, J. B. and Misunas, D. P. A Preliminary Architecture For a Basic Data-Flow Processor. Proceedings of the 2nd Annual Symposium on Computer Architecture, IEEE, December, 1975, pp. 126-132.

10. Dennis, J. B. Data Should Not Change: A Model for a Computer System. 209, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., July, 1981.

11. Dennis, J. B. An Operational Semantics for a Language with Early Completion Data Structures. Presented at the International Colloquium on The Formalization of Programming Concepts, Peniscola, Spain, April 19-25, 1981, 1981.

12. Dennis, J. B., and Gao, G. R., and Todd, K. W. A Data Flow Supercomputer. 213, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., March, 1982.

13. Dennis, J. B. and Stoy, J. E. and Guharoy, B. VIM: An Experimental Multi-User System Supporting Functional Programming. 1984 Conference on High-Level Architecture, 1984.

14. *1981 Conference on Functional Programming and Computer Architecture*, 1981.

15. J. Darlington et al.. *Functional Programming and its Applications : An Advanced Course.* Cambridge University Press, 1982.

16. Friedman, D. P., and D. S. Wise. CONS Should Not Evaluate its Arguments. In *Automata, Languages, and Programming*, unknown, 1976, pp. 257-284.

17. Guharoy, Bhaskar. Data Structure Management in a Data Flow Computer System. Master Th., Massachusetts Institute Technology,1985.

18. Hoare, C.A.R. "Communicating Sequential Processes". *Communciations of the ACM* (August 1978), 666-677.

19. Hughes, J. A. Error Detection and Correction Techniques for DataFlow Systems. 13th Annual Symposium on Fault Tolerant Techniques, IEEE, 1983, pp. 318-321. Also Center For Reliable Computing Memo 83-1, Computing Systems Laboratory, Stanford University.

20. Kahn, G., and D. MacQueen. Coroutines and Networks of Parallel Processes. Information Processing 77: Proceedings of IFIP Congress 77, August, 1977, pp. 993-998.

21. Lampson, B.W. and Sturgis, H. E. Crash Recovery in a Distributed Data Storage System. Xerox PARC, 1979. Internal draft.

22. Leung, C. K. C. Fault Tolerance in Packet Communication Architectures. TR-250, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

23. Liskov, B., and Scheifler, R. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. 210-1, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., Novemeber, 1981.

24. Misunas, D. P. Error Detection and Recovery in a Data-Flow Computer. Proceedings of the 1976 Conference on Parallel Computing, 1976, pp. 117-122.

25. Nelson, B. J. *Remote Procedure Call.* Ph.D. Th., Carnegie-Mellon University, May 1981.

26. Siewiorek and Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

27. Stern, J. A. Backup and Recovery of Online Information in a Computer Utility. TR-116, Laboratory for Computer Science, MIT, Cambridge, Mass., 1974.

28. Stoy, J. E. "VIM: A Dynamic Dataflow Implementation of VAL". *TOPLAS* (). To appear in ACM Transactions On Programming Languages And Systems.

29. Svobodova, L., and Liskov, B., and Clark, D. Distributed Computer Systems: Structure and Semantics. TR-215, Laboratory for Computer Science, MIT, Cambridge, Mass., 1979.

30. Weng, K.-S. An Abstract Implementation for a Generalized Data Flow Language. TR-228, Laboratory for Computer Science, MIT, Cambridge, Mass., 1979.